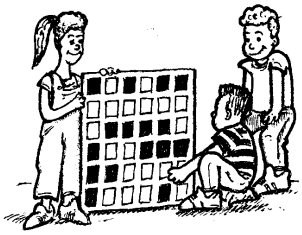
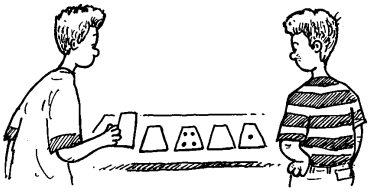
Computer Science Unplugged . . .
off-line activities and games
for all ages

Tim Bell

Ian H. Witten

Mike Fellows





`http://unplugged.canterbury.ac.nz`

Authors' contact details:

Tim Bell,
Department of Computer Science,
University of Canterbury,
Private Bag 4800,
Christchurch, New Zealand
`tim@cosc.canterbury.ac.nz`,
phone (+64 3) 364-2352

Ian H. Witten,
Department of Computer Science,
Waikato University,
Hamilton, New Zealand
`ihw@waikato.ac.nz`,
phone (+64 7) 838-4246

Mike Fellows,
Department of Computer Science,
University of Victoria,
Victoria, BC, Canada V8W 3P6
`mfellows@csr.UVic.ca`,
phone (604) 721-2399

About this book

Many important topics in computer science can be taught without using computers at all. This book unplugs computer science by providing twenty off-line activities, games and puzzles that are suitable for people of all ages and backgrounds, but especially for elementary school children. The activities cover a wide range of topics, from algorithms to artificial intelligence, from binary numbers to boolean circuits, compression to cryptography, data representation to deadlock. By avoiding the use of computers altogether, the activities appeal to those who lack ready access to computers, and are ideal for people who don't feel comfortable using them. The only materials needed are cards, string, crayons and other household items.

Full instructions are given for each activity, and reproducibles are provided wherever possible to minimize the effort required for class preparation. Each activity includes a background section that explains its significance, and answers are provided for all problems. All you need for most of these activities are curiosity and enthusiasm.

These activities are primarily aimed at the five to twelve year-old age group. They have been used in the classroom, in science center demonstrations, in the home, and even for community fun days in a park! But they are by no means restricted to this age range: they have been used to teach older children and adults too.

This book is principally for teachers who would like to give their classes something a bit different from the standard fare, teachers at the elementary, junior high, and high school levels. It is also written for computing professionals who would like to help out in their children's or grandchildren's classrooms, for parents who can use these as family activities, for homeschoolers, for science centers who run educational programs for children, for computer camps or clubs, and for course instructors—including university professors—who are looking for a motivational introduction to a computer science topic. It is designed for anyone who wants to introduce people to key concepts of the information age of which they have no knowledge themselves.

Topics include the Poor Cartographer (graph coloring), the Muddy City (minimal spanning trees), Treasure Hunt (finite-state machines), the Peruvian Coin Flip (cryptographic protocols), Magic Card Flips (error correcting codes), the Chocolate Factory (human–computer interaction), and many more.

So unplug your computer, and get ready to learn what computer science is really about!

Acknowledgments

Many children and teachers have helped us to refine our ideas. The children and teachers at South Park School (Victoria, BC), Shirley Primary School, and Ilam Primary School (Christchurch, New Zealand) were guinea pigs for many activities. We are particularly grateful to Linda Picciotto, Karen Able, Bryon Porteous, Paul Cathro, Tracy Harrold, Simone Tanoa, Lorraine Woodfield, and Lynn Atkinson for welcoming us into their classrooms and making helpful suggestions for refinements to the activities. Gwenda Bensemann has trialed several of the activities for us and suggested modifications. Richard Lynders and Sumant Murugesh have helped with classroom trials. Parts of the cryptography activities were developed by Ken Noblitz. Some of the activities were run under the umbrella of the Victoria "Mathmania" group, with help from Kathy Beveridge. The delightful illustrations were done by Malcolm Robinson and Gail Williams, and have also benefited from advice from Hans Knutson. Matt Powell has provided valuable assistance with the "Unplugged" project.

Special thanks go to Paul and Ruth Ellen Howard, who tested many of the activities and provided a number of helpful suggestions. Peter Henderson, Joan Mitchell, Nancy Walker-Mitchell, Jane McKenzie, Gwen Stark, Tony Smith, Tim A. H. Bell[1], Mike Hallett, and Harold Thimbleby also provided numerous helpful comments.

We owe a huge debt to our families: Judith, Pam, and Roberta for their support, and Andrew, Anna, Hannah, Max, Michael, and Nikki who inspired much of this work,[2] and were often the first children to test an activity.

We welcome comments and suggestions about the activities. Details about contacting the authors are given on page 227 in the conclusion.

[1] No relation to the first author.

[2] In fact, the text compression activity was invented by Michael.

Contents

Introduction

Computers are everywhere. Though you may not realize it, it's unlikely that you'll get through the day without using one. Even if you don't have one at home, and you don't use a bank, and you avoid the checkout at the supermarket—and the corner store—you'll probably end up using a computer disguised as a VCR, microwave, or video game. When you graduate from school, it will be hard to find a career that doesn't involve computers. They seem to be taking over! With computers around every corner, it makes sense to find out how they work, what they can do—and what they can't do. The activities in this book will give you a deeper understanding of what computers are about.

The reason this book is "unplugged" is that we are concerned about presenting the ideas and issues of computer science, and these are often easier to explain with paper and crayons, ordinary materials, and simple activities. We believe you will be surprised and delighted, as we are, with how much entertainment can be had with simple things animated by the ideas important in understanding computers. Rather than talking about chips and disks and ROM and RAM, we want to convey a feeling for the *real* building blocks of computer science: how to represent information in a computer, how to make computers do things with information, how to make them work efficiently and reliably, how to make them so that people can use them.

Pressing social issues are raised by computing technology, like how information can be kept private and whether computers will ever be as intelligent as us. There are performance issues: computers are mind-bogglingly fast, yet people are always complaining that their computer is too slow. And there are human issues: why do people get so frustrated using computers? Are computers getting out of control? Behind these issues lie important technical factors, and the activities in this book will help you understand what they are.

We have selected a wide range of topics from the field of computer science, and packaged them so that they can be learned without using a computer. They will give you a good idea of what computers can and can't do, what they might be able to do in the future, and some of the problems and opportunities that computer scientists face now.

But the main reason that we wrote this book is because computer science is *fun*. You don't believe us?—read on! The subject is bursting with fascinating ideas just waiting to be explored, and we want to share them with people who might not be tuned in to computers, but would be interested in the ideas in computer *science*. These activities will certainly give you something to think about.

The activities and how to use them

The activities are divided into six topics: data (*representing information*), putting computers to work (*algorithms*), telling computers what to do (*representing procedures*), really hard problems (*intractability*), sharing secrets and fighting crime (*cryptography*), and the human face of computing (*interacting with computers*). These areas are representative of the kinds of things that computer scientists study, although the list is far from exhaustive. Each topic has a brief introduction explaining where it fits into the wider picture, followed by several activities.

All the activities begin with a summary of materials needed. An age group is given, but it is merely indicative—the work can be adapted for anyone who has the basic skills, and a great deal depends on the children's background. There is no maximum age. Although nearly all the activities are designed for elementary school children, many have been used with older children, and even to introduce topics to university students. A time is indicated for each activity: this also is very approximate. The activities generally take about 30 to 40 minutes to complete, although they can be adapted to suit the time available. Some can serve as extended projects, and most can be cut down to a five-minute demonstration—perhaps as an illustration in a lecture or seminar. We indicate whether there is a minimum or maximum number of people required for an activity. Although the activities are described for a classroom situation, you will find that most can be undertaken individually or with the help of a second person such as a parent or friend.

Each activity description has the same structure. They begin with a *focus* section, which lists some of the key skills that are developed by the activity, and a *summary* section, which provides some background to the activity. There is a list of *technical terms* that might seem like jargon, but will be useful if you want to pursue the topic into the primary scientific literature. The *materials* part gives a checklist of what is required to undertake the activity with a class.

The next two sections take you through the activity. *What to do* is a step-by-step explanation of how to present it to children. The steps are the result of experimenting with several approaches, but you should feel free to adapt them to suit your group. *Variations and extensions* suggest alternative ways to present the activity, and ideas for extending it.

What's it all about? is intended to fill the teacher in on the wider significance of the activity. Older children might appreciate hearing about some of this—you might even have them read the section themselves. Younger children may not be able to comprehend the bigger picture, and it is often best to let them enjoy the activity in its own right, perhaps with a simplified account of its significance. This section is also intended for parents. Some will be curious, and others will want to talk about the activities with their child at home. It may also prove useful for justifying the activities to parents or supervisors who are not convinced of their value.

Further reading provides references to books and papers on the topic. Many of the references are fairly technical, but we have tried to include ones written for lay people where possible. A general reference that provides an accessible introduction to many of the topics in the activities is David Harel's book *Algorithmics: The Spirit of Computing* , which was published in a slightly less technical form as *The Science of Computing*. Complete bibliographic information about all material mentioned appears at the end of the book.

Many of the activities deserve a reward for good work. We have made the picture on page 6 into a stamp, which we dispense freely on worksheets and the backs of hands in return for

a good effort. Any rubber stamp shop will be able to make one for you from a copy of the picture. Another reason for using this particular stamp is that it reminds children that the work is about computers, which is important because there is not much mention of computers during the activities.

For the teacher

Don't be put off from using these activities just because you feel you don't know much about computers. Despite the technical nature of the topics, this book is intended specifically for teachers who have no background in computer science. We find that such people often enjoy the activities as much as the children. The activities are designed to provide opportunities for teachers and students to learn together about the principles of computer science. To assist you, every activity has a section that explains its background in non-technical terms. Answers to all the problems are provided so that you can confirm that you have things right.

Most of the activities can be used to supplement a mathematics program, but some (particularly Activity 19 about human interface design and Activity 20 about artificial intelligence) are also appropriate for social programs. The *focus* section identifies skills that the children will exercise, ranging from representing numbers in base two to coloring, from logical reasoning to interviewing. Topics and skills can be located using the index at the back of the book.

If you aren't experienced with working in a classroom ...

These activities can be used by computing professionals who would like to communicate computer science ideas to children or other general audiences. They are ideal for those occasions when you are asked to present something about your profession, but realize that the computers that you work with are probably not nearly as impressive as the video games that the children use every day.

If you are going to use these activities in a classroom situation, particularly with younger children, getting some help from a teacher about keeping order in the classroom can make the time more productive, and avoid frustration. There are some fairly safe techniques that will work in general (such as "I'll choose someone sitting quietly with their hand up"), although the best method varies from class to class. Some teachers have reward systems that you can use, or you can bring your own rewards—such as stickers, or the stamp on page 6. There may be a signal that the teacher uses to indicate that everyone must be quiet. This sort of information can be very valuable!

For somewhat exploratory and open-ended activities such as these, it is natural to expect a range of responses from individual students. One effective strategy is to employ those students who rapidly understand what's going on to explain things to others who are having more difficulty. Younger students may become quite animated—more so than the university students to whom you might be accustomed! The goal is to do something interesting and to have fun; a certain amount of chaos is not necessarily a bad thing.

Don't forget to check the list of materials required: it's easy to forget an important piece of equipment. Often the materials will be available from the school (such as balance scales,

crayons, and chalk). If you are not involved in the teaching profession, you may not realize that the probability of a photocopier breaking down is particularly high in the five minutes before a class for which you need a set of handouts.

Many of the activities involve solving problems. In most cases, the intention is *not* to explain how to solve the problem, but to allow the children to understand it and find their own method of solution. Knowing an algorithm that solves a problem has little intrinsic value, but the process of discovering an algorithm (even if it is not the best one) can be very instructive. In some cases there is no good algorithm; rather, the intention is for the children to learn about the inherent complexity of the problem. Sometimes there will be an opportunity to explain to the children how a particular problem applies to the real world, but often it will suffice for children to see the elegance of a solution, or appreciate that some problems don't have easy solutions.

Above all, there is no substitute for enthusiasm and being well prepared. Children appreciate both of these and respond accordingly. And if you are from outside the class you have the advantage that the children may be more attentive because you are offering a break from the daily routine.

For the technically-minded

As well as providing a taste of computer science for children, we have included some material for those who would like to delve a little deeper into the subject. Like this one, these sections are marked "for the technically-minded."

One of our goals is to communicate what computer science is really about. Computer *science* is a rich subject concerned with what computers can and cannot do, how to approach problems, and how to make computers more valuable to their users. Very little computer science is taught in elementary and high school. Most students will do some sort of "computing" work, but usually it is about how to *use* computers rather than how to design them for other people to use. A common misconception is that computer science is about programming. Programming is a fundamental tool, but it is not the end in itself. Computer science is about programming in the same way that astronomy is about telescopes—just as learning astronomy need not involve understanding how a telescope is built, learning about computer science does not need to involve programming. In fact, much of computer science would exist even if computers didn't (although it would probably have a different name!) None of the activities in this book require a computer, but the principles that they teach are widely used in modern computers.

Few youngsters—or even adults—are aware of what computers really can and can't do. For example, many real-life optimization problems, such as time-tabling teachers and classes, or finding the shortest route to make deliveries, take too long to solve optimally on computers—regardless of how fast the computer is. There are even problems, such as solving some equations (known as Diophantine equations), that we can prove will *never* be solved by a computer. There are lots of things that computers can't do.

Many things that computers *can* do are also not widely known. For example, many people use debit cards to pay for goods, where money is transferred directly from their bank account to the store's. However, in the process the bank finds out where the purchase is being made, and could build up a profile of the person's shopping habits. Despite this loss of privacy, debit

cards are widely accepted. Most people are unaware that cryptographic protocols exist which enable the transaction to be carried out reliably *without the bank being able to identify who the money is going to!* This seems incredible—literally unbelievable—to people who have never encountered public key cryptosystems and information hiding protocols. If more people know about such things, there may well be an outcry for systems that protect privacy better. (Activity 16 on information hiding protocols demonstrates a situation where it is possible to exchange information without losing any privacy). Understanding the technical issues involved goes a long way to making informed decisions on privacy issues, just as an understanding of biology goes a long way to making informed decisions on environmental issues.

We hope that this book will take some of the science fiction out of people's understanding of computers. The upcoming generation of computer users deserves a clear view of the technical issues that underpin the myriad of computerized systems that permeate our lives.

To find out more about the "Unplugged" project, visit the web site at

`http://unplugged.canterbury.ac.nz/`.

Instructions: *Have this picture made into a rubber stamp (about half this size) and use it as a reward for good work.*

Part I

Data: the raw material—*Representing information*

Data is the raw material that computers work on. People used to think of computers as giant electronic calculators. But nowadays it's better to think of a computer as a cross between an electronic filing cabinet, a library, and a TV. Calculators work with numbers. But computers work with data of any kind: baseball lore, sports facts, letters, mailing lists, accounts, payrolls, advertisements, magazines, books, encyclopedias, music, movie listings—even movies themselves. Calculators do arithmetic on numbers. Computers can do that too. But more importantly, they can manipulate all sorts of data, looking for high-scoring players, predicting the outcome of baseball games, helping to write letters, address envelopes, balance accounts, print checks, distribute advertisements, lay out magazine pages, find information in books and encyclopedias, play music, look through movie listings—they can even show movies. What is really amazing is that all of this wide range of facts, documents and images are stored on a machine that, at the lowest level, only works with two things: zero and one!

In this part of the book we look at how different kinds of information can be represented by a computer. Internally, all computers store data in an electronic form that is based on a very simple idea: that everything can be coded as a sequence of the digits zero and one. You, the user, do not normally see these things because the data is presented in a human-readable form—who wants to see a bunch of numbers instead of a movie? But to understand what computers do, you need to know what it is that they work with, and how numbers, letters, words, and pictures can be converted into zeros and ones.

For teachers

Representing information is absolutely fundamental to computing. Although the difference is hard to pin down precisely, data, which dictionaries define as "numerical information in a form suitable for processing by computer," is subtly different from information, or "knowledge derived from study, experience, or instruction." We refer to the stuff that is stored and manipulated by computers as *data*, and the real-world entities that are so represented as *information*. Thus data is the raw material of computing, while information is the raw material of computer applications.[3] Information is converted into data for the computer, and data is presented as information to the user.

The five activities in this section provide a broad introduction to information representation. The first is about binary numbers, disguised as a game that even very young children enjoy playing. The second, a kind of "paint by numbers" activity, shows how pictures can be represented as numbers, and embodies conventions used in fax machines for transmitting images over phone lines. The next two activities are about ways of representing information that are efficient and reliable. Data often consumes vast quantities of computer disk space—this is particularly true of multi-media material containing sound and images. Computer users will testify that they never seem to have quite enough disk space, and so computer scientists are concerned not just with how to store data, but how to store it efficiently. Activity 3 shows how ordinary text can be represented as numbers in an efficient way. The next activity is about maintaining the integrity of the data. The media

[3] We use *data* in the singular, because it usually seems like a large—often formidably large—entity in itself, rather than a collection of individual "datums."

on which data is stored (and transmitted) is susceptible to the occasional error, and it is important that the effect of errors is negligible. Disguised as a magic trick, we introduce a widely-used technique to detect and correct minor errors in computer data. The final activity in this section is about how information can be quantified. Because the idea of information is so central to computer science, a whole theory has been developed that enables us to quantify information and find limits on how efficiently it can be stored and transmitted.

Taken together, these activities will help children understand how different kinds of information can be stored on a computer, without taking up more space than necessary, and in a way that decreases the chance of loss of data due to defects in the storage medium. They will also learn about how to measure the information content of data. The activities can be performed in any order, though it is helpful if Activity 1 is done first. The first three are suitable for children in early elementary school, while for the last two the children need to be at the middle elementary level. However, these are lower bounds: even quite advanced children enjoy these activities and learn from them, as do adults.

For the technically-minded

Many computer professionals will be surprised at the content of these activities. Progressing from binary numbers, through picture representation, to text compression, error control, and foundations of information theory, constitutes a rather unusual introduction to computer science. Indeed, many students of the subject are unfamiliar with some of these ideas.

No-one would disagree that information representation is basic to computing. But the traditional fare of integers, floating-point numbers, and character strings conveys an impoverished and pedestrian view of information. Pictures and text are what users see, and in this sense they are the fundamental data types in computing today. Instead of working toward structured data—such as lists, trees, and object hierarchies—the way many computing technology and programming courses do, we pursue a user-oriented track. Efficient storage of information, and its integrity, are the major issues of concern. A feeling for how information is quantified underpins our understanding of what it is that computers work with. These topics provide a view of computing that young children can relate to, and they lend themselves naturally to simple but intriguing activities.

Activity 1

Count the dots—*Binary numbers*

Age group	Early elementary and up.
Abilities assumed	Counting up to 15 or 31, matching, sequencing.
Time	10 to 40 minutes.
Size of group	From individuals to the whole class.

Focus

Representing numbers in base two.

Patterns and relationships in powers of two.

Summary

All data in a modern digital computer is ultimately stored and transmitted as a series of zeros and ones. This activity demonstrates how numbers and text can be represented using just these two symbols.

Technical terms

Binary number representation; binary to decimal conversion; bits and bytes; character sets.

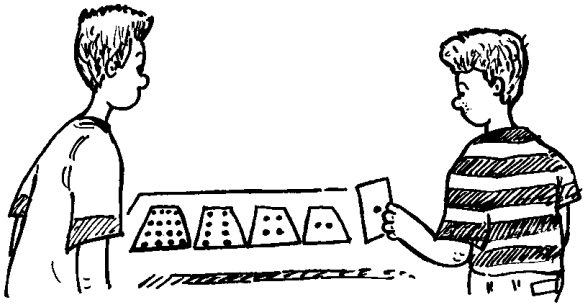


Figure 1.1: Initial layout of the binary cards

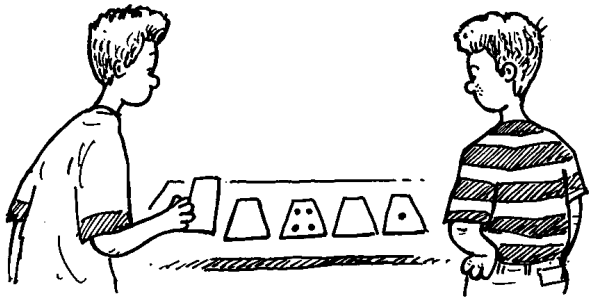


Figure 1.2: Flipping the cards to show five dots

Materials

Each child will need:

one set of five cards from the blackline master on page 17 (the blackline master has two sets),

a copy of the blackline master on page 18, and

a pen or pencil.

What to do

1. Seat the children where they can see you, and give each child a set of cards.
2. The children should lay their cards out, as in Figure 1.1, with the 16-dot card to their left.

   Some children will be tempted to put the cards in the opposite order, so you should check that they are in descending numeric order from left to right. For younger children, do not use the 16-dot card.

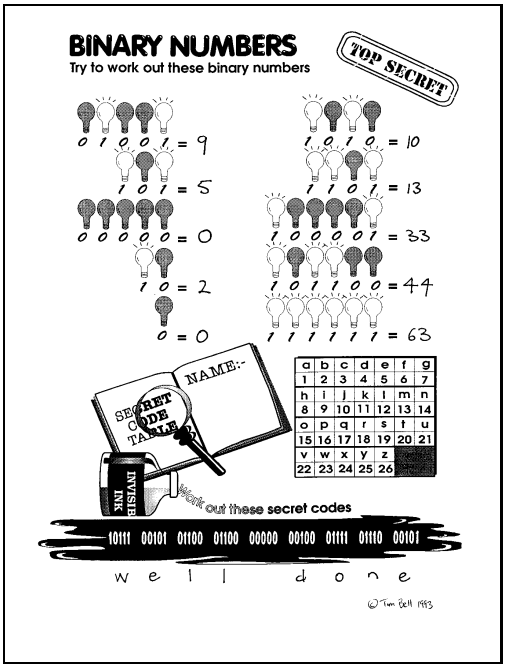


Figure 1.3: Solution to the worksheet on page 18

3. Have the children work out which cards to flip over so that exactly five dots are showing. The only (correct) way to do this is to have the 4-dot and 1-dot cards face up, and the rest face down (Figure 1.2). Each card must be either face up or face down, with all or none of its dots showing. Be prepared for some novel ways of getting five dots—it is not unusual for children to produce the requisite number by using spare cards to cover up three of the dots on the eight card!

4. Now get the children to show other numbers of dots, so that they explore which numbers can be represented.

   Ask for numbers such as three (requires cards 2 and 1), twelve (8 and 4), nineteen (16, 2 and 1) and so on. For those who find the combination for a number quickly, ask if they can find another way to get the number (there is only one way to display each number, and they are likely to discover this eventually).

   Discuss what the biggest number is that can be made with the cards (it is 31 for five cards, 15 for four cards). The smallest? (Often the number one will be offered first, but the correct answer is zero.) Is there any number between the smallest and largest that can't be represented? (No—all numbers can be represented, and each has a unique representation.)

5. For older children, ask them to display the numbers 1, 2, 3, 4, ... in sequence, and see if they can work out a procedure for incrementing the number of dots displayed on the cards by one (the number of dots increases by one if you flip all cards from right to left until you turn one face up).

6. This part of the activity uses zeros and ones to represent whether a card is face up or not.

   Tell the children that we will use a 0 to show that a card is hidden, and a 1 if its face is showing. For example, the pattern in Figure 1.2 is represented by 00101. Give them some other numbers to work out (e.g. 10101 represents 21, 11111 represents 31). With some practice the children will be able to convert in both directions. You could ask children to take turns calling out the day of the month that they were born on using zeros and ones, and have the rest of the class interpret the date.

   This representation is called the binary system, also known as base two.

7. Use the worksheet on page 18 to extend the exercise. (A completed worksheet is shown in Figure 1.3.)

   The worksheet uses a light bulb that is switched on to represent a card that is showing, and a light bulb that is off to represent a hidden card. The first few patterns should be easy to work out. For example, the first pattern has the 8 and 1 cards showing, so the value represented is $8 + 1 = 9$. For the patterns with fewer than five light bulbs, the children should use only the smaller valued cards. For example, the second pattern has only three light bulbs, which correspond (from left to right) to the 4-, 2-, and 1-dot cards respectively. See if the children can work this out for themselves.

   The six-bulb questions are designed to make the children think about how many dots should be on a sixth card. The number of dots on each card is double the number on the previous one, so the sequence is 1, 2, 4, 8, 16, 32, 64 .... Thus a 32-dot card would be added (at the left) to solve a problem that needs six cards.

   The code at the bottom of the worksheet uses the numbers 1 to 26 to represent the letters of the alphabet. (A zero can be used to represent a space.) The children must work out what each number in the code is, and look up the corresponding letter in the table. This shows how a textual message can be converted to a series of zeros and ones. The children can then write coded messages for each other.

Variations and extensions

Instead of using cards with dots, the exercise can be done with the lengths of rods (a set of rods of lengths 1, 2, 4, 8 and 16 units can be used to create any length from 0 to 31) or with weights (a set of weights of 1, 2, 4, 8 and 16 units can be combined to produce any weight from 0 to 31).

Instead of calling out sequences like 01101, try using beeps—call out a high-pitched beep for a one and a low-pitched one for a zero. This activity is noisy in the classroom, but children find it memorable! Modems and fax machines use tones like this to transmit information, although the tones are sent so quickly that they blend to make a continuous screeching sound. If the children aren't familiar with this, they could try calling a fax machine number to hear what it sounds like.

Any objects that have two states can be used to represent numbers. Figure 1.4 shows some different ways of representing the number nine (01001). A particularly challenging method is to use your fingers. If a finger is up it represents a one; down represents zero. Counting on your

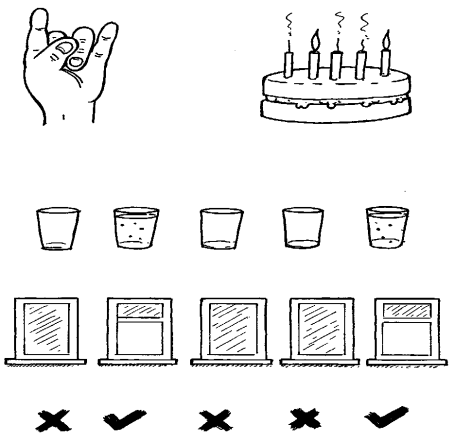


Figure 1.4: Some unusual ways of representing the number nine (01001 in binary)

fingers using the binary system enables you to go up to 31 on one hand, and 1023 on two hands. It requires some dexterity, and you have to watch out for rude gestures along the way! For a real challenge, try using your toes as well—this will allow you to count up to more than a million. (How many exactly? Two hands give 1024 possibilities, 0 through 1023. Hands and toes give $1024 \times 1024 = 1,048,576$ possibilities, 0 through 1,048,575.)

Older children will enjoy extending the sequence 1, 2, 4, 8, 16, 32 . . . The sequence contains an interesting relationship: if you add the numbers from the beginning from left to right, the sum will always be one less than the next number in the sequence.

Another property of binary numbers is that you can double the number by inserting a zero on the right-hand side of a number. For example, 1001 (9) doubled is 10010 (18). Older children should be able to explain why this happens. (All of the places containing a one are now worth twice their previous value, and so the total number doubles. The same effect occurs in base ten, where inserting a zero on the right of a number multiplies it by ten.)

Binary numbers are closely related to the guessing game in which one person thinks of a number and someone else tries to guess it by asking questions of the form "is it greater than or equal to x?" For example, suppose the number is known to be less than 32. A sensible first question would be "is it less than 16?" The yes/no answers to the questions are given by the zero/one bits in the binary representation of the number. This is explored in detail in Activity 5.

The five-bit code used for letters does not allow both upper- and lower-case letters to be represented. You could have the children work out how many different characters a computer has to represent (including digits, punctuation, and special symbols such as $), and consequently how many bits are needed to store a character. (With two lots of 26 letters, 10 digits, and a few punctuation marks, there are bound to be more than 64 codes needed, so at least seven bits are necessary. Seven bits allows for 128 characters, and this is more than sufficient.) Most current computers use a representation called ASCII (American Standard Code for Information Interchange), which is based on using seven bits per character. Longer codes that allow for the languages of non-English speaking countries are now becoming common.

What's it all about?

Modern digital computers almost exclusively use the system described above to represent information. The system is called binary because only two different digits are used. It is also known as base two (as opposed to base ten, which humans normally use). Each zero or one is called a "bit", the term being a contraction of *binary digit*. A bit is usually represented in a computer's main memory by a transistor that is switched on or off, or a capacitor that is charged or discharged. On magnetic disks (floppy disks and hard disks), bits are represented by the direction of a magnetic field on a coated surface, either North-South or South-North. CD-ROMs store bits optically—the part of the surface corresponding to a bit either does or does not reflect light. When data must be transmitted over a telephone line or radio link, the ones and zeros are commonly represented by high- and low-pitched tones.

One bit on its own can't represent much, so they are usually grouped together as in the exercise above. It is very common to store bits in groups of eight, which can represent numbers from 0 to 255. A group of eight bits is called a byte.

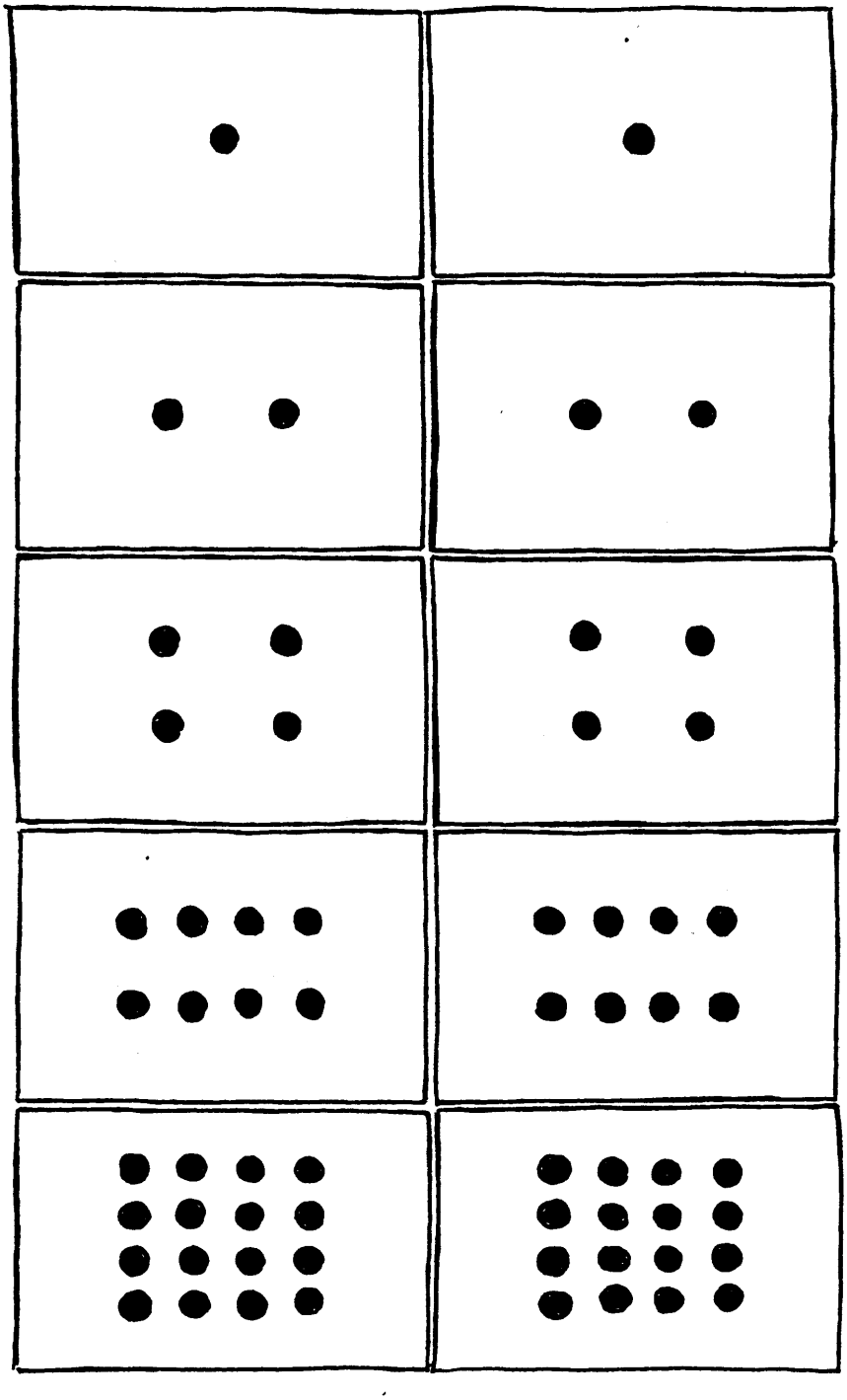
As well as representing numbers, this code can also represent the characters in a word-processor document. A byte is often used to represent a single character in a text—the numbers 0 to 255 are more than enough to encode all the upper- and lower-case letters, digits, punctuation, and many other symbols.

To represent larger numbers, several bytes are grouped together. Two bytes (16 bits) can represent 65,536 different values, and four bytes can represent over 4 billion values. The speed of a computer is affected by the number of bits it can process at once. For example, a 32-bit computer can perform arithmetic and manipulations on 32-bit numbers, whereas a 16-bit computer must break large numbers into 16-bit quantities, making it slower.

Normally we don't see the bits and bytes in a computer directly because they are automatically converted to characters and numbers when they are displayed, but ultimately bits and bytes are all that a computer uses to store numbers, text, and all other information.

Further reading

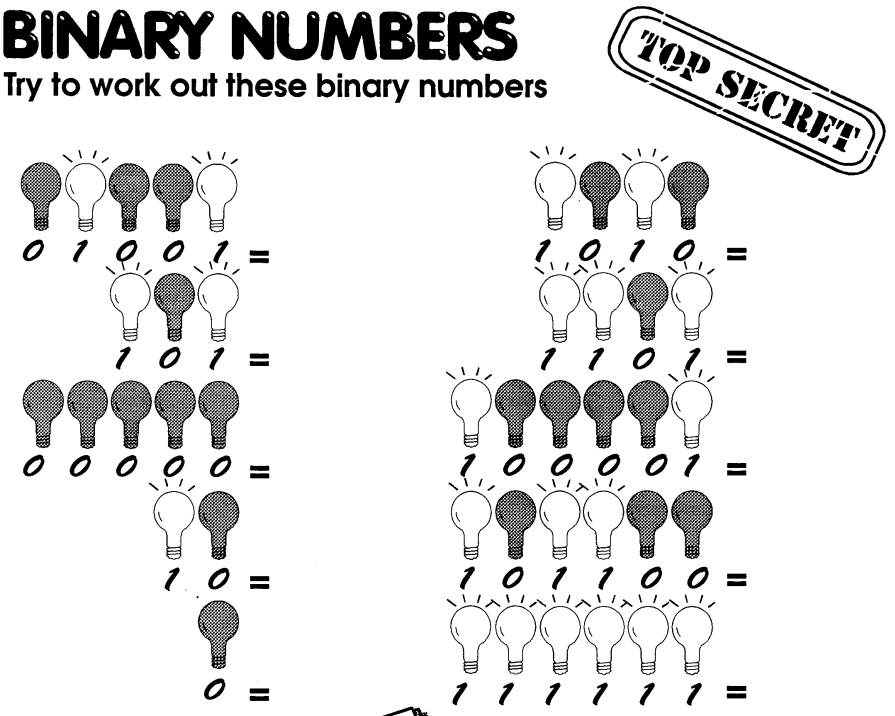
Most introductory computing texts discuss the binary number system. *My friend Arnold's book of Personal Computers* by Gareth Powell has a whole chapter on binary numbers.



Instructions: *Copy this page onto card, and cut out the boxes to make two sets of five cards.*

BINARY NUMBERS

Try to work out these binary numbers



a	b	c	d	e	f	g
1	2	3	4	5	6	7
h	i	j	k	l	m	n
8	9	10	11	12	13	14
o	p	q	r	s	t	u
15	16	17	18	19	20	21
v	w	x	y	z		
22	23	24	25	26		

Work out these secret codes

10111 00101 01100 01100 00000 00100 01111 01110 00101

Instructions: *Work out the numbers represented by the lightbulbs at the top of the page. Also, there is a message coded in binary at the bottom of the page; work out the numbers and look them up in the table to get the message.*

Activity 2

Color by numbers—*Image representation*

Age group	Early elementary and up.
Abilities assumed	Elementary counting, and some concentration. Activity 1 (Count the Dots) is helpful, but not essential, preparation. Experience with graphing is also useful.
Time	20 minutes or more.
Size of group	From individuals to the whole class.

Focus

Representation.

Coloring.

Pictures.

Summary

Computers are often used to store drawings, photographs and other pictures. This activity shows how pictures can be represented efficiently as numbers in a computer.

Technical terms

Raster images; pixels; image compression; run-length coding; facsimile machines.

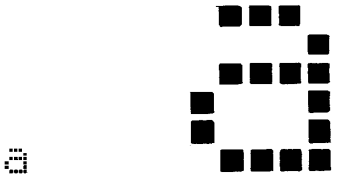


Figure 2.1: The letter “a” from a computer screen, and a magnified view showing the pixels that make up the image.

Materials

Each child will need:

a copy of the blackline master on page 25, and

a pencil and eraser.

You will need:

an overhead projector transparency of Figure 2.1 (or draw it on the classroom board).

What to do

1. Discuss what facsimile (fax) machines do (arranging for the children to send and/or receive faxes would be excellent preparation for this activity). Ask for other places where computers store pictures (for example, a drawing program, a game with graphics in it, or a multi-media system).

   Explain that computers can really only store numbers (if they have done the activity on binary numbers then they will already have some appreciation of this). Have the children suggest how a picture might be represented using only numbers.

2. Demonstrate how images are displayed on a computer screen, as follows:

   Computer screens are divided up into a grid of small dots called *pixels*. The word pixel is a contraction of “picture element.” On a black and white screen, each pixel is either black or white. Figure 2.1 shows a picture of a letter “a” that has been magnified so that the dots are visible. (This image should be shown on an overhead projector, or drawn on the board.) When a computer stores a picture, all that it needs to store is which dots are black and which are white.

3. Using the image of Figure 2.1, demonstrate how pictures can be represented by numbers on the blackboard, as follows:

   Begin by writing down the number of consecutive white pixels in the first line of the image (in Figure 2.1 there is only one white pixel at the start of the first line). Then write

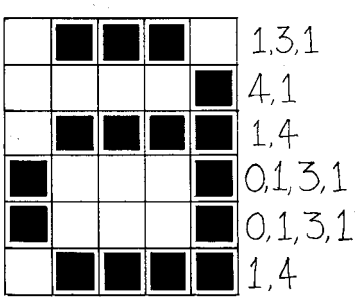


Figure 2.2: The image of Figure 2.1 coded using numbers

down the number of consecutive black pixels (three), and so on until the whole line has been coded. Thus the first line of Figure 2.1 will be represented as 1, 3, 1. Each row of pixels in the image is transcribed using this method. For example, the second line of Figure 2.1 is represented by 4, 1, since it has four white pixels followed by one black one. Figure 2.2 shows the final representation of Figure 2.1. Notice that lines beginning with a black pixel have a zero at the beginning to indicate that there are no white pixels at the start of the line.

There may be some confusion with this demonstration if you use a chalkboard, because coloring in a pixel makes it white rather than black!

4. Have the children "decode" the pictures on the worksheet on page 25.

   Hand out the worksheets and get the children to decipher the images there (see Figure 2.3 for the completed images; Figure 2.4 contains a reduced version which makes the pictures clearer). The "test" picture on the right is the easiest, and the "being" on the left is the most complex.[1] The squares on the grid should be colored in completely, and made as dark as possible. If the squares on the blackline master are too small, you can use quad-ruled paper. It is important to use pencil, as mistakes are easily made! Children should mark each line of numbers as they finish with it. Some may prefer to circle all the numbers that represent black pixels to help them keep track of where they are up to—it is very easy to get confused about which numbers are for black and which are for white.

   Note that each line always begins with the number of white pixels. If the first pixel is black, the line will begin with a zero.

Variations and extensions

The image will be clearer if the children draw on a sheet of tracing paper on top of the grid, so that the final image can be viewed without the grid.

Instead of coloring in the grid the class could use squares of sticky paper on a larger grid, or place any objects on a grid. The codes could even be used for a cross-stitch pattern.

[1] The images are taken from the excellent children's drawing program *KidPix*, by Brøderbund Software.

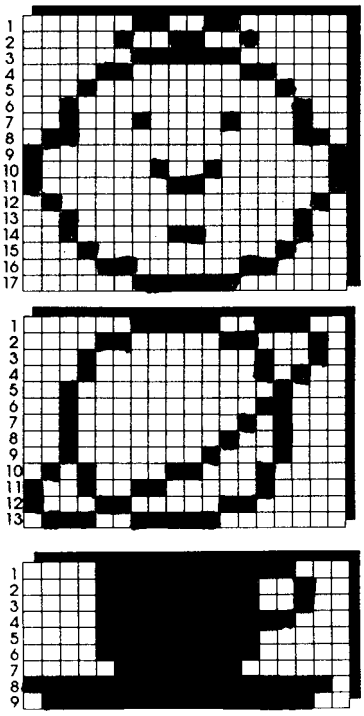


Figure 2.3: The completed images

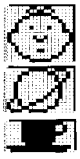


Figure 2.4: The completed images reduced

Children can try drawing their own pictures on a grid (or try to copy one from a computer screen). They can code the picture as numbers and give it to friends to decipher.

In practice there is usually a limit to the maximum length of a run of pixels because the length is being represented as a binary number. Ask the children how they would represent a run of twelve black pixels if they could only use numbers up to seven. (A good way is to code a run of seven black pixels, followed by a run of zero white, then a run of five black.)

The method can be extended to colored images by using a number to represent the color (e.g. 0 is black, 1 is red, 2 is green etc.) Two numbers are now used to represent a run of pixels: the first gives the length of the run as before, and the second specifies the color.

What's it all about?

The simplest way to represent a black and white picture on a computer is to represent white pixels with a zero (say) and black with a one. However, it is common for images to contain large blocks of white pixels (especially in the margins) and runs of black pixels (e.g. a horizontal line). Facsimile documents commonly have about 100 pixels per inch, so a row of white pixels at the top of a 7 inch wide page could take 700 bits of storage.

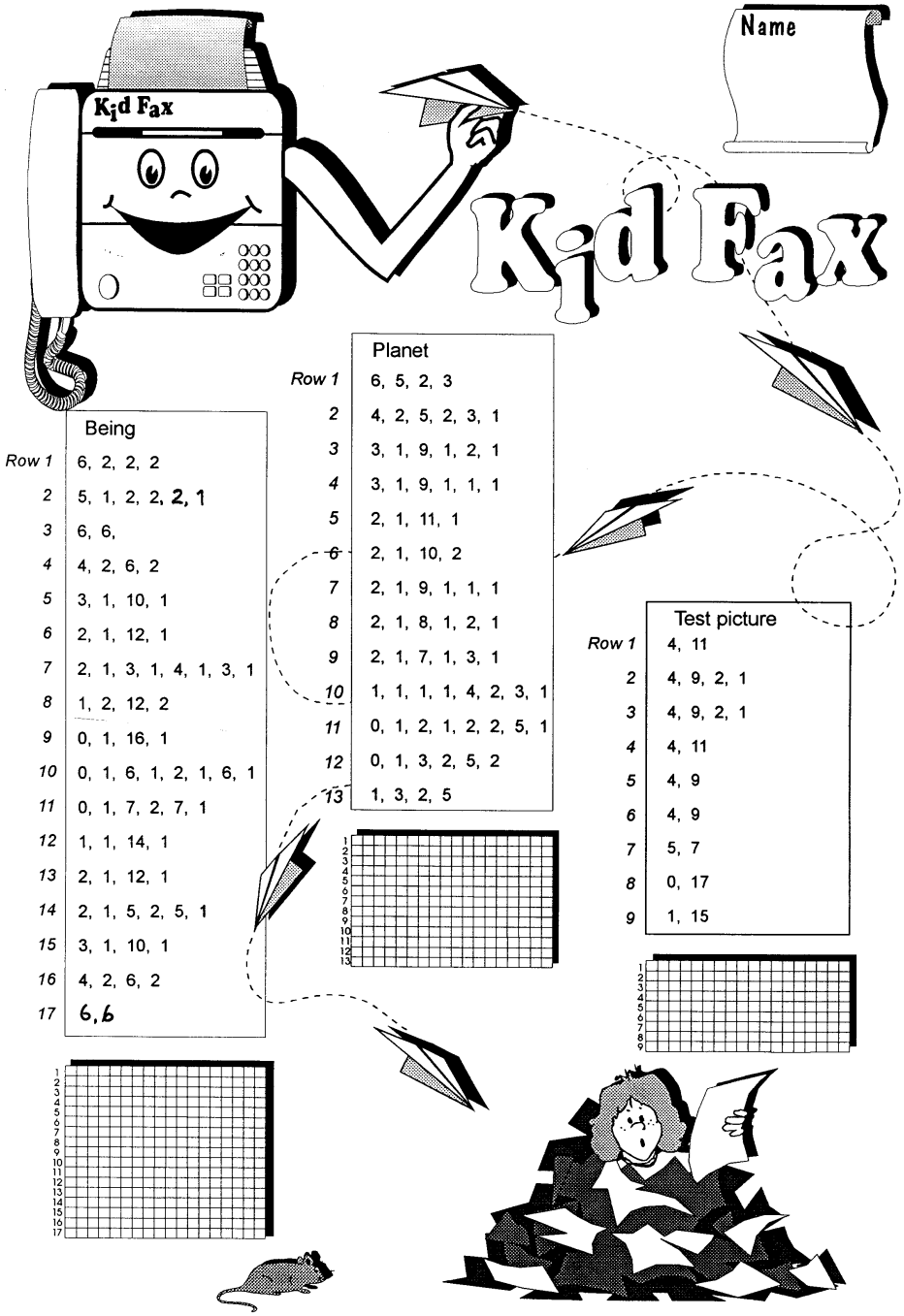
The "run-length coding" that we used in the activity above is much more efficient. It represents the run-length of 700 using the binary system (see Activity 1), which can be done in 10 bits. This example is a little extreme, but nevertheless significant savings can be made.

Saving space using techniques like this is called *compression*, and it is crucial to the viability of working with images on computers. For example, fax images are generally compressed to about a seventh of their original size. Without compression they would take seven times as long to transmit, which would make fax a much less attractive means of communication. Images stored on computer disks are often compressed to a tenth or even a hundredth of their original size. Without compression there might be room for just a handful of images on the disk, but with compression many more images can be stored. The advantage is even greater when storing moving pictures, which normally involve 25 or more images each second. The compression method used in this activity is related to the method used on most common fax machines. A host of other methods are also available. The ones that give the best compression are "lossy"—they change the image very slightly so that it can be stored much more efficiently.

Is there a down-side?—can "compressing" an image ever *expand* it? Well, yes. A checkerboard image in which pixels alternate black and white would be much more efficient to store using one bit for each pixel than using one number for each pixel, because it will be necessary to represent the number as several bits to allow for the possibility of longer runs. Although this particular case is unlikely, some real images—such as halftone images, like illustrations in newspapers, that are made up of lots of tiny dots—do not compress well and may even expand. It is sometimes necessary to tailor the representation method to the kind of image being represented.

Further reading

Representing images on computers is discussed in depth by Netravali and Haskell in *Digital pictures: representation and compression*. The standard method for coding on fax machines is described by Hunter and Robinson in a paper published in 1978 entitled "International digital facsimile coding standards." A more recent standard for coding images is described in the book *JPEG: Still Image Data Compression Standard* by Pennebaker and Mitchell.



Instructions: *Use the numbers to color in the squares. There is a row of numbers for each line in the picture. For example, the line 4, 9, 2, 1 means to leave 4 squares empty, color in 9, leave 2 empty, and color in the next one.*

Activity 3

You can say that again!—*Text compression*

Age group	Early elementary and up.
Abilities assumed	Copying written text.
Time	10 minutes or more.
Size of group	From individuals to the whole class.

Focus

Writing.

Copying.

Repetition in written text.

Summary

Despite the massive capacity of modern computer storage devices, there is still a need for systems to store data as efficiently as possible. By coding data before it is stored, and decoding it when it is retrieved, the storage capacity of a computer can be increased at the expense of slightly slower access time. This activity shows how text can be coded to reduce the space needed to store it.

Technical terms

Text compression; Ziv-Lempel coding.

Materials

Each child will need:

a copy of the blackline master on page 32.

You will also need:

a copy of other poems or text for the children to encode.

What to do

1. Give each child a copy of the blackline master on page 32, and have them "decode" the message by filling in empty boxes with the contents of the box that their arrow points to. For example, the first empty box should contain the letter *e*. The first empty box on the second line refers to the phrase *Pease porridge* on the first line. The completed worksheet is shown in Figure 3.1.

2. Now have the children write their own representation of some text using boxes and arrows. The goal is to have as few of the original characters left as possible. The arrows should always point to an earlier part of the text to ensure that it can be decoded if the boxes are filled in from top to bottom, left to right. The children can either choose their own text, make some up, or encode some that has been provided. Many nursery rhymes and poems for children can be coded particularly effectively because they tend to have a lot of repetition, at least in the rhyming parts[1]. Books such as Dr. Seuss's "Green eggs and ham" also provide a good source of compressible text.

Variations and extensions

The arrow-and-box code requires that arrows always point to an earlier piece of text, although it is only the first letter pointed to that needs to be earlier. For example, Figure 3.2 shows a coded version of the word "Banana." Even though the missing text points to part of itself, it can be decoded correctly provided the letters are copied from left to right—each letter becomes available for copying before it is needed. This kind of self-referential representation is useful if there is a long run of a particular character or pattern.

On computers the boxes and arrows are represented by numbers. For example, the code in Figure 3.2 might be represented as "Ban(2,3)", where the 2 means count back two characters

[1] A good rhyme is "A dillar, a dollar / a ten o'clock scholar / what makes you come so soon / you used to come at ten o'clock / and now you come at noon."

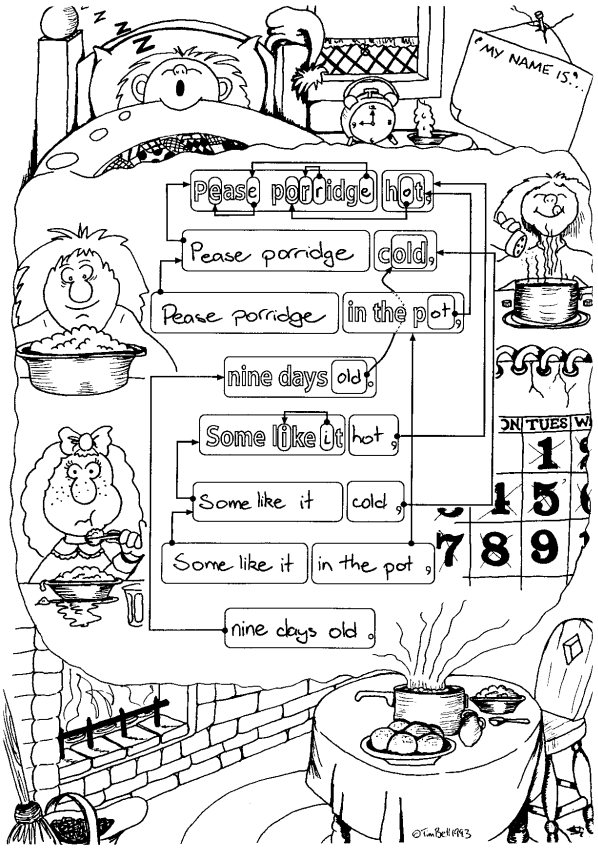


Figure 3.1: The completed worksheet

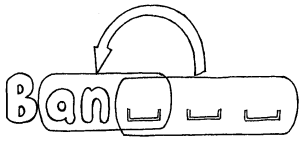


Figure 3.2: A self-referential code

to find the starting point for copying, and the 3 means copy three consecutive characters. On a computer the pair of numbers typically take up a similar amount of space to two letters, so matches of only one character are not normally encoded. Children could try encoding and decoding texts using the number representation.

To get a feel for how this technique would work on real text, children could be given a photocopied page containing a quantity of text, and starting about half-way down the page, cross out any text that could be replaced by a pointer to an earlier occurrence. The earlier occurrences should only be groups of two or more letters, since there is no saving if a single letter is substituted with two numbers. The goal is to get as many letters crossed out as possible.

What's it all about?

The storage capacity of computers is growing at an unbelievable rate—in the last 25 years, the amount of storage provided on a typical computer has grown about a millionfold—but instead of satisfying demand, the growth is fueling it. The ability of computers to store whole books suggests the possibility of storing libraries of books on computer; being able to show a high quality image on a computer suggests displaying movies; and the availability of high capacity CD-ROM simplifies the distribution of data and programs that had previously been limited by the size of a floppy disk. Anyone who owns a computer will have experienced the variation of Parkinson's law that states that the data always expands to fill the storage available for it.

An alternative to buying more storage space is to *compress* the data on hand. This activity illustrates the compression process: the representation of the data is changed so that it takes up less space. The process of compressing and decompressing the data is normally done automatically by the computer, and the user need not even be aware that it is being done except that they might notice that the disk holds more, and that it takes a little more time to get files from the disk. Essentially some computing time is being traded for storage space. Sometimes the system can even be faster using compression because there is less to read from the disk, and this more than compensates for the small amount of time needed to decompress the material.

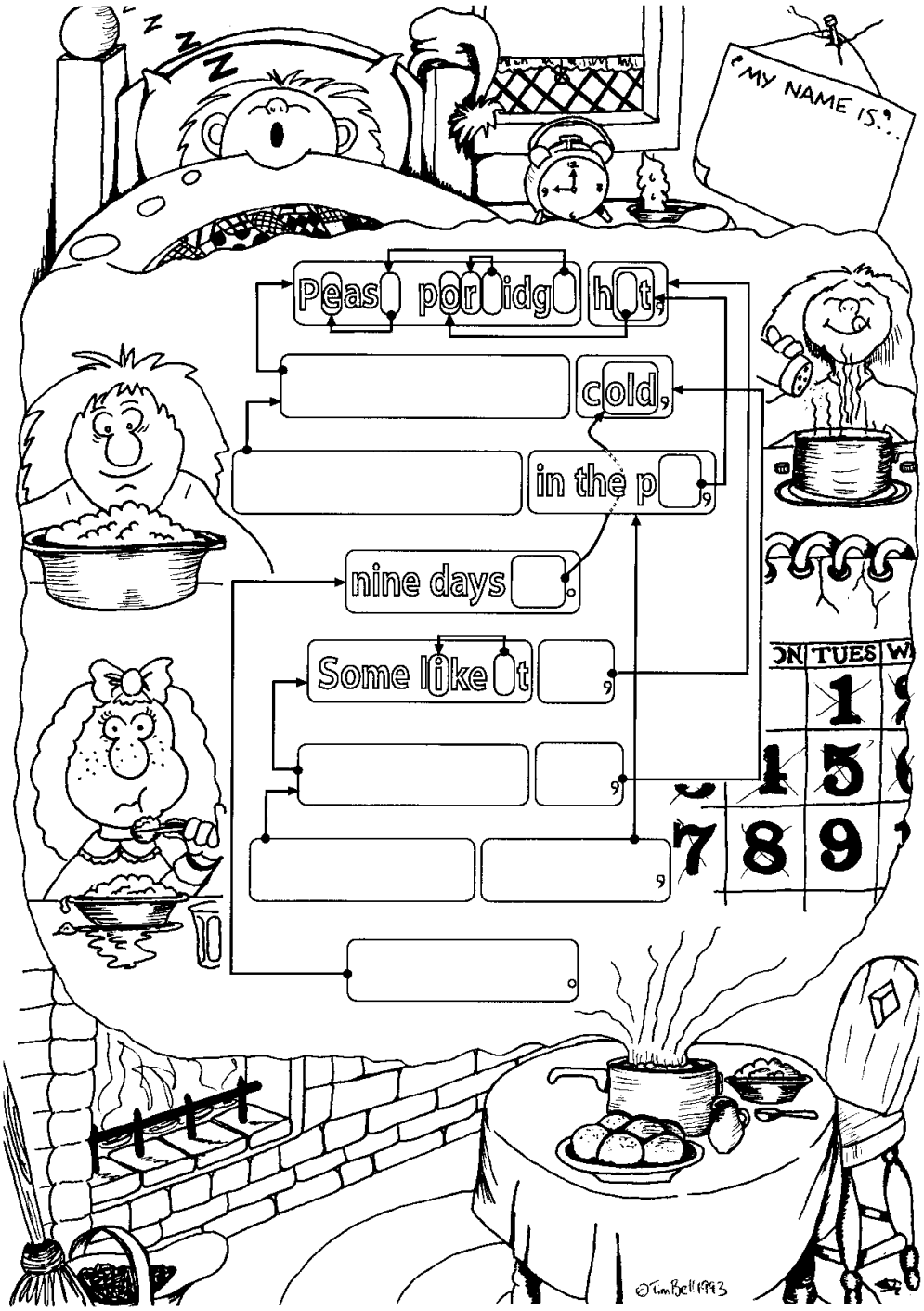
Many methods of compression have been invented. The principle of pointing to earlier occurrences of chunks of text is a popular technique, often referred to as "Ziv-Lempel coding," or LZ coding, after two Israeli professors who published several important papers in the 1970s and 1980s about this kind of compression. It is very effective because it adapts to whatever sort of data is being coded—it is just as suitable for Spanish, or even Japanese which uses a completely different alphabet, as it is for English. It even adapts to the subject of the text, since any word that is used more than once can be coded with a pointer. LZ coding will typically halve the size of the data being compressed. It is used in popular archiving programs with names like *Zip* and *ARC*, and in "disk doubling" systems. It is also used in high-speed modems, which are devices that allow you to interact with computers over ordinary telephone lines. Here it reduces the amount of data that needs to be transmitted over the phone line, noticeably increasing the apparent speed of transmission.

Another class of compression methods is based on the idea that frequently occuring letters should have shorter codes than rare ones (Morse code uses this idea.) Some of the best methods (which tend to be slower) use the idea that you can have a good guess at what the next letter will

be if you know the last few letters. We will encounter this principle in Activity 5.

Further reading

A comprehensive introduction to compression methods can be found in the books *Text compression* by Bell, Cleary and Witten, and *Managing Gigabytes: Compressing and indexing documents and images* by Witten, Moffat and Bell—although these are aimed at university-level computer science students rather than lay people. If you are interested in computer programming, you will enjoy Mark Nelson's *The data compression book*, which is a practically-oriented guide to the subject. Dewdney's *Turing Omnibus* discusses a technique called "Huffman coding" in a section about text compression.



Instructions: *Fill in each blank box by following the arrow attached to the box and copying the letters in the box that it points to.*

Activity 4

Card flip magic—*Error detection and correction*

Age group	Middle elementary and up.
Abilities assumed	Requires counting and recognition of (small) odd and even numbers. Children will get more out of it if they have learned binary number representation (see Activity 1, Count the Dots).
Time	About 30 minutes.
Size of group	From individuals to the whole class.

Focus

Odd and even numbers.

Patterns.

Magic tricks.

Summary

When data is transmitted from one computer to another, we usually assume that it gets through correctly. But sometimes things go wrong and the data is changed accidentally. This activity uses a magic trick to show how to detect when data has been corrupted, and to correct it.

Technical terms

Error detecting codes, error correcting codes, parity.

Materials

Each pair of children will need:

a pile of approximately 25 identical cards, as described below.

To demonstrate to larger groups you will need:

a set of about 40 cards with magnets on them, and a metal board for the demonstration.

What to do

This activity is presented in the form of teaching the children a "magic" trick. Their interest is easily gained by first performing the trick, and then offering to show them how to do it. As with any magic trick, a certain amount of drama is helpful in making the presentation effective. The children will need to be sitting where they can see you.

The trick requires a pile of identical, two-sided cards. For example, the cards could be red on one side and white on the other. An easy way to make them is to cut up a large sheet of cardboard that is colored on one side only. A pack of playing cards is also suitable.

For the demonstration it is easiest if you have a set of cards that have magnets on them. It is possible to buy strips of magnetic material with a peel-off sticky back, and these are ideal. Make about forty cards, half with a magnet on the front and half with one on the back. Alternatively, fridge magnets can be used; glue them back to back in pairs, and paint one side. You can then lay the cards out vertically on a metal board (e.g. a whiteboard), which is easy for the class to see. When the children come to do the trick they can lay the cards out on the floor in front of them.

1. Have one or two children lay out the cards for you on the magnetic board, in a rectangular shape. Any size rectangle is suitable, but about five by five cards is good. (The larger the layout, the more impressive the trick.) The children can decide randomly which way up to place each card. Figure 4.1 shows an example of a five by five random layout.

2. Casually add another row and column to the layout "just to make it a bit harder" (Figure 4.2). Of course, these cards are the key to the trick. The strategy is to choose the extra cards to ensure that there is an even number of colored cards in each row and column (this is explained in more detail below).

3. Select a child, and while you cover your eyes and look away, have the child swap a card—just one card—for one of the opposite color (they only need to flip it over if it is on the floor). For example, in Figure 4.3, the third card in the fourth row has been flipped. You

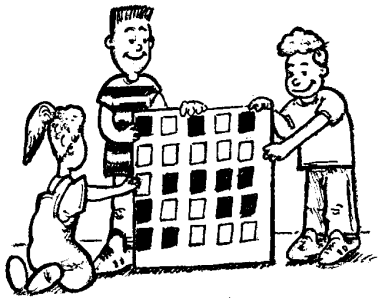


Figure 4.1: An initial random five by five layout of cards

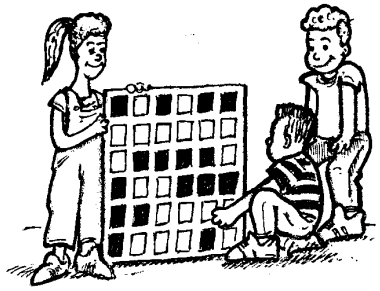


Figure 4.2: The cards with an extra row and column added

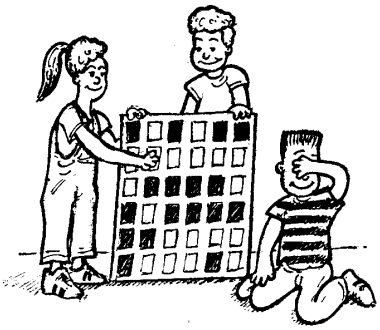


Figure 4.3: One card has been flipped

then uncover your eyes, study the cards, and identify which one was flipped. Because of the way the cards were set up, the row and column containing the changed card will now have an odd number of colored cards, which quickly identifies the offending card. Flip the card back, and have a couple more children choose a card to flip. Children will likely be quite impressed with your ability to repeatedly find the flipped card. The trick works with any number of cards, and is particularly impressive if a lot of cards are used, although it is important to rehearse the trick in this case.

4. Have the children try to guess how the trick is done. This is a good exercise in reasoning, and also helps to establish that the method is not obvious.

5. At this point you offer to teach the trick to the children. We have yet to see this offer refused!

   It is helpful to have the children work in pairs. Get each pair to lay out their own cards in a four by four square, choosing randomly whether each card is showing colored or white.

   Check that children can remember the concept of odd and even numbers, and that they appreciate that zero is an even number. Then get them to add a fifth card to each row and column, making sure that the number of colored cards is even (if it is already even then the extra card should be white, otherwise it should be colored). The technical name for the extra card is a *parity* card, and it can be helpful to teach the children the term at this point.

   Point out what happens if a card is flipped—the row and column of the flipped card will have an odd number of colored cards, and so the flipped card is the one where the offending row and column intersect. Each member of a pair can now take turns at doing the "trick."

   Once they are comfortable with the trick they may wish to get together with another pair and make up a larger rectangle of cards. You might even try pooling all the children's cards together to make one huge square.

Variations and extensions

Instead of cards, you can use just about any object at hand that has two "states." For example, you could use coins (heads or tails), sticks (pointing left-right or up-down), or cups (upside-down or right-way-up). If the activity is to be related to binary representation (Activity 1), the cards could have a zero on one side and a one on the other. This makes it easier to explain the wider significance of the exercise—that the cards represent a message in binary, to which parity bits are added to protect the message from errors.

There are lots of other things that children can discover by experimenting with the cards. Get them to think about what happens if two cards are flipped. In this case it is not possible to determine exactly which two cards were flipped, although it is always possible to tell that something has been changed. Depending on where the cards are in relation to each other, it can be possible to narrow it down to one of two pairs of cards. A similar thing happens with three card flips: you can always tell that the pattern has been changed, but it's not possible to determine exactly which cards have been flipped. With four flips it is possible that all the parity bits will be correct afterwards, and so the error could go undetected.

Another interesting exercise is to consider the lower right-hand card. If you choose it to be the correct one for the column above, then will it be correct for the row to its left? (The answer is yes, which is fortunate as it saves having to remember whether it applies to the row or the column. Children can probably "prove" this to themselves by trying to find a counter-example.)

The description above uses *even parity*—it requires an even number of colored cards. It is also possible to use *odd parity*, where each row and column has an odd number of colored cards. However, the lower right-hand card only works out the same for its row and column if the number of rows and columns of the layout are either both odd or both even. For example, a 5 by 9 layout or a 12 by 4 layout will work out fine, but a 3 by 4 layout won't. Children could be asked to experiment with odd parity and see if they can discover what is happening to the corner bit.

An everyday use of a related kind of error checking occurs in the International Standard Book Number (ISBN) given to published books. This is a ten-digit code, usually found on the back cover of a book, that uniquely identifies it. The final (tenth) digit is not part of the book identification, but is a check digit, like the parity bits in the exercise. It can be used to determine if a mistake has been made in the number. For example, if you order a book using its ISBN and get one of the digits wrong, this can be determined using only the checksum, so that you don't end up waiting for the wrong book.

The checksum is calculated using some straightforward arithmetic. You multiply the first digit by ten, the second by nine, the third by eight, and so on, down to the ninth digit multiplied by two. Each of these values is then added together to form a value which we will call s. For example, the ISBN 0-13-911991-4 gives a value

$$s = (0\times 10)+(1\times 9)+(3\times 8)+(9\times 7)+(1\times 6)+(1\times 5)+(9\times 4)+(9\times 3)+(1\times 2) = 172.$$

You then take the remainder after dividing s by 11, which is 7 for the example. If the remainder is zero then the checksum is zero, otherwise subtract the remainder from 11 to get the checksum. For the example, the checksum (at the end of the ISBN), is therefore $11-7=4$.

Figure 4.4: A bar code (UPC) from a grocery item

If the last digit of the ISBN wasn't a four then we would know that a mistake had been made somewhere.

With this formula for a checksum it is possible to come up with the value 10, which requires more than one digit. This is solved in an ISBN by using the character X for a checksum of 10. It isn't too hard to find a book with an X for the checksum—one in every 11 should have it.

Have the children try to check some real ISBN checksums. Now get them to see if they can detect whether one of the following common errors has been made in a number:

- a digit has its value changed;
- two adjacent digits are swapped with each other;
- a digit is inserted in the number; and
- a digit is removed from the number.

Have the children think about what sort of errors could occur without being detected. For example, if one digit increases and another decreases then the sum might still be the same.

A similar kind of check digit (using a different formula) is used on bar codes (universal product codes or UPCs) such as those found on grocery items (Figure 4.4). If a bar code is misread, the final digit is likely to be different from its calculated value, in which case the scanner beeps and the checkout operator must re-scan the code.

What's it all about?

Imagine that you are depositing $10 cash into your bank account. The teller types in the amount of the deposit, and it is sent to a central computer to be added to your balance. But suppose some interference occurs on the line while the amount is being sent, and the code for $10 is changed to $1,000. No problem if you are the customer, but clearly it is in the bank's interest to make sure that the message gets through correctly!

This is just one example of where it is important to detect errors in transmitted data. Just about anywhere that data is transmitted between computers, it is crucial to make sure that a receiving computer can check that the incoming data has not been corrupted by some sort of electrical interference on the line. The transmission might be the details of a financial transaction, a fax of a document, some electronic mail, or a file of information. And it's not just transmitted data that we are concerned about: another situation where it is important to ensure

that data has not been corrupted is when it is read back from a disk or tape. Data stored on this sort of medium can be changed by exposure to magnetic or electrical radiation, by heat, or by physical damage. In this situation, not only do we want to know if an error has occurred, but if possible we want to be able to reconstruct the original data—unlike the transmission situation, we can't ask for corrupted disk data to be re-sent! Another situation where retransmission is not feasible is when data is received from a deep space probe. It can take many minutes, even hours, for data to come in from a remote probe, and it would be very tedious to wait for retransmission if an error occurred. Often it is not even possible to retransmit the data, since space probes generally don't have enough memory to store images for long periods.

People have come to expect that when they store a document on their word processor, or send a message by electronic mail, they won't find the occasional character changed. Sometimes major errors happen, as when a whole disk gets wiped out, but very minor errors just don't seem to occur. The reason is that computer storage and transmission systems use techniques that ensure that data can be retrieved accurately.

Being able to recognize when the data has been corrupted is called *error detection*. Being able to reconstruct the original data is called *error correction*. A simple way to achieve error detection and correction is to transmit the same data three times. If an error occurs then one copy will be different from the other two, and so we know to discard it . . . or do we? What if, by chance, two of the copies happen to be corrupted in the same way? What's more, the system is very inefficient because we have to store or send three times as much data as before. All error control systems require some sort of additional data to be added to our original data, but we can do better than the crude system just described.

All computer data is stored and transmitted as sequences of zeros and ones, called *bits*, so the crux of the problem is to make sure that we can deliver these sequences of bits reliably. The "card flip" game uses the two sides of a card to represent a zero or one bit respectively, and adds parity bits to detect errors. The same technique is used on computers. Adding a single parity bit to a row of bits will allow us to detect whether a single bit has been changed. By putting the bits into imaginary rows and columns (as in the game), and adding parity bits to each row and column to ensure that it has an even number of ones, we can not only detect if an error has occurred, but *where* it has occurred. The offending bit is changed back, and so we have performed error correction.

In practice computers often use more complex error control systems that are able to detect and correct multiple errors. Nevertheless, they are closely related to the parity scheme. Even with the best error control systems there will always be a tiny chance that an error goes undetected, but it can be made so remote that it is less likely than the chance of a monkey hitting random keys on a typewriter producing the complete works of Shakespeare.

And to finish, a joke that is better appreciated after doing this activity:

Q: What do you call this: "Pieces of nine, pieces of nine"?

A: A parroty error.

Further reading

The parity method of error detection is relatively crude, and there are many other methods around that have better properties. Some of the more important ideas in error coding are *Hamming distances, CRC (cyclic redundancy check), BCH (Bose-Chaudhuri-Hocquenghem) codes, Reed-Solomon Codes,* and convolutional codes. Details about these sorts of codes can be found in Richard Hamming's book *Coding and Information Theory*, and Benjamin Arazi's *A commonsense approach to the theory of error correcting codes.* The ISBN of Hamming's book is 0-13-139139-9, an interesting number for a book about coding. Less comprehensive, but more accessible, information about error correction can be found in *The Turing Omnibus* by Dewdney, and in *Computer Science: An Overview* by Brookshear.

Activity 5

Twenty guesses—*Information theory*

Age group	Can be simplified to suit middle elementary and up.
Abilities assumed	Comparing numbers and working with ranges of numbers.
Time	20 to 30 minutes.
Size of group	From two people to the whole class.

Focus

Deduction.

Ranges of numbers.

Asking questions.

Summary

Computer science is very much concerned with information. Computer systems store information, retrieve it, analyze it, summarize it, and exchange it with other computers. Since information is so fundamental to the subject, it is important to know how to quantify it. Information theory provides a way of measuring information, and includes laws that define limits on how efficiently it can be stored or transmitted. This activity studies an important method of measuring information content.

Technical terms

Information theory; Shannon theory; coding; data compression; error detection and correction; entropy.

Materials

You will need:

a writing surface, such as blackboard and chalk, and

cards with answers to questions written on them (shown in Table 5.1), as described below.

What to do

1. Have a discussion with the children about *information*. Ask for their definition of information. Ask them how we might measure how much information there is in, say, a book. They might suggest the number of pages, or the number of words. But what if it is a particularly boring book, or particularly interesting—does one have more information than the other? What if the book contained nothing but the phrase "Blah blah blah" repeated over and over? Would 400 pages of that contain more information than a 400 page telephone directory?

   It soon becomes apparent that although we have an intuitive notion of information content, it is hard to quantify. Explain to the group that computer scientists measure information by how surprising a message (or book!) is. Telling you something that you know already—for example, when a friend who always walks to school says "I walked to school today"—doesn't give you any information, because it isn't surprising. If your friend said instead, "I got a ride to school today in a helicopter," that *would* be surprising, and would therefore convey a lot of information.

   How can the surprise value of a message be measured? One way is to see how hard it is to guess the information. If your friend says "Guess how I got to school today," and they had walked, you would probably guess right first time. It might take a few more guesses before you got to a helicopter, and even more if they had traveled by spaceship.

   The aim of this following activity is to get a feel for how much information messages contain, based on how difficult they are to guess. It is really just a sophisticated version of the game of twenty questions, although the children are allowed to ask more than twenty questions if necessary.

2. Select a child to stand in front of the group and answer questions about some information that is on a card which you give them. The group is initially told what kind of information it is—whether it is a number, a sequence of numbers, or a sentence; and if it is a number, what range it lies in. The group then asks questions of the child who has the card, but the child can only give yes/no answers. A good starter is to have the group guess a number

Type of message	Sample message	Notes
A number between 1 and 100	67	
A number between 1 and 1000	387	For older children, use numbers between 1 and 1,000,000.
Any whole number	145	For older children, use larger numbers.
The age of the child answering the questions	10	Or use your own age if you aren't easily offended!
A sequence of numbers	{2,4,6,8,10,12}	Explain that there are six numbers, and that they follow a pattern. Encourage the children to guess them one at a time, from first to last.
	{1,3,2,4,3,5,4,6,5,7}	These numbers go up two, down one, up two, down one, etc.
A sentence	The switch is on.	The sentence should be guessed one letter at a time, from left to right.

Table 5.1: Suggested messages for *Twenty Questions* (see text for notes about each)

between 1 and 100. They will typically ask questions like "Is it more than 50?" or "Is it between 20 and 60?" In the latter case, you should clarify whether "between" is inclusive or exclusive *before* an answer is given. Check that the child is answering the questions correctly, because one erroneous answer can be very misleading.

Record the questions and answers on the chalkboard (at least in shorthand) so that the group can refer back to them. It can be convenient to have the child with the card choose who can ask the next question.

Once the number has been guessed, count up the number of questions that were asked, and point out that this is a measure of the amount of "information" that the number was worth.

3. Now try the same game with some of the "messages" suggested in Table 5.1, taking note of how many questions are needed for each one.

   It may become boring for the children if you do *all* the message types in Table 5.1, depending on the age group, but they will probably enjoy this game in small doses.

   After they have been guessing numbers within a range, discuss the strategies that they used. The best strategy (which they will probably latch on to) is to halve the range of

possibilities with each question. You are always able to identify a number between 1 and 100 in just seven guesses this way, although children will be tempted to make wild guesses that do not usually pay off.

When the range is increased to 1,000, the natural reaction is to think that it will take ten times the effort, but in fact only three more questions are needed. Numbers between 1 and 1,000,000 can be found in just twenty questions. Every time the range doubles, just one extra question is needed—the number of questions increases in proportion to the logarithm of the size of the range.

For larger numbers, the children may devise other strategies—such as guessing a digit at a time. This is quite reasonable, for it is often a good idea to break a large "message" down into simple parts—although the halving strategy is always the most efficient in terms of the number of guesses taken.

When the "message" can be *any* whole number, the children will need to find an upper bound first. A typical sequence of questions begins "Is it less than 100?", "less than 1000?" etc. Discuss the strategy with them. To determine where they were headed, find out what questions they would have asked had the answers to these initial questions all been "no." Any increasing sequence of numbers will eventually find a bound. For example, one could simply say "Is it greater than or equal to 1?", then "2?", "3?" and so on, but this would take a long time for large numbers. A better strategy is to multiply the bound by 10 (or some other number) on each question, giving the questions "Is it less than 10?", "100?", "1,000?", and so on.

When the message is someone's age, it will probably only take a couple of guesses to find the answer. Discuss with the class why this is the case, even though an age can be any number between 1 and 100 or more. (The reason is that some ages are more likely than others. This relates to the "information content" discussed earlier—it would be very surprising if a grade 2 child was a hundred years old, or if the teacher was twelve years old.)

A similar effect occurs with the sequence of numbers. Once the beginning of the sequence is known (e.g. $\{2,4,6\}$), the next one is very predictable. Once they figure out the relationship, there is very little information in the latter part of the sequence.

Guesses for the letters of the sentence should be of the form "Is the next letter an *A*?", "Is the next letter a vowel?", or "Is the next letter one of *B*, *D*, or *P*?". Note that spaces between words should also count as "letters." Discuss with the children which letters were the hardest to guess. (Usually the first letter of a word is the hardest. Many of the letters are very predictable, and will be guessed first time.) The letters that are guessed quickly contain very little information; in fact, many of them could be left out and the message would still be intelligible—consider the sentence: "Ths sntnc hs th vwls mssng."

Variations and extensions

If the strategy for asking questions is pre-determined, it is possible to transmit a message without having to ask the questions.

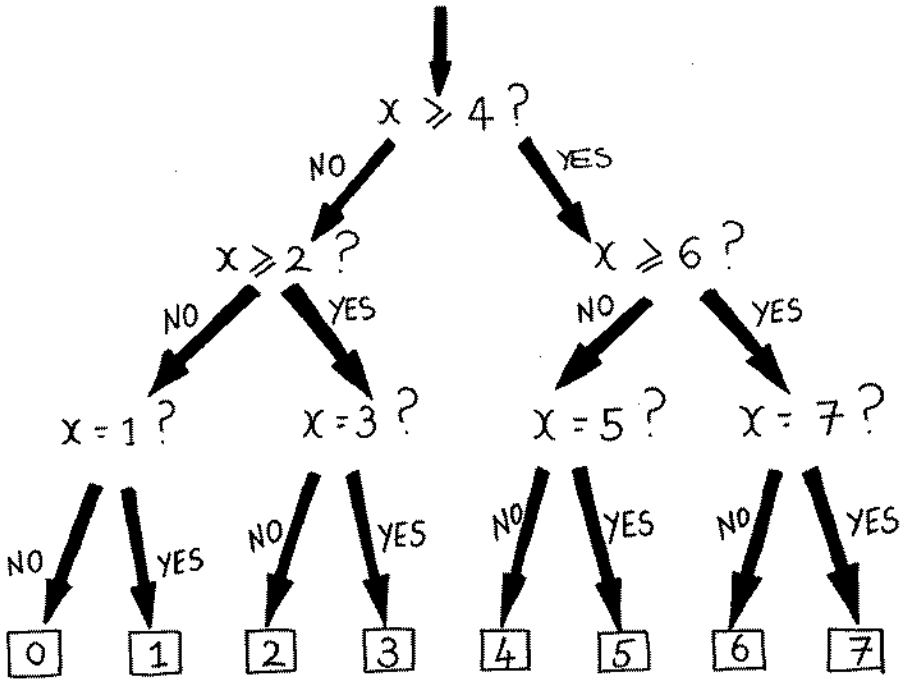


Figure 5.1: A strategy for guessing a number between 0 and 7

For example, Figure 5.1 shows a strategy for guessing a number between 0 and 7. The first question is "Is it greater than or equal to four?" If the answer is yes, we follow the right-hand path, and ask if it is greater than or equal to six. If that answer is no, we follow the left-hand path from there, and ask if the number is five. If that answer is no, the number must be four; otherwise it is five.

If the person answering the questions knew that this strategy was being followed, they need only give the sequence of answers and we would be able to figure out the message. For example, the answers "No, yes, yes" would represent the number three.

The structure shown in Figure 5.1 is called a *decision tree.* Children could design and use a decision tree for guessing numbers between 0 and 15. Have them think about the kind of tree they would use for someone's age (it might be biased towards smaller numbers). What about letters in a sentence? (In this case the decision might depend upon what the previous letter was.)

What's it all about?

We encountered the idea of a binary digit or "bit" of information in Activity 1, and the answer to a single yes/no question corresponds to exactly one bit of information—whether it is a simple question like "Is it more than 50?" or a more complex one like "Is it between 20 and 60?" Indeed, in the number-guessing game, if the questions are chosen in a certain way, the sequence of answers is just the binary representation of the number. Three is 011 in binary and is represented

by the answers "No, yes, yes" in the tree of Figure 5.1, which is the same if we write no for 0 and yes for 1. The binary notation is discussed in detail in Activity 1.

The celebrated American mathematician (and juggler, and unicyclist) Claude Shannon defined the concept of "entropy" in 1948 as, roughly, the information content of a message. The amount of information in a message is a slightly slippery concept because it depends on your knowledge of what the message might be. For example, the information in a single toss of a coin is normally one bit: heads or tails. But if the coin happens to be a biased one that turns up heads seven times out of eight, then the information is no longer one bit—believe it or not, it's less. How can you find out what a coin toss was with less that one yes/no question? Simple—just use questions like "are the next *two* coin tosses both heads?" For a sequence of tosses with the biased coin just described, the answer to this will be "yes" 49/64, or about 77%, of the time, and on average you will be asking less than one question per coin toss! So the entropy depends not just on the *number* of possible outcomes—in the case of a coin toss, two—but also on their *probabilities*. Improbable events convey more information, just like helicoptering versus walking to school, but probable ones happen more frequently (by definition!), keeping the average information low.

Shannon was fascinated by the question of how much information is contained in messages written in a natural language like English. He performed an experiment in which people had to guess sentences letter by letter. He showed subjects text up to a certain point, and asked what they thought the next letter was. If they were wrong, they were told so and asked to guess again, and again, and again, until eventually they guessed correctly. For example, one of Shannon's texts began "there is no reverse on a motorcycle a friend of mine found this out rather dramatically the other day . . . " (he omitted punctuation and capitalization, to make life easier). A typical result of the experiment is:

t	h	e	r	e	•	i	s	•	n	o	•	r	e	v	e	r	s	e	•	o	n
1	1	1	5	1	1	2	1	1	2	1	1	15	1	17	1	1	1	2	1	3	2

•	a	•	m	o	t	o	r	c	y	c	l	e	•	a	•	f	r	i	e	n	d
1	2	2	7	1	1	1	1	4	1	1	1	1	1	3	1	8	6	1	3	1	1

where bullets represent spaces and the numbers below the letters record how many guesses it took the subject to get that letter correct. On the basis of no information about the sentence, this subject guessed that its first letter would be *t*—and in fact was correct. Knowing this, the next letter was guessed correctly as *h* and the following one as *e*. The fourth letter was not guessed first time. Seeing *the*, the subject probably guessed space; then, when told that was wrong, tried letters such as *n*, *i* and *s* before getting the *r*, which is correct, on the fifth attempt. Out of 44 symbols the first guess was correct 28 times, the second six times, the third three times, the fourth, fifth, sixth, seventh and eighth once each, while on only two occasions were more than eight guesses necessary.

Results like this are typical of prediction by a good subject with ordinary literary prose. In fact, we ourselves have tried this very sentence on many adult audiences and found that, in an astonishingly high proportion of cases, the number of guesses taken are *exactly* the same as those given above. From experiments such as these, Shannon showed that the entropy of

ordinary English text is between 0.6 and 1.3 bits per letter; more recent experiments indicate that the upper bound is more typical of people's performance. One bit per letter would mean that on average there are two possibilities for each letter, while two bits means four possibilities. With 1.3 bits, on average there are two and a half possibilities for each letter.

In principle, the decision tree idea of Figure 5.1 is a way of removing the interaction from the experiment by getting subjects to consider all possibilities in advance. You can imagine eliciting from a subject a decision tree for each possible context. The one for no context would say "guess *t*, then, if incorrect, guess *a*, then *o*, then *s*, . . . ," because *t* is the most likely letter to begin a sentence, then *a*, and so on. The one to use when the context was *the* says something like "guess space, then, if incorrect, guess *n*, *i*, *s*, *r*, . . . ". The one to use when the context was *q* says "guess *u*, then, if incorrect, guess space (for phrases like *coq a vin*), *a* (for words like *Qantas*), . . . " In practice, it would be impossibly tedious to elicit decision trees like this, even for a few contexts, let alone a complete set of them (and what is a "complete set"?—language is infinite, for it has limitless possibilities).

The concept of entropy relates directly to both compression (Activity 2 and Activity 3) and error detection/correction (Activity 4). You cannot compress a message to occupy less space than its entropy, at least on average. And although people's models of English text seem to indicate an inherent entropy of around 1.3 bits per letter, in practice computer models, which effectively embody decision trees as described above, are far less sophisticated: the best ones compress English text to around 2 bits per letter. Since letters are usually stored in 8-bit bytes, compression can reduce text files to around a quarter of their original size—which is certainly a worthwhile saving.

Compression *removes* redundancy from a message, whereas error detection and correction *insert* it. For example, adding the extra row and column of cards in Figure 4.2 increases the number of cards from 25 to 36. It is this extra information that allows changes in the cards—more generally, "errors"—to be detected. A good way of compressing the resulting array of cards would be to remove the extra row and column entirely, for they can always be regenerated! But this is the point: compression and error detection/correction work in opposite directions. In practice, communication engineers find it best to compress a message first to remove all redundancy from it, and then add just the right kind of redundancy that is needed to combat the kind of error, or "noise," that occurs in the communication channel.

There are other applications of prediction, aside from compression. For example, using the kind of decision tree that might be used to compress text, you can build a computer interface that predicts what the user is going to type next! Obviously predictions cannot always be correct—otherwise users could go away and leave the computer to do their work for them—but they might be correct often enough to support useful communication. The *Reactive Keyboard* is a device that displays several different predictions on a computer "menu," and lets the user select the correct one by pointing at it. While it is unlikely to help a skilled typist, except perhaps when entering documents such as legal contracts that contain a lot of standard "boilerplate" paragraphs, some physically disabled people, for whom regular keyboards are difficult to use, find it a boon.

As we have seen, information theory has a variety of applications in computer science, from finding limits on how much a file can be compressed, to predicting what people are going to type. It is fundamental for a machine whose modern day application is primarily to store, retrieve, and

transmit information.

Further reading

Bell, Cleary and Witten's book on *Text compression* gives a good introduction to the subject of information theory and entropy, although it is aimed at university-level computer science students rather than lay people. Shannon's original work was published in 1949 in a little book entitled *The mathematical theory of communication*, which also includes a non-mathematical, and very readable, account of the subject by Warren Weaver. An article by Donald Knuth called "Supernatural numbers" discusses strategies for guessing arbitrarily large numbers; it appears in the book *The Mathematical Gardner*, edited by D.A. Klarner. *The Reactive Keyboard* and other "keyboard acceleration" devices for the disabled are described in a book by Darragh and Witten.

Part II

Putting computers to work—*Algorithms*

Now that we can represent information inside computers, let's put them to work for us. Computers operate by following a procedure—a list of instructions—that you set out for them. The tough part is coming up with a method for doing what you want, and writing it as a procedure for the computer. The activities in this section will help you think about methods for doing things, and in the following one we'll take a look at how you tell computers what to do.

Here's an example of a method you might come up with. Suppose you're sorting a deck of cards into order. How do you do it? One way is to go through the deck and pull out the smallest card, put it on the table, then go through and pull out the smallest card left in the deck, put it on top of the first, and repeat the process until all the cards have been extracted. Is there a faster way? Yes, there is—in fact there are *much* faster ways of getting the deck into order, and soon you will find out how. "But," you're thinking, "computers don't play cards with real decks!" No, but they do sort things into order. Pretty well every time you get information out of a computer, it has been sorted in some way. Computers do a lot of sorting. "But," you're thinking, "computers are really fast, so what's the big deal about finding a faster way?" Well, they're not *that* fast. Sorting the telephone list for your city could take a computer days using a slow method—or even months, for a large city. A lot of the numbers would be out of date by the time the book was printed!

Sorting is one example. Another is finding things in a large collection of data—that can take a long time too, unless you have a good method. Or deciding how to connect up networks: where are the best places to build roads to link up towns? Or deciding how to send things through networks: which way to go? To understand how to put computers to work, you need to know about finding good methods for jobs like these. The activities in this section will give you an opportunity to try out different methods.

For teachers

An algorithm is a set of instructions for completing a task. The word derives from the name of Mohammed ibn Musa Al-Khowarizmi—Mohammed, son of Moses, from Khowarizm—who joined an academic center known as the House of Wisdom in Baghdad around 800 AD. His works transmitted the Hindu art of reckoning to the Arabs and thence to Europe. When they were translated into Latin in 1120 AD the first words were "Dixit Algorismi . . . "—thus said Algorismi. The 1963 Oxford dictionary on my desk still has the word as "algorism" rather than algorithm. School children will already be familiar with algorithms for addition, subtraction, multiplication, and division. In this section they will encounter algorithms that work with problems that don't necessarily involve numbers.

The idea of an algorithm is absolutely central to computer science: algorithms are how we get computers to solve problems. The five activities in this section expose children to problems for which efficient algorithms have been found. They provide an opportunity for children to explore these algorithms, and (in some cases) discover efficient algorithms for themselves. The first three are about searching and sorting, tasks that are traditionally used to begin introductory courses on algorithms. Both are important tasks that computers spend a lot of their time on. For both, a simplistic algorithm is obvious—but usually impractical in realistic situations because of its slowness. Much faster algorithms, which

are still easy to understand, have been developed that improve speed by a millionfold on large-scale problems. Searching and sorting are related: sorting a list into order makes it much easier to search, and this provides an opportunity for children to think about the importance of ordered lists in everyday life. The remaining two activities are practical exercises that are concerned with networks: how to construct networks efficiently and economically, and how the activities of several people have to be coordinated in order to use a network effectively.

These activities will help children understand that some algorithms are *far* faster than others, that good algorithms may not be obvious and sometimes require real insight, that some apparently sequential problems can be broken up into parts that can be done in parallel, and that the communication and coordination involved in sharing resources requires considerable deliberation. They can be completed successfully by children at the early to middle elementary level, except that the first activity is slightly more advanced—though it too can be accomplished by younger children under close supervision. The activities can be done in any order, except that it is helpful, though not essential, for Activity 7 to precede Activity 8. Activity 8 is suitable for the playground or gym.

For the technically-minded

For any task there will be many different algorithms that accomplish it successfully. Computer scientists seek fast algorithms for the kind of tasks that computers perform. A faster algorithm may save you having to buy a faster computer. And a slow algorithm might be useless—if it took a month to sort a telephone directory into order, the book would be out of date before it was printed; likewise it would not be much good being able to predict tomorrow's weather if it took two days of computer time to do so!

There have been some remarkable breakthroughs where people have found a fast algorithm for a task and saved hours, days, or more of computer time. It is not unusual for a good algorithm to speed up the solution to a large problem by a factor of a million or a billion. This is significant if the fast version of the algorithm takes one second—a million seconds is nearly two weeks and a billion is over 30 years. Many of the algorithms that have been discovered have made it possible to solve problems that previously took an infeasible length of time to solve. People are still discovering improved algorithms, and there are many important problems for which efficient solutions still elude researchers.

A mid-1980s article by Jon Bentley, a prominent computer scientist, compared the amount of speed-up obtained in the solution to a small problem (solving a three-dimensional partial differential equation, Poisson's equation) by improvements in hardware and in software over a long period of time. The conclusion was that in the four decades since 1947, improvements in computer hardware from the Manchester Mark I computer to the Cray-2 supercomputer had accelerated the solution of the problem by a factor of half a million. Algorithmic improvements over a three-decade period (1947–78) contributed a speed-up of a quarter of a million. Each of these improvements alone reduces the running time from centuries to hours. Together, they multiply to reduce the run time to under a

second. Algorithmic development ceased in 1978 because it can be shown that no faster algorithms exist for this problem. Hardware improvements, of course, continue. But note that the hardware/software comparison depends critically on the problem size. In this case the time taken was reduced from being proportional to the seventh power of the size of the problem, to being proportional to its cube. This means that for a problem ten times as large, the hardware speed-up would remain the same whereas the software speed-up would, in this particular case, be ten thousand times greater!

Activity 6

Battleships—*Searching algorithms*

Age group	Older elementary, although with sufficient supervision the activity can be undertaken by younger children.
Abilities assumed	Comparing four-digit numbers (can be adapted for smaller numbers), and calculating sums with two-digit results.
Time	About 30 minutes.
Size of group	At least a pair of people, suitable for the whole class.

Focus

Logical reasoning.

Comparison.

Inequalities.

Summary

Computers are often required to “look things up” in large collections of data. It is tempting to think that the machine might as well simply search through all the data until the desired item is found. However, even on very fast computers, this is prohibitively slow in practice, because the quantities of data involved are often very large. This activity introduces the ideas underlying two widely-used techniques that enable computers to search for data very quickly.

Technical terms

Search algorithms; linear search; binary search; hash tables.

Materials

Each pair of children will need:

one copy of the game sheets on pages 61 to 66: 1A, 1B for the first game, 2A, 2B for the second, and 3A, 3B for the third (they are easier to use if enlarged on to double size paper),

means to conceal the game sheets from each other, and

a pen or pencil for each child.

You will also need:

a few copies of the remaining, supplementary, game sheets on pages 67 to 72: 1A′, 1B′ for the first game, 2A′, 2B′ for the second, and 3A′, 3B′ for the third.

What to do

1. Group the children into pairs. Each child is given a game sheet for the first game. One member of each pair receives sheet 1A (page 61); the other receives 1B (page 62). They should not show their sheets to each other (the sheets can be hidden by putting another piece of paper or book on top of them, or a small screen could be placed between the children).

   Each child circles one of the 26 battleships on the top line of their game sheet, and they tell their partner its four-digit number. This is "their" ship that the other child must try to "sink." The children then take turns guessing where their partner's circled ship is. They call out the letter corresponding to a location (A to Z), and their partner tells them the number of the ship at that letter. If the person who fires a shot "misses," they cross out the corresponding unnumbered ship on their sheet. The game finishes when both children have located their partner's ship. Their score for the game is the number of "shots" they fired (i.e. guesses they made), which is recorded on the game sheet.

   The spare sheets 1A′, 1B′ (pages 67 and 68) can also be used for this activity. They are provided for children who would like to play more games, or who get to see what is on their partner's sheet (whether accidentally or intentionally!) Sheets 2A′, 2B′ and 3A′, 3B′ (pages 69 to 72) provide alternatives for the later games, for the same reasons.

2. Once the children have finished this game, it is helpful to have a class discussion. Collect the scores. Point out the minimum and maximum, and ask what are the minimum and maximum possible (they are 1 and 26 respectively, assuming that the children don't shoot at the same ship twice).

3. Give out sheets 2A and 2B (again, each child in a pair receives a different sheet). The rules for this version of the game are the same, but the numbers on the ships are now in ascending order. Point this out and ask if the children think this game will be easier than the first one. They should discover that, in this new version, they are able to cross off several ships with one "shot." For example, if we are seeking ship 3423 and discover that position J contains 8934, all positions to the right of J can be eliminated since their ships must have numbers greater than 8934. Have the children play the game, seeing if they can take advantage of the ordering.

4. After they have finished sheets 2A and 2B (and, if necessary, 2A′ and 2B′), collect the scores and discuss them again. Ask a child with a low (good) score what strategy they used (inevitably some of the strategies may include looking at their partner's sheet, or just plain good luck). Discuss which ship it is best to choose first (the one in the middle is best). Point out (or have the students work out) that once the number of the middle ship is known, we know which half the chosen ship must be in. Ask which location should be chosen next (the best one is in the middle of the half that the ship is known to be in). In fact, the best strategy is always to choose the middle of the region that must contain the ship. Each shot halves the number of locations that the ship might occupy, and so we quickly narrow down its location. If the strategy is applied correctly, any ship can be found in five shots.

5. The third game, on sheets 3A and 3B, uses another strategy for locating ships quickly. Knowing the number of a ship, you can calculate which column (0 to 9) it is in. This is done by summing the digits of the ship number and taking the last digit of the result. For example, to locate a ship numbered 2345, add the digits 2+3+4+5, giving 14. The last digit of the sum is 4, so that ship must be in column 4. The sheets are arranged so that the rule works for all ships.

   With this arrangement, when a child is told the number of their partner's ship they can immediately calculate which column it is in. Teach the children how to do this. Once the column is known, guesses are needed to ascertain which of the ships in the column is the desired one. Have them play the game with this new strategy. It is possible to play more than one game on each sheet, because only one column of ships is revealed per game. (Note that, unlike the other games, the spare sheets 3A′ and 3B′ must be used as a pair, because the pattern of ships in columns must correspond.)

6. Collect and discuss the scores as before. Check that the children have realized that the new strategy is often faster than the previous one, but can occasionally be very slow. Ask which ships would be very quick to find (the ones that are alone in their columns) and which would be hard to find (the ones whose columns contain lots of other ships).

7. Discuss the advantages of each of the three ways of playing the game. (The second strategy is faster than the first, but the first one doesn't require the ships to be sorted into order. The third strategy is usually faster than the other two, but it is possible, by chance, for it to be very slow. In the worst case, if all the ships end up in the same column, it is just as slow as the first strategy.)

Variations and extensions

Have the children construct their own games using the three formats. For the second game (A2 and B2) it is crucial that the numbers are in ascending order. For the third game (A3 and B3) they will need to work out which column each ship occupies. Discuss how you could choose numbers to make the game very hard (the hardest game is when all of the ships are in the same column). How likely is this if the numbers are chosen randomly? (Not very likely—the ships are being placed in randomly chosen boxes, so the chances of all the ships ending up in the same box are very low, the odds are one in 10^{25} that all ships will randomly end up in the same box.) What would you do to make the game as easy as possible? (You should try to get the same number of ships into each column.)

Discuss what would happen if the ship being sought isn't there. For the first game (1A and 1B), 26 shots would be necessary to discover this. For the second, the "divide-and-conquer" strategy still applies: it takes no more than six shots to discover that the ship is missing. The success of the third strategy depends on how many ships appear in the relevant column. If there are none, we know immediately to give up; otherwise all ships in the column must be checked.

For the second strategy—the one where the list is sorted—older children might like to consider how many shots would be required if there were a hundred locations (about six shots), a thousand locations (about nine), or a million (about nineteen). Notice that the number of shots increases very slowly compared to the number of ships. (One extra shot is required each time the size doubles, so it is proportional to the logarithm of the number of ships.)

Closely related to the second strategy is the guessing game of Activity 5, where one person thinks of a number between one and (say) a hundred, and a second person guesses it using only questions that have yes/no answers. Even with a very large range of possibilities, just a few guesses are needed if the strategy is used correctly. (If you double the range, only one more guess is required since each guess halves its size.)

For the third strategy, the number of guesses you might expect to have to make is determined by the distribution of ship numbers over columns. It is easiest to locate ships if they are spread out as much as possible. Note that rearranging the digits of a number does not change its column (e.g. 1234 is stored in the same column as 4321). To avoid over-popular columns, we really want a way of assigning each ship to a "random" column based on its number—though the assignment must be done using a repeatable formula. This kind of mapping is called *pseudo-random*.

Instead of the children hiding their sheets from each other, removable stickers can be placed over the numbers and removed with each "shot." Since children can now look at each others' sheets while playing, they can check that their partner is doing their job correctly. However, a lot of stickers are required, only opaque (preferably black) stickers can be used, and it takes a long time to put them all on the sheets.

It is also possible to use cards for this activity, particularly for a small group. Children place the cards upside down in front of them, and their partner chooses which one to turn over. This version, like the one with stickers, is more interesting to play and children aren't disadvantaged by an incompetent partner. However, the children must be able to lay the cards out correctly, perhaps with help from an adult. For an even more dramatic simulation, the numbers could be placed under heavy stones or bricks, so that considerable effort is required to test each "shot."

What's it all about?

People often need to locate information stored somewhere inside a computer. Checkout computers need to locate product descriptions and prices from bar-code numbers, bank tellers need to look up the balance for a customer's account number, and on-line library catalogues need to be able to find the titles of books written by a given author. Computers can store a lot of information, and they need to sift through it to locate what has been requested.

The data that the computer is given to look up, such as a bar code number, customer's account number, or author's name, is called a *search key*. The general form of the problem is that the computer must locate some extra information related to the search key, such as the product price, account balance, or book titles, although sometimes we are just interested in its existence, such as when searching a dictionary to check that a word is spelled correctly.

Computers can process information very quickly, and you might think that to find something they should just start at the beginning of their storage and keep looking until the desired information is found. This is analogous to the first strategy used in the battleships game. But it is very slow—even for computers. For example, suppose a supermarket has 10,000 different products on its shelves. When a bar code is scanned at a checkout, the computer must look through up to 10,000 numbers to find the product name and price. Even if it takes only one thousandth of a second to check each code, ten seconds would be needed to go through the whole list. Imagine how long it would take to check out the groceries for a family of four!

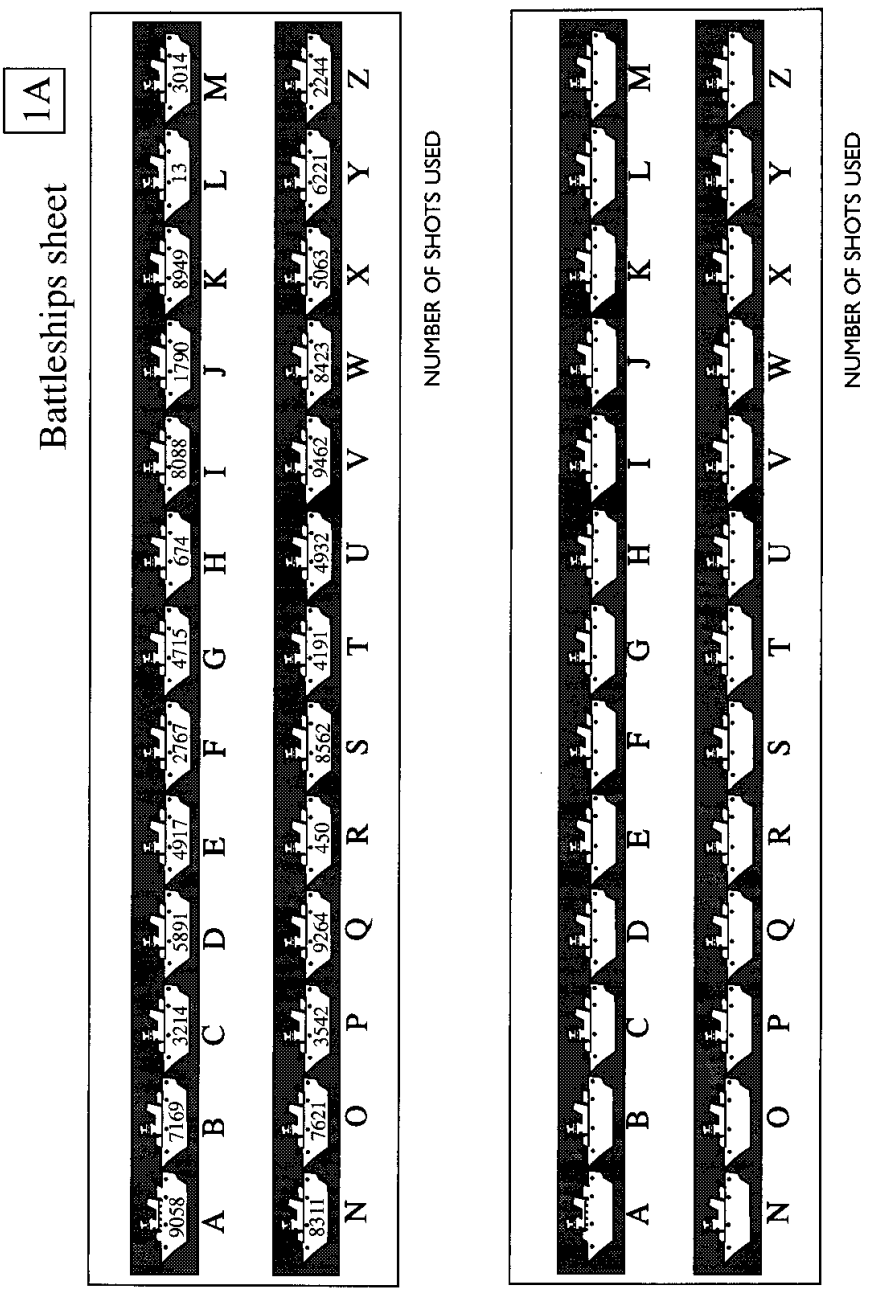
A better strategy, corresponding to the second version of battleships, is *binary search*. In this method, the numbers have previously been sorted into order. Checking the middle item of the list will identify which half the search key is in. The process is repeated on each successive half until the list has been narrowed down to just one item. People use a similar strategy when looking up a dictionary or phone book, and it's much quicker than starting at the beginning and working through until you find what you are looking for. Imagine the problems of searching a telephone book in an ideographic language like Chinese, where there is no alphabetic ordering! Returning to the supermarket example, the 10,000 items can now be searched with fourteen probes, which might take two hundredths of a second—hardly noticeable.

A third strategy for finding data, corresponding to the final battleships game, is called *hashing*. Here the search key is manipulated to indicate exactly where to look for the information. For us, this manipulation involved summing the digits of the key and using the last digit of the sum. This produced ten possible locations (we called them "columns"). In practice, a much larger number of locations is used to ensure that relatively few keys end up at the same one. This means that most of the time the computer will find what it is looking for straight away. However, there is a chance (usually a very small one) that several keys end up in the same location, in which case the computer will need to search through them until it finds the one it is seeking. Until quite recently, the Hong Kong telephone directory worked like this: names were sorted into groups according to the number of strokes they contained, and having found the right group, you had to scan through it to find the name you wanted.

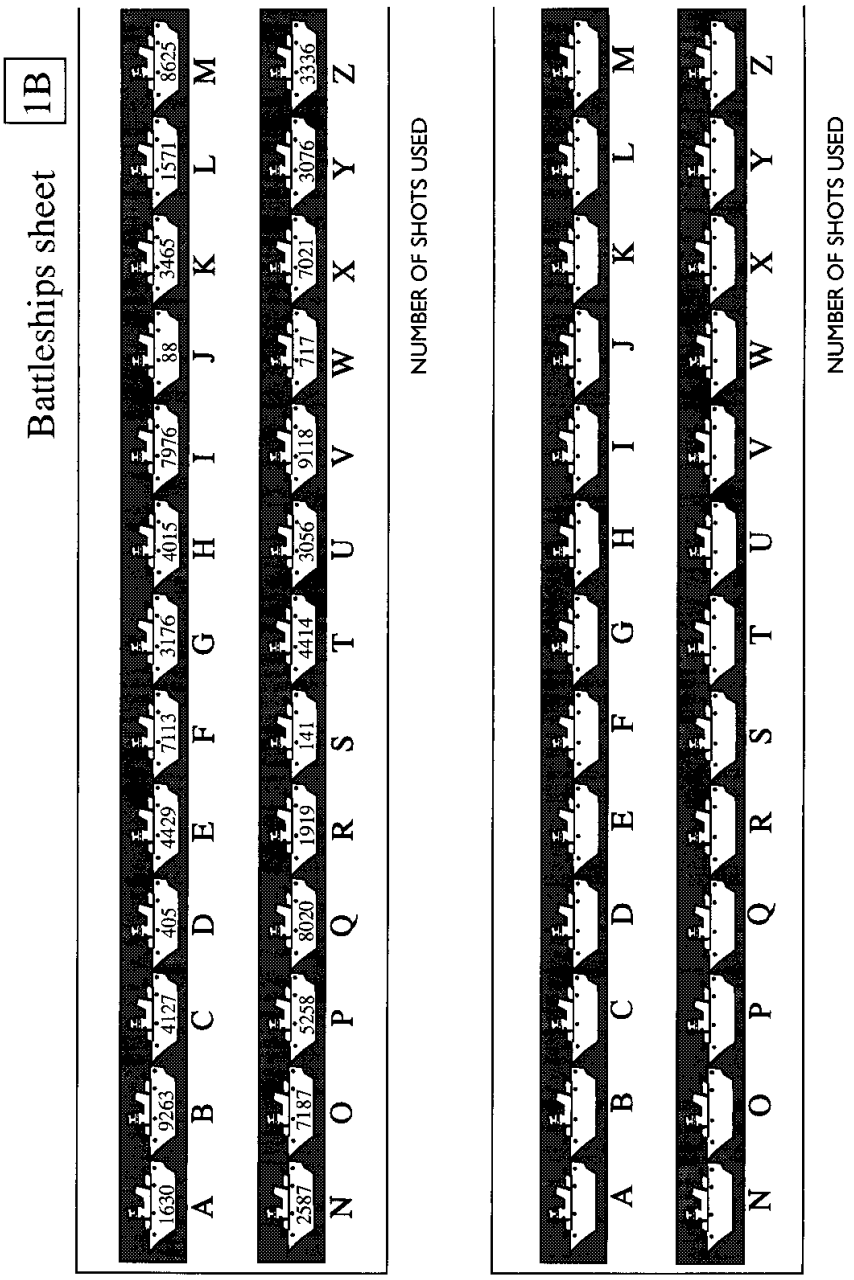
Computer programmers usually use some version of the hashing strategy for searching, unless it is important to keep the data in order, or if an occasional slow response is unacceptable—as in a life support system.

Further reading

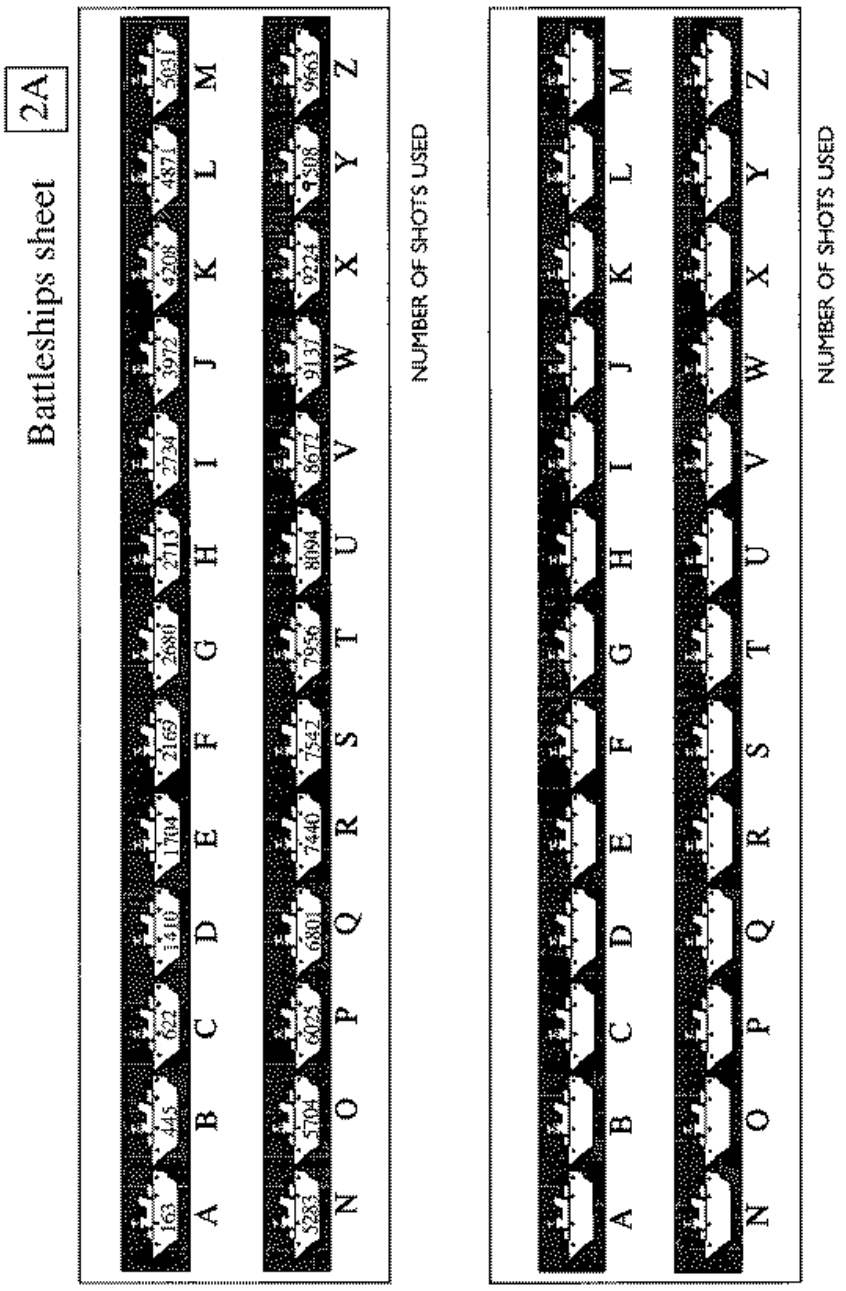
Most computer science texts discuss the three methods—linear search, binary search, and hashing—and there are innumerable variations and adaptations. Harel's *The Science of Computing*. gives a good introduction. More details can be found in Kruse's *Data Structures and Program Design*, and in Dale and Lilley's *Pascal Plus Data Structures, Algorithms, and Advanced Programming*. Search techniques for mass storage devices like disks are covered by *File Structures: An Analytic Approach* by Betty Salzberg. Dewdney's *Turing Omnibus* has a section on storage by hashing. One of the most comprehensive texts on searching (and sorting—see Activity 7) is Knuth's classic 1973 tome *The Art of Computer Programming, Volume 3: Sorting and Searching*, although this is very detailed and is really suitable only for the professional—or the fanatic!



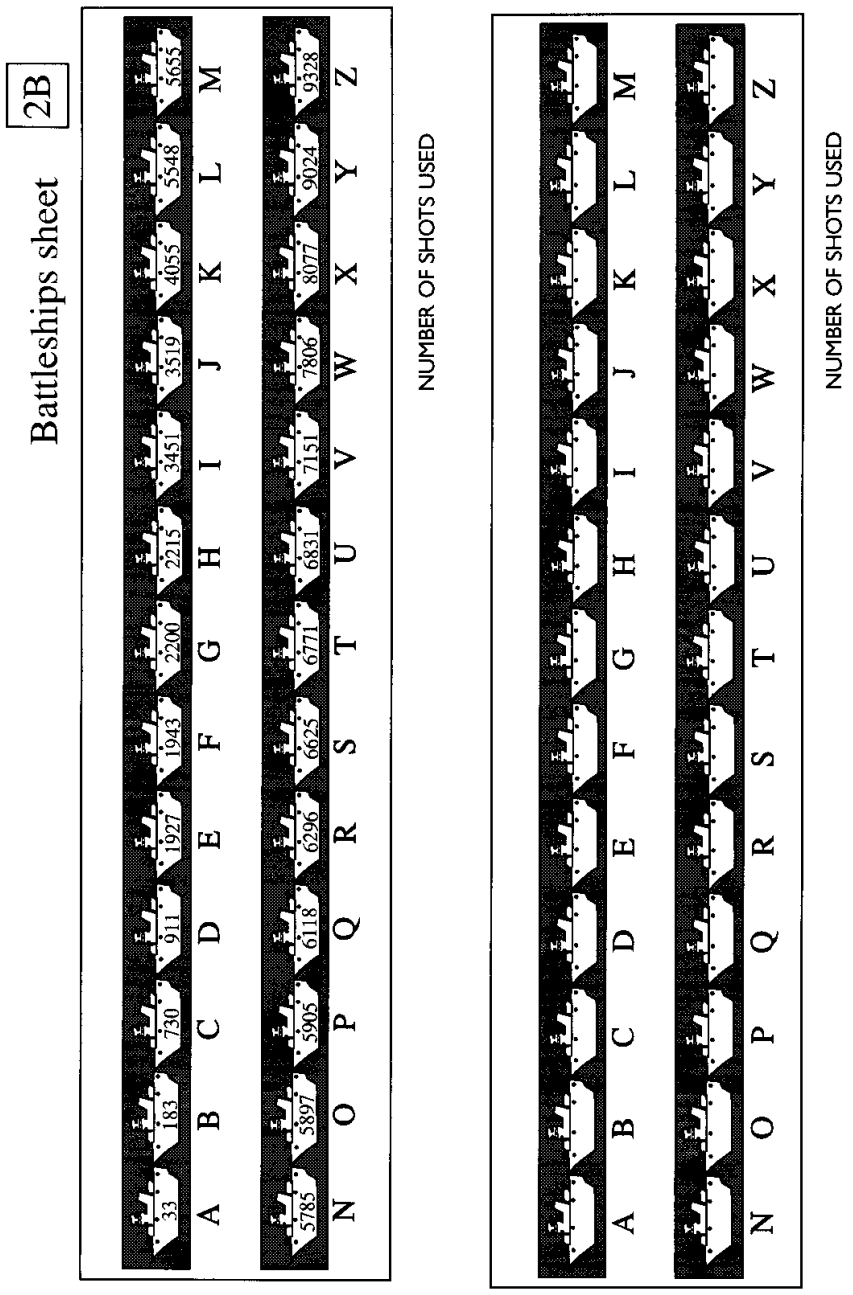
Instructions: *Your opponent should have sheet 1B. Choose one of the ships, tell your opponent its number, and then take turns trying to locate each other's ship.*



Instructions: *Your opponent should have sheet 1A. Choose one of the ships, tell your opponent its number, and then take turns trying to locate each other's ship.*

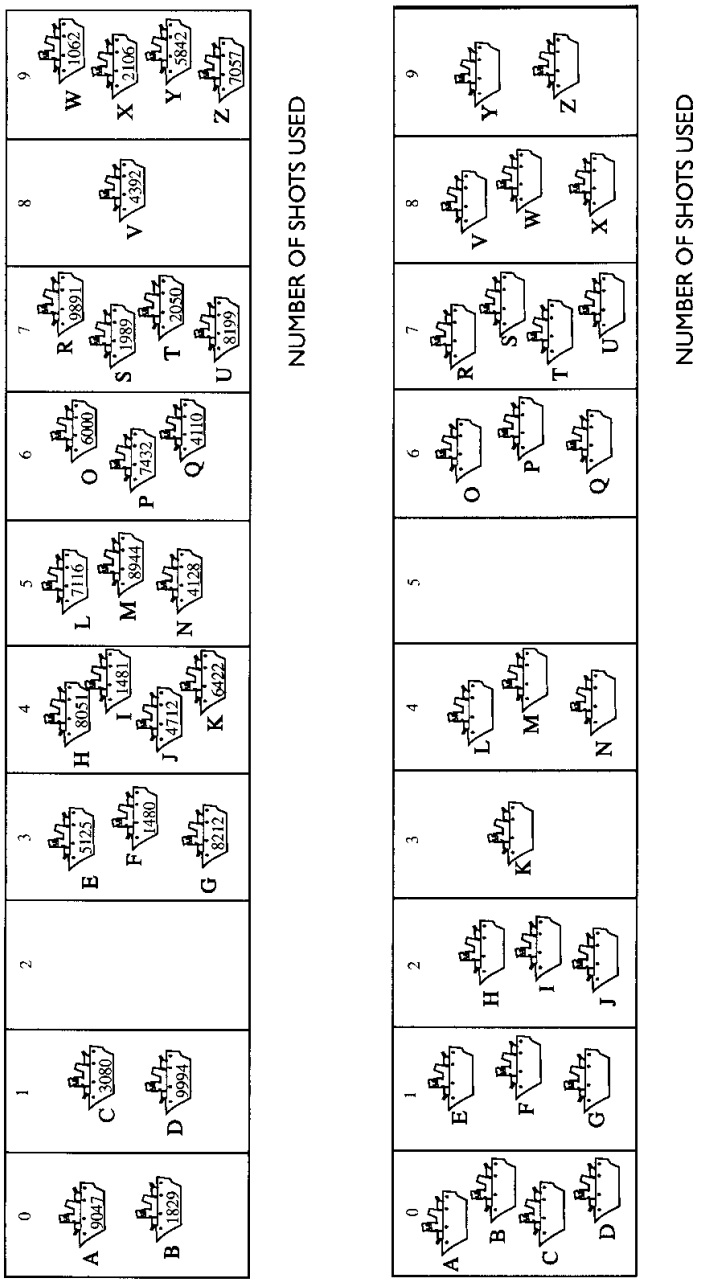


Instructions: *Your opponent should have sheet 2B. Choose one of the ships, tell your opponent its number, and then take turns trying to locate each other's ship. The numbers on the ships are in increasing order.*



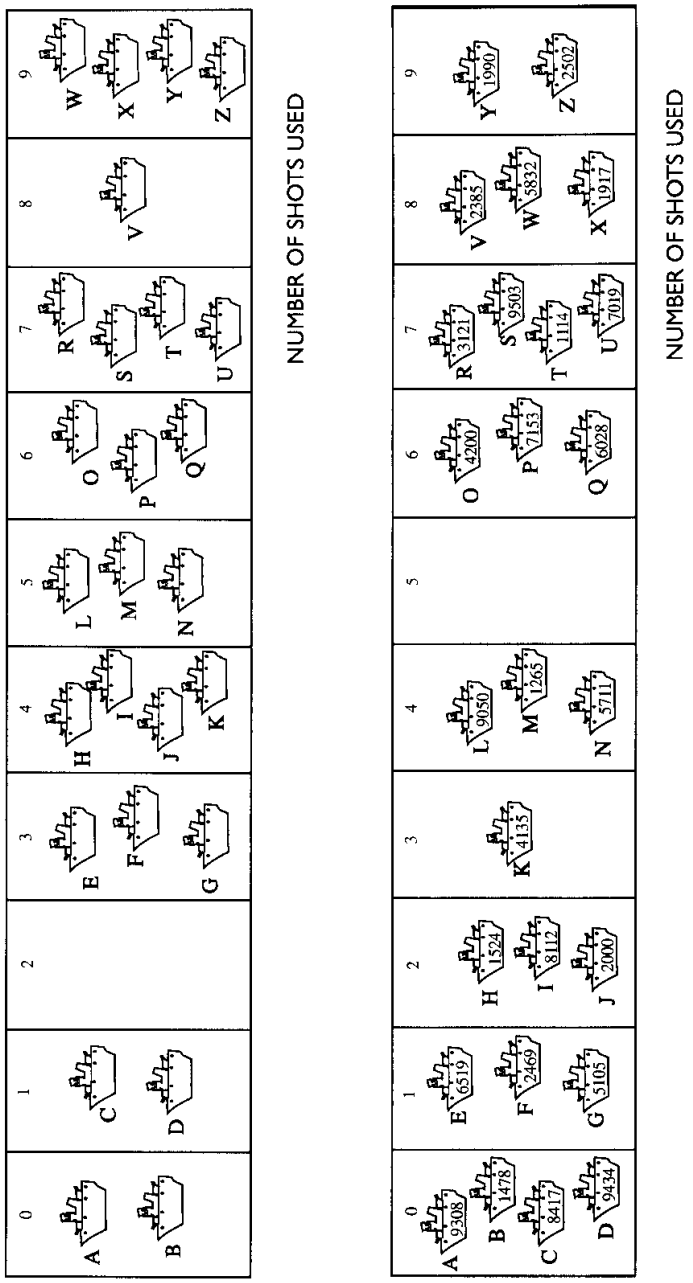
Instructions: *Your opponent should have sheet 2A. Choose one of the ships, tell your opponent its number, and then take turns trying to locate each other's ship. The numbers on the ships are in increasing order.*

Battleships sheet 3A

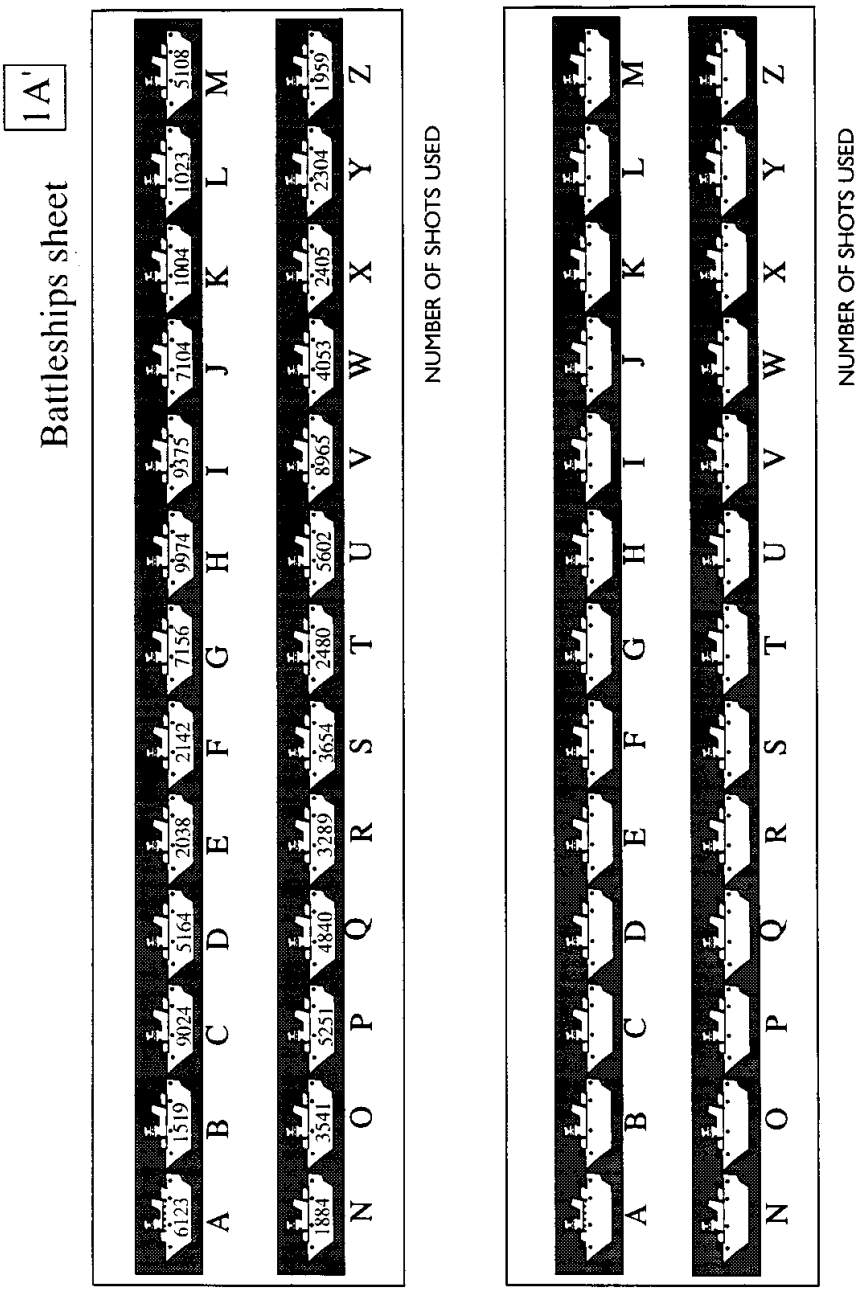


Instructions: *Your opponent should have sheet 3B. Choose one of the ships, tell your opponent its number, and then take turns trying to locate each other's ship. The box that a ship is in can be calculated by adding up its digits, and taking the last digit of the sum.*

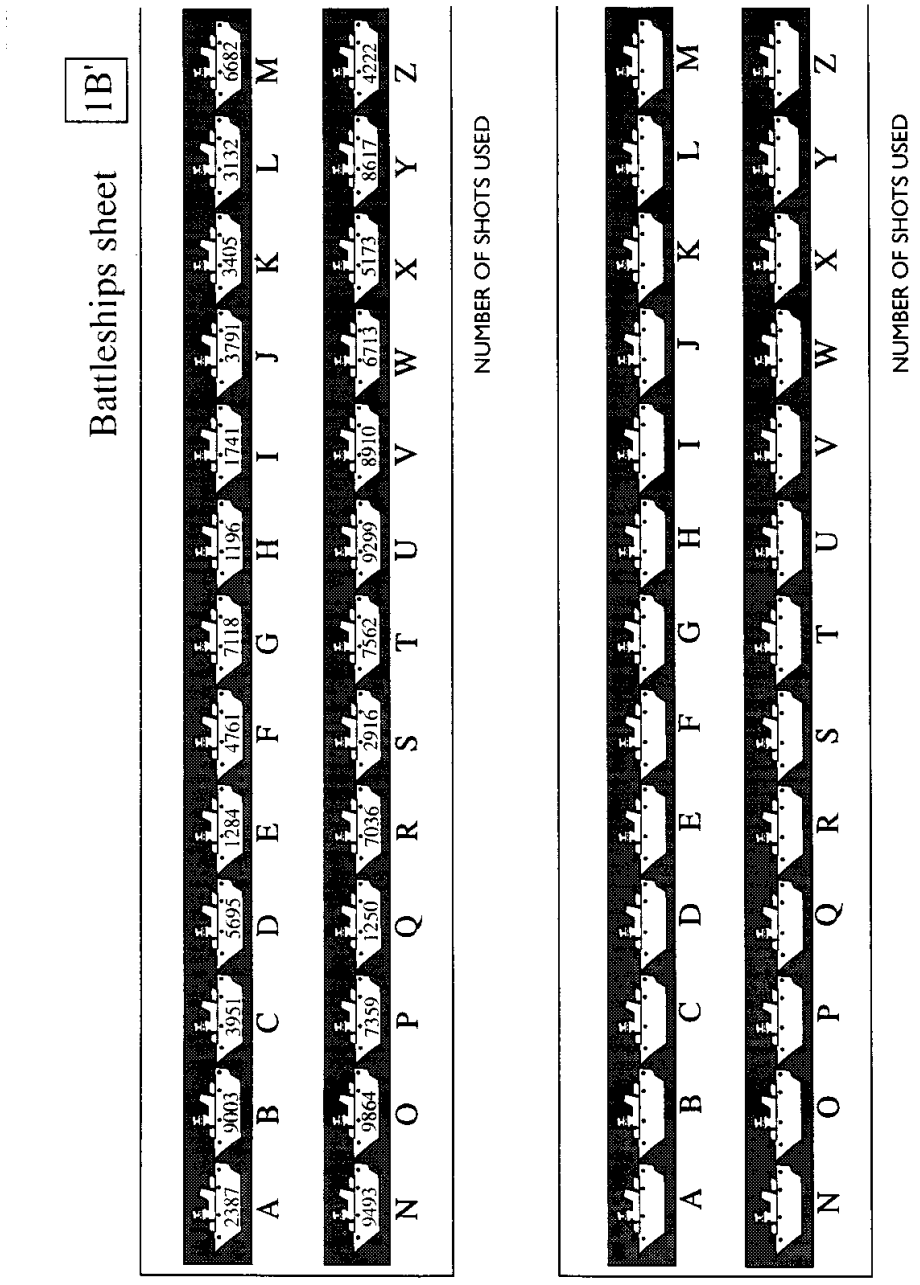
Battleships sheet 3B



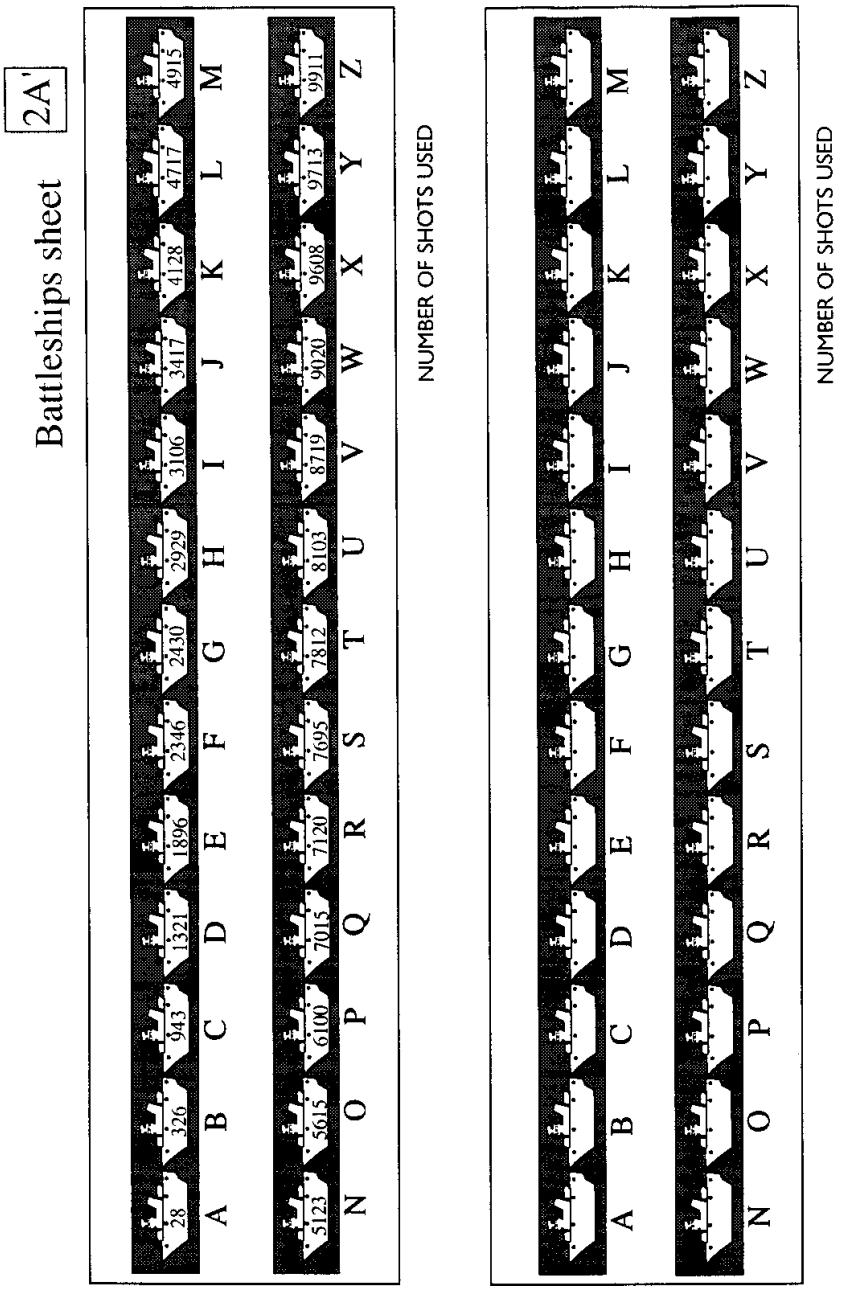
Instructions: *Your opponent should have sheet 3A. Choose one of the ships, tell your opponent its number, and then take turns trying to locate each other's ship. The box that a ship is in can be calculated by adding up its digits, and taking the last digit of the sum.*



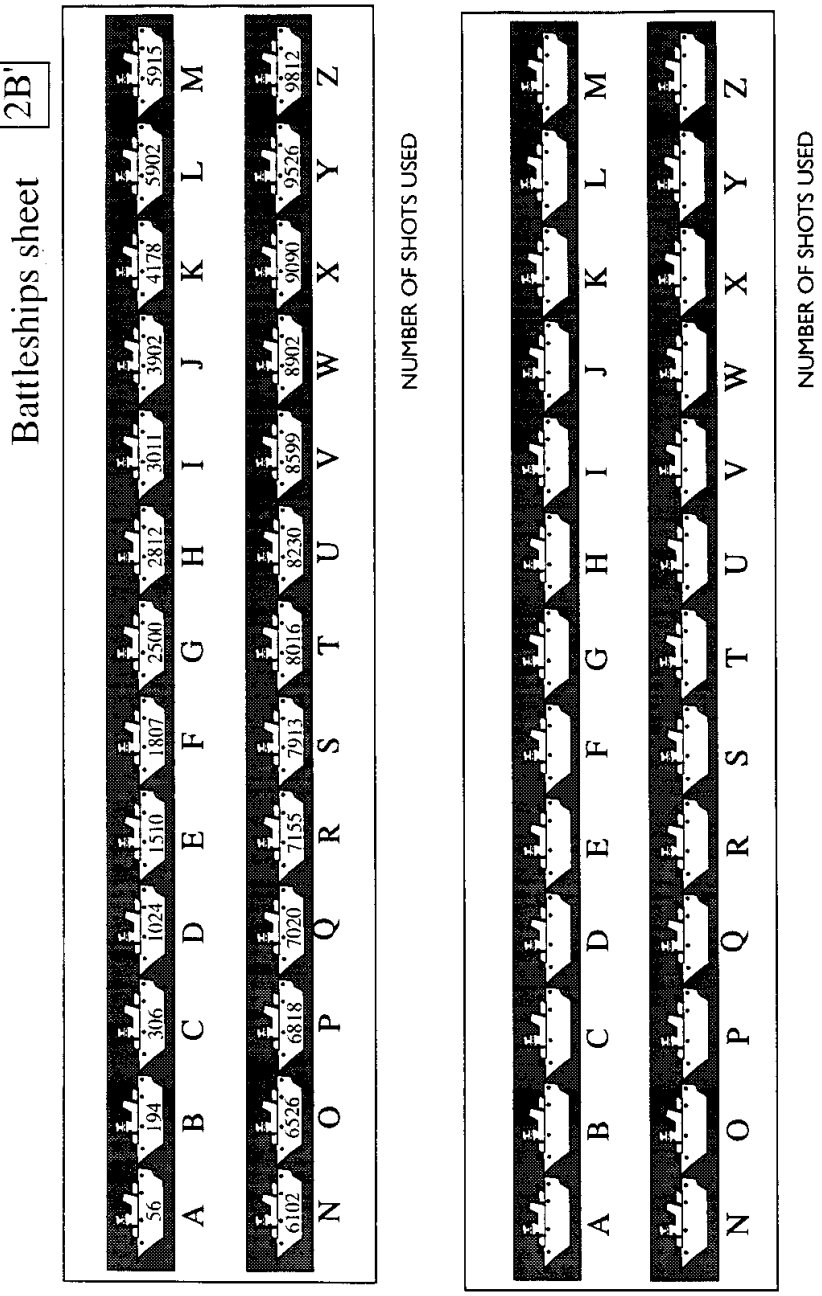
Instructions: *Your opponent should have sheet 1B′. Choose one of the ships, tell your opponent its number, and then take turns trying to locate each other's ship.*



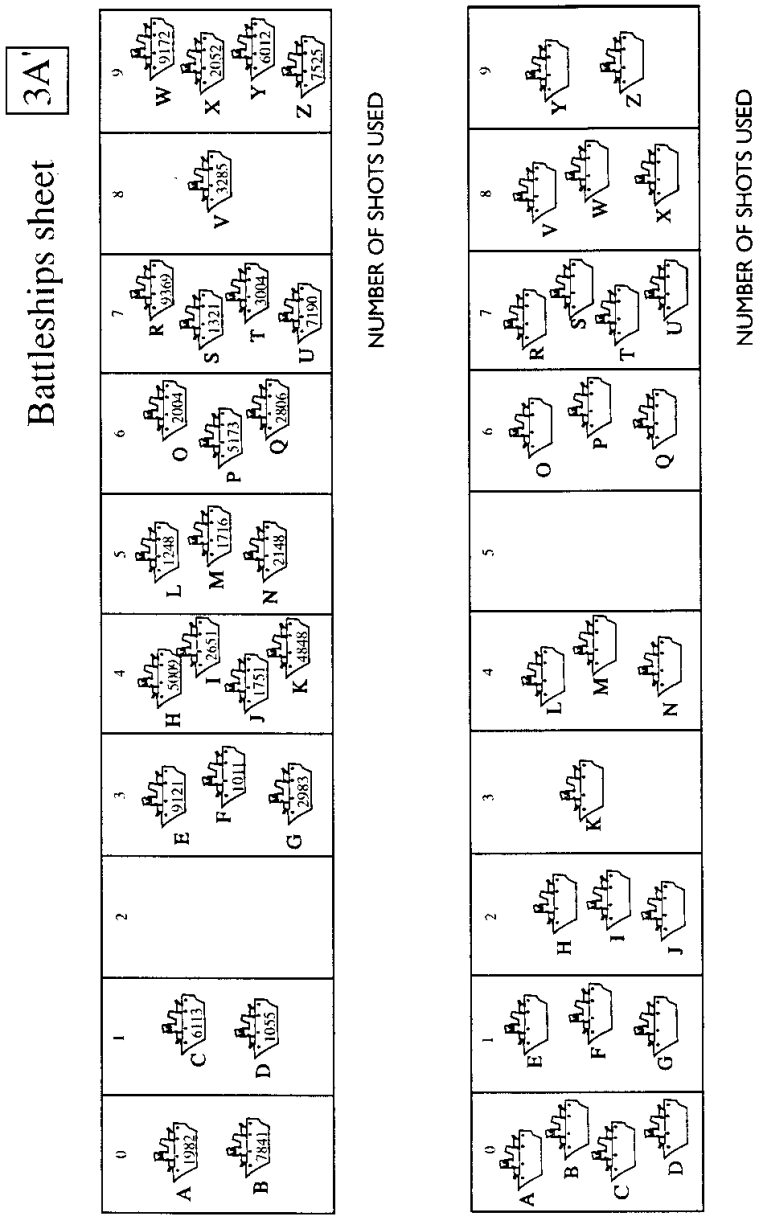
Instructions: *Your opponent should have sheet 1A′. Choose one of the ships, tell your opponent its number, and then take turns trying to locate each other's ship.*



Instructions: *Your opponent should have sheet 2B′. Choose one of the ships, tell your opponent its number, and then take turns trying to locate each other's ship. The numbers on the ships are in increasing order.*



Instructions: *Your opponent should have sheet 2A′. Choose one of the ships, tell your opponent its number, and then take turns trying to locate each other's ship. The numbers on the ships are in increasing order.*



Instructions: *Your opponent should have sheet 3B′. Choose one of the ships, tell your opponent its number, and then take turns trying to locate each other's ship. The box that a ship is in can be calculated by adding up its digits, and taking the last digit of the sum.*

Battleships sheet 3B′

Instructions: *Your opponent should have sheet 3A′. Choose one of the ships, tell your opponent its number, and then take turns trying to locate each other's ship. The box that a ship is in can be calculated by adding up its digits, and taking the last digit of the sum.*

Activity 7

Lightest and heaviest—*Sorting algorithms*

Age group	Early elementary and up.
Abilities assumed	Using balance scales, ordering.
Time	10 to 40 minutes.
Size of group	From individuals to the whole classroom.

Focus

Comparing.

Ordering.

Inequalities.

Summary

Computers are often used to put lists into some order, whether alphabetic, numeric, or by date. If you use the wrong method, it can take a long time to sort a large list into order, even on a fast computer. Fortunately several fast methods are known for sorting. In this activity children will encounter different methods for sorting, and see how a clever method can perform the task much more quickly than a simple one.

Technical terms

Sorting algorithms; insertion sort; selection sort; quicksort; recursion; divide and conquer; merging; mergesort; insertion sort; bubble sort.

Materials

Each child or group of children will need:

a set of about eight containers of the same size but different weights (e.g. milk cartons or film canisters filled with sand), and

balance scales.

What to do

In computer science, the term *sorting* usually refers to putting a list into alphabetical or numeric order. This is different from the common meaning in schools, where sorting involves placing objects into categories, or grouping identical objects together.

1. Discuss the computer science meaning of sorting, and see if the children can think of places where putting things into order is important (such as names in the telephone book, entries in a dictionary, the index of a book, the books on the shelves in a library, the letters in a postal worker's bag, a class roll, names in an address book, a list of files on a computer). Have the children think about the consequences if these things were not in order (usually the problem is that it takes a long time to locate an object in an unsorted list).

   Point out that sorting lists helps us find things quickly, and also makes extreme values easy to see. If you sort the marks for a class test into order, the lowest and highest marks become obvious.

2. The children will be using balance scales with a set of about eight containers that are of different weights, but look identical. Having identical containers ensures that they must compare the weights using the scales instead of visually. If the weights can be compared simply by picking them up, assign a person for each balance scale to handle the weights under the direction of the children who are making the decisions. The only clues available to the children should arise from comparisons between pairs of weights on the scales.

   It can be helpful for the teacher to keep track of the order of the weights by writing a code on each one that the children won't be able to interpret. For example, if you know the letters of the Greek alphabet you could label the containers from lightest to heaviest with α, β, etc.

   Check that the children can use the balance scales to find the lightest and heaviest of two objects.

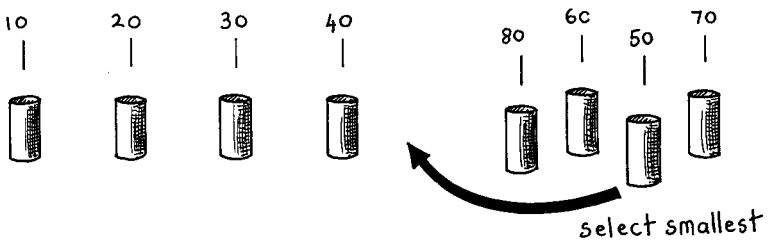


Figure 7.1: Sorting weights using the selection sort method

3. Have the children find the lightest of the eight or so weights they have been given, using only the balance scales. (The best way to do this is to go through each object in turn, keeping track of the lightest one so far. That is, compare two objects, and keep the lighter one. Now compare that with another, keeping the lighter from the comparison. Repeat until all the objects have been used.)

4. Discuss how you would check whether the weights are sorted into ascending order. (Check that the first object is lighter than the second, the second is lighter than the third, and so on. If each pair is in order then the whole list must be in order.)

5. Have the children sort three containers into ascending order of weight, using only comparisons on the balance scales. This can easily be done with three comparisons, and sometimes just two will suffice—if the children realize that the comparison operator is transitive (that is, if A is lighter than B and B is lighter than C, then A must be lighter than C).

6. Have the children sort all of the objects (about eight) into ascending order, using whatever strategy they wish. This might be very time consuming. When they think they have finished, check their ordering by comparing each adjacent pair of objects on the balance scales.

7. Have the children sort their weights into order using the following method, which is called *selection sort*. They should count how many comparisons they make to sort the objects.

   This is how selection sort works. Find the lightest weight in the set using the method described above, and put it to one side. Next, find the lightest of the remaining weights, which belongs next in ascending order of weight, and remove it. Repeat this until all the weights have been removed. Figure 7.1 shows the fifth weight being selected and added to the end of the sorted list.

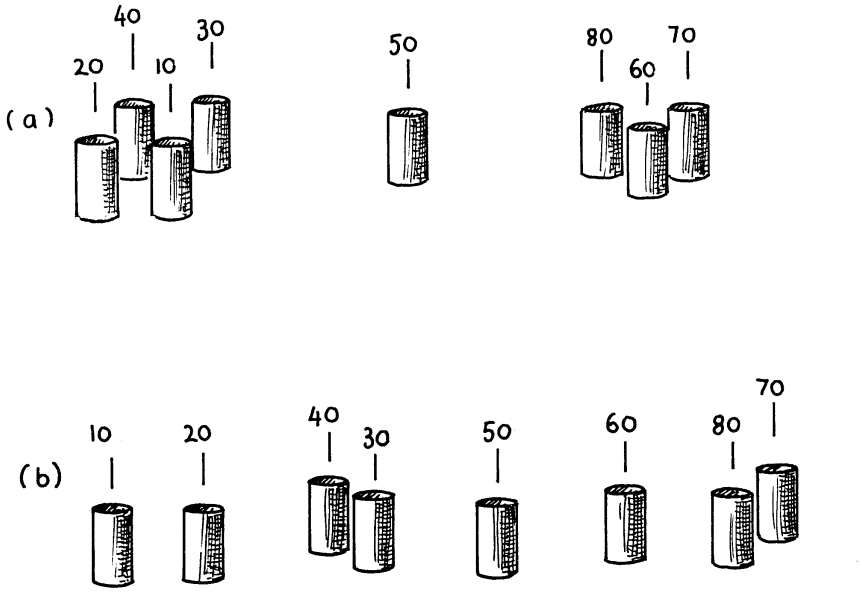


Figure 7.2: Sorting weights using the quicksort method

8. If the children have sufficient mathematical background, they should be able to calculate how many comparisons are required to sort a group of objects using this method. (To find the minimum of n objects requires $n-1$ comparisons. For example, to find the minimum of two objects requires only one comparison, and to find the minimum of five objects requires four comparisons. To sort eight objects using selection sort, there will be seven comparisons to find the smallest one, six to find the next smallest, five for the next, and so on, giving a total of $7+6+5+4+3+2+1=28$ comparisons.)

9. Have the children try out the following method of sorting, which is called *quicksort*. As its name might imply, quicksort is a lot faster than selection sort, particularly for larger lists. In fact, it is one of the best methods known. Have the children count how many comparisons they need to make when they sort their objects using quicksort.

   This is how quicksort works. Choose one of the objects at random, and place it on one side of the balance scales. Now compare each of the remaining objects with it, and put them in one of two groups, those that are lighter, and those that are heavier. Put the lighter group on the left, the chosen object in the middle, and the heavier ones on the right (it is possible that there may be no objects in one of the groups). Now repeat this procedure on each of the groups—that is, use the balance to divide the groups into subgroups. Keep repeating on the remaining groups until no group has more than one object in it. Once all the groups have been divided down to single objects, the objects will be in ascending order.

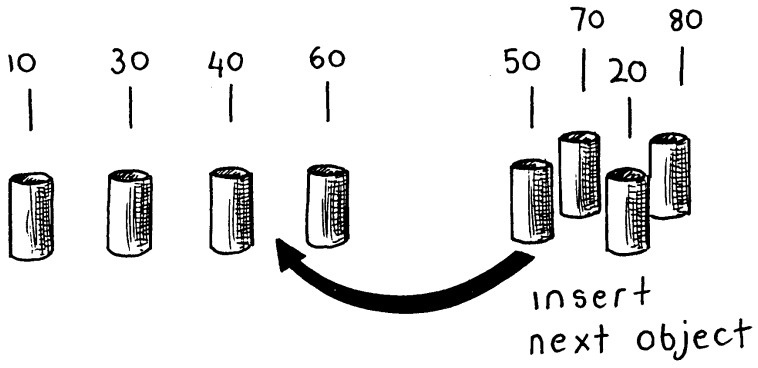


Figure 7.3: Sorting weights using the insertion sort method

Figure 7.2 shows how this process might go. In Figure 7.2a the 50 gram weight has been chosen at random, and the balance was used to group the lighter weights to the left and the heavier ones to the right. In Figure 7.2b the lighter group has been split using the 20 gram weight, and the heavier one was split using the 60 gram weight. In both cases the split is uneven, and in the second case one of the two subgroups has no objects in it! Once the two groups of two objects are split up—which is trivial because they are so small—the sorting will be complete.

In practice the way that the objects are split up will depend on which objects are chosen at random. For example, if the 80 gram weight were chosen as the first one to divide the groups in Figure 7.2, the groups would be very unbalanced. Nevertheless, the list will always end up in increasing order. Unbalanced splits simply increase the number of comparisons required to complete the sorting.

10. Compare the number of comparisons that were made for selection sort and quicksort. If different groups have performed the experiment, have them share their results with the whole class. (Using selection sort on 8 items will always take 28 comparisons. In the example in Figure 7.2, quicksort used only 14 comparisons to sort the 8 items. Sometimes it will take more, depending on which weights are chosen as the splitting point. In the worst case, if the randomly chosen weight always happens to be the lightest or heaviest, it will take the same number of comparisons as selection sort, but this is unlikely. It will never be worse than selection sort, and usually a lot better.)

Variations and extensions

Many different methods for sorting have been invented. More advanced classes may wish to try sorting weights using other methods. In addition to selection sort and quicksort, the following methods are common.

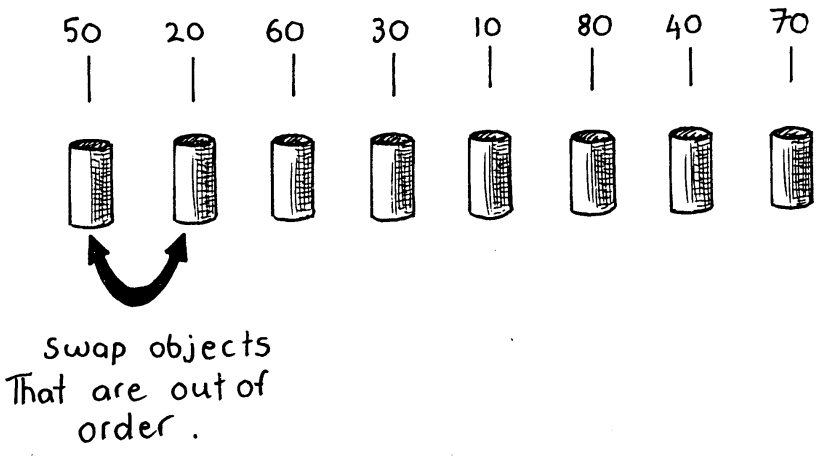


Figure 7.4: Sorting weights using the bubble sort method

Insertion sort works by removing each object from an unsorted group and inserting it into its correct position in a growing list (see Figure 7.3). With each insertion the group of unsorted objects shrinks and the sorted list grows, until eventually the whole list is sorted. This method is often used by card players to sort a hand into order.

Bubble sort involves repeatedly going through the list, swapping any adjacent objects that are in the wrong order (Figure 7.4). The list is sorted when no swaps occur after a pass through the list. This method is not very efficient, but some people find it easier to understand than the others.

Mergesort is one of the fastest known sorting methods, even though it might sound like hard work. The objects are distributed randomly into two groups of equal size (or nearly equal size if there is an odd number of objects). Each of these half-size groups is then sorted into order (we won't worry yet about how this is done), and then the two groups are merged together. Merging two sorted lists together is easy: you repeatedly remove the smaller of the two objects at the front of the two lists. In Figure 7.5 the 40 and 60 gram weights are at the front of the two lists respectively, so the 40 gram one is removed and added to the end of the final sorted list. This leaves the 50 gram weight at the front of the first list, which will be compared with the 60 gram weight and removed next.

There remains the question of how to sort the two half-sized lists. Simple—they are sorted using mergesort! This means they will each be divided into quarter-sized lists, sorted, and merged back to give a sorted half-sized list. Each time a list is divided into two, it is sorted using mergesort, unless it only contains one item. In that case it is already sorted, and so nothing needs to be done. Eventually, all the lists will be divided down to individual items, and so there is no danger of repeatedly having to perform mergesort *ad infinitum*.

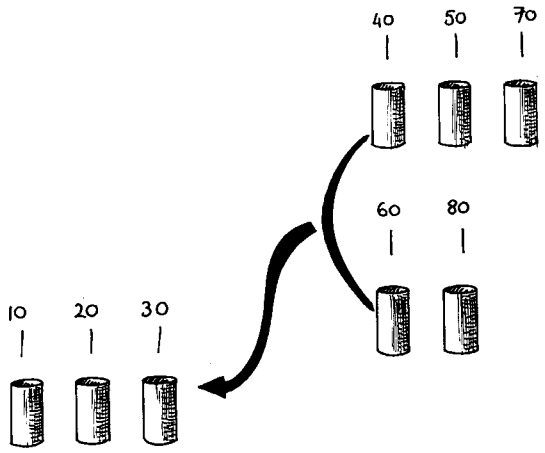


Figure 7.5: Sorting weights using the mergesort method

The relative efficiency of each sorting method becomes more obvious when a larger number of objects (two dozen or more) are being sorted. Write random numbers on about 50 cards, and have children try to sort them into increasing order using selection sort and quicksort. Selection sort will require 1225 comparisons of the values, whereas quicksort will require only about 271 on average. This is quite a saving!

Older children may enjoy a short cut for adding up the number of comparisons that selection sort makes. We pointed out above that n objects will take $1+2+3+4\cdots+n-1$ comparisons to sort. Adding up these numbers is made easy by regrouping them. For example, to add up the numbers $1+2+3+\cdots+20$, regroup them as

$$\begin{aligned} & (1+20)+(2+19)+(3+18)+(4+17)+(5+16)+ \\ & \quad (6+15)+(7+14)+(8+13)+(9+12)+(10+11) \\ = \;& 21\times 10 \\ = \;& 210. \end{aligned}$$

In general, the sum $1+2+3+4\cdots+n-1=n(n-1)/2$.

Another approach to sorting is described in Activity 8 on sorting networks.

What's it all about?

We are used to working with ordered lists. Telephone directories, dictionaries and book indexes all use alphabetical order, and life would be far more difficult if they didn't. Information is much easier to find in a sorted list. Also, if a list of numbers (such as a list of expenses) is sorted into order, the extreme cases become apparent because they are at the beginning and end of the list. Duplicates are also easy to find, because they end up adjacent.

Sorting is an important task that computers perform frequently. In the 1970s it was estimated that most computers spent a quarter of their time doing sorting—and for some it was over half.[1] A good deal of computer output is sorted: lists of names and address need to be in alphabetical order, and files displayed on the screen are often shown alphabetically, or in order of when they were created or last modified. Unsorted output can be confusing because people expect the order to be of some significance—it looks more professional if a list of names is printed in alphabetical order, even when it is not strictly necessary.

Using a balance scale for sorting weights is an accurate model of what happens on computers, because the basic operation in sorting is to compare two values to see which is the larger. Most sorting programs are based on this operation, and their efficiency is measured in terms of how many comparisons they require to sort a list, because that is a good indicator of how long they will take regardless of the computer they run on.

Researchers have worked for decades on the sorting problem, and literally dozens—if not hundreds—of methods have been invented. Those described above are the most common, and many others are related to them. They can be divided into two groups, those that are relatively slow (insertion sort, selection sort, and bubble sort), and those that are fast (quicksort and mergesort). The slower ones are only useful for special situations, and are just too slow if there is a large amount of data to be sorted. Quicksort and mergesort are standard methods on many computers.

Both quicksort and mergesort introduce an important and powerful concept in computer science: recursion. This is where a method (algorithm) is described in terms of itself. Both methods work by dividing a list into parts, and then performing the same kind of sort on each of the parts. This approach is called *divide-and-conquer*. The list is divided repeatedly until it is small enough to conquer. For both methods, the lists are divided until they contain only one item. It is trivial to sort one item into order!

Merging two sorted lists, which is the basis of mergesort, is an efficient method for the more general problem of matching items from two groups. For example, a postal worker takes a sorted pile of envelopes and "merges" it with a sorted street of mailboxes—the next letter to be delivered will always be at the top of the pile. This technique is also used for the controversial practice of computer matching, where two agencies (such as social welfare and the tax department) look for clients who are enrolled in both to detect fraud—for example, if someone is declaring an income to the tax department, they probably should not be collecting an unemployment benefit. Lists from the agencies are matched by sorting each into order, and then matching duplicates in the same way that a postal worker matches envelopes to mailboxes.

Further reading

Sorting is a standard topic in computer science courses, and is discussed in many introductory texts, such as Kruse's *Data Structures and Program Design*. Harel touches on sorting several times in *Algorithmics*. The classic (albeit slightly dated) text on sorting is Knuth's *The Art*

[1] Nowadays most computer time is spent idling or running those programs that draw changing patterns or pictures on the screen ("screen savers"), because greatly reduced prices have removed the necessity to have a machine performing useful work all the time.

of Computer Programming, Volume 3: Sorting and Searching, published in 1973. Dewdney discusses mergesort, and an interesting method called heapsort, in the *Turing Omnibus*.

Activity 8

Beat the clock—*Sorting networks*

Age group	Middle elementary and up.
Abilities assumed	Identifying the larger and smaller of two numbers. Activity 7 on sorting algorithms is helpful, but not essential, preparation.
Time	10 to 30 minutes.
Size of group	At least six children.

Focus

Comparing

Ordering

Developing algorithms

Cooperative problem solving

Summary

Even though computers are fast, there is a limit to how quickly they can solve problems. One way to speed things up is to break a job up into pieces and have a different computer process each piece simultaneously, a strategy that is called "parallel computing." This activity shows how sorting items into numerical order, normally thought of as a sequential kind of process, can be carried out more quickly using parallelism.

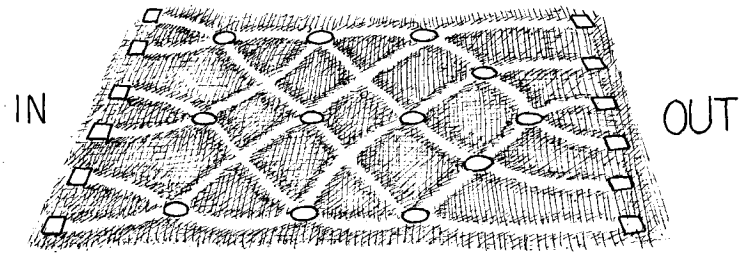


Figure 8.1: A sorting network

Technical terms

Parallel computation; concurrency; sorting.

Materials

Chalk or other material for marking on the ground,

two sets of six cards, such as the ones in the blackline master on page 89, and

a stopwatch.

What to do

1. Before working with the children, mark out the network shown in Figure 8.1 on the ground. It needs to be large enough for children to run around it following the lines, and the circles and squares must be obvious. There are several ways of marking the network. Possibilities include

   - drawing with chalk on an asphalt playground,
   - construction site ribbon "stapled" to the grass with small pieces of masking tape,
   - masking tape on the floor,
   - string or streamers attached to the ground with tape, and
   - mowing paths through long grass.

2. If the children have not already done the activity on sorting (Activity 7), discuss the frequent need for computers to sort lists of items into order (Activity 7 gives information about this).

3. Show the children the sorting network (Figure 8.1) drawn on the ground, and explain how they will use it, as follows.

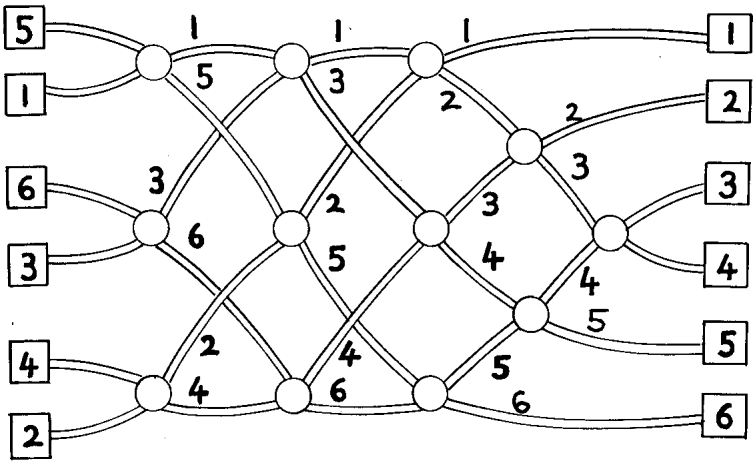


Figure 8.2: An example flow of numbers through a sorting network

Divide the children up into teams of six. One team uses the network at a time. Give each member of the team a card with a number from 1 to 6 on it (see the blackline master on page 89). The members of a team start at the six squares on the left-hand side of the network, in a random order. Children advance along the lines marked, and when they reach a circle they wait for someone else to arrive. When there are two children in a circle they compare cards. The one with the smaller number takes the exit to their left, and the other to their right (it can be helpful if the line to the right is thicker to remind them that the larger number goes that way). When they arrive at the six squares on the right they should have ended up in ascending numerical order of cards. Figure 8.2 shows an example of the numbers flowing through the network of Figure 8.1, with the sorted numbers coming out on the right.

4. Once the children have understood the goal, and have had a trial run, use the stopwatch to time how long each team takes to get through the network. Now use cards with larger numbers (such as the three-digit ones in the blackline master on page 89). For older children, make up cards with even larger numbers that will take some effort to compare—or with words, to be compared alphabetically. The idea is to see which team can get through the network quickest. In the rush it is possible that the numbers will end up unsorted, or that a lone child will be left standing in the middle of the network. In both cases the team has made an error, and must start again.

Variations and extensions

Once the children have understood the operation of a node (circle) in the network, where the smaller value goes left and the other goes right, all sorts of variations can be explored using this as a building block. One simple question that they should be able to answer is “what happens

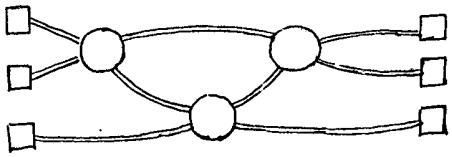


Figure 8.3: A smaller sorting network

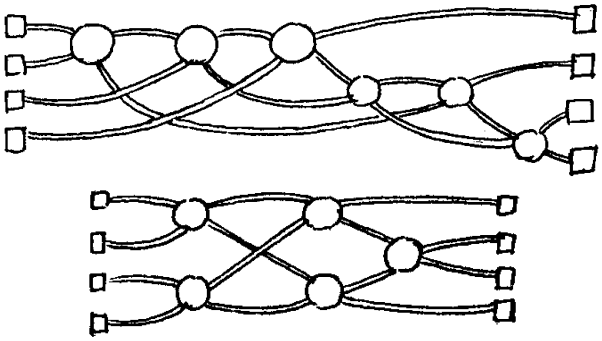


Figure 8.4: Two different networks for sorting four input values

if the smaller one goes right instead of left and vice versa?" (The numbers will be sorted in reverse order.) Another question for them to try to answer is "does it work if the network is used backwards?" (It will not necessarily work, and the children should be able to find an example of an input that comes out in the wrong order.)

The children can try to design smaller and larger networks. For example, Figure 8.3 shows a network that sorts just three numbers, which children should try to come up with on their own.

The network for sorting a particular number of inputs is not unique. For example, Figure 8.4 shows two different networks that will sort four inputs. Which is the faster? (The second one is. Whereas the first requires all comparisons to be done serially, one after the other, the second has some being performed at the same time. The first network is an example of serial processing, whereas the second uses parallel processing to run faster.)

Figure 8.5 shows a larger sorting network which can be used if a group is keen to tackle something a bit more challenging.

The networks can also be used to find the minimum or maximum value of the inputs. For example, Figure 8.6 shows a network with eight inputs, and the single output will contain the minimum of the inputs (the other values will be left at the dead ends in the network).

Discuss processes from everyday experience that can (and can't) be accelerated using parallelism. For example, cooking a meal would be a lot slower using only one cooking element, because the items would have to be cooked one after the other. Likewise, when building a house, it is possible to have several people work on different tasks at the same time. What kinds of jobs can be completed more quickly by employing more people? What kinds of jobs can't?

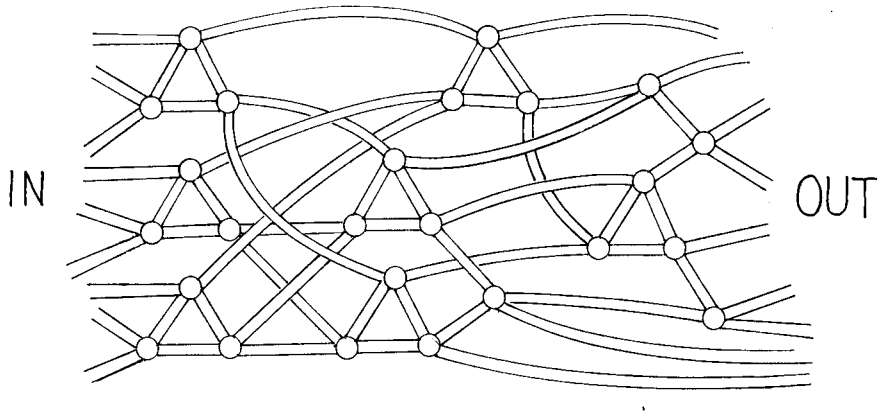


Figure 8.5: A larger sorting network

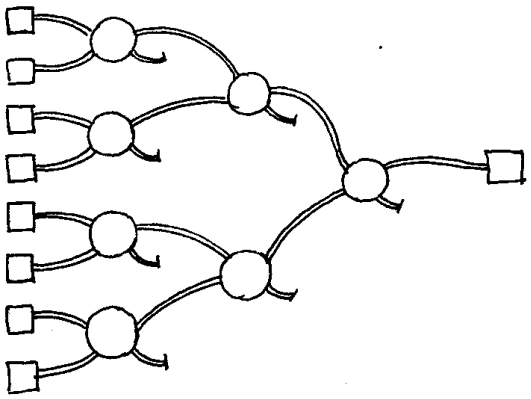


Figure 8.6: A network that finds the smallest of eight numbers

What's it all about?

The power of computers is increasing at a mind-boggling rate. Their speed doubles approximately every two years, and their memory capacity doubles even more often, yet their price remains roughly constant. With this exponential rate of growth of readily available computing power, it is hard to understand why people persistently complain that their computer is too slow. Yet they do, and those who can afford to are quick to upgrade when faster models come out. As we see what computers are capable of and find more uses to put them to, our expectations rise dramatically. Today's lowly word processor would have been unimaginable only 25 years ago, and a kid's drawing program would have been a research project back then.

Some people need faster computers so badly that they can't even wait a couple of years until computers are faster. Motivated by these needs, computer scientists explore ways to make computations happen more quickly without having to get faster computers. One way of doing this is to write programs that do their task using fewer computational steps. This approach is explored in the searching and sorting activities (Activities 6 and 7).

Another approach to solving problems faster is to have several computers work on different parts of the task at the same time. This can only be done if the task can be broken up into independent parts. This is not always possible, but computer scientists have found that a lot of everyday tasks can be parallelised, including searching a database and sorting data into order. This activity shows how the problem of sorting a list into order can be decomposed into a number of comparison operations, many of which can be carried out at the same time.

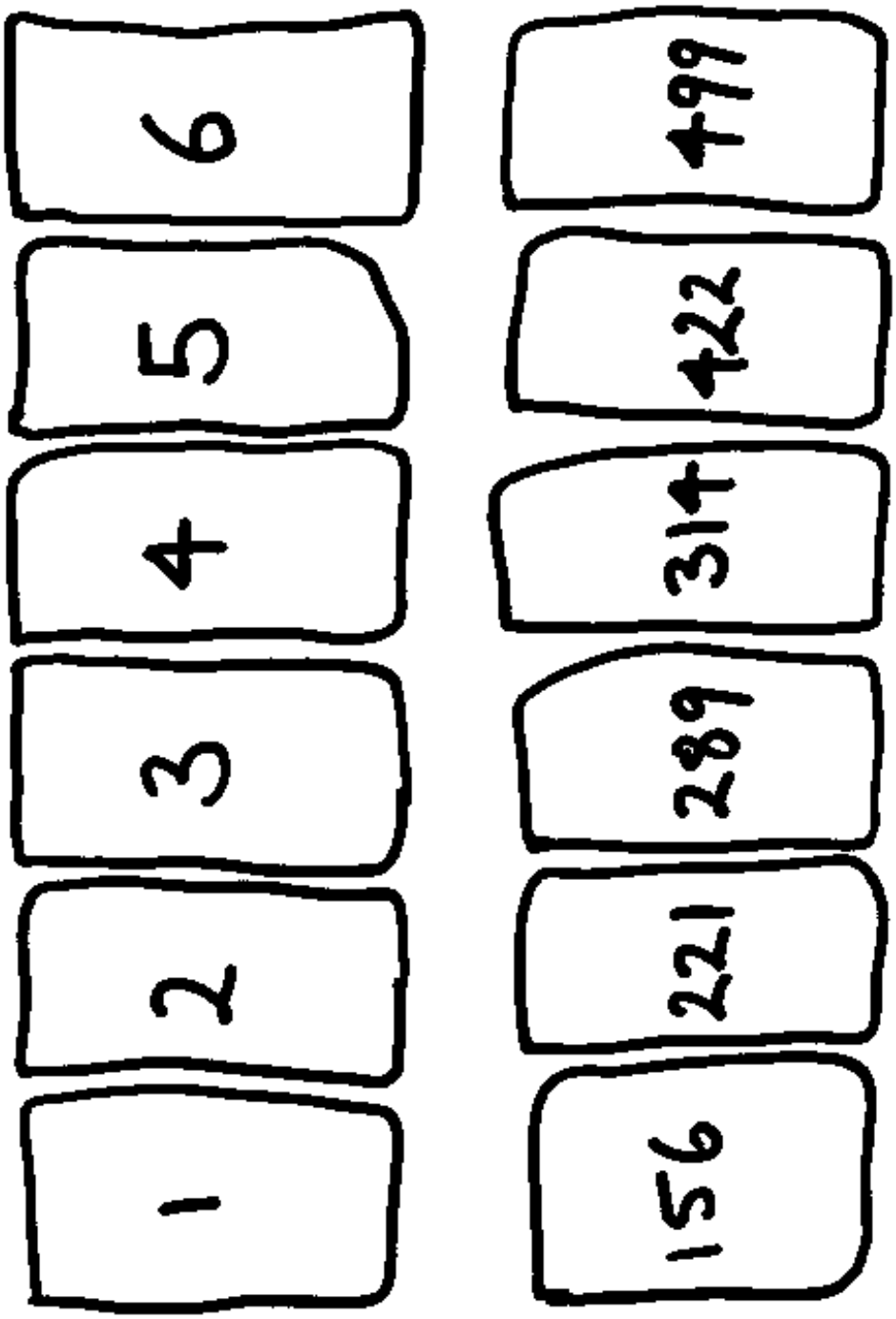
For example, in Figure 8.2, up to three comparisons are made in parallel. Even though a total of twelve comparisons will be made in the process of sorting the numbers, many of them are performed simultaneously, and the overall time will be that required for just five comparison steps. This parallel network sorts the list more than twice as quickly as a system that can only perform one comparison at a time.

Isn't parallel processing expensive? Well, yes, it is a bit. But most of the money you spend on a computer goes into the screen, the keyboard, the box, the power supply, the disk unit, ... even the desk you put it on. The processor chip is really quite cheap, and having a dozen (or even a hundred) of them inside the box instead of just one isn't going to affect the price of the computer as much as you might think.

Not all tasks can be completed more quickly by using parallel computation. As an analogy, imagine one person digging a ditch ten feet long. The task could be completed almost ten times as quickly if ten people each dug one foot of the ditch. However, the same strategy could not be applied to a ditch ten feet deep—the second foot is not accessible until the first foot has been dug. And although one woman can bear a child in nine months, it is not possible to speed up the process and have nine women work together to bear a child in one month!

Further reading

Chapter 10 of *Algorithmics* by David Harel discusses parallelism and concurrency, including sorting networks, and the analogies with ditch-digging and pregnancy.



Instructions: *These are two sets of six cards for use with the sorting networks activity.*

Activity 9

The muddy city—*Minimal spanning trees*

Age group	Middle elementary and up.
Abilities assumed	Need to be able to count to about 50.
Time	About 20 to 30 minutes.
Size of group	From individuals to the whole classroom.

Focus

Puzzle solving.

Optimization.

Planning.

Summary

Our society is linked by many networks: telephone networks, utility supply networks, computer networks, and road networks. For a particular network there is usually some choice about where the roads, cables, or radio links can be placed. The following activity explores one way to optimize the choice of links between objects in a network.

Technical terms

Minimal spanning trees; greedy algorithms; graph algorithms.

Materials

Each child will need:

a copy of the blackline master on page 96 (it is easier for the children to use if it is enlarged onto double-sized paper), and

counters or squares of cardboard (approximately 40 per child).

What to do

Once upon a time there was a city that had no roads. Getting around the city was particularly difficult after rainstorms because the ground became very muddy—cars got stuck in the mud and people got their boots dirty. The mayor of the city decided that some of the streets must be paved, but didn't want to spend more money than necessary because the city also wanted to build a swimming pool. The mayor therefore specified two conditions: (1) Enough streets must be paved so that it was possible for everyone to travel from their house to anyone else's house by a route consisting only of paved roads, possibly via other houses, and (2) the paving should be accomplished at a minimum total cost.

The map on page 96 shows the layout of the city. The number of paving stones between each house represents the cost of paving that route. The problem comes down to figuring out the least number of paving stones needed to allow people to get from any house to any other.

1. Hand out a copy of the blackline master to each child and explain the muddy city problem using the story above (you will need to adapt the story to suit the age of the children).

2. Give out the counters and let the children try to find efficient solutions by putting counters where they think paving stones should be laid. Keep the class informed as children come up with better and better solutions. Figure 9.1 shows two optimal solutions, both with a cost of just 23 paving stones. As this example illustrates, there can be more than one solution to this kind of problem, each with the same total cost.

3. Discuss the strategies that the children used to solve the problem.

   Some of the children will have developed the strategy of starting with an empty map, and gradually adding counters as necessary until all of the houses are linked. This is a good strategy; in fact, you are guaranteed to find the optimal solution if you add the paths in increasing order of length, but don't add any paths that link houses that are already linked. Different solutions are generated by changing the order in which paths of the same length are added.

   Another strategy that the children might develop is to start with all of the paths paved, and then remove redundant paths. This can lead to an optimal solution, but requires considerably more effort.

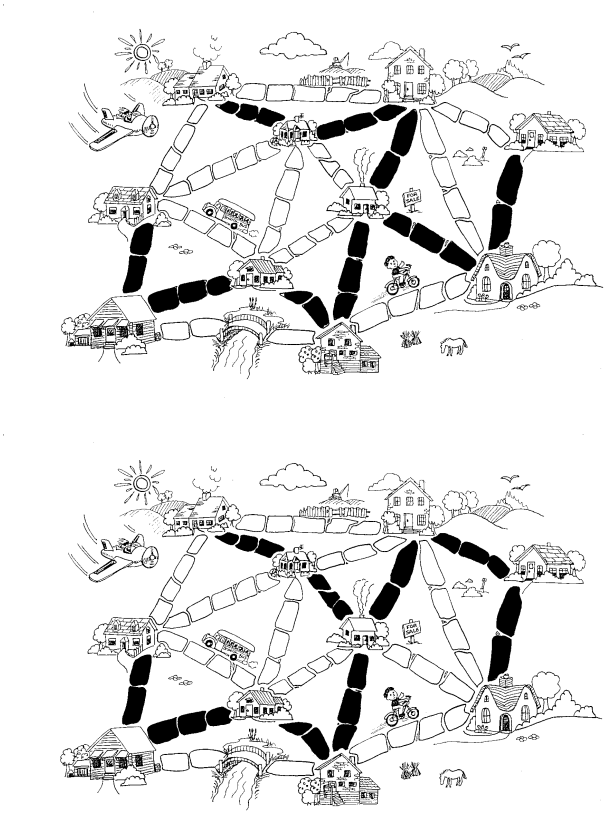


Figure 9.1: Two solutions to the muddy city problem on page 96

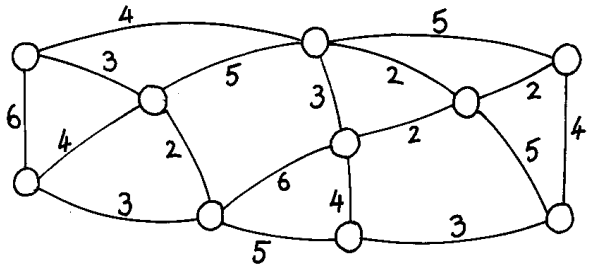


Figure 9.2: An abstract representation of a (different) muddy city

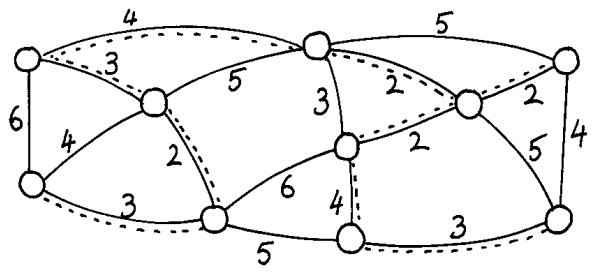


Figure 9.3: A solution to the muddy city problem in Figure 9.2

Variations and extensions

The children could experiment with other cities. Figure 9.2 shows a more abstract representation of a city that is quicker to draw and easier to work with. The houses are represented by circles, the muddy roads by lines, and the length of a road is given by the number beside the line. Computer scientists and mathematicians often use this kind of abstract representation of relationships for such problems. They call it a *graph*. This may be confusing at first because the term "graph" is used in mathematics to mean a chart displaying numerical data, such as a bar graph, but the graphs that computer scientists use are not related to these. Notice that the lengths in Figure 9.2 are not to scale. The dotted lines in Figure 9.3 show a solution.

You could have the children think about how many roads or connections are needed if there are n houses in the city. It turns out that an optimal solution will always have exactly $n - 1$ connections in it, as this is always sufficient to link up the n houses, and adding one more would create unnecessary alternative routes between houses.

Another extension is to have the children look for real networks that can be represented by a graph, such as the network of roads between local cities, or airplane flights around the country. The muddy city algorithm may not be much use for these networks, because it simply minimizes the total length of the roads or flight paths. It guarantees that you can get between any two points, but does not take into account the convenience of the route. However, there are many other algorithms that can be applied to graphs, such as finding the shortest distance between two points, or the shortest route that visits all the points. Representing the network abstractly as a graph is a useful first step towards solving these problems.

What's it all about?

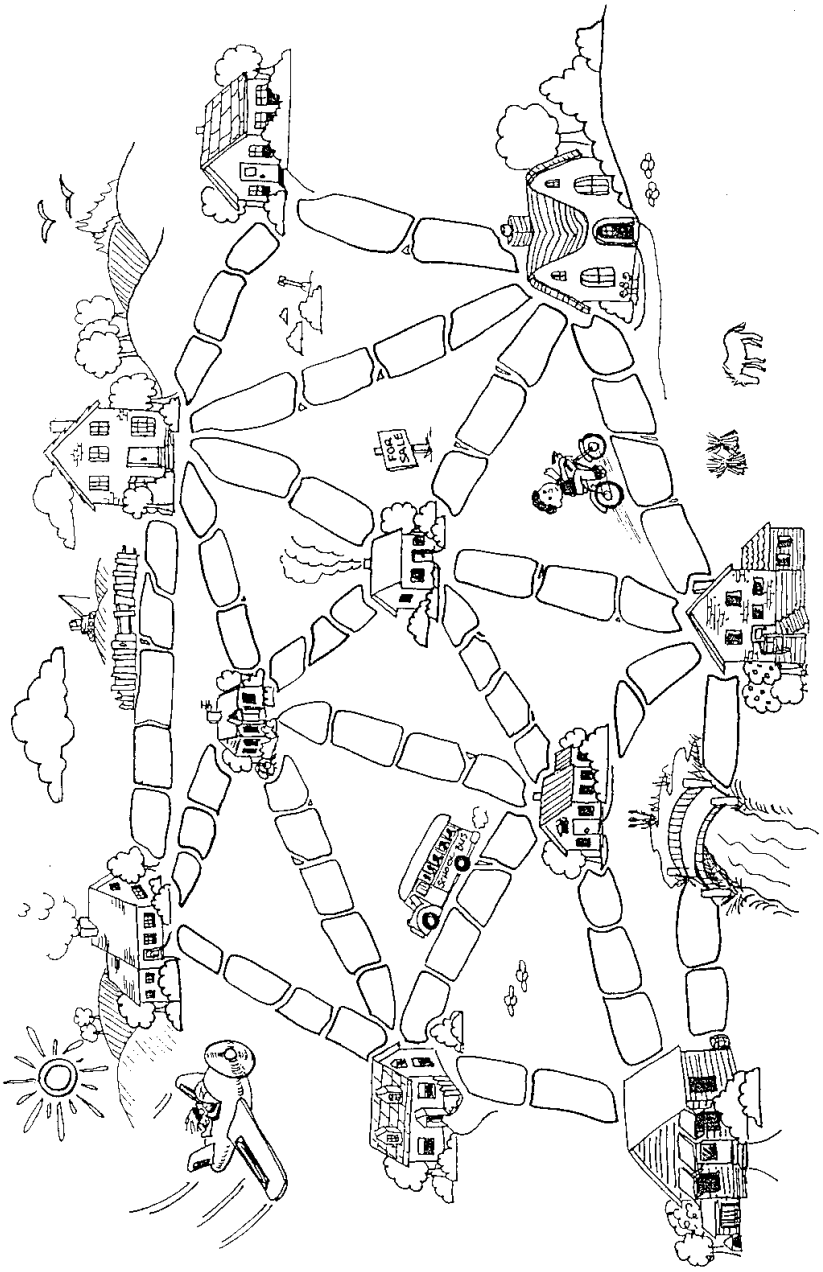
Suppose you are designing how a utility such as electricity, gas, or water should be delivered to a new community. A network of wires or pipes is needed to connect all the houses to the utility company. Every house needs to be connected into the network at some point, but the route taken by the utility to get to the house is not usually very critical, just so long as a route exists. The task of designing a network with a minimal total length is called the *minimal spanning tree* problem.

Minimal spanning trees aren't only useful in gas and power networks; they help us solve problems in applications as diverse as computer networks, telephone networks, oil pipelines, and airline routes—although, as noted above, when routing people you have to be careful to take into account the convenience of the route as well as its cost. They are also useful as one of the steps for solving other problems on graphs, such as the "traveling salesperson problem" which seeks the shortest route that visits every point in the network.

There are efficient algorithms (methods) for solving minimal spanning tree problems. A simple but method that gives an optimal solution is to start with an no connections, and add them in increasing order of size, only adding connections that join up part of the network that wasn't previously connected. This approach was mentioned earlier, and we have known children to discover this algorithm for themselves. It is called Kruskal's algorithm after J.B. Kruskal, who published it in 1956.

Further reading

Harel discusses minimal spanning trees in *Algorithmics*, and also shows how they can be used to help with the "traveling salesperson." Dewdney's *Turing Omnibus* has a section on minimal spanning trees, which discusses Kruskal's algorithm.



Instructions: *Find the minimum number of paving stones that need to be used so that you can get from any house to any other house. (The bridge doesn't need to be paved.)*

Activity 10

The orange game—*Routing and deadlock in networks*

Age group	Middle elementary and up.
Abilities assumed	Working in a group to solve a problem.
Time	20 minutes or more.
Size of group	About five children or more, can be adapted for a single person.

Focus

Cooperative problem solving.

Logical reasoning.

Planning.

Summary

When people are contending for scarce resources, careful planning is needed. When there are dependencies amongst the demands for a resource (such as cars using roads, or messages getting through networks) there is the possibility of "deadlock," where a stalemate is reached and someone must back off before anyone can proceed. This activity gives children experience with a situation where they are contending for resources. A cooperative procedure is required for everyone to accomplish the desired result without becoming deadlocked.

Figure 10.1: Playing the game with six children and eleven oranges; child D has an empty hand

Technical terms

Routing; deadlock; message passing.

Materials

Each child will need:

two oranges (or some other tokens that can be labeled), and

a name tag or sticker.

What to do

1. Form the children into groups of about five or more, and seat each group in a circle. (It can be helpful to start with just one group and have everyone else watch first. The game can take as long as 15 minutes, but there is no need to watch it to completion.)

2. The members of the group are given two oranges each (or balls, or cards), except for one child who has only one. The children are labeled with a letter of the alphabet, and their oranges are labeled with the same letters, two oranges per letter. The children wear name tags or stickers showing their letter. The oranges are mixed up so that people don't have any with their own letter on. The children hold one orange in each hand. Figure 10.1 shows a possible arrangement of children and oranges.

3. The object of the game is for the children to pass the oranges around so that each child ends up holding their corresponding oranges—that is, child A is holding both oranges labeled A, and so on. They must follow two rules:

   (a) Only one orange may be held in a hand.

   (b) An orange can only be passed to an empty hand of an immediate neighbor in the circle.

   The rules mean that for the arrangement in Figure 10.1, the only allowed move is for child C or child E to hand child D an orange. In the case of C this could be either the B or E orange.

   Children will quickly find that if they are "greedy" (hold onto their own oranges as soon as they get them) then the group might not be able to attain its goal. It may be necessary to emphasize that individuals don't "win" the game, but that the puzzle is solved when *everyone* has their oranges.

   If the children are having difficulty making progress, it may be helpful to suggest that they imagine a wall between child A and F, which is equivalent to the children being in a line rather than a circle. (One strategy that works is to sort the letters into alphabetical order simply by swapping letters that are out of order.)

4. If a group finishes early they could try one of the variations suggested below.

5. Afterwards, have the children discuss the strategies that they used to solve the problem.

Variations and extensions

Many variations of this game are possible. Some possibilities are:

1. Try the activity with a larger or smaller circle.

2. Have the children come up with variant rules.

3. Carry out the activity without any talking.

4. Try a different configuration for message passing, such as sitting in a line, or having more than two neighbors for some children (see Figure 10.2).

5. This problem can be simulated by a single person manipulating the tokens; cooperation is now automatic, but strategies can still be explored.

Discuss where children have experienced the problem of deadlock. Some examples might be a traffic jam or gridlock in a road network, getting players around bases in baseball, or trying to get a lot of people through a doorway at once.

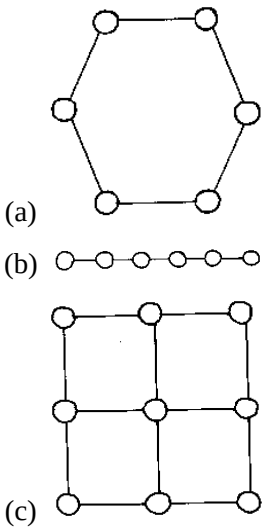


Figure 10.2: Some configurations for the orange game: (a) circle, (b) line, (c) grid. The lines show where oranges can be passed.

What's it all about?

Routing and deadlock are important problems in many different kinds of networks, such as road systems and traffic jams, telephone and computer systems. Engineers spend a lot of time figuring out how to solve these problems—and how to design networks for which the problems are easier to solve.

Routing, congestion and deadlock can present frustrating problems in many different kinds of networks. Just think of your favorite rush-hour traffic! It has happened several times in New York City that the traffic in the streets has become so congested that it deadlocks: no-one can move their car! Many times when the computers are "down" in businesses (such as banks) the problem is caused by a communication network deadlock. Designing networks so that routing is easy and efficient and congestion is minimized is a difficult problem faced by many kinds of engineers.

A classic (though somewhat dated) example of deadlock in computing is when two people on a shared computer are running a program that uses both a tape reader and a printer. Person A issues a command to read some data from a computer tape, and print it out. To do this, the computer must allocate the tape reader and the printer to person A. Suppose the tape reader gets allocated to person A, but before the printer is allocated, person B's program makes a request for the same resources, and is allocated the printer first. The two programs are now deadlocked: neither can proceed until the other releases a resource.

A more likely scenario on modern computers is when users are competing for data in a database. If a piece of data (such as a customer's bank balance) is being updated, it is important to "lock" it during the update because otherwise a simultaneous update from another source could result in the balance being recorded incorrectly. Deadlock could occur when two people

are accessing the same data in a different order. Such a situation must be prevented because when it occurs the people involved will have to wait indefinitely for a response. While it may not seem to be very likely to happen, it is a fact that if it is possible for a system to deadlock, it certainly will. This sounds like Murphy's law—that if things can go wrong they will—but it is a lot stronger than Murphy's law because it doesn't just happen by bad luck: you can *prove* that if a system that is subject to random inputs can deadlock, it surely will eventually. And things become even more complicated if multiple requests are involved in the deadlock situation.

One of the most exciting developments in computer design is the advent of parallel computing, where hundreds or thousands of PC-like processors are combined (in a network, of course) to form a single powerful computer. Many problems like the Orange Game must be played on these networks continuously (but much faster!) in order for these parallel computers to work.

Further reading

Many books about computer operating systems, such as Tanenbaum's book *Operating Systems*, have a section on deadlock. Tanenbaum's book includes the Ostrich Algorithm (ignore the problem) and the Banker's algorithm (analagous to a banker with a limited amount of money to lend in a community). This kind of problem is also discussed in Peterson's book *Petri Net Theory and the Modeling of Systems*.

Part III

Telling computers what to do—*Representing procedures*

Computers follow instructions—millions of instructions every second. To tell a computer what to do, all you have to do is give it the right instructions. But that's not as easy as it sounds! The list of instructions is called a *program*, and the language that the instructions are written in is called a *programming language*. To actually write a program you need to learn the language. And just as there are lots of different natural languages—English, French, Japanese, and so on—there are lots of different programming languages too.

The activities in this section will give you some idea of what it's like to use a programming language. We can't teach you one—that would take too long, and it would depend on the computer you had, and all the little details involved might become frustrating and boring. But the really interesting part is not learning the language; that's routine. The interesting part is having to communicate using a fixed set of instructions that are interpreted literally. And we can show you what *that* is like.

When you tell people what to do by giving instructions (or they tell you!), they use common sense to interpret what is meant. If someone says "go through that door," they don't mean to actually smash through the door—they mean go through the door*way*, if necessary opening the door first! Computers are different. Indeed, when they are attached to mobile robots you need to be careful to take safety precautions to avoid them causing damage and danger by interpreting instructions literally—by trying to go through doors. Dealing with something that obeys instructions exactly, without "thinking," takes some getting used to. That's what these activities are about.

For teachers

Programming is the very essence of computing, and writing instructions for machines is the essence of programming. The two activities in this section convey a feeling for what it is like to communicate to literal-minded machines using a fixed set of instructions. Both are suitable for middle elementary children and up. Both are group activities that can be undertaken in the classroom, or even (in the case of the first activity and part of the second) outside. There are no special prerequisites, and they can be tackled in any order.

The first activity, in the guise of a treasure hunt, introduces the idea of states and transitions between states that is fundamental to the operation of a computer. A few people giving directions act as the states, and the children must build up a map of how the overall "machine" works. The second illustrates the difficulty of giving instructions in a form that can be interpreted literally, particularly without the ready feedback and non-verbal cues that we take for granted in interpersonal communication.

Activity 11 is about something that is technically known by the rather forbidding term of *finite-state automaton*. In ordinary language, an automaton is a self-operating machine, one that works spontaneously. "Finite-state" means that the number of states the machine can be in is not infinite. With these definitions in mind the term "finite-state automaton" smacks of technical mumbo-jumbo. All machines have a finite number of states—how could it be infinite?—What does it mean to work "spontaneously"? And indeed, this is a case where taking the term apart and looking at its component words doesn't really help understanding. In computer science, a finite-state automaton is an important abstraction

of a *very* rudimentary kind of "machine": one that, in this activity, we represent by a map rather than by anything mechanical. Although in general we avoid acronyms in this book—they are the bane of modern technical writing!—in this case we revert to the commonly-used abbreviation "FSA" because it is no less informative than the phrase it stands for.

We also introduce the acronym RISC for "reduced instruction-set computer," which is explained in Activity 12. Like many others, this acronym is used, or rather misused, in adjectival form, in phrases like "RISC computer." ' Similar misuse occurs in everyday speech, For example, everyone talks about "PIN numbers" even though the letters stand for "personal identification number," and New Zealanders invariably refer to "Lake Rotorua" despite the fact that in the Maori language *roto* means *lake* and so Lake Rotorua is redundant.

So don't worry about the technical terms. The ideas that they describe are simple, and you can use the jargon to impress people!

For the technically-minded

The art of programming is something about which computer scientists wax lyrical. Fred Brooks, a renowned programming manager in IBM during its heyday in the 1950s and 60s when it seemed to be an eternal, invincible force in computing, extolled . . .

> . . . the delight of working in such a tractable medium. The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures. Yet the program construct, unlike the poet's words, is real in the sense that it moves and works, producing visible outputs separate from the construct itself. It prints results, draws pictures, produces sounds, moves arms. The magic of myth and legend has come true in our time. One types the correct incantation on a keyboard, and a display screen comes to life, showing things that never were nor could be.

Although you may be astounded to find a technical subject described in such poetic terms, any programmer will confirm that these words do indeed accurately reflect the joys of their craft (on a good day).

We cannot hope in this book to convey to children the joy of programming, to let them experience first-hand the ecstasy (and the agony!) of creating things that never were nor could be. Instead, we try merely to communicate the notion of instructions as things that are taken very literally. A wider view of the programmer's craft—the tractability of the medium, the feeling that the only limits to complexity are imposed by one's own mind, the sheer joy of making things that work—these are the kind of things that computer professionals who are visiting a classroom may be able to describe from their own experience.

Activity 11

Treasure hunt—*Finite-state automata*

Age group	Middle elementary and up.
Abilities assumed	Simple map reading.
Time	15 minutes or more.
Size of group	Around ten or more.

Focus

Maps.

Abstract representations.

Recognizing patterns.

Summary

Computers programs often need to process a sequence of symbols. The symbols might be letters or words in a document, or the text of another computer program. Computer scientists often use a simple but powerful idea called a finite-state automaton to process these symbols. A finite-state automaton involves little more than following simple instructions on a map. This activity shows how the idea works, and what it can be used for.

Technical terms

Finite-state automata; languages; parsing; compilers.

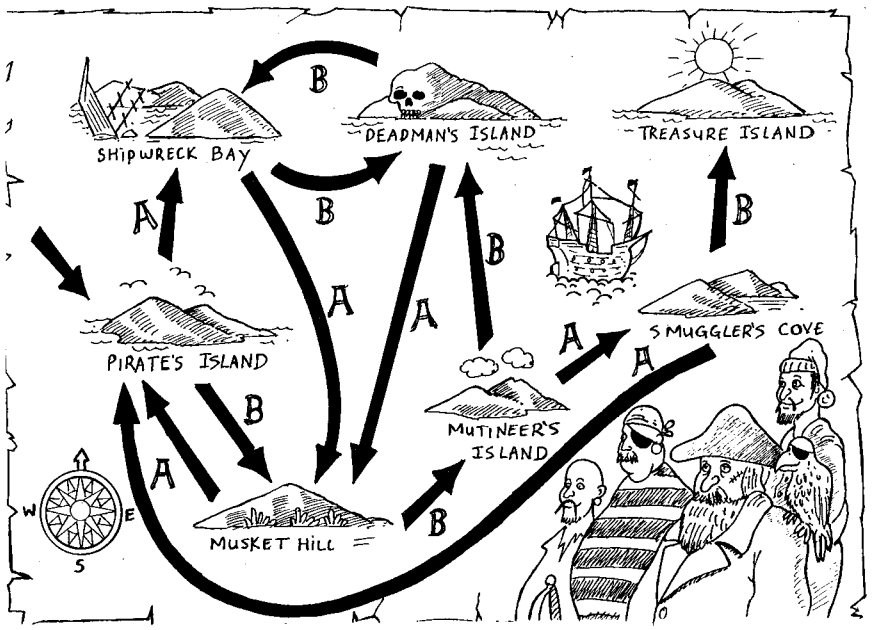


Figure 11.1: The final map

Materials

Each child will need:

- a copy of the blackline master on page 116, and
- a pen or pencil.

You will need:

- one set of cards made from the blackline master on page 117,
- cards or stickers to label the people at the islands with the names on page 116,
- prizes (such as a stamp or stickers), and
- blank paper, cards, and labels for creating new maps.

What to do

In this activity, seven people are positioned around the room (or further apart if outdoors), each representing an island. Pirate ships sail along a fixed set of routes between the islands (these are very well-organized pirates who offer a scheduled shuttle service for children). The seven

"islands" are labeled with the names on page 116, either by holding a card or wearing a sticker or badge.

Each island has two departing ships, A and B. Figure 11.1 shows a map of the routes available. Unfortunately the pirates don't let children have this map because they want to keep the route to Treasure Island secret. The goal is to find a sequence of ship rides that will get from Pirate's Island to Treasure Island. The only way to find this is for the children to "travel" between islands, until they find a route to Treasure Island. For example, the sequence of choices AABAAAB is one of many that achieves the goal for Figure 11.1.

The children start by going to the person representing Pirate's Island, and saying whether they would like to take route A or B. The people looking after the islands have the corresponding instruction card from the blackline master on page 117. They use the card to tell arriving children where their chosen route will take them. For example, a child arriving at Pirate's Island who asks to take route A will be sent to Shipwreck Bay. The child continues to move around the islands by asking for route A or B. The children start with a blank map—the blackline master on page 116—and use it to note where each route takes them as they move between the islands. Without their record of the routes, they could easily end up going around in circles!

Here is a step-by-step description of the activity.

1. First demonstrate what the children are required to do by using a small example.

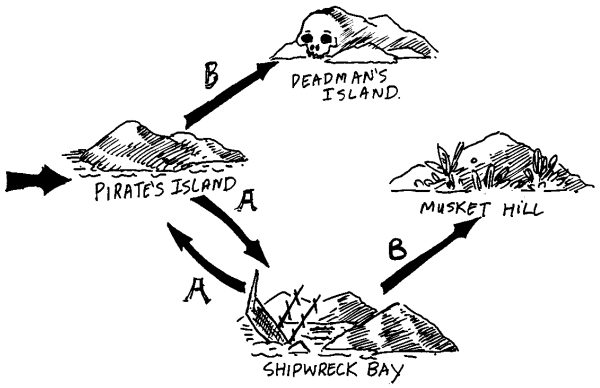
   A map with four islands, only two of which have associated directions, is shown in Figure 11.2a, and the cards for the islands are in Figure 11.2b. Choose two children (or, preferably, adult helpers) and give them the example cards and the labels *Pirate's Island* and *Shipwreck Bay*. Have them stand some distance apart. Show all the children a copy of page 116 (but do not hand it out yet) and explain that

   - it is a map of some islands, with a choice of two routes, A and B, from each one,
   - the goal is to find a route to get to Treasure Island, and
   - there is a prize at Treasure Island for getting there with a correct route marked on the map.

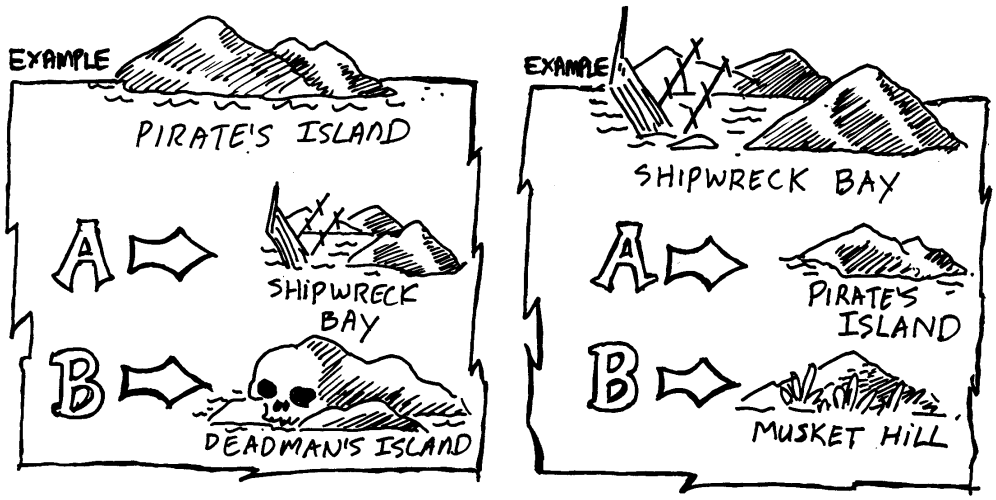
   Demonstrate the game by going to the person representing Pirate's Island and asking for route A. The person at Pirate's Island should direct you to Shipwreck Bay. Show the children how to mark this on the map by drawing an arrow from Pirate's Island to Shipwreck Bay, labeled with an A. At Shipwreck Bay, ask for route A again. You should be directed back to Pirate's Island. Mark this route on the map.

   Emphasize that they need to write down the routes as they go along—in the excitement children can forget about recording routes, and become hopelessly lost. Also, they often forget to label the route with A or B, which makes things difficult if they need to re-use the route later.

   The routes in the small example of Figure 11.2 are different from those in the main activity, so collect in the example cards to ensure that they don't cause confusion with the next map that will be used.



(a)



(b)

Figure 11.2: A small example: (a) map, (b) cards

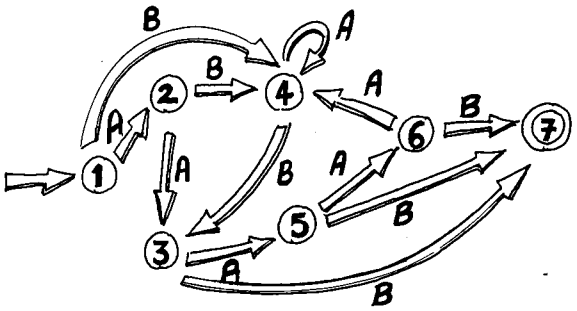


Figure 11.3: A map with a loop

2. Now it's time to set up all seven islands with a new set of routes, and have the children attempt to navigate their way to Treasure Island.

   Give out the instruction cards (from the blackline master on page 117) to the people who will be the islands, and position them around the room (or around a playground). It is safer if these people are adults, but older children will catch on quickly enough. A seventh "island" (Treasure Island) is required to check maps and administer prizes. Give out the blank maps (page 116) and send the children off one at a time to start at Pirate's Island.

   The people at the islands should check that children are writing down the routes as they move around. The person at Treasure Island needs to check that their map is correct (it needn't be complete, but it must show at least one path from Pirate's Island to Treasure Island). Correct maps receive a prize, which might be a stamp (see page 6) or sticker.

   Children who finish quickly can be asked to find an alternative route.

3. After everyone has found a route, gather the children together to discuss the routes they found to get to the treasure. A complete map (Figure 11.1) should be constructed on the classroom board, and the children can trace other routes. Point out routes that involve loops—for example, BBAAAAAB and BBAAAAAAAB both get to Treasure Island.

4. The game can be repeated with different layouts. Figure 11.3 shows an alternative way of expressing a map. The islands are shown as numbered circles, and the final island (with the treasure) has a double circle. Notice the loop from island number 4 back to itself. This may seem pointless, but we will explore uses for it later. The children moving around the islands can construct a map on a blank piece of paper this time, drawing the islands as a number in a circle.

   Children will probably enjoy designing their own layouts and creating the cards for the islands.

5. Back at the classroom board, the children can explore different maps. Figure 11.4 shows three that can be discussed. For each of the sample maps they should try to work out

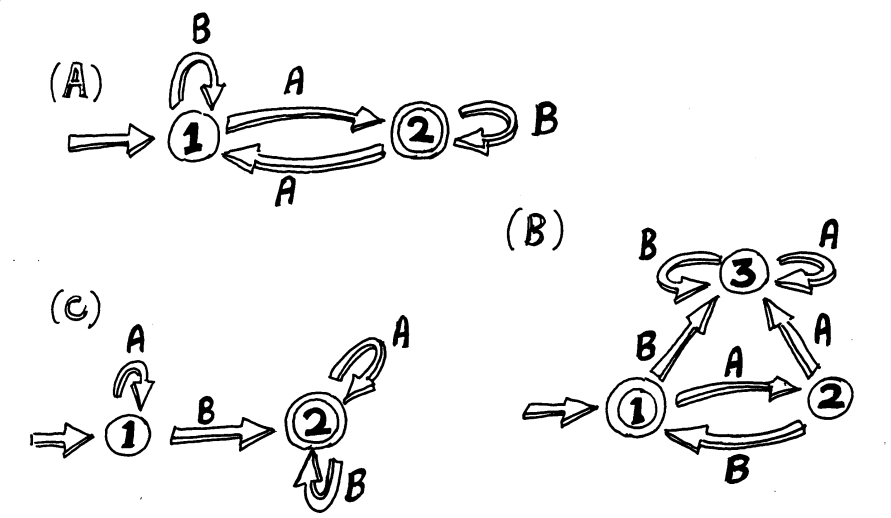


Figure 11.4: Sample maps for classroom discussion

a general description of the sequence of routes that get to the island with the double circle. The map in Figure 11.4a will finish at the double circle (island 2) only if the sequence has an odd number of A's (for example, AB, BABAA, or AAABABA). The one in Figure 11.4b only gets to the double circle with a strictly alternating sequences of A's and B's (AB, ABAB, ABABAB, . . .). The one in Figure 11.4c requires that the sequence contains at least one B (the only sequences not suitable are A, AA, AAA, AAAA, . . .).

Variations and extensions

Instead of having the children discover the map, each could be given a different sequence of A's and B's, and they could report which island they ended up at, receiving a reward if their answer is correct.

Various games and puzzles can be based on this kind of "map." For example, Figure 11.5 illustrates a way of constructing sentences by choosing random paths through the map and noting the words that are encountered.

The following puzzle has a solution that can be represented clearly by a map of the type we have been using. Figure 11.6a shows a sequence of coin tosses supposedly produced by an automatic coin-tossing machine. We want to find out if the machine is rigged. It looks like there may be a pattern in the tosses, but it is hard to see what it is. Show the sequence to the class, and see if they can identify a predictable pattern.

The pattern in Figure 11.6a can be explained using the map in Figure 11.6b. Following the sequence, the first *h* takes you from island 1 to island 2. The second *h* goes from 2 to 4. Notice that there is no route from island 2 on a *t*. This means we would be stuck if a *t* ever came up while we were at island 2. Tracing through the sequence in Figure 11.6a reveals that we never

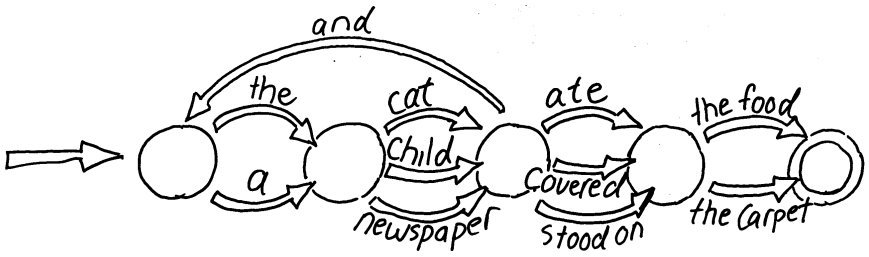
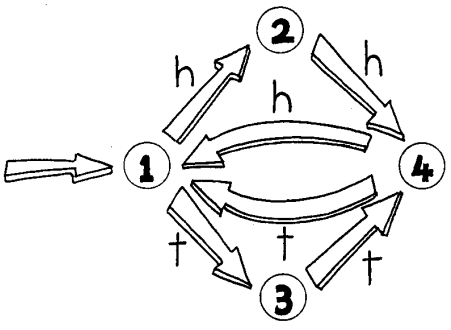


Figure 11.5: A map for generating sentences

h	h	t	h	h	t	h	h	h	t	t	h	h	h	h	t	t	h	t	t	t	h	h	h	h	t	h	h	h	t	t	t
h	h	h	t	t	t	h	h	h	h	h	h	t	t	h	t	t	t	t	t	h	t	t	h	t	t	t	h	h	h	t	t	h
h	h	t	h	h	h	h	h	h	h	h	h	t	t	h	h	h	t	t	t	t	h	h	h	h	h	t	t	t	t	t	t	t

(a)



(b)

Figure 11.6: (a) A sequence of coin-tosses; (b) a finite-state automaton that explains it

get stuck like this, and so the pattern is now revealed in the structure of the map: the first two coin tosses of every group of three will be the same. Every third toss is completely predictable! (You can see this from the sequence too: every third toss, starting with the second symbol in the sequence, is a copy of the previous one.)

What's it all about?

The "maps" used in this activity are a kind of representation that computer scientists call a *finite-state automaton*, or FSA for short. They are usually represented on paper with diagrams like the ones in Figure 11.4. We say that they are made up of *states* (the circles) and *transitions* (the arrows). The goal is to get to an *accepting state* (the double circle). An FSA begins in a specified starting state, and reads symbols from its input, one by one. After reading each symbol it moves to the next state indicated by the transition labeled with that symbol. For example, in Figure 11.3, suppose the input is ABAABB. We start in state 1, and the first A takes us to state 2. The B takes us to state 4, and the next two A's keep us in state 4. The next B takes us to state 3, and the final B takes us to state 7.

The double circle around state 7 means that we "accept" any input that leads to that state. Therefore the input ABAABB is accepted. The input ABAAAA would be rejected—it doesn't fit the definition of an "acceptable" input.

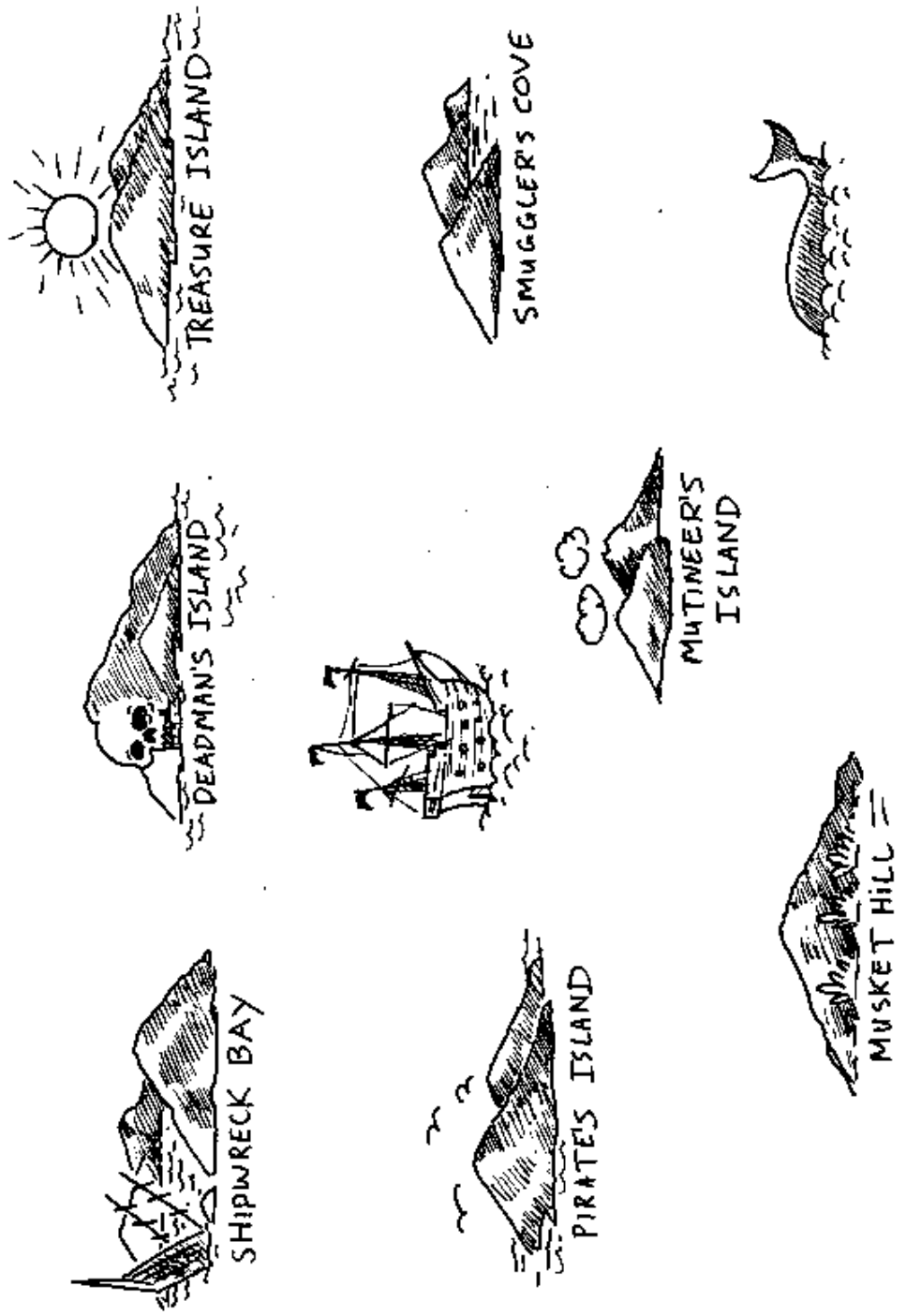
Although the examples have mainly used just A's and B's, FSAs normally work with all letters of the alphabet, as well as other characters such as digits and punctuation. They aren't usually built as real machines, but are simulated by computer programs instead. They are mainly useful as a tool for thinking about recognizing patterns in the computer's input. For example, a compiler is a program that converts programs from languages such as Pascal and C to a form that the computer can use. Compilers use finite-state automata to recognize parts of the language—such as numbers and keywords—in the input.

Finite-state automata are also used to explain or generate output. Suppose you want to get a computer to imitate the way that people can learn the structure of sequences of symbols from just a few examples. You might instruct the computer to construct a finite-state automaton that represents the patterns in the input, and can generate more output symbols that fit the pattern. The coin-tossing puzzle was an example of this: the automaton in 11.6b was generated by feeding the sequence of Figure 11.6a into a program that looks for small FSAs that could generate the sequence. This is a simple form of learning: the computer has been fed some raw information, the example sequence, and has generated a concise description or "explanation" for it that could be used to predict more symbols in the sequence. Finite-state automata are used in data compression to make predictions of what character will appear next in the input so that they don't have to be transmitted explicitly. They are also useful in human–computer interaction, where the transitions represent a series of interactions between a person and a computer.

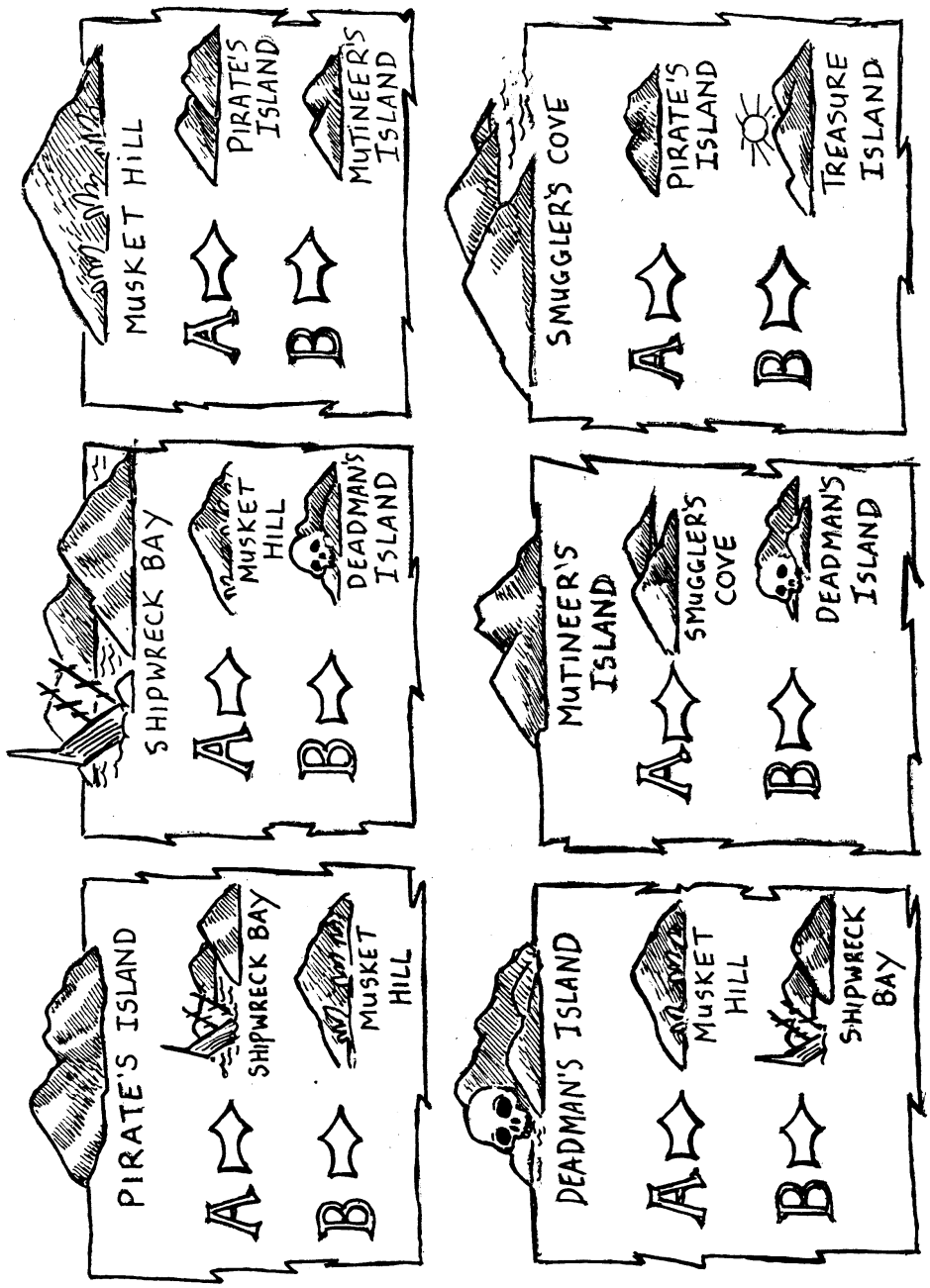
FSAs aren't the only sort of abstract machines that computer scientists use. Other models are used for different purposes, with names such as *push-down automaton*, *LR parser*, and *Turing machine*. Like the FSA, each is simple to describe, and represents a new way of approaching problems that enable us to deal with their complexity a lot more easily.

Further reading

Harel's *Algorithmics* and Dewdney's *Turing Omnibus* both have sections on finite-state automata. The rigged coin-tossing machine is introduced in the book *Thinking with a teachable machine* by John Andreae. A thorough technical treatment of the area is given by Hopcroft and Ullman in *Introduction to Automata Theory, Languages, and Computation*.



Instructions: *Start at Pirate's Island, and ask for either the A or B route. Write down the routes that you discover, and keep going until you find a way to get to Treasure Island.*



Instructions: *Give one of these cards to each of the people who is looking after an "island."*

Activity 12

Marching orders—*Programming languages*

Age group	Middle elementary and up.
Abilities assumed	Giving and following simple instructions.
Time	About 30 minutes or more.
Size of group	At least two people, suitable for the whole class.

Focus

Communication.

Language.

Giving and following instructions.

Summary

Computers are usually programmed using a “language,” which is a limited vocabulary of instructions that can be obeyed. One of the most frustrating things about programming is that computers always obey the instructions to the letter, even if they produce a crazy result. This activity gives children some experience with this aspect of programming.

Technical terms

Programming languages; instruction sets; RISC computing.

Materials

Each child will need:

pencil and paper, and

tape measure, yard-stick, or something similar to measure approximate distances.

You will also need:

cards with pictures such as the ones in Figure 12.1.

What to do

1. Discuss whether it would be good if people followed instructions exactly. For example, what would happen if you pointed to a closed door and said "Go through that door."

   Explain that computers work by following lists of instructions, and that they do exactly what the instructions say—even if they are incorrect (or ludicrous). This activity involves some experiments that show how difficult it is to give instructions if they are followed to the letter.

2. The first exercise involves communicating a picture. Choose a child and give them a simple image, such as one of the pictures in Figure 12.1 (for younger children, simpler images like Figure 12.1a are best). Their task is to stand in front of the class and describe the picture so that the other children can reproduce it on their own piece of paper without seeing it. The object is to see how quickly and accurately the exercise can be completed—it is not a competition between the class and the child giving the instructions. The children are allowed to ask questions to clarify the instructions. Repeat this with a few images. You should choose images whose complexity matches the children's ability.

3. Repeat the exercise, but this time do not allow the class to ask questions. The children will find this a lot more difficult, as there is no opportunity to make clarifications, and they are completely dependent on the ability of the child at the front to give clear instructions. It is best to use a simpler image for this exercise, as the children can get lost very quickly.

4. Now try the exercise with the instructing child hidden behind a screen, without allowing any questions, so that the only communication is in the form of instructions. The children will likely find that the lack of visual feedback makes the task very challenging.

   Point out that this form of communication is most like the one that computer programmers experience when writing programs. They give a set of instructions to the computer, and don't find out the effect of the instructions until afterwards. This should help the children to realize that it is very difficult to write a program that works correctly the first time. The *What's it all about?* section below gives some material for further discussion about the reliability of computer programs.

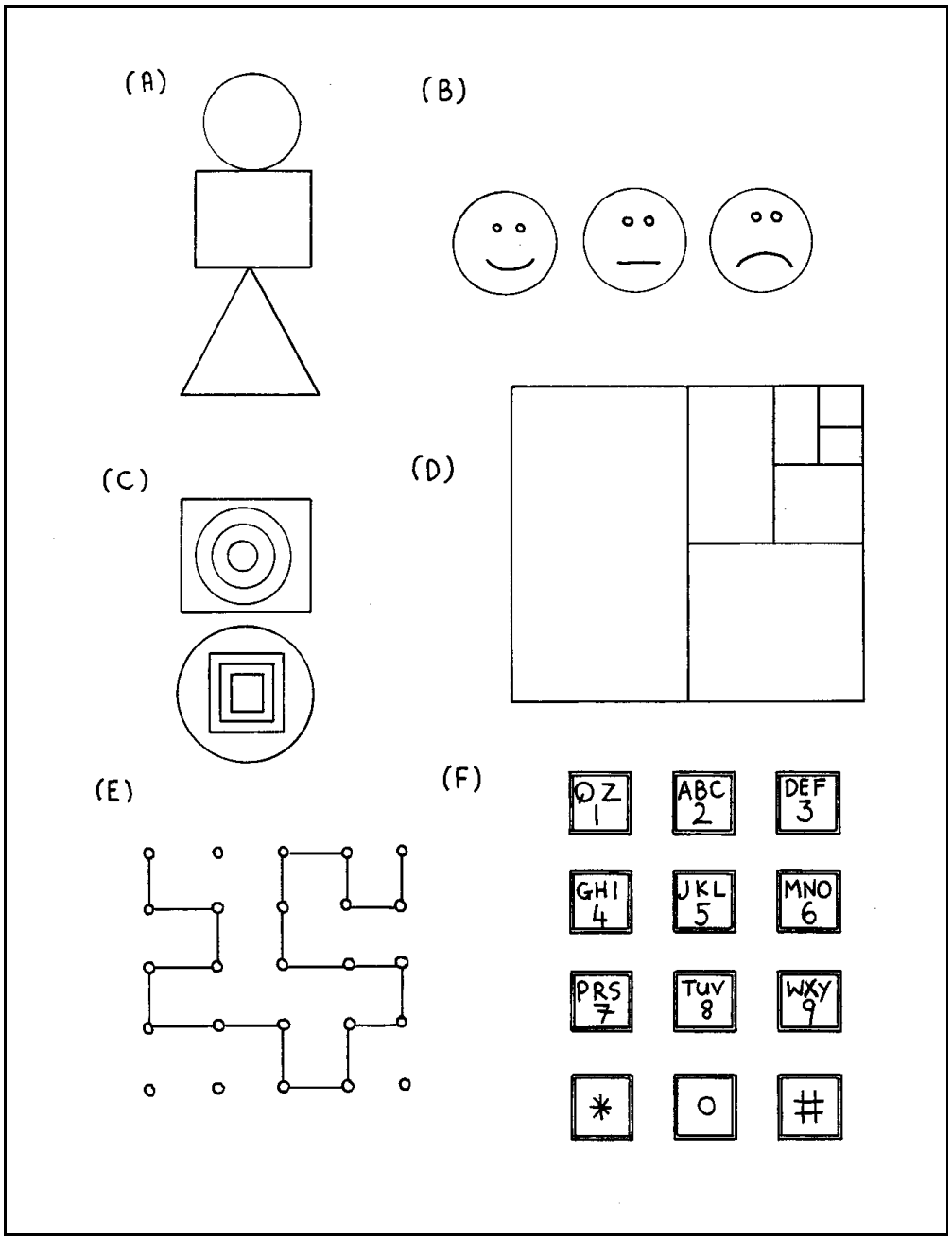


Figure 12.1: Some pictures for children to describe

5. This exercise can be extended by having the children write down instructions to carry out a given activity, such as constructing a paper dart or getting to some particular place.

   For example, they could write instructions for each other about how to get to a mystery location around the school using only instructions of the form "Go forward x meters," "turn left," (90 degrees), and "turn right" (90 degrees). Have them follow each others' instructions and evaluate how accurate they are. Children should be encouraged to refine their instructions until they have the desired effect.

   An important issue in computing is the number of different instructions that are available. For example, the "turn right" instruction above could be eliminated because the same effect can be achieved by doing "turn left" three times. Discuss the trade-off between having lots of instructions available (it's quicker to specify a route) and having only a few (it's easier to remember them, but a longer list of instructions is needed).

   Ask the children what instructions they would like to have available for navigating around the schoolyard. Can they think of places that can't be reached with a limited instruction set? (For example, if there is no instruction for going up, it may not be possible to get to a room that is upstairs.)

Variations and extensions

There are all sorts of activities that can be described by a list of instructions drawn from a small, specialized vocabulary. For example, the children could locate a book in the library using instructions like "Go up one shelf," "Point to the first book on that shelf," "Point to the next book to the right," and so on. Children can create their own limited set of instructions, and then try using them to describe an activity. The main element is that instructions must be written down and given in advance, and followed exactly.

Other examples of this kind of instruction include cake recipes and knitting patterns.

What's it all about?

Computers operate by following a list of instructions, called a *program*, that has been written to carry out a particular task—whether it be word processing, accounting, game playing, or controlling a nuclear power plant. Programs are written in languages that have been specially designed for controlling the operation of computers. Literally hundreds of different languages have been invented, with names like Ada, BASIC, C, C++, COBOL, FORTRAN, LISP, Pascal, and RPG. Each language has a limited set of instructions that provide the vocabulary with which the programmer specifies what they want the computer to do. In principle, if a task can be programmed in one language, it can be programmed in any other one as well. This is an important idea in computer science that is known as the "Church-Turing thesis." However, some languages are more suitable for some purposes than others, and hence the apparent babel of invented tongues.

Regardless of what language they use, programmers must become adept at specifying *exactly* what they want the computer to do. Unlike human beings, a computer will carry out

instructions to the letter even if they are patently ridiculous. The importance of getting the instructions right becomes clearer when one considers the consequences of an error in the program of a computer in an automated teller machine, a space shuttle launch, a nuclear power plant, or the signals on a train track. Even an error in a word processing program can ruin a software company if their program gets a reputation for losing people's work regularly. Errors like this are commonly called "bugs" in honor (so it is said) of a moth that was once removed ("debugged") from an electrical relay in an early 1940s electronic calculating machine.

It is a fact of life that large-scale computer software systems will never be bug-free. This became a major issue when the USA was working on the Strategic Defense Initiative ("Star Wars") program, a computer controlled system that was intended to form an impenetrable defense against nuclear attack. Some computer scientists claimed that it could never work because of the complexity and inherent unreliability of the software required. Software needs to be tested carefully to find as many bugs as possible, and it wouldn't be feasible to test this system since one would have to fire missiles at the United States to be sure that it worked!

There is a trade-off between having very large, complicated programming languages that make all sorts of instructions available, and small, simple ones that might have as few as only two instructions. This trade-off arose in a small way in the exercise, where the "turn right" instruction could be removed from the instruction language since the same effect could be achieved by issuing a turn left command three times. The advantage of complex languages is that the resulting programs are generally shorter, and hence easier to understand. The advantage of small languages is that they are easier to learn, and usually very efficient for computers to run. The so-called *reduced instruction set computers* (or RISC computers) use very small instruction sets so that they can run very quickly, in contrast to *complex instruction set computers* (CISC). Fortunately humans rarely have to program in the cut-down RISC languages, because systems are available (called compilers) that convert a program from a language that is easy for a human to read into the RISC language which runs very efficiently on the computer.

RISC computers generally run faster because they can be constructed to be "lean and mean," and are now generally favored over traditional CISC machines. However, people tend to use programming languages that are richer and result in shorter, more elegant programs, that are compiled automatically into a form that the RISC machine can use.

Further reading

Harel discusses programming languages in Chapter 3 of *Algorithmics*.

Part IV

Really hard problems—*Intractability*

Are there problems that are too hard even for computers? Yes. We will see in Activity 20 that just having a conversation—chatting—is something computers can't do, not because they can't speak but because they can't understand or think of sensible things to say. But really it's not that they can't do it, more that we don't know just how we do it ourselves and so we can't tell the computer what to do. In this section we're going to look at problems where it's easy to tell the computer what to do—by writing a program—but the computer can't do what we want because it takes far too long: millions of centuries, perhaps. Not much good buying a faster computer: if it were a hundred times faster it would still take millions of years; even one a million times faster would take hundreds of years. That's what you call a *hard* problem—one where it takes far longer than the lifetime of the fastest computer imaginable to come up with a solution!

The activities in Part II on algorithms showed you how to find ways of making computer programs run more efficiently. In this section we look at problems for which *no* efficient solutions are known, problems that take computers millions of centuries to solve. And we will encounter what is surely the greatest mystery in computer science today: that *no-one knows* whether there's a more efficient way of solving these problems! It may be just that no-one has come up with a good way yet, or it may be that there is no good way. We don't know which. And that's not all. There are thousands of problems that, although they look completely different, are equivalent in the sense that if an efficient method is found to solve one, it can be converted into an efficient method to solve them all. In these activities you will learn about these problems.

For teachers

There are three activities in this section. The first involves coloring maps and counting how many colors are needed to make neighboring countries different. The second requires the ability to use a simple street map, and involves placing ice-cream vans at street corners so that nobody has to go too far to get an ice-cream. The third is an outdoor activity that uses string and pegs to explore how to make short networks connecting a set of points.

The activities provide a hands-on appreciation of the idea of complexity—how problems that are very simple to state can turn out to be incredibly hard to solve. And these problems are not abstruse. They are practical questions that arise in everyday activities such as mapping, school time-tabling, and road building. The computational underpinning rests on a notion called "NP-completeness" that is explained in the *What's it all about?* sections at the end of each activity. Although the activities themselves can be tackled in any order, these sections are intended to be read in the order in which they appear. By the time you reach the end you will have a firm grip on the most important open question in contemporary computer science.

The technical name for this part is "intractability" because problems that are hard to solve are called *intractable.* The word comes the Latin *tractare* meaning to draw or drag, leading to the modern usage of *tractable* as easy to handle, pliant, or docile. Intractable problems are ones that are not easily dealt with because it would take too long to come up with an answer. Although it may sound esoteric, intractability is of great practical interest because a breakthrough in this area would have major ramifications for many different lines of research. For example, most cryptographic codes rely on the intractability

of some problems, and a criminal who managed to come up with an efficient solution could have a field day decoding secrets and selling them, or—more simply—just making phoney bank transactions. We will look at these things in Part V—Cryptography.

For the technically-minded

To make precise the idea of really hard problems, we begin with a measure of the size of a problem, and work out how the amount of computation required to solve it grows as the problem size increases. For example, searching for an item in an unordered list—perhaps searching through a city telephone directory for the person whose telephone number is 982–1735—is an operation whose time, on average, grows in direct proportion to the size of the list. This is known as a *linear-time* algorithm. Searching an ordered list—perhaps searching the directory for the number of a particular person—can be done using the binary search strategy (Activity 6) in a time which, on average, grows logarithmically with the size of the list, a *logarithmic-time* algorithm. Sorting the list into order in the first place using the selection sort method (Activity 7) takes time that grows with the square of the size of the list, a *quadratic-time* algorithm. (Insertion and bubble sorts are also quadratic-time algorithms.) Sorting using the quicksort and mergesort methods (also in Activity 7) actually takes average time proportional to $n \log n$ for a list of length n—which is a lot faster than a quadratic or n^2 algorithm.

For some problems we have a lot of trouble finding fast algorithms. How bad can it get? Well, there are algorithms that take cubic time, quartic time, indeed n^q time for any given value of q. These are all called "polynomial-time" algorithms because the time taken grows polynomially with the problem size. Polynomial growth may sound bad, especially for high-order polynomials, but it can get much worse than that. Exponential-time algorithms are ones whose execution time grows with 2^n (or indeed q^n) for a problem of size n. The problem size is an exponent, which means that once n becomes large enough, the problem outgrows any polynomial function, whatever the power.

What we encounter in these activities are problems for which the only algorithms known are exponential ones. The great unsolved question of computer science is that no-one knows whether polynomial-time algorithms exist for these problems.

Activity 13

The poor cartographer—*Graph coloring*

Age group	Early elementary and up.
Abilities assumed	Coloring in.
Time	30 minutes or more.
Size of group	Suitable for individuals to the whole class.

Focus

Problem solving.

Logical reasoning.

Algorithmic procedures and complexity.

Communication of insights.

Summary

Many optimization problems involve situations where certain events cannot occur at the same time, or where certain members of a set of objects cannot be adjacent. For example, anyone who has tried to time-table classes or meetings will have encountered the problem of satisfying the constraints on all the people involved. Many of these difficulties are crystallized in the map coloring problem, in which colors must be chosen for countries on a map in a way that makes bordering countries different colors. This activity is about that problem.

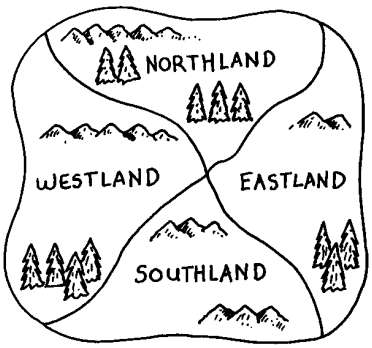


Figure 13.1: A sample map to be colored

Technical terms

Graph coloring; exponential time algorithms; heuristics.

Materials

For each child you will need:

a copy of the blackline masters on pages 138 to 141,

movable small colored markers (e.g. counters or poker chips), and

four crayons of different colors (or colored pencils, felt tips etc.)

You will also need:

a blackboard or similar writing surface.

What to do

This activity revolves around a story in which the children have been asked to help out a cartographer, or map-maker, who is coloring in the countries on a map. It doesn't matter which color a country is, so long as it's different to all bordering countries. For example, Figure 13.1 shows four countries. If we color Northland red, then Westland and Eastland cannot be red, since their border with Northland would be hard to see. We could color Westland green, and it is also acceptable to color Eastland green because it does not share a border with Westland. (If two countries meet only at a single point, they do not count as sharing a border and hence can be made the same color.) Southland can be colored red, and we end up needing only two colors for the map.

In our story, the cartographer is poor and can't afford many crayons, so the idea is to use as few colors as possible.

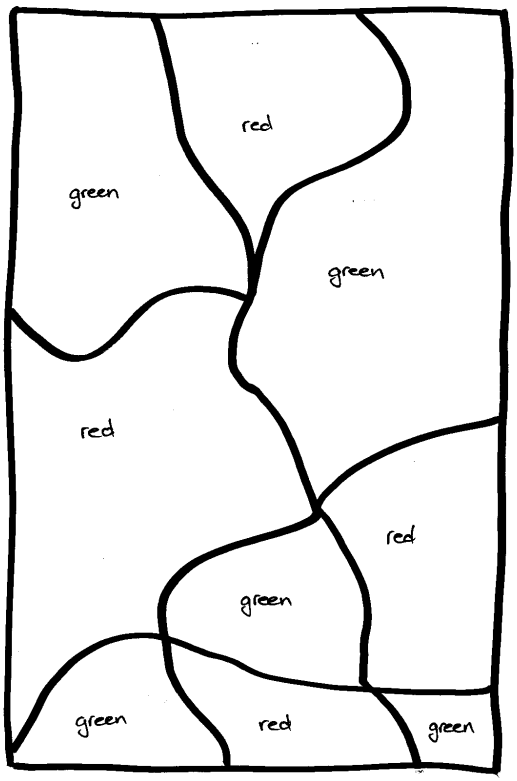


Figure 13.2: Solution for coloring the map on page 138 using just two colors.

1. Describe the problem that the children will be working on, demonstrating the coloring process on a blackboard.

2. Give out the blackline master on page 138. This map can be colored correctly using only two colors. Although restricting the number of colors to just two might sound particularly challenging, the task is quite simple compared with maps that require more colors because there is very little choice about what color each country can be.

   Have the children try to color the map in with only two colors. In the process they may discover the "has-to-be" rule: once one country is colored in, any bordering country has to be the opposite color. This rule is applied repeatedly until all countries are colored in. It is best if the children can discover this rule for themselves, rather than being told it, as it will give them a better insight into the process. Figure 13.2 shows the only possible solution for the map on page 138 (of course, the choice of colors is up to the child, but only two different ones are required).

   The children may also discover that it is better to use place-holders, such as colored counters, instead of coloring the countries straight away, since this makes it easier for

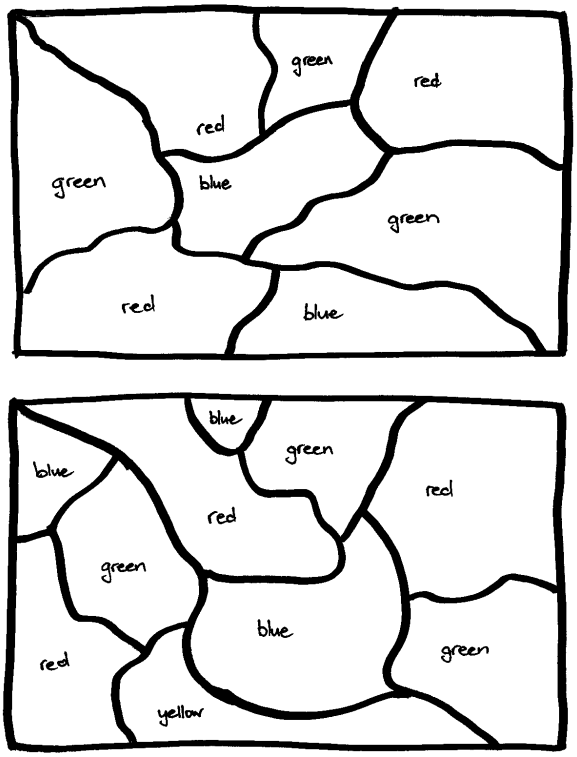


Figure 13.3: Solutions for coloring the maps on page 139 using just three and four colors respectively.

them to change their mind.

3. As children complete each exercise they can be given the next sheet to try. The map at the top of page 139 can be colored correctly using three colors, while the one at the bottom requires four. Possible solutions are shown in Figure 13.3. The map on page 140 is a simpler three-color map, with a possible solution shown in Figure 13.4.

   For older children, ask them to explain how they know that they have found the minimum number of colors. For example, at least three colors are required for the map in Figure 13.4 because the map includes a group of three countries (the largest three), each of which has borders with the other two.

4. If a child finishes all the sheets early, ask them to try to devise a map that requires five different colors. It has been proved that *any* map can be colored with only four colors, so this task will keep them occupied for some time! In our experience children will quickly find maps that they believe require five colors, but of course it is always possible to find a

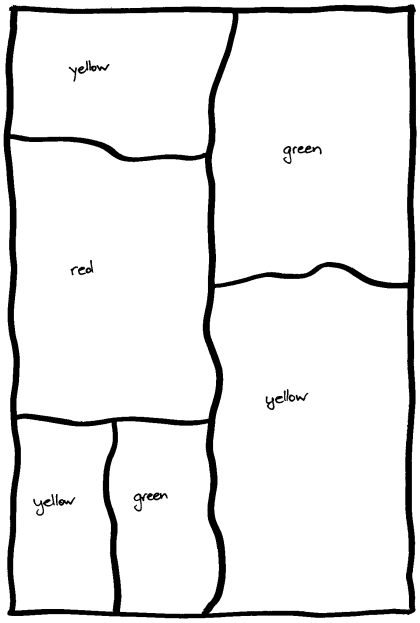


Figure 13.4: Solution for coloring the map on page 140 using just three colors.

four-color solution to their maps.

Variations and extensions

There is a simple way to construct maps that require only two colors, like the one on page 141 (solution in Figure 13.5). This map was drawn by overlaying closed curves (lines whose beginning joins up with their end). You can draw any number of these curves, of any shape, on top of each other, and you will always end up with a map that can be colored with two colors. Children can experiment with creating this type of map.

Four colors are always enough to color a map drawn on a sheet of paper or on a sphere (that is, a globe). One might wonder (as scientists are paid to do) how many colors are needed for maps drawn on weirder surfaces, such as the torus (the shape of a donut). In this case, one might need five colors, and five is always enough. Children might like to experiment with this.

There are many other entertaining variations on the map-coloring problem that lead off into directions where much is currently unknown. For example, if I am coloring a map on a sheet of paper by myself, then I know that if I work cleverly, four colors will be enough. But suppose that instead of working alone I am working with an incompetent (or even adversarial) partner. Assume that I work cleverly, while my partner only works “legally” as we take turns coloring countries on the map. How many crayons need to be on the table in order for me in my cleverness to be able to make up for my partner’s legal but not very bright (or even subversive) moves? Presently we know that 33 crayons will always be enough, but we don’t know that this

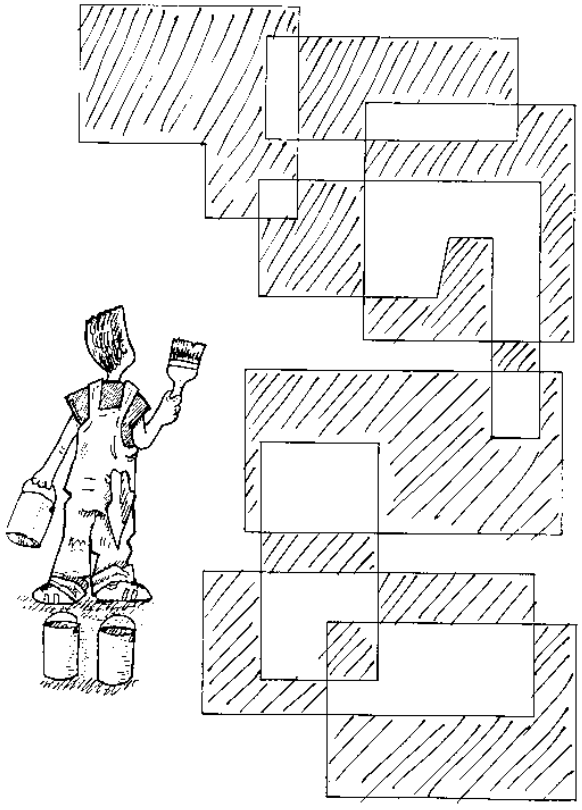


Figure 13.5: Solution for coloring the map on page 141 using just two colors. The colors are shown in this figure as shaded and white.

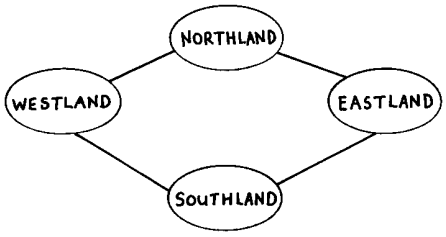


Figure 13.6: An example of a graph

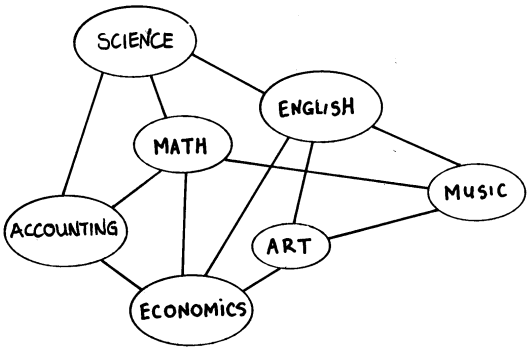


Figure 13.7: Another graph

many is ever actually required. (Experts conjecture that fewer than 10 colors are sufficient.) Children might enjoy acting out this situation, which is rather like a two-person game.

In a variation know as empire coloring, we start with two different maps on two sheets of paper having the same number of countries. Each country on one of the maps (say, the Earth) is paired with exactly one country on the other map (which might be colonies on the Moon). In addition to the usual coloring requirement of different colors for countries that share a border (for both maps) we add the requirement that each Earth country must be colored the same as its colony on the Moon. How many colors do we need for this problem? The answer is currently unknown.

What's it all about?

The map coloring problem that we have explored in this activity is essentially to find the minimum number of colors—two, three, or four—that are necessary to color a particular map. The conjecture that any map can be colored using only four colors was formulated in 1852, but it was not proved until 1976. Computer science is full of unsolved problems, and knowing that the four-color theorem was proved after more than 120 years of attention from researchers is an encouragement to those working on other problems whose solution has eluded them for decades.

Map coloring belongs to a general class of problems known as "graph coloring." In computer science, a graph is an abstract representation of relationships. An example is shown in Figure 13.6.

As mentioned in Activity 9 on the Muddy City, the term *graph* is used in a different sense in mathematics to mean a chart displaying numerical data, such as a bar graph, but the graphs that computer scientists use are not related to these. In computer science, graphs are drawn using circles or large dots, technically called "nodes," to denote objects, with lines between them to indicate some sort of relationship between the objects. The graph in Figure 13.6 happens to represent the map of Figure 13.1. The nodes represent the countries, and a line between two nodes indicates that those two countries share a common border. On the graph, the coloring rule is that no connected nodes should be allocated the same color. Unlike a map, there is no limit to the number of colors that a general graph may require, because many different constraints may be drawn in as connecting lines, whereas the two-dimensional nature of maps restricts the possible arrangements. The "graph coloring problem" is to find the minimum number of colors that are needed for a particular graph.

Figure 13.7 shows another graph. Here the nodes correspond to subjects in a school. A line between two subjects indicates that at least one student is taking both subjects, and so they should not be timetabled for the same period. Using this representation, the problem of finding a workable timetable using the minimum number of periods is equivalent to the coloring problem, where the different colors correspond to different periods. Graph coloring algorithms are of great interest in computer science, and are used for many real-world problems, although they are probably never used to color in maps!—our poor cartographer is just a fiction.

There are literally thousands of other problems based on graphs. Some are described elsewhere in this book, such as the minimal spanning tree of Activity 9 and the dominating sets of Activity 14. Graphs are a very general way of representing data and can be used to represent all sorts of situations, such as routes available for travel between cities, connections between atoms in a molecule, paths that messages can take through a computer network, connections between components on a printed circuit board, and relationships between the tasks required to carry out a large project. For this reason, problems involving graph representations have long fascinated computer scientists.

Many of these problems are very difficult—not difficult conceptually, but difficult because they take a long time to solve. For example, to determine the most efficient solution for a graph coloring problem of moderate size—such as finding the best way to timetable a school with thirty teachers and 800 students—would take years, even centuries, for a computer using the best known algorithm. The problem would be irrelevant by the time the solution was found—and that's assuming the computer doesn't break down or wear out before it finishes! Such problems are only solved in practice because we are content to work with sub-optimal, but still very good, solutions. If we were to insist on being able to guarantee that the solution found was the very best one, the problem would be completely intractable.

The amount of computer time needed to solve coloring problems increases exponentially with the size of the graph. Consider the map coloring problem. It can be solved by trying out all possible ways to color the map. We know that at most four colors are required, so we need to evaluate every combination of assigning the four colors to the countries. If there are n countries, there are 4^n combinations. This number grows very rapidly: every country that is

added multiplies the number of combinations by four, and hence quadruples the solution time. Even if a computer were invented that could solve the problem for, say, fifty countries in just one hour, adding one more country would require four hours, and we would only need to add ten more countries to make the computer take over a year to find the solution. This kind of problem won't go away just because we keep inventing faster and faster computers!

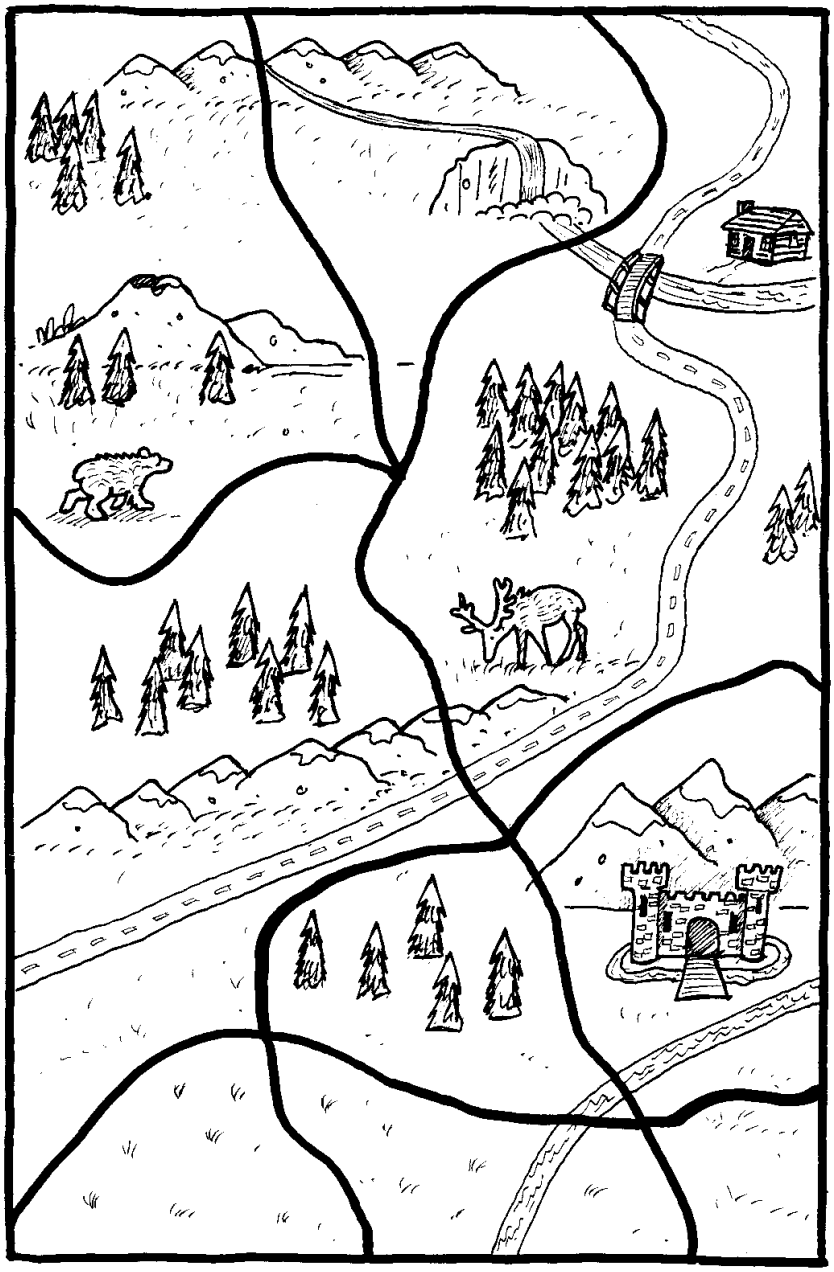
Graph coloring is a good example of a problem whose solution time grows exponentially. For very simple instances of the problem, such as the small maps used in this activity, it is quite easy to find the optimal solution, but as soon as the number of countries increases beyond about ten, the problem becomes very difficult to do by hand, and with a hundred or more countries, even a computer would take many years to try out all the possible ways of coloring the map in order to choose the optimal one.

Many real-life problems are like this, but must be solved anyway. Applied computer scientists use methods that give good, but not perfect, answers. These *heuristic* techniques are often very close to optimal, very fast to compute, and give answers that are close enough for all practical purposes. Schools can tolerate using one more classroom than would be needed if the timetable were perfect, and perhaps the poor cartographer could afford an extra color even though it is not strictly necessary.

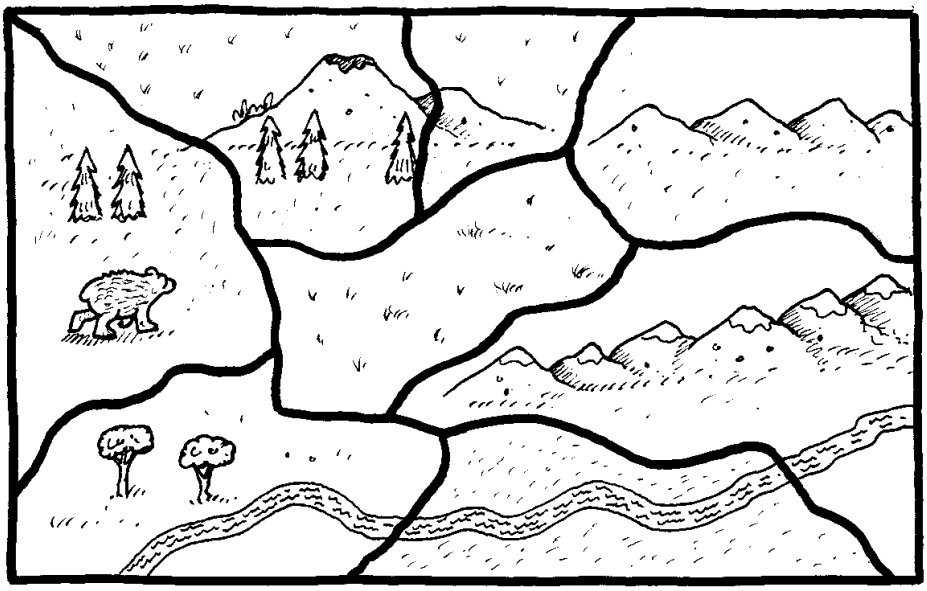
No-one has proved that there isn't an efficient way to solve this sort of problem on conventional computers, but neither has anyone proved that there is, and computer scientists are sceptical that an efficient method will ever be found. We will learn more about this kind of problem in the next two activities.

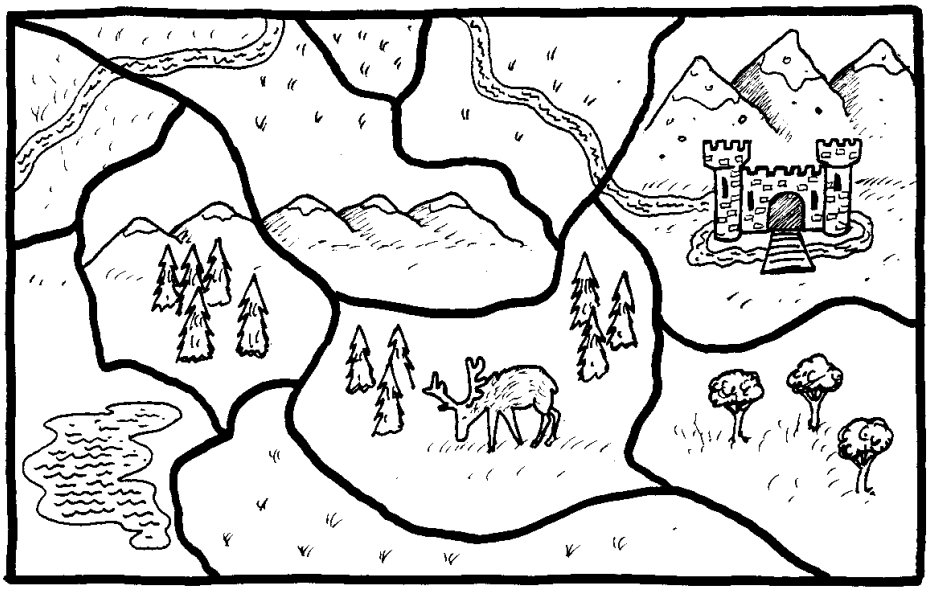
Further reading

Harel discusses the four-color theorem, including its history, in *Algorithmics*. More aspects of the map-coloring problem are discussed in *This is MEGA-Mathematics!* by Casey and Fellows. More information about graph coloring can be found in Beineke and Wilson's book *Selected Topics in Graph Theory*, and Garey and Johnson's book *Computers and Intractability*.

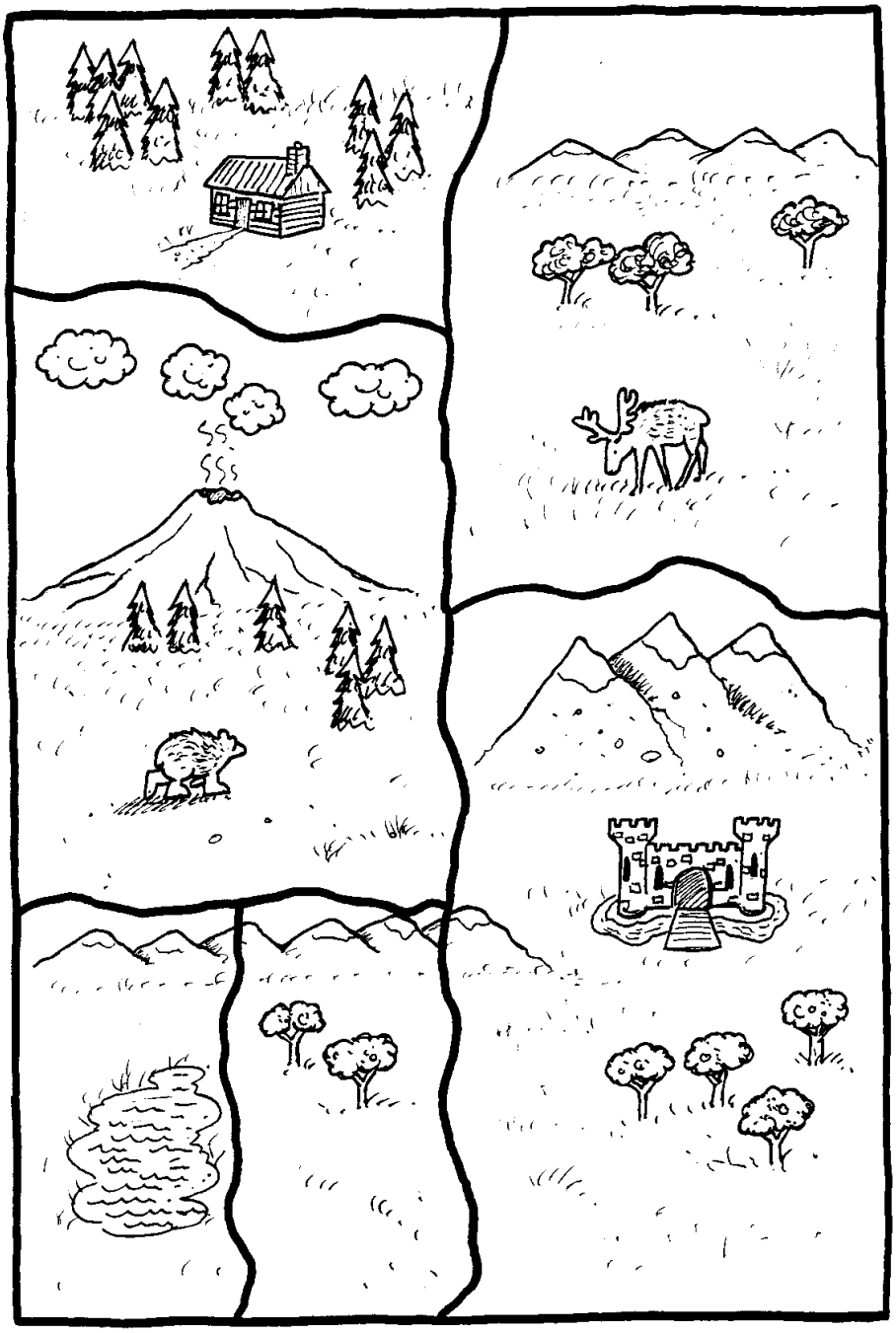


Instructions: *Color in the countries on this map with as few colors as possible, but make sure that no two bordering countries are the same color.*

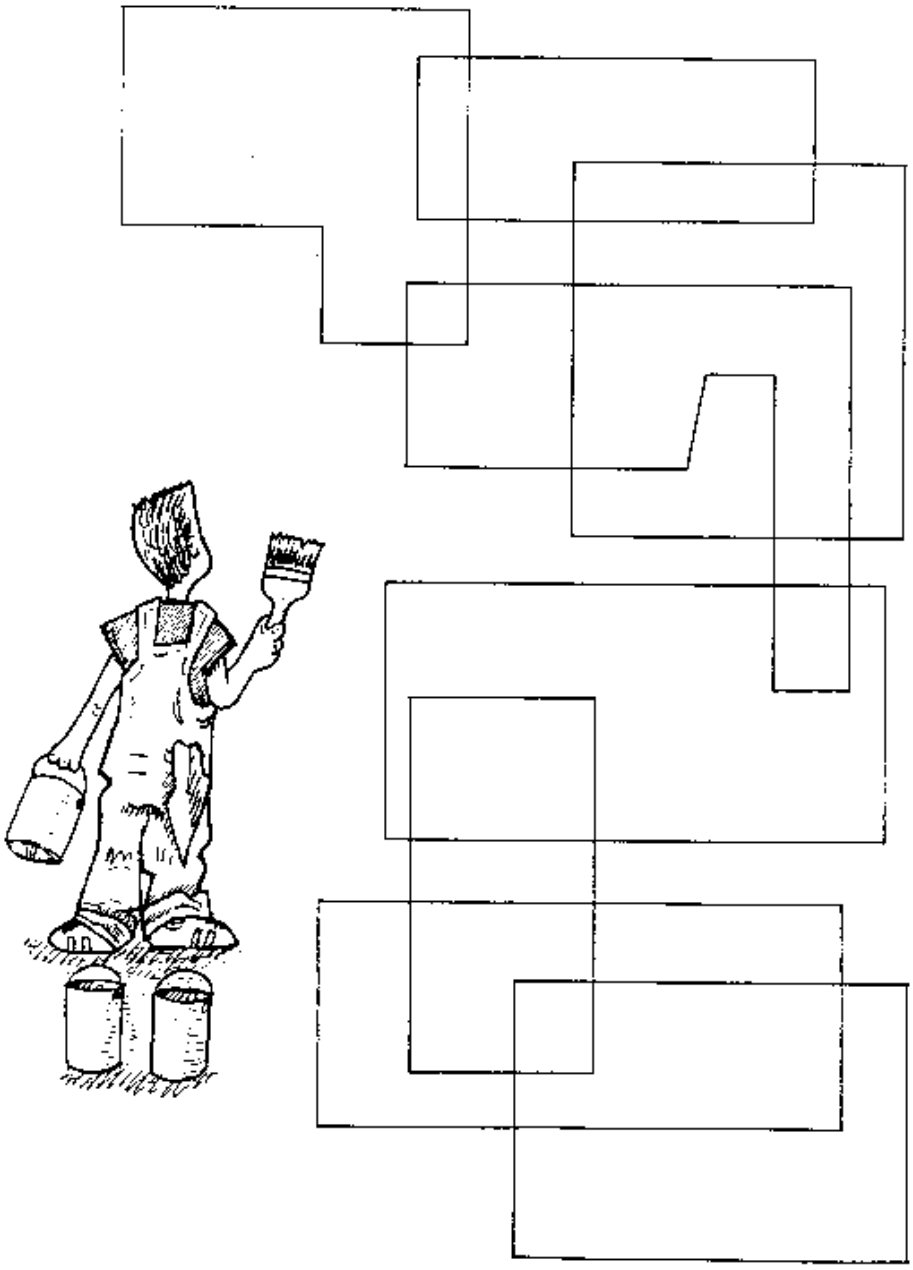




Instructions: *Color in the countries on these maps with as few colors as possible, but make sure that no two bordering countries are the same color.*



Instructions: *Color in the countries on this map with as few colors as possible, but make sure that no two bordering countries are the same color.*



Instructions: *Color in the countries on this map with as few colors as possible, but make sure that no two bordering countries are the same color.*

Activity 14

Tourist town—*Dominating sets*

Age group	Middle elementary and up.
Abilities assumed	Using a simple map.
Time	One or more sessions of about 30 minutes.
Size of group	From individuals to the whole class.

Focus

Maps.

Relationships.

Puzzle solving.

Iterative goal seeking.

Summary

Many real-life situations can be abstracted into the form of a network or "graph" of the kind used for coloring in Activity 13. Networks present many opportunities for the development of algorithms that are practically useful. In this activity, we want to mark some of the junctions, or "nodes," in such a way that all other nodes are at most one step away from one of the marked ones. The question is, how few marked nodes can we get away with? This turns out to be a surprisingly difficult problem.

Technical terms

Graphs; dominating sets; NP-complete problems; brute-force algorithms; greedy algorithms.

Materials

Each group of children will need:

a copy of the blackline master on page 149, and

several counters or poker chips of two different colors.

You will need

an overhead projector transparency of the blackline master on page 150, or a blackboard to draw it on.

What to do

Shown on page 149 is a map of Tourist Town. The lines are streets and the dots are street corners. The town lies in a very hot country, and in the summer season ice-cream vans park at street corners and sell ice-creams to tourists. We want to place the vans so that anyone can reach one by walking to the end of their street and then at most one block further. (It may be easier to imagine people living at the intersections rather than along the streets; then they must be able to get ice-cream by walking at most one block.) The question is, how many vans are needed and on which intersections should they be placed?

1. Divide the children into small groups, give each group the Tourist Town map and some counters, and explain the story.
2. Show the children how to place a counter on an intersection to mark an ice-cream van, and then place counters of another color on the intersections one street away. People living at those intersections (or along the streets that come into them) are served by this ice-cream van.
3. Have the children experiment with different positions for the vans. As they find configurations that serve all houses, remind them that vans are expensive and the idea is to have as few of them as possible. It is obvious that the conditions can be met if there are enough vans to place on all intersections—the interesting question is how few you can get away with.
4. The minimum number of vans for Tourist Town is six, and a solution is shown in Figure 14.1. But it is very difficult to find this solution! After some time, tell the class that six vans suffice and challenge them to find a way to place them. This is still quite a hard problem: many groups will eventually give up. Even a solution using eight or nine vans can be difficult to find.

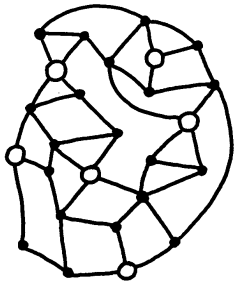


Figure 14.1: Solution to the ice-cream van problem for Tourist Town

5. The map of Tourist Town was made by starting with the six map pieces in the blackline master on page 150, each of which obviously requires only one ice-cream van, and connecting them together with lots of streets to disguise the solution. The main thing is not to put any links between the solution intersections (the open dots on page 150), but only between the extra ones (the solid dots). Show the class this using an overhead projector, or by drawing it on the board.

6. Get the children to make their own difficult maps using this strategy. They may wish to try them on their friends and parents, and will get a kick out of devising puzzles that they can solve but others can't! These are examples of what is called a "one-way function": it's easy to come up with a puzzle that is very difficult to solve—unless you're the one who created it in the first place. One-way functions play a crucial role in crpytography (see Activities 17 and 18).

Variations and extensions

There are all sorts of situations in which one might be faced with this kind of problem in town planning: locating mailboxes, wells, fire-stations, and so on. And in real life, the map won't be based on a trick that makes it easy to solve. If you really have to solve a problem like this, how would you do it?

There is a very straightforward way: consider all possible ways of placing ice-cream vans and check them to see which is best. With the 26 street corners in Tourist Town, there are 26 ways of placing one van. It's easy to check all 26 possibilities, and it's obvious that none of them satisfies the desired condition. With two vans, there are 26 places to put the first, and, whichever one is chosen for the first, there are 25 places left to put the second (obviously you wouldn't put both vans at the same intersection): $26 \times 25 = 650$ possibilities to check. Again, each check is easy, although it would be very tedious to do them all. Actually, you only need to check half of them (325), since it doesn't matter which van is which: if you've checked van 1 at intersection A and van 2 at intersection B then there's no need to check van 1 at B and van 2 at A. You could carry on checking three vans (2600 possibilities), four vans (14950 possibilities), and so on.

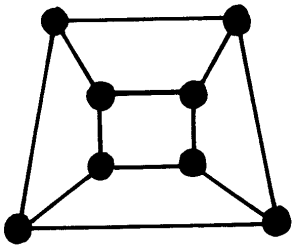


Figure 14.2: An easy town for ice-cream sellers

Clearly, 26 vans are enough since there are only 26 intersections and there's no point in having more than one van at the same place. Another way of assessing the number of possibilities is to consider the total number of configurations with 26 intersections and any number of vans. Since there are two possibilities for each street corner—it may or may not have a van—the number of configurations is 2^{26}, which is 67,108,864.

This way of solving the problem is called a "brute-force" algorithm, and it takes a long time. It's a widely held misconception that computers are so fast they can do just about anything quickly, no matter how much work it involves. But that's not true. Just how long the brute-force algorithm takes depends on how quick it is to check whether a particular configuration is a solution. To check this involves testing every intersection to find the distance of the nearest van. Suppose that an entire configuration can be tested in one second. How long does it take to test all 2^{26} possibilities for Tourist Town? (Answer: 2^{26} is about 67 million; there are 86,400 seconds in a day, so 2^{26} seconds is about 777 days, or around two years.) And suppose that instead of one second, it took just one thousandth of a second to check each particular configuration. Then the same two years would allow the computer to solve a 36-intersection town, because 2^{36} is about 1000 times 2^{26}. Even if the computer was a million times faster, so that one million configurations could be checked every second, then it would take two years to work on a 46-intersection town. And these are not very big towns! (How many intersections are there in your town?)

Since the brute-force algorithm is so slow, are there other ways to solve the problem? Well, we could try the greedy approach that was so successful in the muddy city (Activity 9). We need to think how to be greedy with ice-creams—I mean how to apply the greedy approach to the ice-cream van problem. The way to do it is by placing the first van at the intersection that connects the greatest number of streets, the second one at the next most connected intersection, and so on. However, this doesn't necessarily produce a minimum set of ice-cream van positions—in fact, the most highly connected intersection in Tourist Town, which has five streets, isn't a good place to put a van (check this with the class).

Let's look at an easier problem. Instead of being asked to find a minimum configuration, suppose you were given a configuration and asked whether it was minimal or not. In some cases, this is easy. For example, Figure 14.2 shows a much simpler map whose solution is quite

straightforward. If you imagine the streets as edges of a cube, it's clear that two ice-cream vans at diagonally opposite cube vertices are sufficient. Moreover, you should be able to convince yourself that it is not possible to solve the problem with fewer than two vans. It is much harder—though not impossible—to convince oneself that Tourist Town cannot be serviced by less than six vans. For general maps it is extremely hard to prove that a certain configuration of ice-cream vans is a minimal one.

What's it all about?

One of the interesting things about the ice-cream problem is that *no-one knows* whether there is an algorithm for finding a minimum set of locations that is significantly faster than the brute-force method! The time taken by the brute-force method grows exponentially with the number of intersections—it is called an *exponential-time* algorithm. A *polynomial-time* algorithm is one whose running time grows with the square, or the cube, or the seventeenth power, or any other power, of the number of intersections. A polynomial-time algorithm will always be faster for sufficiently large maps—even (say) a seventeenth-power algorithm—since an exponentially-growing function outweighs any polynomially-growing one once its argument becomes large enough. (For example, if you work it out, whenever n is bigger than 117 then n^{17} is smaller than 2^n). Is there a polynomial-time algorithm for finding the minimum set of locations?—no-one knows, although people have tried very hard to find one. And the same is true for the seemingly easier task of checking whether a particular set of locations is minimal: the brute-force algorithm of trying all possibilities for smaller sets of locations is exponential in the number of intersections, and polynomial-time algorithms have neither been discovered nor proved not to exist.

Does this remind you of map coloring (Activity 13)? It should. The ice-cream van question, which is officially called the "minimum dominating set" problem, is one of a large number—thousands—of problems for which it is not known whether polynomial-time algorithms exist, in domains ranging from logic, through jigsaw-like arrangement problems to map coloring, finding optimal routes on maps, and scheduling processes. Astonishingly, all of these problems have been shown to be equivalent in the sense that if a polynomial-time algorithm is found for one of them, it can be converted into a polynomial-time algorithm for all the others—you might say that they stand or fall together.

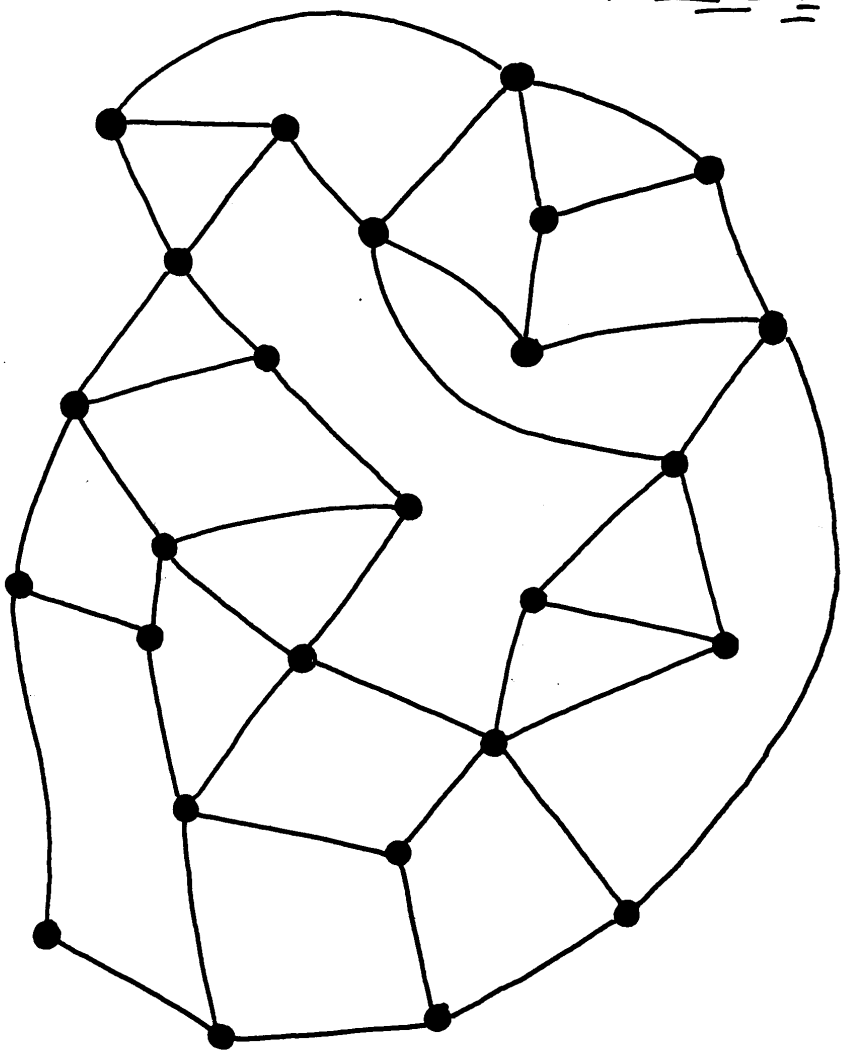
These problems are called *NP-complete*. NP stands for "non-deterministic polynomial." This jargon means that the problem could be solved in a reasonable amount of time if you had a computer that could try out an arbitrarily large number of solutions at once. You may think this is a pretty unrealistic assumption, and indeed it is. It's not possible to build this kind of computer, since it would have to be arbitrarily large! However, the concept of such a machine is important in principle, because it appears that NP-complete problems cannot be solved in a reasonable amount of time without a non-deterministic computer.

Furthermore, this group of problems is called *complete* because although the problems seem very different—for example, map-coloring is very different from placing ice-cream vans—it turns out that if an efficient way of solving one of them is found, then that method can be adapted to solve *any* of the problems. That's what we meant by "standing or falling together."

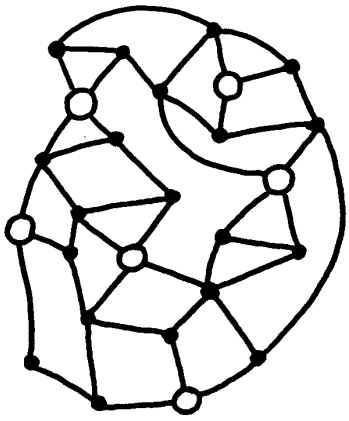
There are thousands of NP-complete problems, and researchers have been attacking them, looking for efficient solutions, for several decades without success. If an efficient solution had been discovered for just one of them, then we would have efficient solutions for them all. For this reason, it is strongly suspected that there is no efficient solution. But proving that the problems necessarily take exponential time is the most outstanding open question in theoretical computer science—possibly in all of mathematics—today.

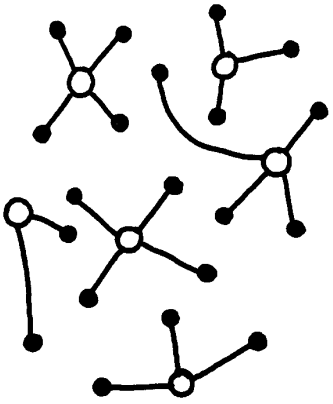
Further reading

Harel's book *Algorithmics* introduces several NP-complete problems and discusses the question of whether polynomial-time algorithms exist. Dewdney's *Turing Omnibus* also discusses NP-completeness. The standard computer science text on the subject is Garey & Johnson's *Computers and Intractability*, which introduces several hundred NP-complete problems along with techniques for proving NP-completeness. However, it is fairly heavy going and is really only suitable for the computer science specialist.



Instructions: *Work out how to place ice-cream vans on the street intersections so that every other intersection is connected to one that has a van on it.*





Instructions: *Display this to the class to show how the puzzle was constructed.*

Activity 15

Ice roads—*Steiner trees*

Age group	Middle elementary and up.
Abilities assumed	Measuring and adding up the lengths of several pieces of string. The children need to be able to tie the string to pegs.
Time	20 minutes or more. This activity requires a fine day as the children will be working outside on the grass.
Size of group	From individuals to a class working in groups of three or four.

Focus

Spatial visualization.

Geometric reasoning.

Algorithmic procedures and complexity.

Summary

Sometimes a small, seemingly insignificant, variation in the specification of a problem makes a huge difference in how hard it is to solve. This activity, like the Muddy City problem (Activity 9), is about finding short paths through networks. The difference is that here we are allowed to introduce new points into the network if that reduces the path length. The result is a far more difficult problem that is not related to the Muddy City, but is algorithmically equivalent to the cartographer's puzzle (Activity 13) and Tourist Town (Activity 14).

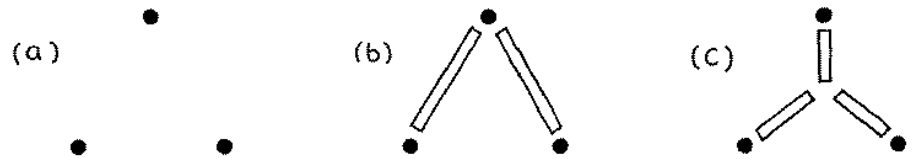


Figure 15.1: Connecting three drill sites with ice roads: (b) is ok but (c) is better

Technical terms

Steiner trees; shortest paths; NP-complete problems; minimal spanning trees; networks.

Materials

Each group of children will need

five or six pegs to place in the ground (tent pegs are good, although a coat hanger cut into pieces which are then bent over is fine),

several meters of string or elastic,

a ruler or tape measure, and

pen and paper to make notes on.

What to do

The previous activity, Tourist Town, took place in a very hot country; this one is just the opposite. In the frozen north of Canada (so the story goes), in the winter on the huge frozen lakes, snowplows make roads to connect up drill sites so that crews can visit each other. Out there in the cold they want to do a minimum of road building, and your job is to figure out where to make the roads. There are no constraints: highways can go anywhere on the snow—the lakes are frozen and covered. It's all flat.

The roads should obviously travel in straight stretches, for to introduce bends would only increase the length unnecessarily. But it's not as simple as connecting all the sites with straight lines, because adding intersections out in the frozen wastes can sometimes reduce the total road length—and it's total length that's important, not travel time from one site to another.

Figure 15.1a shows three drill sites. Connecting one of them to each of the others (as in Figure 15.1b) would make an acceptable road network. Another possibility is to make an intersection somewhere near the center of the triangle and connect it to the three sites (Figure 15.1c). And if you measure the total amount of road that has been cleared, this is indeed a better solution. The extra intersection is called a "Steiner" point after the Swiss mathematician Jacob Steiner (1796–1863), who stated the problem and was the first to notice that the total length can

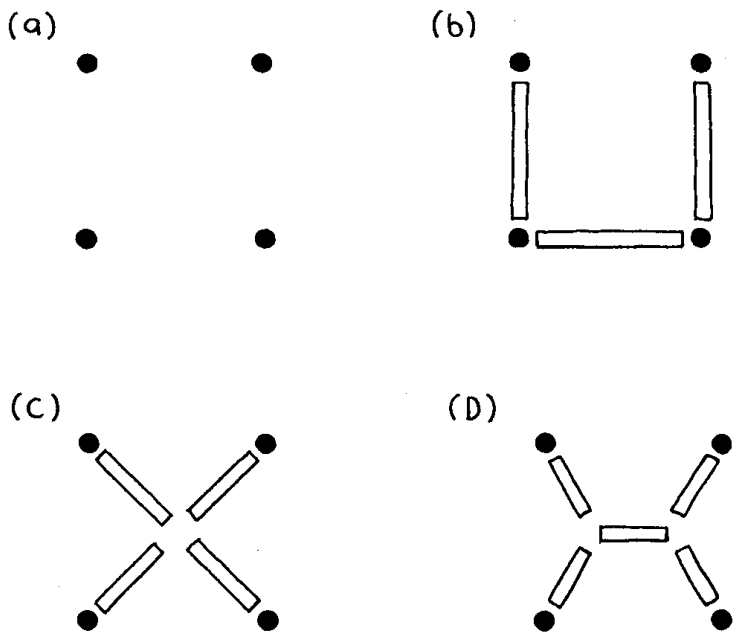


Figure 15.2: Connecting four drill sites with ice roads: (b) is ok, (c) is better and (d) better still

be reduced by introducing new points. You could think of a Steiner point as a new, fictitious, drill site.

1. Describe the problem that the children will be working on. Using the example of Figure 15.1, demonstrate to the children that with three sites, adding a new one sometimes improves the solution by reducing the amount of road-building.

2. The children will be using four points arranged in a square, as illustrated in Figure 15.2a. Go outside and get each group to place four pegs in the grass in a square about 1 meter by 1 meter.

3. Get the children to experiment with connecting the pegs with string or elastic, measuring and recording the minimum total length required. At this stage they should not use any Steiner points. (The minimum is achieved by connecting three sides of the square, as in Figure 15.2b, and the total length is 3 meters.)

4. Now see if the children can do better by using one Steiner point. (The best place is in the center of the square, Figure 15.2c. Then the total length is $2\sqrt{2} = 2.83$ meters.) Suggest that they might do even better using two Steiner points. (Indeed they can, by placing the two points as in Figure 15.2d, forming 120 degree angles between the incoming roads. The total length is then $1 + \sqrt{3} = 2.73$ meters.)

5. Can the children do better with three Steiner points? (Two points are best. No advantage is gained by using more.)

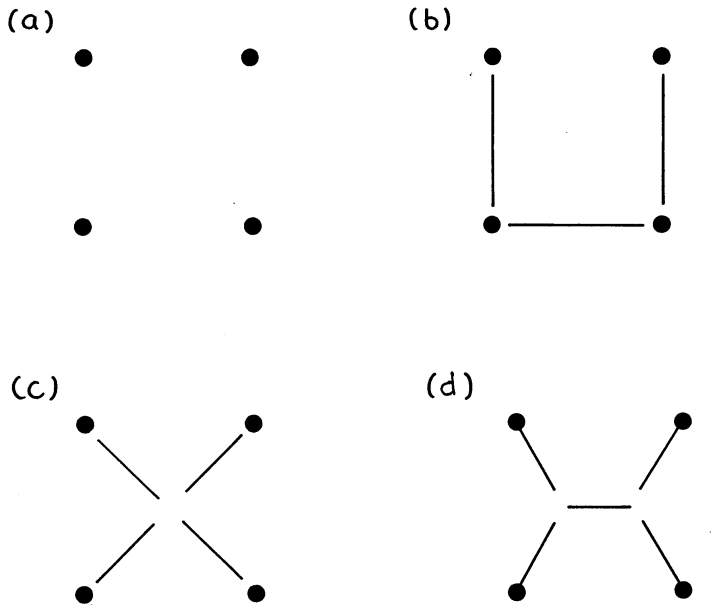


Figure 15.3: Connecting sites in a rectangle: (b) is ok, (c) not so good, and (d) the best

6. Discuss with the children why these problems seem hard. (It's because you don't know where to put the Steiner points, and there are lots of possibilities to try out.)

Variations and extensions

1. An interesting experiment for groups that finish the original activity early is to work with a rectangle about 1 meter by 2 meters (Figure 15.3). The children will find that adding one Steiner point makes things worse, but two give an improved solution. (The lengths are 4 meters for Figure 15.3b, $2\sqrt{5} = 4.47$ meters for Figure 15.3c, and $2+\sqrt{3} = 3.73$ meters for Figure 15.3d.) See if they can figure out why the one-point configuration does so much worse for rectangles than for squares. (It's because when the square of Figure 15.2 is stretched into the rectangle of Figure 15.3, the amount of stretch gets added just once into Figures 15.3b and 15.3d, but both diagonals increase in Figure 15.3c.)

2. Older children can work on a larger problem. Two layouts of sites to connect with ice roads are given in the blackline masters on pages 161 and 162. They can experiment with different solutions either using new copies of the blackline master, or by writing with removable pen on a transparency over the top of the blackline master. (The minimal solution for the first example is shown in Figure 15.4, while two possible solutions for the second, whose total length is quite similar, are given in Figure 15.5.) Figure 15.5

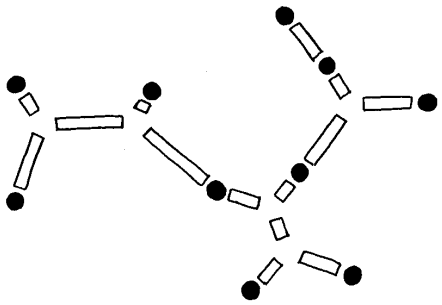


Figure 15.4: Minimal Steiner tree for the first example

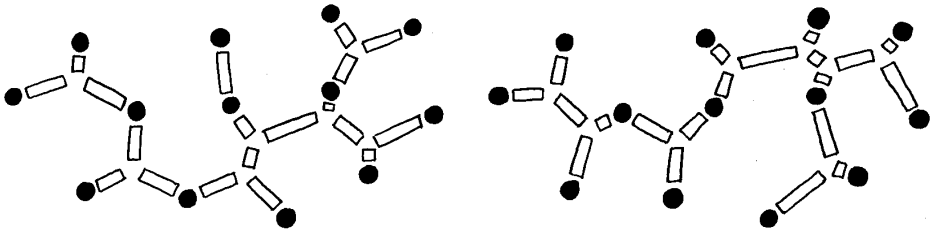


Figure 15.5: Two possible Steiner trees for the second example

illustrates why these kinds of problem are so hard—there are so many choices about where to put the Steiner points!

3. Ladder networks like that in Figure 15.6 provide another way to extend the problem. Some minimal Steiner trees are shown in Figure 15.7. The one for a two-rung ladder is just the same as for a square. However, for a three-rung ladder the solution is quite different—as you will discover if you try to draw it out again from memory! The solution for four rungs is like that for two two-rung ladders joined together, whereas for five rungs it is more like an extension of the three-rung solution. In general, the shape of the minimal Steiner tree for a ladder depends on whether it has an even or odd number of rungs. If it

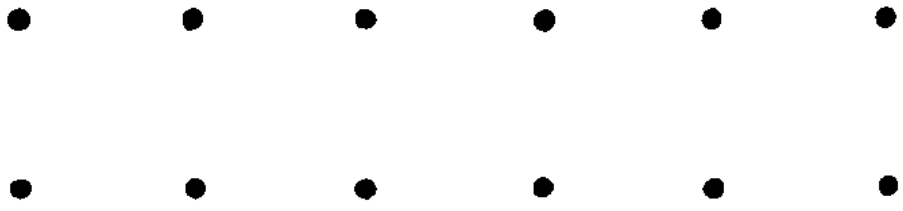


Figure 15.6: A ladder with six rungs

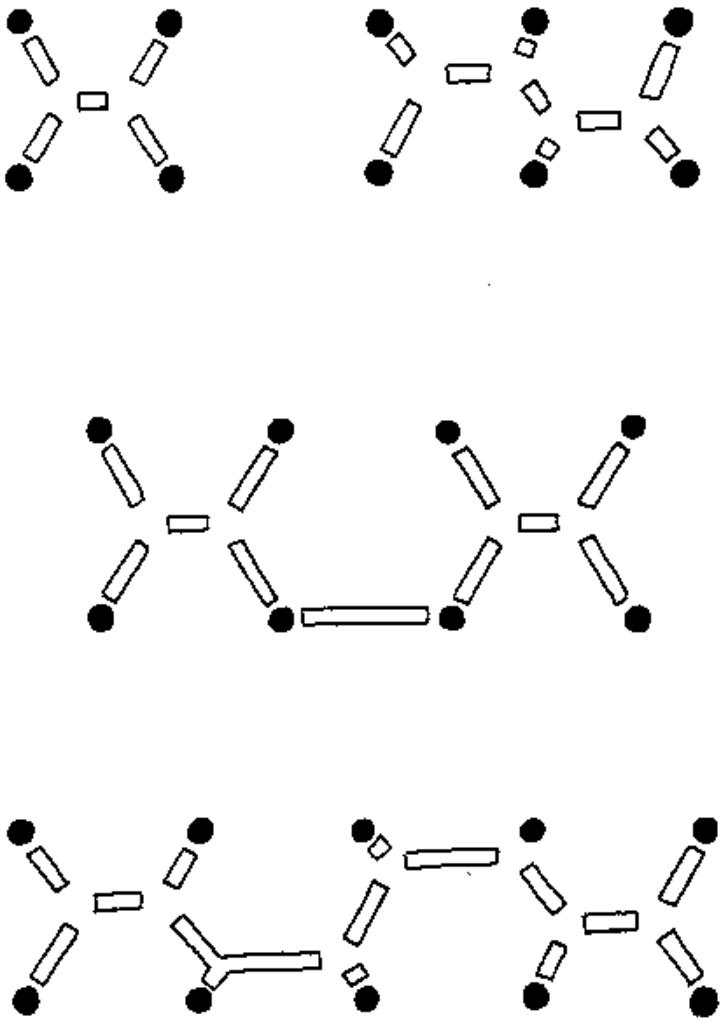


Figure 15.7: Minimal Steiner trees for ladders with two, three, four and five rungs

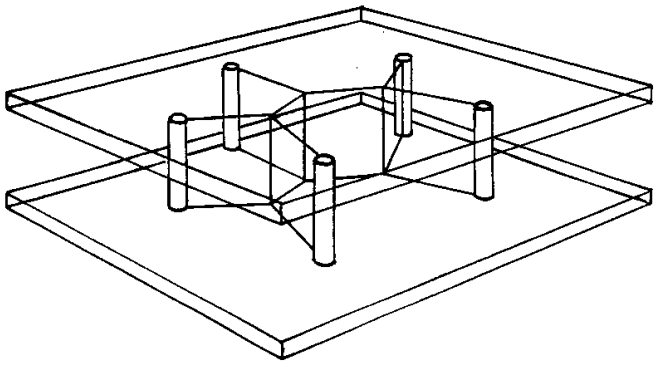


Figure 15.8: Making a Steiner tree with soap bubbles

is even, it is as though several two-rung ladders were joined together. Otherwise, it's like a repetition of the three-rung solution. But proving these things rigorously is not easy.

4. Another interesting activity is to construct soap-bubble models of Steiner trees. You can do this by taking two sheets of rigid transparent plastic and inserting pins between them to represent the sites to be spanned, as shown in Figure 15.8. Now dip the whole thing into a soap solution. When it comes out, you will find that a film of soap connects the pins in a beautiful Steiner-tree network.

   Unfortunately, however, it isn't necessarily a *minimal* Steiner tree. The soap film does find a configuration that minimizes the total length, but the minimum is only a local one, not necessarily a global one. There may be completely different ways of placing the Steiner points to give a smaller total length. For example, you can imagine the soap film looking like the first configuration in Figure 15.5 when it is withdrawn from the liquid on one occasion, and the second configuration on another.

What's it all about?

The networks that we've been working on are *minimal Steiner trees*. They're called "trees" because they have no cycles, just as the branches on a real tree grow apart but do not (normally) rejoin and grow together again. They're called "Steiner" trees because new points, Steiner points, can be added to the original sites that the trees connect. And they're called "minimal" because they have the shortest length of any tree connecting those sites. In the Muddy City (Activity 14) we learned that a network connecting a number of sites that minimizes the total length is called a minimal spanning tree: Steiner trees are just the same except that new points can be introduced.

It's interesting that while there is a very efficient algorithm for finding minimal spanning trees (Activity 14)—a greedy one that works by repeatedly connecting the two closest so-far-unconnected points—there is no general efficient solution to the minimal Steiner problem.

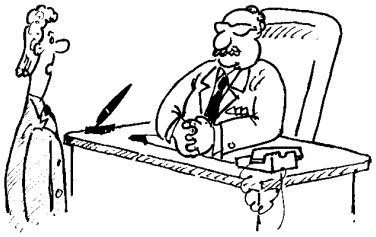
Steiner trees are much harder because you have to decide where to put the extra points. In fact, rather surprisingly, the difficult part of the Steiner tree problem is not in determining the precise location of the Steiner points, but in deciding *roughly* where to put them: the difference between the two solutions in Figure 15.5, for example. Once you know what regions to put the new points in, fine-tuning them to the optimum position is relatively simple. Soap films do that very effectively, and so can computers.

Surprisingly, finding minimal Steiner trees can save big bucks in the telephone business. When corporate customers in the US operate large private telephone networks, they lease the lines from a telephone company. The amount they are billed is not calculated on the basis of how the wires are actually used, but on the basis of the shortest network that would suffice. The reasoning is that the customer shouldn't have to pay extra just because the telephone company uses a round-about route. Originally, the algorithm that calculated how much to charge worked by determining the minimal spanning tree. However, it was noticed by a customer—an airline, in fact, with three major hubs—that if another hub was created at an intermediate point the total length of the network would be reduced. The telephone company was forced to reduce charges to what they would have been if there was a telephone exchange at the Steiner point! Although, for typical configurations, the minimal Steiner tree is only 5% or 10% shorter than the minimal spanning tree, this can be a worthwhile saving when large amounts of money are involved. The Steiner tree problem is sometimes called the "shortest network problem" because it involves finding the shortest network that connects a set of sites.

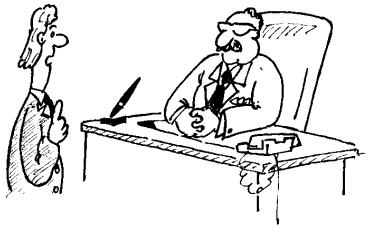
If you have tackled the two preceding activities, the cartographer's puzzle and tourist town, you will not be surprised to hear that the minimal Steiner tree problem is NP-complete. As the number of sites increases, so does the number of possible locations for Steiner points, and trying all possibilities involves an exponentially-growing search. This is another of the thousands of problems for which it simply isn't known whether exponential search is the best that can be done, or whether there is an as-yet-undiscovered polynomial-time algorithm. What is known, however, is that if a polynomial-time algorithm is found for this problem, it can be turned into a polynomial-time algorithm for graph coloring, for finding minimal dominating sets—and for all the other problems in the NP-complete class.

We explained at the end of the previous activity that the "NP" in NP-complete stands for "non-deterministic polynomial," and "complete" refers to the fact that if a polynomial-time algorithm is found for one of the NP-complete problems it can be turned into polynomial-time algorithms for all the others. The set of problems that are solvable in polynomial time is called P. So the crucial question is, do polynomial-time algorithms exist for NP-complete problems—in other words, is $\mathrm{P} = \mathrm{NP}$? The answer to this question is not known, and it is one of the great mysteries of modern computer science.

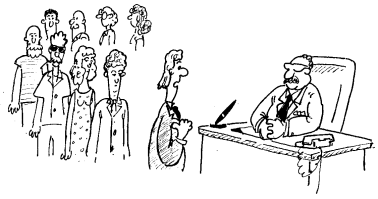
Problems for which polynomial-time algorithms exist—even though these algorithms might be quite slow—are called "tractable." Problems for which they do not are called "intractable," because no matter how fast your computer, or how many computers you use together, a small increase in problem size will mean that they can't possibly be solved in practice. It is not known whether the NP-complete problems—which include the cartographer's puzzle, tourist town, and ice roads—are tractable or not. But most computer scientists are pessimistic that a polynomial-time algorithm for NP-complete problems will ever be found, and so proving that a problem is NP-complete is regarded as strong evidence that the problem is inherently intractable.



"I can't find an efficient algorithm, I guess I'm just too dumb."



"I can't find an efficient algorithm, because no such algorithm is possible."



"I can't find an efficient algorithm, but neither can all these famous people."

Figure 15.9: What to do when you can't find an efficient algorithm: three possibilities

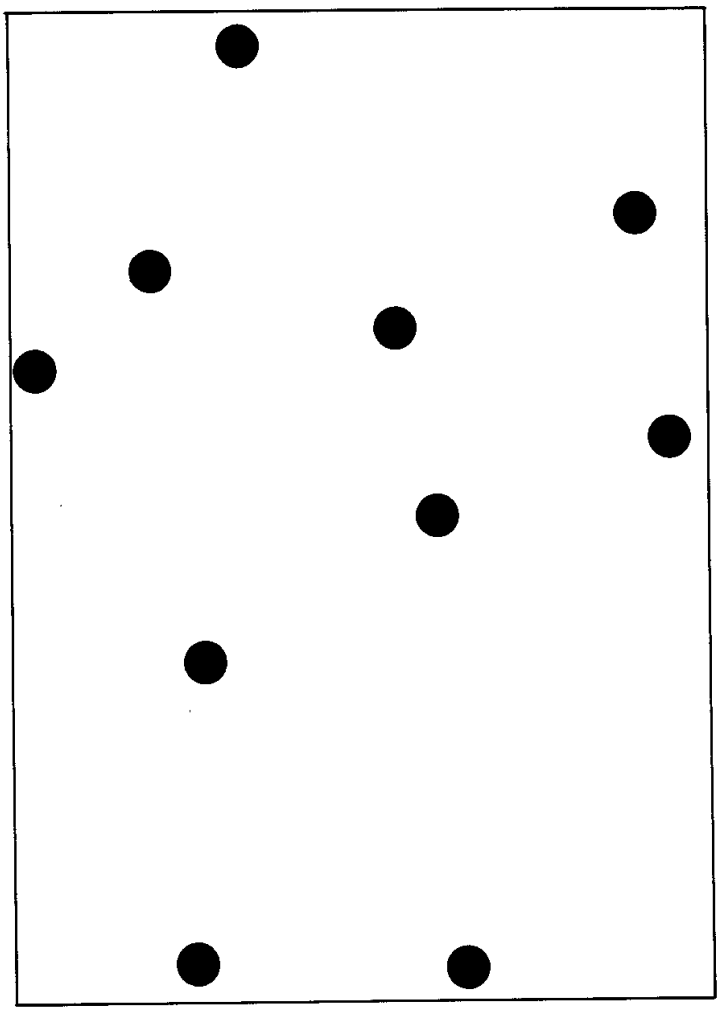
What can you do when your boss asks you to devise an efficient algorithm that comes up with the optimal solution to a problem, and you can't find one?—as surely happened when the airline hit upon the fact that network costs could be reduced by introducing Steiner points. Figure 15.9 shows three ways you can respond. If you could *prove* that there isn't an efficient algorithm to come up with the optimal solution, that would be great. But it's very difficult to prove negative results like this in computer science, for who knows what clever programmer might come along in the future and hit upon an obscure trick that solves the problem. So, unfortunately, you're unlikely to be in a position to say categorically that no efficient algorithm is possible—that the problem is intractable. But if you can show that your problem is NP-complete, then it's actually true that thousands of people in research laboratories have worked on problems that really are equivalent to yours, and also failed to come up with an efficient solution. That may not get you a bonus, but it'll get you off the hook!

Further reading

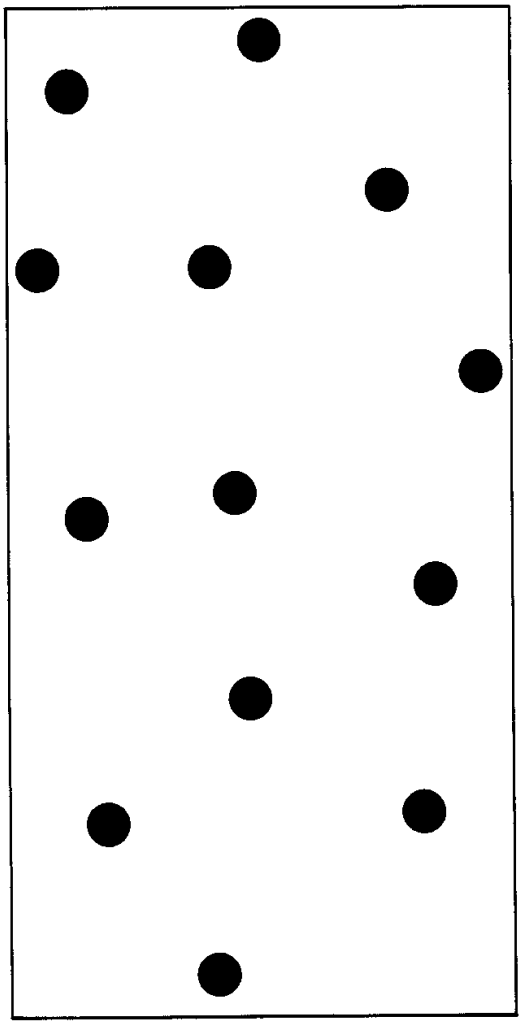
An entertaining videotape entitled *The shortest network problem* by R.L. Graham, in the University Video Communications Distinguished Lecture Series, introduces minimal Steiner trees and discusses some of the results that are known about them. Much of the material in this activity was inspired by that video. Figure 15.9 is from Garey and Johnson's classic book *Computers and Intractability*.

The "Computer recreations" column of *Scientific American*, June 1984, contains a brief description of how to make Steiner trees using soap bubbles, along with interesting descriptions

of other analog gadgets for problem solving, including a spaghetti computer for sorting, a cat's cradle of strings for finding shortest paths in a graph, and a light-and-mirrors device for telling whether or not a number is prime. These also appear in a section about analog computers in Dewdney's *Turing Omnibus*.



Instructions: *Find a way of linking these drill sites with the shortest possible ice roads.*



Instructions: *Find a way of linking these drill sites with the shortest possible ice roads.*

Part V

Sharing secrets and fighting crime—*Cryptography*

You've heard of spies and secret agents using hidden codes or magic invisible writing to exchange messages. Well, that's how the subject of "cryptography" started out, as the art of writing and deciphering secret codes. During the Second World War, the English built special-purpose electronic code-breaking machines and used them to crack military codes. And then computers came along and changed everything, and cryptography entered a new era. Massive amounts of computation, that would have been quite unimaginable before, could be deployed to help break codes. When people began to share computer systems with each other, there were new uses for secret passwords. When computers were linked up in networks, there were new reasons to protect information from people who would have liked to have got hold of it. When electronic mail arrived, it became important to make sure that people who sign messages are really who they say they are. Now that people can buy and sell goods using computers, we need secure ways of placing orders and sending cash on computer networks. And the growing threat of a terrorist attacking a computer system makes computer security ever more important.

Cryptography probably makes you think of computers storing secret passwords, and jumbling up the letters of messages so that the enemy can't read them. But the reality is very different. Modern computer systems *don't* store secret passwords, because if they did, anyone who managed to get access to them would be able to break through all the security in the system. That would be disastrous: they could make phoney bank transfers, send messages pretending to be someone else, read everyone's secret files, command armies, bring down governments. Nowadays, passwords are handled using the "one-way functions" that you learned about in Activity 14. And encryption is *not* just jumbling up the letters of messages: it's done using techniques involving really hard problems—like the "intractable" ones introduced in Part IV.

Using cryptography, you can do things that you might think are impossible. In this section you will discover a simple way to calculate the average age of the people in a group without anyone having to let anyone else know what their age is. You will find out how two people who don't trust each other can toss a coin and agree on the outcome even though they are in different cities and can't both see the coin being tossed. And you will find a way to encode secret messages that can only be decoded by one person, even though everyone knows how to encode them.

For teachers

The activities that follow provide hands-on experience with modern cryptographic techniques—which are very different from what most people conjure up when they think of secrecy and computers.

There are two key ideas. The first is the notion of a "protocol," which is a formal statement of a transaction. Protocols may bring to mind diplomats, even etiquette, but computers use them too! Seemingly difficult tasks can be accomplished by surprising simple protocols. Activity 16, which only takes a few minutes, shows how a group of people, cooperating together, can easily calculate their average age (or income), without anyone finding out any individual's age (or income). The second key idea is the role that computational complexity—intractability—can play when interacting with others through computers. Activity 17 shows how two people who don't necessarily trust each other can agree on

the outcome of a coin toss when they are connected only by telephone. (This activity also introduces, as an aside, the idea of Boolean logic circuits and how to work with them.) Activity 18 shows how people can use computational techniques to encrypt messages securely, even though the method for performing the encoding is public knowledge.

Some of these activities—particularly the last one—are hard work. You will have to motivate your class by instilling into the children a sense of wonder that such things can be done at all, for the activities really do accomplish things that most people would think were impossible. It is vital to create this sense of wonder, communicate it, and pause frequently to keep it alive throughout the activity so that children do not miss the (amazing!) forest for the (perhaps rather tiresome) trees. These activities are among the most challenging and technically intricate in the book. If they turn out to be too difficult, please skip to Part VI, which has a completely different, non-technical, character.

For the technically-minded

As computers encroach upon our daily lives, the application of cryptography is potentially rather tendentious. Most people simply don't realize what modern cryptographic protocols are capable of. The result is that when large institutions—both governmental and commercial—set up systems that involve personal information, it tends to be technocrats who make the key decisions on how things are to be handled, what is to be collected, what is to be made available, and to whom. If people had a better understanding of the possibilities opened up by modern technology, they would be able to participate more actively in such decisions, and society might end up with a different information infrastructure.

This material on information-hiding protocols, cryptographic protocols, and public-key encryption is generally considered to be pretty advanced. But the ideas themselves are not difficult. It's the technicalities, not the underlying concepts, that are hard to understand. In practical situations involving electronic commerce, the technicalities are buried inside computer software, which renders the new technologies of encryption very easy to use. But it's also important to understand the ideas on which they are based, in order to gain insight into what can be done.

Cryptographic systems are of great interest to governments, not just because they want to keep official communications secure, but because of concerns that encrypted communication could be used by people involved in illegal activities such as drug trafficking and terrorism. If such people use encryption then wire-tapping becomes useless unless a decryption method is available. These concerns have created a lot of debate between people concerned with law enforcement, who want to limit the strength of cryptographic systems, and civil libertarians, who are uncomfortable with the government having access to the private communications. The US government has restricted the use of some cryptographic methods by deeming them to be munitions—like bombs and guns, anyone can set up a secure communication system given the right information and some technical ability, but they are dangerous in the wrong hands. One recent debate was over the "Clipper Chip," a system that has an extra password called a *key escrow*, which is held by a govern-

ment agency that allows it to decode any message encrypted by the chip. The FBI and US Justice department wanted this chip to be widely used for communications, but this has drawn considerable opposition because of threats to privacy. All sorts of cryptographic systems are technically feasible, but they aren't necessarilly politically acceptable!

Cryptographic ideas have many applications other than keeping messages secret. Like verifying that messages really were sent by the people who said they sent them—this is "authentication," and without it electronic commerce is impossible. There are ways to let people vote by computer without anyone else being able to find out who they voted for—even those who run the computer system—yet still prevent people from voting more than once. And you can even play cards over the phone—which may sound silly until you realize that making business deals is a lot like playing poker.

These things sound impossible. How could you even begin to shuffle a deck of cards over the phone if you're in competition with the person at the other end and so can't trust them? How could you possibly detect that someone has intercepted a message, modified it, and then passed it off as the original? Yet if you can't, you can't conduct business electronically. You *have* to prevent technically-minded criminals from forging authorizations for withdrawals from bank accounts by intercepting the phone line between a point-of-sale terminal and the bank. You *have* to prevent business competitors from wreaking havoc by generating false orders or false contracts. With modern cryptographic techniques such miracles can be done, and these activities show how.

There are many interesting books about codes and code-breaking. *Codebreakers: the inside story of Bletchley Park* edited by Hinsley and Stripp, gives first-hand accounts of how some of the first computers were used to break codes during the Second World War, significantly shortening the war and saving many lives. *The Kids' Code and Cipher Book* by Nancy Garden provides children with activities relating to data encryption, mainly in the form of a running story that involves increasingly difficult codes. However, this book focuses primarily on written codes rather than the sort used by computers, and they are considerably easier to use (and to crack) than the kind discussed in this activity.

Activity 16

Sharing secrets—*Information hiding protocols*

Age group	Middle elementary and up.
Abilities assumed	Adding three digit numbers competently; understanding the concept of *average* and how to calculate it.
Time	About 5 minutes.
Size of group	At least three children, preferably more.

Focus

Calculating an average.

Random numbers.

Cooperative tasks.

Summary

Cryptographic techniques enable us to share information with other people, yet still maintain a surprisingly high level of privacy. This activity illustrates a situation where information is shared, and yet none of it is revealed: a group of children will calculate their average age without anyone having to reveal to anyone else what their age is.

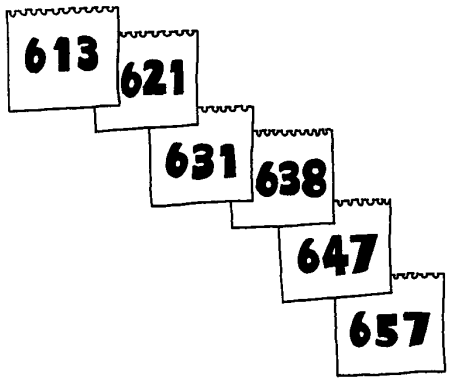


Figure 16.1: Pages of the pad used for finding the average age of five children

Technical terms

Computer security, cryptography, cryptographic protocol, averaging.

Materials

Each group of children will need:

a small pad of paper, and

a pen.

What to do

This activity involves finding the average age of a group of children, without anyone having to reveal what their age is. Alternatively, one could work out the average income (allowance) of the children in the group, or some similar personal detail. Calculating these statistics works particularly well with adults, because older people can be more sensitive about details like age and income. You will need at least three children in the group.

1. Explain to the group that you would like to work out their average age, without anyone telling anyone else what their age is. Ask for suggestions about how this might be done, or even whether they believe it can be done.

2. Select about six to ten children to work with. Give the pad and pen to the first child, and ask them to secretly write down a randomly chosen three-digit number on the top sheet of paper. In the example in Figure 16.1, 613 has been chosen as the random number.

3. Have the first child tear off the first page, add their age to the random number, and write it on the second sheet on the pad. In Figure 16.1, the first child's age is 8, so the second sheet shows 621. They should keep the page that was torn off (and not show it to anyone.)

4. The pad is then passed to the second child, who adds their age to the number on the top, tears off the page, and writes the total on the next page. In the example, the second child is 10 years old.

5. Continue this process having a child tear off the top page and add their age to the number on it, until all the children have had the pad.

6. Return the pad to the first child. Have that child subtract their original random number from the number on the pad. In the example, the pad has been around five children, and the final number, 657, has the original number, 613, subtracted from it, giving the number 44. This number is the sum of the children's ages, and the average can be calculated by dividing by the number of children; thus the average age of our example group is 8.8 years old.

7. Point out to the children that so long as everyone destroys their piece of paper, no-one can work out an individual's age unless two people decide to cooperate.

Variations and extensions

This system could be adapted to allow secret voting by having each person add one if they are voting yes, and zero if they are voting no. Of course, if someone adds more than one (or less than zero) then the voting would be unfair, although they would be running the risk of arousing suspicion if everyone voted yes, since the number of yes votes would be more than the number of people.

What's it all about?

Computers store a lot of personal information about us: our bank balance, how much tax we owe, how long we have held a driver's license, our credit history, examination results, medical records, and so on. Privacy is very important! But we do need to be able to share some of this information with other people. For example, when paying for goods at a store using a bank card, we recognize that the store needs to verify that we have the funds available.

Often we end up providing more information than is really necessary. For example, if we perform an electronic transaction at a store, they will probably discover who we bank with, what our account number is, and what our name is. Furthermore, the bank finds out where we have done our shopping. In principle, a bank could create a profile of someone by monitoring things like where they buy gas or groceries, how much they spend on these items each day, and when these places are visited. If we had paid by cash then none of this information would have been revealed. Most people wouldn't worry too much about this information being shared, but there is the potential for it to be abused, whether for targeted marketing (for example, sending travel advertisements to people who spend a lot on air tickets), discrimination (such as offering better service to someone whose bank usually only takes on wealthy clients), or even blackmail (such as threatening to reveal the details of an embarrassing transaction). If nothing else, people might change the way they shop if they think that someone might be monitoring them.

This loss of privacy is fairly widely accepted, yet cryptographic protocols exist that allow us to make electronic financial transactions with the same level of privacy as we would get with cash. It might be hard to believe that money can be transferred from your bank account to a store's account without anyone knowing where the money was coming from or going to. This activity makes such a transaction seem a little more plausible: both situations involve limited sharing of information, and this can be made possible by a clever protocol.

Further reading

David Chaum has written a paper with the provocative title "Security without identification: transaction systems to make Big Brother obsolete." The paper is quite readable, and gives simple examples of information hiding protocols, including how completely private transactions can be made using "electronic cash." It can be found in *Communications of the ACM*, October 1985.

Activity 17

The Peruvian coin flip—*Cryptographic protocols*

Age group	Older elementary and up.
Abilities assumed	Requires counting, and recognition of odd and even numbers. Some understanding of the concepts *and* and *or* is helpful. Children will get more out of this activity if they have learned binary number representation (see Activity 1, Count the dots), the concept of *parity* (see Activity 4, Card flip magic), and have seen the example of one-way functions in Activity 14, Tourist Town.
Time	About 30 minutes.
Size of group	From individuals to the whole classroom.

Focus

Boolean logic.

Functions.

Puzzle solving.

Summary

This activity shows how to accomplish a simple, but nevertheless seemingly impossible task—making a fair random choice by flipping a coin, between two people who don't necessarily trust each other, and are connected only by a telephone.

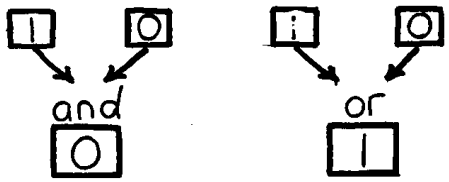


Figure 17.1: An *and*-gate and an *or*-gate

Technical terms

Distributed coin-tossing, computer security, cryptography, cryptographic protocol, *and*-gate, *or*-gate, combinatorial circuit.

Materials

Each group of children will need:

a copy of the reproducible sheet on page 183, and

about two dozen small buttons or counters of two different colors.

What to do

This activity was originally devised when one of the authors (MRF) was working with children in Peru, hence the name. You can customize the story to suit local conditions.

The women's soccer teams of Lima and Cuzco have to decide who gets to be the home team for the championship game. The simplest way would be to flip a coin. But the cities are far apart, and Alicia, representing Lima, and Benito, representing Cuzco, cannot spend the time and money to get together to flip a coin. Can they do it over the telephone? Alicia could flip and Benito could call heads or tails. But this won't work because if Benito called heads, Alicia can simply say "sorry, it was tails" and Benito would be none the wiser. Alicia is not naturally deceitful but this, after all, is an important contest and the temptation is awfully strong. Even if Alicia were truthful, would Benito believe that if he lost?

This is what they decide to do. Working together, they design a circuit made up of *and*-gates and *or*-gates, as explained below. In principle they can do this over the phone, although admittedly in practice it could turn out to be more than a little tedious (fax machines would help!). During the construction process, each has an interest in ensuring that the circuit is complex enough that the other will be unable to cheat. The final circuit is public knowledge.

The rules of *and*-gates and *or*-gates are simple. Each "gate" has two inputs and one output (Figure 17.1). Each of the inputs can be either a 0 or a 1, which can be interpreted as *false* and *true*, respectively. The output of an *and*-gate is one (*true*) only if both inputs are one (*true*), and zero (*false*) otherwise. For example, the *and*-gate in Figure 17.1 has a one and a zero on its

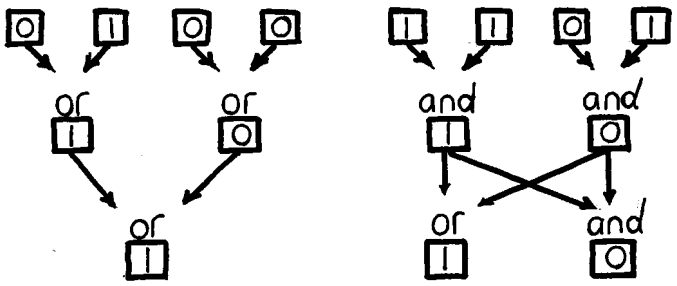


Figure 17.2: Combinations of *or*-gates and *and*-gates

inputs (at the top), so the output (the square at the bottom) is a zero. The output of an *or*-gate is one (*true*) if either (or both) of the inputs is one (*true*), and zero (*false*) only if both the inputs are zero. Thus in Figure 17.1 the output of the *or*-gate is a one for the inputs zero and one.

The output of one gate can be connected to the input of another (or several others) to produce a more complicated effect. For example, in the left-hand circuit of Figure 17.2 the outputs from two *or*-gates are connected to the inputs of a third *or*-gate, with the effect that if any of the four inputs is a one then the output will be a one. In the right-hand circuit of Figure 17.2 the outputs of each of the top two *and*-gates feeds into the lower two gates, so the whole circuit has two values in its output.

For the Peruvian coin flip we need even more complex circuits. The circuit in the reproducible sheet on page 183 has six inputs and six outputs. Figure 17.3 shows a worked example for one particular set of input values.

The way that this circuit can be used to flip a coin by telephone is as follows. Alicia selects a random input to the circuit, consisting of six binary digits (zeros or ones), which she keeps secret. She puts the six digits through the circuit and sends Benito the six bits of output. Once Benito has the output, he must try to guess whether Alicia's input has an even or an odd number of ones—in other words, she must guess the *parity* of Alicia's input. If the circuit is complex enough then Benito won't be able to work out the answer, and his guess will have to be a random choice (in fact, he could even toss a coin to choose!) Benito wins—and the playoff is in Cuzco—if his guess is correct; Alicia wins—and the playoff is in Lima—if Benito guesses incorrectly. Once Benito has told Alicia his guess, Alicia reveals her secret input so that Benito can confirm that it produces the claimed output.

1. Divide the children into small groups, give each group the circuit and some counters, and explain the story. The situation will probably be more meaningful to the children if they imagine one of their sports captains organizing the toss with a rival school. Establish a convention for the counter colors—red is 0, blue is 1, or some such—and have the children mark it on the legend at the top of the sheet to help them remember.

2. Show the children how to place counters on the inputs to show the digits that Alicia chooses. Then explain the rules of *and*-gates and *or*-gates, which are summarized at the bottom of the sheet (consider getting the children to color these in).

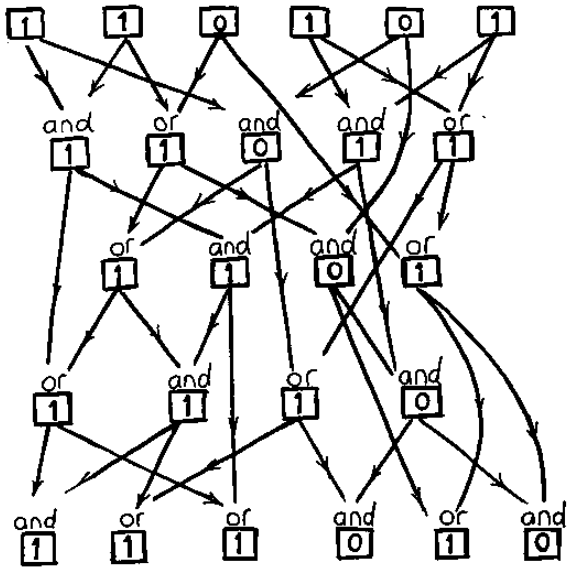


Figure 17.3: An example of Alicia's work

3. Show how to work through the circuit, placing counters at the nodes, to derive the corresponding output. This must be done accurately and takes some care; Table 17.1 (which should *not* be given to the children) shows the output for each possible input for your own reference in case of any doubt.

4. Now each group should elect an Alicia and a Benito. The group can split in half and each half side with Alicia or Benito respectively. Alicia should choose a random input for the circuit, calculate the output, and tell it to Benito. Benito guesses the parity of the input (whether it has an odd or even number of ones in it). It should become evident during this process that Benito's guess is essentially random. Alicia then tells everyone what the input was, and Benito wins if she guessed the correct parity. Benito can verify that Alicia's didn't change her chosen input by checking that it gives the correct output from the circuit.

   At this point the coin toss has been completed.

Benito can cheat if, given an output, he can find the input that produced it. Thus it is in Alicia's interests to ensure that the function of the circuit is *one-way*, in the sense discussed in Activity 14, to prevent Benito cheating. A one-way function is one for which the output is easy to calculate if you know what the input is, but the input is very difficult to calculate for a given output.

Alicia can cheat if she can find two inputs of opposite parity that produce the same output. Then, whichever way Benito guesses, Alicia can reveal the input that shows him to be wrong. Thus it is in Benito's interests to ensure that the circuit does not map many different inputs to the same output.

Input	000000	000001	000010	000011	000100	000101	000110	000111
Output	000000	010010	000000	010010	010010	010010	010010	010010
Input	001000	001001	001010	001011	001100	001101	001110	001111
Output	001010	011010	001010	011010	011010	011010	011010	011111
Input	010000	010001	010010	010011	010100	010101	010110	010111
Output	001000	011010	001010	011010	011010	011010	011010	011111
Input	011000	011001	011010	011011	011100	011101	011110	011111
Output	001010	011010	001010	011010	011010	011010	011010	011111
Input	100000	100001	100010	100011	100100	100101	100110	100111
Output	000000	010010	011000	011010	010010	010010	011010	011010
Input	101000	101001	101010	101011	101100	101101	101110	101111
Output	001010	011010	011010	011010	011010	011010	011010	011111
Input	110000	110001	110010	110011	110100	110101	110110	110111
Output	001000	011010	011010	011010	011010	111010	011010	111111
Input	111000	111001	111010	111011	111100	111101	111110	111111
Output	001010	011010	011010	011010	011010	111010	011010	111111

Table 17.1: Input-Output function for the circuit in Figure 17.3

5. See if the children can find a way for Alicia or Benito to cheat.

   From the first line of Table 17.1 you can see that several different inputs generate the output 010010—for example, 000001, 000011, 000101, etc. Thus if Alicia declares the output 010010, she can choose input 000001 if Benito guesses that the parity is even, and 000011 if he guesses that it is odd.

   With this circuit, it is hard for Benito to cheat. But if the output happens to be 011000, then the input must have been 100010—there is no other possibility (you can see this by checking right through Table 17.1). Thus if this is the number that Alicia happens to come up with, Benito can guess even parity and be sure of being correct. A computer-based system would use many more bits, so there would be too many possibilities to try (each extra bit doubles the number of possibilities).

6. Now ask the groups of children to devise their own circuits for this game. See if they can find a circuit that makes it easy for Alicia to cheat, and another that makes it easy for Benito to cheat. There is no reason why the circuit has to have six inputs, and it may even have different numbers of inputs and outputs.

Variations and extensions

1. An obvious problem in practice is the cooperation that is needed to construct a circuit acceptable to both Alicia and Benito. This might make the activity fun for the kids, but is likely to render the procedure inoperable in practice—particularly over the phone! However, there is a simple alternative in which Alicia and Benito construct their circuits independently and make them publicly available. Then Alicia puts her secret input through *both* circuits, and joins the two outputs together by comparing corresponding bits and making the final output a one if they are equal and zero otherwise. In this situation, neither participant can cheat if the other doesn't, for if just one of the circuits is a one-way function then the combination of them both is also a one-way function.

The next two variations relate not to cryptographic protocols or the coin-tossing problem *per se*, but rather to the idea of circuits constructed out of *and* and *or* gates. They explore some important notions in the fundamentals not only of computer circuits, but of logic itself. This kind of logic is called Boolean algebra , named after the mathematician George Boole (1815–64).

2. The children may have noticed that the all-zero input, 000000, is bound to produce the all-zero output, and likewise the all-one input 111111 is bound to produce the all-one output. (There may be other inputs that produce these outputs as well; indeed, there are for the example circuit—000010 produces all zeros, while 110111 produces all ones.) This is a consequence of the fact that the circuits are made up of *and* and *or* gates. By adding a *not*-gate (Figure 17.4), which takes just one input and produces the reverse as output (i.e. $0 \rightarrow 1$ and $1 \rightarrow 0$), the children can construct circuits that don't have this property.

3. Two other important kinds of gate are *and-not* and *or-not*, which are like *and* and *or* but followed by a *not.* Thus a *and-not* b is *not* (a *and* b). These do not allow any functionally

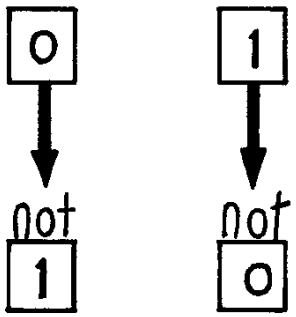


Figure 17.4: The *not*-gate

different circuits to be achieved, since their effect can always be obtained with the corresponding *and* or *or* gate, followed by *not*. However, they have the interesting property that all other gate types can be made out of *and-not* gates, and also out of *or-not* gates.

Having introduced *and-not* and *or-not*, challenge the children to discover whether any of the gates can be made from other gates connected together, and further, if they can be made from just one type of gate connected together. Figure 17.5 shows how the three basic gates, *and*, *or* and *not*, can be constructed from *and-not* gates, in the top row, and *or-not* gates, in the bottom row.

What's it all about?

Recent years have seen huge increases in the amount of commerce being conducted over computer networks, and it is essential to guarantee secure interchange of electronic funds, confidential transactions, and signed, legally binding, documents. The subject of *cryptography* is about communicating in secure and private ways. Two decades ago, computer science researchers discovered the counter-intuitive result that secrecy can be guaranteed by techniques that ensure that certain information is kept *public*. The result is the so-called "public key cryptosystem" of Activity 18, Kid Krypto, that is now regarded as the only completely secure way of exchanging information.

Cryptography is not just about keeping things secret, but about placing controls on information that limit what others can find out, and about establishing trust between people who are geographically separated. Formal rules or "protocols" for cryptographic transactions have been devised to allow such seemingly impossible things as unforgeable digital signatures and the ability to tell others that you possess a secret (like a password) without actually revealing what it is. Flipping a coin over the telephone is a simpler but analogous problem, which also seems, on the face of it, to be impossible.

In a real situation, Alicia and Benito would not design a circuit themselves, but acquire a computer program that accomplishes the transformation internally. Probably neither would be interested in the innards of the software. But both would want to rest assured that the other is

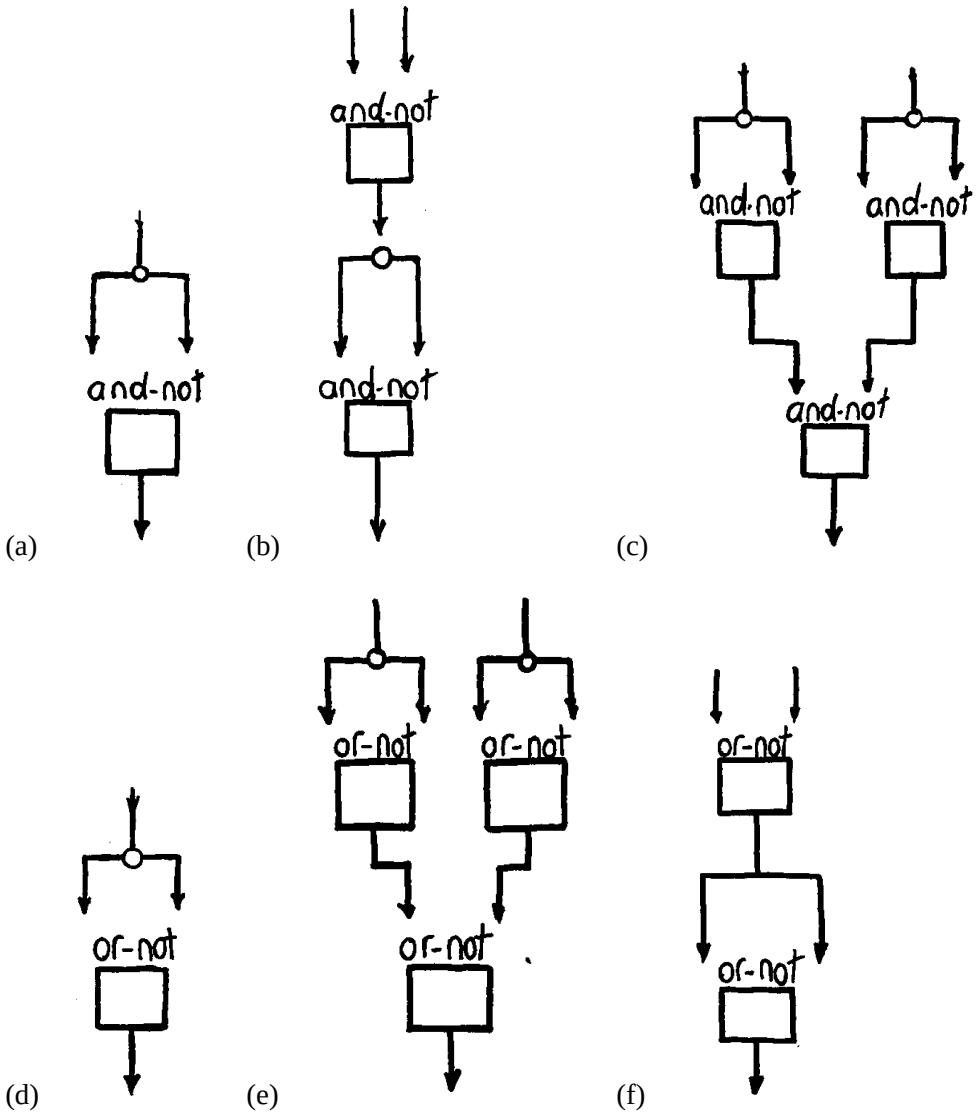


Figure 17.5: Making the three basic gates from *and-not* and *or-not* gates. (a) and (d) are *not*-gates, (b) and (e) are *and*-gates, while (c) and (f) are *or*-gates.

unable to influence the outcome of the decision, no matter how good their computer skills and how hard they tried.

In principle, any disputes would have to be resolved by appeal to a neutral judge. The judge would be given the circuit, Alicia's original binary number, the output that she originally sent Benito, and the guess that Benito sent in return. Once the interchange is over, all this is public information, so both participants will have to agree that this is what the outcome was based on. The judge will be able to put Alicia's original number through the circuit and check that the output is as claimed, and therefore decide whether the decision has been made fairly. Needless to say, the very fact that there is a clear procedure to check that the rules have been followed makes it unlikely that a dispute will arise. Compare with the situation where Alicia flips an actual coin and Benito calls heads or tails—no judge would take on that case!

A circuit as small as the one illustrated would not be much use in practice, for it is easy to come up with Table 17.1 and use it to cheat. Using thirty-two binary digits in the input would provide better protection. However, even this does not *guarantee* that it is hard to cheat—that depends on the particular circuit. Other methods could be used, such as the one-way function introduced in Activity 14, Tourist Town. Methods used in practice often depend on the factoring of large numbers, which is known to be a hard problem (although, as we will learn at the end of the next activity, it is not NP-complete). It is easy to check that one number is a factor of another, but finding the factors of a large number is very time consuming. This makes it more complex for Alicia and Benito (and the judge) to work through by hand, although, as noted above, in practice this will be done by off-the-shelf software.

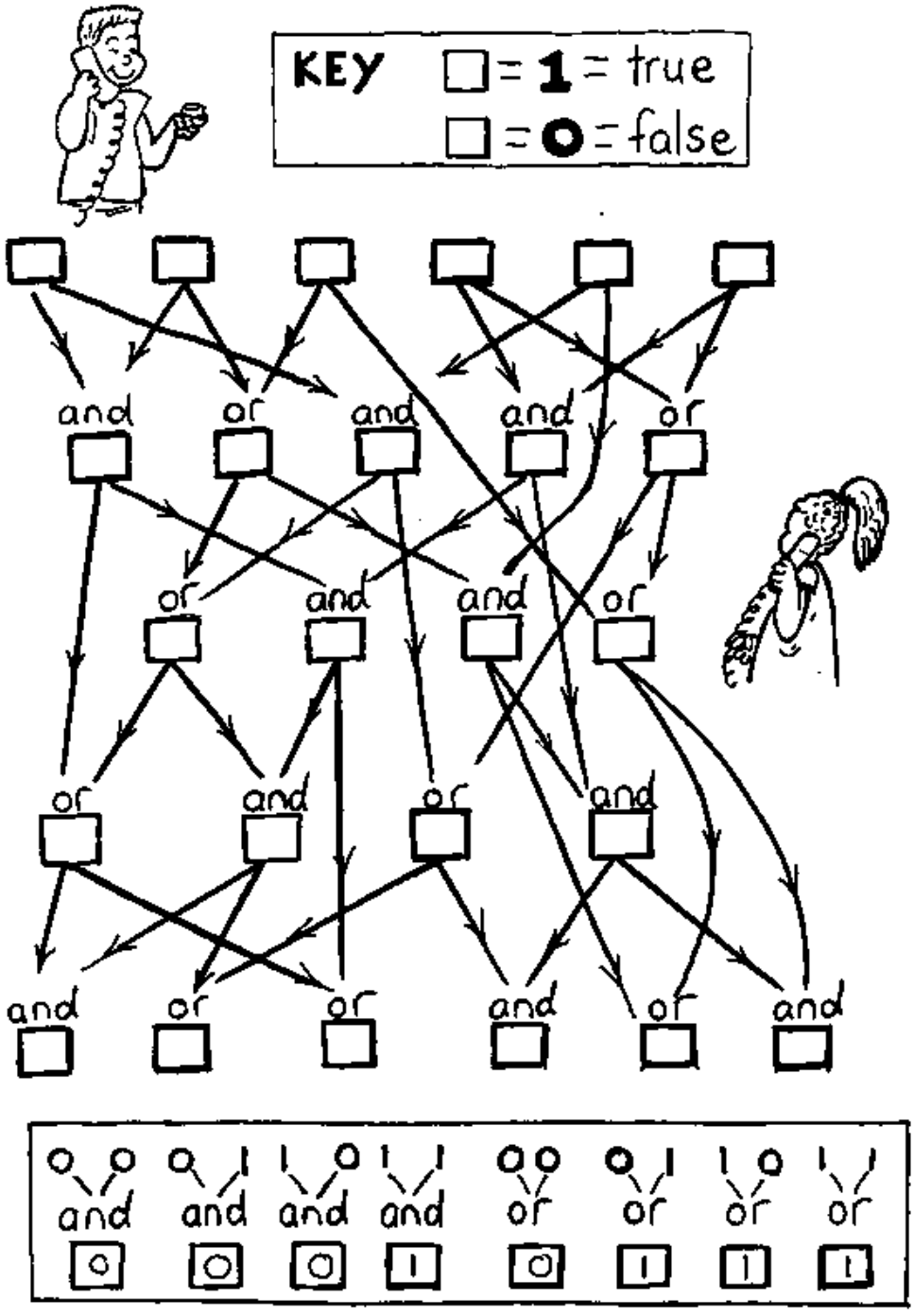
Digital signatures are based on a similar idea. By making public the output of the circuit for the particular secret input that she has chosen, Alicia is effectively able to prove that she is the one who generated the output—for, with a proper one-way function, no-one else can come up with an input that works. No-one can masquerade as Alicia! To make an actual digital signature, a more complex protocol is needed to ensure that Alicia can sign a particular message, and also to ensure that others can check that Alicia was the signatory even if she claims not to be. But the principle is the same.

Another application is playing poker over the phone, in an environment in which there is no referee to deal the cards and record both player's hands. Everything must be carried out by the players themselves, with recourse to a judge at the end of the game in the event of a dispute. Similar situations arise in earnest with contract negotiations. Obviously, players must keep their cards secret during the game. But they must be kept honest—they must not be allowed to claim to have an ace unless they actually have one! This can be checked by waiting until the game is over, and then allowing each player to inspect the other's original hand and sequence of moves. Another problem is how to deal the cards while keeping each player's hand secret until after the game. Surprisingly, it is possible to accomplish this using a cryptographic protocol not dissimilar to the coin-tossing one.

Cryptographic protocols are extremely important in electronic transactions, whether to identify the owner of a smart card, to authorize the use of a cellphone for a call, or to authenticate the sender of an electronic mail message. The ability to do these things reliably is crucial to the success of electronic commerce.

Further reading

Harel's book *Algorithmics* discusses digital signatures and associated cryptographic protocols. It also shows how to play poker over the phone, an idea that was first raised in 1981 in a chapter called "Mental poker", in the book *The Mathematical Gardener*, edited by D.A. Klarner. *Cryptography and data security* by Dorothy Denning is an excellent computer science text on cryptography. Dewdney's *Turing Omnibus* has a section on Boolean logic that discusses the building blocks used for the circuits in this activity.



Instructions: *Choose some inputs for this circuit and work out what the outputs are.*

Activity 18

Kid krypto—*Public-key encryption*

Age group	Junior high and up.
Abilities assumed	This is the most technically challenging activity in the book. While rewarding, it requires careful work and sustained concentration to complete successfully. Children should already have studied the example of one-way functions in Activity 14, Tourist Town, and it is helpful if they have completed the other activities in this section (Activity 16, Sharing Secrets, and Activity 17, the Peruvian coin flip). The activity also uses ideas covered in Activity 1, Count the dots, and Activity 5, Twenty guesses.
Time	About 30 minutes.
Size of group	Requires at least two people, can be done with a whole class.

Focus

Puzzle solving.

Secret codes.

Summary

Encryption is the key to information security. And the key to modern encryption is that using only *public* information, a sender can lock up their message in such a way that it can only be unlocked (*privately*, of course) by the intended recipient.

It is as though everyone buys a padlock, writes their name on it, and puts them all on the same table for others to use. They keep the key of course—the padlocks are the kind where you just click them shut. If I want to send you a secure message, I put it in a box, pick up your padlock, lock the box and send it to you. Even if it falls into the wrong hands, no-one else can unlock it. With this scheme there is no need for any prior communication to arrange secret codes.

This activity shows how this can be done digitally. And in the digital world, instead of picking up your padlock and using it, I *copy* it and use the copy, leaving the original lock on the table. If I were to make a copy of a physical padlock, I could only do so by taking it apart. In doing so I would inevitably see how it worked. But in the digital world we can arrange for people to copy locks without being able to discover the key!

Sounds impossible? Read on.

Technical terms

Public-key cryptosystems, encryption, decryption, NP-complete problems.

Materials

The children are divided into groups of about four, and within these groups they form two subgroups. Each subgroup is given a copy of the two maps on page 193. Thus for each group of children you will need:

two copies of the blackline master on page 193.

You will also need:

an overhead projector transparency of page 194, and

a way to annotate the diagram.

What to do

Amy is planning to send Bill a secret message. Normally we might think of secret messages as a sentence or paragraph, but in the following exercise Amy will send just one character — in fact, she will send one number that represents a character. Although this might seem like a simplistic message, bear in mind that she could send a whole string of such "messages" to make up a sentence, and in practice the work would be done by a computer. And sometimes even small messages are important — one of the most celebrated messages in history, carried by Paul Revere, had only two possible values.

We will see how to embed Amy's number in an encrypted message using Bill's public lock so that if anyone intercepts it, they will not be able to decode it. Only Bill can do that, because only he has the key to the lock.

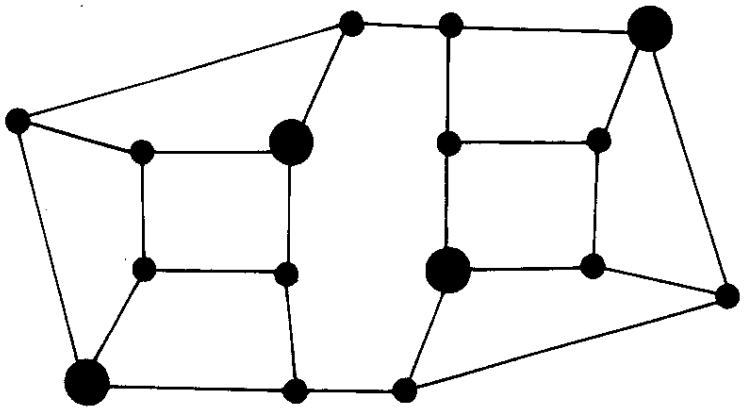


Figure 18.1: Bill's private map

We will lock up messages using *maps*. Not Treasure Island maps, where X marks the spot, but street maps like the one of Tourist Town on page 149, where the lines are streets and the dots are street corners. Each map has a public version—the lock—and a private version—the key.

Shown on page 194 is Bill's public map. It's not secret: Bill puts it on the table (or a web page) for everyone to see, or (equivalently) gives it to anyone who might want to send him a message. Amy has a copy; so has everyone else. Figure 18.1 shows Bill's private map. It's the same as his public map, except that some of the street corners are marked as special by enlarging them. He keeps this version of the map secret.

This activity is best done as a class, at least to begin with, because it involves a fair amount of work. Although not difficult, this must be done accurately, for errors will cause a lot of trouble. It is important that the children realize how surprising it is that this kind of encryption can be done at all—it seems impossible (doesn't it?)—because they will need this motivation to see them through the effort required. One point that we have found highly motivating for school children is that using this method they can pass secret notes in class, and even if their teacher knows how the note was encrypted, the teacher won't be able to decode it.

1. Put Bill's public map (page 194) on the overhead projector. Decide which number Amy is going to send. Now place random numbers on each intersection on the map, so that the random numbers add up to the number that Amy wishes to send. Figure 18.2 gives an example of such numbers as the upper (non-parenthesised) number beside each intersection. Here, Amy has chosen to send the number 66, so all the unbracketed numbers add up to 66. If necessary, you can use negative numbers to get the total down to the desired value.

2. Now Amy must calculate what to send to Bill. If she sent the map with the numbers on, that would be no good, because if it fell into the wrong hands anybody could add them up and get the message.

   Instead, choose any intersection, look at it and its three neighbors—four intersections in

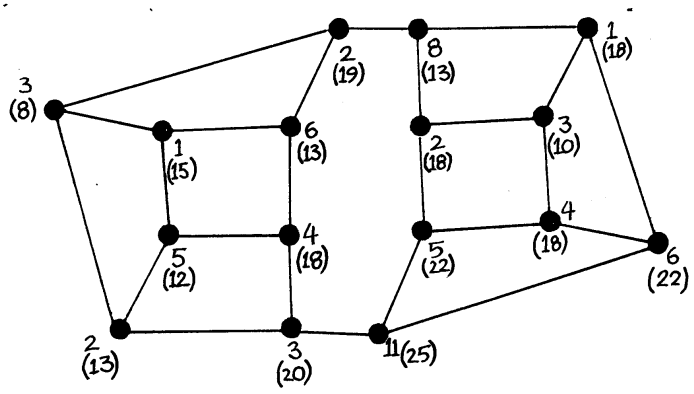


Figure 18.2: Amy's calculations, using Bill's public map

all—and total the numbers on them. Write this number at the intersection in parentheses or using a different color of pen. For example, the rightmost intersection in Figure 18.2 is connected to three others, labeled 1, 4, 11, and is itself labeled 6. Thus it has a total of 22. Now repeat this for all the other intersections in the map. This should give you the number in parentheses in Figure 18.2.

3. Amy will send to Bill his map, with only the parenthesised numbers on it.

   Erase the original numbers and the counts, leaving only the numbers that Amy sends; or write out a new map with just those numbers on it. See if any of the children can find a way to tell from this what the original message was. They won't be able to.

4. Only someone with Bill's private key can decode the message to find the message that Amy originally wanted to send. On the coded message mark the special enlarged nodes in Bill's private map (Figure 18.1).

   To decode the message, Bill looks at just the secret marked intersections and adds up the numbers on them. In the example, these intersections are labeled 13, 13, 22, 18, which add up to 66, Amy's original message.

5. How does it work? Well, the map is a special one. Suppose Bill were to choose one of the marked intersections and draw around the intersections one street distant from it, and repeat the procedure for each marked intersection. This would partition the map into non-overlapping pieces, as illustrated in Figure 18.3. Show these pieces to the children by drawing the boundaries on the map. The group of intersections in each partition is exactly the ones summed to give the transmitted numbers for the marked intersections, so the sum of the four transmitted numbers on those intersections will be the sum of all the original numbers in the original map; that it, it will be the original message!

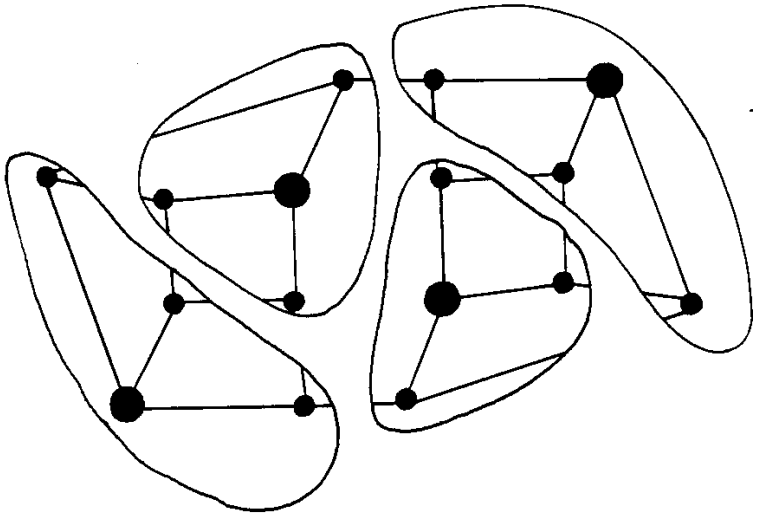


Figure 18.3: The regions that comprise Bill's private map

Phew! It seems a lot of work to send one letter. And it *is* a lot of work to send one letter—encryption is not an easy thing to do. But look at what has been accomplished: complete secrecy using a public key, with no need for any prior arrangement between the participants. You could publish your key on a noticeboard and *anyone* could send you a secret message, yet no-one could decrypt it without the private key. And in real life all the calculation will be done by a software package that you acquire, so it's only a computer that has to work hard.

Perhaps your class would like to know that they have joined the very select group of people who have actually worked through a public-key encryption example by hand—practising computer scientists would consider this to be an almost impossible task and virtually no-one has ever done it!

Now, what about eavesdropping? Bill's map is like the ones in the Tourist Town activity (Activity 14), where the marked intersections are a minimal way of placing ice-cream vans to serve all street corners without anyone having to walk more than one block. We saw in Tourist Town that it's easy for Bill to make up such a map by starting with the pieces shown in Figure 18.3, and it's very hard for anyone else to find the minimal way to place ice-cream vans except by the brute-force method. The brute-force method is to try every possible configuration with one van, then every configuration with two vans, and so on until you hit upon a solution. No-one knows whether there is a better method for a general map—and you can bet that lots of people have tried to find one!

Providing Bill starts with a complicated enough map with, say, fifty or a hundred intersections, it seems like no-one could ever crack the code—even the cleverest mathematicians have tried hard and failed. (But there is a caveat: see below under *What's it all about?*)

6. Having been through one example with the whole class, divide the children into groups of, say, four. Give each pair of each group the public map on the blackline master on

page 193. Each pair should choose a "message" (any integer), encode it with the public key, and give the resulting map to the other group. The other group can try to decode it, but they are unlikely to be successful until they are given (or work out!) the private map. Then give out the private map and see if they can now decode it correctly.

7. Now each pair can design their own map, keeping the private version secret and giving the public version to the other pair—or indeed "publishing" it on the classroom board. The principle for designing maps is just the same as was discussed in the Tourist Town activity, and extra streets can be added to disguise the solution. Just be careful not to add extra streets into any of the "special" points. That would create an intersection from which *two* ice-cream vans could be reached in one hop, which is all right for the tourist town situation but would cause havoc when encrypting. That is because the special points no longer decompose the map into *non-overlapping* pieces, as illustrated in Figure 18.3, and this is essential for the trick to work.

What's it all about?

It's clear why you might want to send secret messages over computer networks that no-one but the intended recipient could decode, no matter how clever they were or how hard they tried. And of course there are all sorts of ways in which this can be done *if* the sender and receiver share a secret code. But the clever part of public-key encryption is that Amy can send Bill a secure message without any secret prior arrangement, just by picking up his lock from a public place like a web page.

Secrecy is only one side of cryptography. Another is *authentication*: When Amy receives a message from Bill, how does she know that it really comes from him and not from some imposter? Suppose she receives electronic mail that says, "Darling, I'm stuck here without any money. Please put $100 in my bank account, number 0241-45-784329 – love, Bill." How can she know whether it really comes from Bill? Some public-key cryptosystems can be used for this, too. Just as Amy sends Bill a secret message by encoding it with his public key, he can send her a message *that only he could have generated* by encoding it with his *private* key. If Amy can decode it with Bill's public key, then it must have come from him. Of course, anyone else could decode it too, since the key is public, but if the message is for her eyes only, Bill can then encode it a second time with Amy's public key. This dual encoding provides both secrecy and authentication with the same basic scheme of public and private keys.

Now is the time to admit that while the scheme illustrated in this activity is very similar to an industrial-strength public-key encryption system, it is *not* in fact a secure one—even if quite a large map is used.

The reason is that although there is no known way of finding the minimal way to place ice-cream vans on an arbitrary map, and so the scheme is indeed secure from this point of view, there happens to be a completely different way of attacking it. The idea is unlikely to occur to schoolchildren, at least up to high school level, but you should at least know that it exists. You might say that the scheme we have been looking at is school-child secure, but not mathematician-secure. Please ignore the next paragraph if you are not mathematically inclined!

Number the intersections on the map 1, 2, 3, ... Denote the original numbers that are assigned to intersections by b_1, b_2, b_3, ..., and the numbers that are actually transmitted by t_1, t_2, t_3, ... Suppose that intersection 1 is connected to intersections 2, 3, and 4. Then the number that is transmitted for that intersection is

$$t_1 = b_1 + b_2 + b_3 + b_4.$$

Of course, there are similar equations for every other intersection—in fact, there are the same number of equations as there are unknowns b_1, b_2, b_3, ... An eavesdropper knows the public map and the numbers t_1, t_2, t_3, ... that are transmitted, and can therefore write down the equations and solve them with an equation-solving computer program. Once the original numbers have been obtained, the message is just their sum—there is actually no need ever to discover the decryption map. The computational effort required to solve the equations directly using Gaussian elimination is proportional to the cube of the number of equations, but because these equations are sparse ones—most of the coefficients are zero—even more efficient techniques exist. Contrast this with the exponential computational effort that, as far as anyone knows, is the best one can do to come up with the decryption map.

We hope you don't feel cheated! In fact, the processes involved in real public-key cryptosystems are virtually identical to what we have seen, except that the techniques they use for encoding are different—and really are infeasible to do by hand. The original public-key method, and still one of the most secure, is based on the difficulty of factoring large numbers.

What are the factors of the 100-digit number 9,412,343,607,359,262,946,971,172,136,294, 514,357,528,981,378,983,082,541,347,532,211,942,640,121,301,590,698,634,089,611,468, 911,681? Don't spend too long! They are 86,759,222,313,428,390,812,218,077,095,850,708,048, 977 and 108,488,104,853,637,470,612,961,399,842,972,948,409,834,611,525,790,577,216,753. There are no other factors: these two numbers are prime. Finding them is quite a job: in fact, it's a several-month project for a supercomputer.

Now in a real public-key cryptosystem, Bill might use the 100-digit number as his public key, and the two factors as the private key. It would not be too difficult to come up with such keys: all you need is a way of calculating large prime numbers. Find two prime numbers that are big enough (that's not hard to do), multiply them together, and—hey presto, there's your public key. Multiplying huge numbers together is no big deal for a computer. Given the public key, no-one can find your private key, unless they have access to several months of supercomputer time. And if you're worried that they might, use 200-digit primes instead of 100-digit ones—that'll slow them down for years! In practice, people use 512-bit keys, which is about 155 decimal digits.

We still haven't given a way to encode a message using a prime-number based public key in such a way that it can't be decoded without the private key. In order to do this, life is not quite as simple as we made out above. It's not the two prime numbers that are used as the private key and their product as the public key, instead it's numbers derived from them. But the effect is the same: you can crack the code by factoring the number. Anyway, it's not difficult to overcome these difficulties and make the scheme into a proper encryption and decryption algorithm, but let's not go into that here. This activity has already been enough work!

How secure is the system based on prime numbers? Well, factoring large numbers is a problem that has claimed the attention of the world's greatest mathematicians for several cen-

turies, and while methods have been discovered that are significantly better than the brute-force method of trying all possible factors, no-one has come up with a really fast (that is, polynomial-time) algorithm. (No-one has proved that such an algorithm is impossible, either.) Thus the scheme appears to be not just school-child secure, but also mathematician-secure. But beware: we must be careful. Just as there turned out to be a way of cracking Bill's code without solving the Tourist Town problem, there may be a way of cracking the prime-number codes without actually factoring large numbers. People have checked carefully for this, and it seems OK.

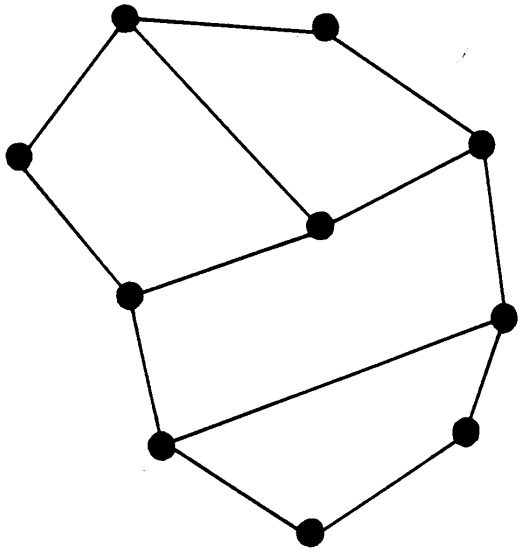
Another worry is that if there are just a few possible messages, an interloper could encrypt each of them in turn using the public key, and compare the actual message with all the possibilities. Amy's method avoids this because there are many ways of encrypting the same message, depending on what numbers were chosen to add up to the code value. In practice, cryptographic systems are designed so that there are just too many possible messages to even begin to try them all out, even with the help of a very fast computer.

It is not known whether a fast method for solving the prime factorization problem exists. No one has managed to devise one, but also it has not been proven that a fast method is impossible. If a fast algorithm for solving this problem is found, then many currently used cryptographic systems will become insecure. In Part IV we discussed *NP-complete* problems, which stand or fall together: if one of them is efficiently solvable then they all must be. Since so much (unsuccessful) effort has been put into finding fast algorithms for these problems, they would seem like excellent candidates for use in designing secure cryptosystems. Alas, there are difficulties with this plan, and so far the designers of cryptosystems have been forced to rely on problems (such as prime factorization) that might in fact be easier to solve than the NP-complete problems—maybe a lot easier. The answers to the questions raised by all this are worth many millions of dollars to industry and are regarded as vital to national security. Cryptography is now a very active area of research in computer science.

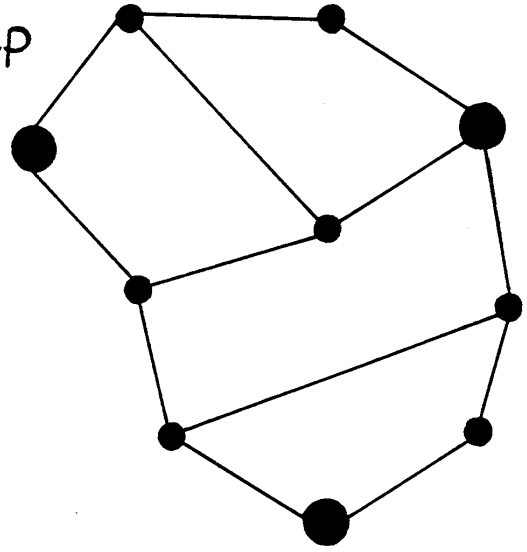
Further reading

Harel's book *Algorithmics* discusses public-key cryptography; it explains how to use large prime numbers to create a secure public-key system. The standard computer science text on cryptography is *Cryptography and data security* by Dorothy Denning, while a more practical book is *Applied cryptography* by Bruce Schneier. Dewdney's *Turing Omnibus* describes another system for performing public key cryptography.

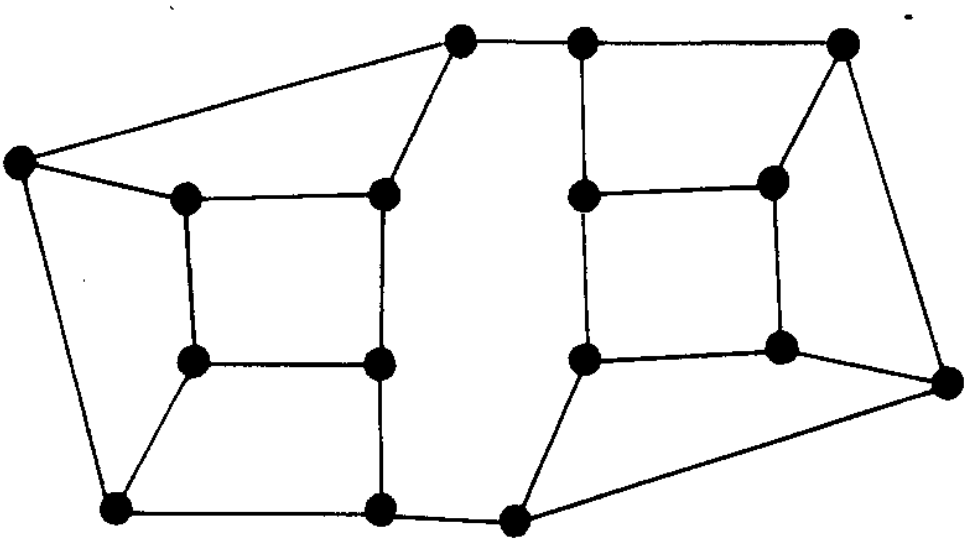
public Map



private Map



Instructions: *Use these maps as described in the text to encrypt and decrypt messages.*



Instructions: *Put this blackline master on an overhead projector transparency and use it to demonstrate the encoding of a message.*

Part VI

The human face of computing—*Interacting with computers*

Why are computers so hard to get along with? Everyone you ask will tell you stories about how difficult computers are to use, how they never seem to do what you really want them to, how they keep going wrong and make ridiculous mistakes. Computers seem to be made for wizards, not for ordinary people. But they *should* be made for ordinary people, because computers are everyday tools that help us to learn, work, and play better.

The part of a computer system that you interact with is called its "user interface." It's the most important bit! Although you might think of what the program actually *does* as the main thing and the user interface as just how you get into it, a program is no good at all if you can't interact with it and make it do what you want. User interfaces are very difficult to design and build, and it has been estimated that when writing programs, far more effort goes into the interface than into any other part. Some video games have excellent user interfaces, interfaces that need no complicated instructions and become almost invisible as you are drawn into the game. Countless software products which are otherwise very good have been complete flops because they have strange user interfaces. And whole industries have been built around a clever interface idea—like the word processor or spreadsheet—that promotes access to computational functions which are really quite elementary in themselves.

But why do we have to have user interfaces at all? Why can't we just talk to our computers the way we do to our friends? Good question. Maybe someday we will; maybe not. But certainly not yet: there are big practical limitations on how "intelligent" computers can be today. The activities that follow will help you understand the problems of user interface design, and help you to think more clearly about the limitations of computers and be wary of the misleading hype that is often used to promote computer products.

For teachers

Computing is not so much about calculation as it is about *communication*. Computing *per se* really has no intrinsic value; it is only worthwhile if the results are somehow communicated to the world outside the computer, and have some influence there. And a lot of the activities in this book are about communication. *Representing data* (Part I) shows how different kinds of information can be communicated to a computer or between computers. *Representing processes* (Part III) is about how to communicate processes to a computer to tell it how to accomplish certain tasks—after all, "programming" is really only explaining to a computer, in its own language! *Cryptography* (Part V) is about how to communicate in secret, or to communicate bits of secrets without revealing all.

The activities that follow are about how people communicate with computers. While the rest of the book is based on well understood technical ideas, this part is not. That makes it both easier, in that no special knowledge is required of the children, and more difficult, in that a certain level of maturity is needed to understand what the activities are about and relate them to a broader context. These activities contain more detailed explanations than most of the others because it is necessary to give you, the teacher, enough background material to be in a position to help draw out some of the implications in class discussion.

There are two activities in this section. The first is about the area known as the "human–computer interface," commonly abbreviated to HCI. In order to "unplug" this aspect of

computing without depending on prior knowledge of a particular example of a computer system, we have invented a design exercise that does not really involve computers—but does introduce fundamental principles that are used in the design of human–computer interfaces. Because human interface design is culture-dependent, there are not necessarily any "right" answers in this activity, which may frustrate some children. The second activity is about the area known as "artificial intelligence," or AI. It involves a guessing game that stimulates children into thinking about what computers can and can't do.

For the technically-minded

Human–computer interaction is fast becoming one of the hottest research areas in computer science as people realize how much the success of a software product depends on its user interface. The subject draws heavily on a wide range of disciplines outside computer science, such as psychology, cognitive science, linguistics, sociology—even anthropology. Few computer scientists have training in these areas, and HCI represents an important growth area for people who are interested in the "softer" side of the subject.

Artificial intelligence is a topic that often raises hackles and causes disputes. In this book we have tried to steer a middle path between AI afficionados who believe that intelligent machines are just around the corner, and AI sceptics who believe that machines cannot in principle be intelligent. Our goal is to encourage children to think independently about such issues, and to promote a balanced view.

The activities here draw heavily on two eminently readable books, Don Norman's *The design of everyday things* and John Haugeland's *Artificial intelligence: the very idea*, which we enthusiastically recommend if you are interested in pursuing these issues further.

Computers involve another important kind of communication, one that is not touched upon in this book: communication between people who are building a computer system. Students who learn about computers and make their way into the job market—perhaps having graduated in computer science from university—are invariably surprised by how much interpersonal communication their job entails. Computer programs are the most complex objects ever constructed by humankind, with millions or perhaps billions of intricately interlocking parts, and programming projects are tackled by close-knit teams that work together and spend a great deal of their time communicating. Once the product is complete, there is the job of communicating with customers through user manuals, courses, "help" phonelines, and the like—not to mention the problem of communicating with potential customers through demonstrations, displays, and advertising. We haven't yet found a way to realistically "unplug" for children the interpersonal communication aspect of computing, so this book doesn't address it. But it is the kind of thing that computer professionals who are visiting a classroom may be able to describe from their own experience and bring out in discussion.

Activity 19

The chocolate factory—*Human interface design*

Age group	Middle elementary and up.
Abilities assumed	No specific abilities required.
Time	An hour or more.
Size of group	From small groups to the whole classroom.

Focus

Design.

Reasoning.

Awareness of everyday objects.

Summary

The aim of this activity is to raise awareness of human interface design issues. Because we live in a world where poor design is rife, we have become accustomed (resigned?) to putting up with problems caused by the artifacts we interact with, blaming ourselves ("human error," "inadequate training," "it's too complicated for me") instead of attributing the problems to flawed design. The issue is greatly heightened by computers because they have no obvious purpose—indeed, they are completely *general* purpose—and their appearance gives no clues about what they are for nor how to operate them.

Technical terms

Interface design; affordances; mapping; transfer effects; population stereotypes; icons; user interface evaluation

Materials

Each group of children will need:

a copy of the blackline master on pages 209 and 210, and

a copy of the images on page 211, either on overhead projector transparency or on cards that can be displayed to the class, and

one or more of the six cards contained in the blackline master on page 212. Cut the sheet into individual cards and divide them between the groups.

What to do

The great chocolate factory is run by a race of elf-like beings called Oompa-Loompas.[1] These Oompa-Loompas have terrible memories and no written language. Because of this, they have difficulty remembering what to do in order to run the chocolate factory, and things often go wrong. Because of this, a new factory is being designed that is supposed to be very easy for them to operate.

1. Divide the children into small groups and explain the story.

2. The first problem the Oompa-Loompas face is getting through the doors carrying steaming buckets of liquid chocolate. They cannot remember whether to push or pull the doors to open them, or slide them to one side. Consequently they end up banging into each other and spilling sticky chocolate all over the place. The children should fill out the "doors" worksheet on page 209. More than one box is appropriate in each case. For some of the doors (including the first one) it is not obvious how to open them, in which case the children should record what they would try first. Once they have filled out their own sheets, have the whole group discuss the relative merits of each type of door, particularly with regard to how easy it is to tell how it works, and how suitable it would be to use if you are carrying a bucket of hot chocolate. Then they should decide what kind of doors and handles to use in the factory.

   Follow this activity with a class discussion. Table 19.1 comments briefly on each door in the worksheet. Real doors present clues in their frames and hinges as to how they open, and there are conventions about whether doors open inwards or outwards. Identify the kinds of door handles used in your school and discuss their appropriateness (they may be

[1] With apologies to Roald Dahl. You'll know about the Oompa-Loompas if you've read his wonderful tale *Charlie and the Chocolate Factory*. If not, never mind: the plot is not relevant to this activity.

Plain door	Can't see how to open this one at all, except that since it has no handle, it must require pushing rather than pulling.	Labeled door	The label is like a tiny user manual. But should a door need a user manual? And the Oompa Loompas can't read.
Hinge door	At least you can see which is the side that opens.	Bar door	It's clear that you are supposed to push the bar, but which side?
Handle door	Handles like this are usually for pulling—or sliding.	Knob door	The knob shows what to grasp, but not whether to push or pull; it probably doesn't slide.
Panel door	It's clear that you push this. What else could you do?	Glass door	The small vertical bar on this side signals "pull"; the longer horizontal one on the other signals "push".
Sliding door	This one's only for sliding.		

Table 19.1: About the doors in the worksheet

quite *in*appropriate!) Do doors normally open inwards or outwards into corridors?—and why? (Answer: They open into rooms so that when you come out you won't bash the door into people walking along the corridor, although in some situations they open outwards to make evacuation easier in an emergency.)

The key concept here is what are called the *affordances* of an object, which are its visible features—both fundamental and perceived—whose appearance indicates how the object should be used. Affordances are the kinds of operation that the object permits, or "affords." For example, it is (mostly) clear from their appearance that chairs are for sitting, tables are for placing things on, knobs are for turning, slots are for inserting things into, buttons are for pushing. And computers are for ... what? They have no affordances that indicate their functionality, apart from very low-level ones such as input (e.g. keyboard) and output (e.g. screen) capabilities.

Doors are very simple objects. Complex things may need explaining, but simple things should not. When simple objects need pictures, labels, or instructions, then design has failed.

3. The pots containing different kinds of chocolate have to cook at different temperatures. In the old chocolate factory the stoves were as shown in the blackline master on page 210. The left-hand knob controlled the rear left heating element, the next knob controlled the front left element, the next one controlled the front right, and the right-hand knob controlled the rear right element. The Oompa-Loompas were always making mistakes, cooking the chocolate at the wrong temperature, and burning their sleeves when reaching across the elements to adjust the controls. The children should recall how the controls are laid out on their cookers at home and come up with a better arrangement for the new factory.

   Follow this activity with a class discussion. Figure 19.1 shows some common arrangements. All but the one at the lower left have the controls at the front, to avoid having to reach across the elements. In the design at the top left, there are so many possible mappings from controls to burners (24 possibilities, in fact) that eight words of labeling are needed. The "paired" arrangement in the top center is better, with only four possible mappings (two for the left cluster and two for the right); it requires just four labeling words. The design at the top right specifies the control–burner relationship diagrammatically rather than linguistically (which is good for the Oompa-Loompas!). The lower three designs need no labels. The left-hand one has a control by each burner, which is awkward and dangerous. The other two involve relocating the burners slightly, but for different reasons: in the center design they are moved to leave room for the controls, while in the right-hand one they are rearranged to make the correspondence clear.

   The key concept here is the *mapping* of controls to their results in the real world. Natural mapping, which takes advantage of physical analogies and cultural standards, leads to immediate understanding. The spatial correspondences at the bottom of Figure 19.1 are good examples—they are easily learned and always remembered. Arbitrary mappings, as in the top arrangements, need to be labeled, or explained and memorized.

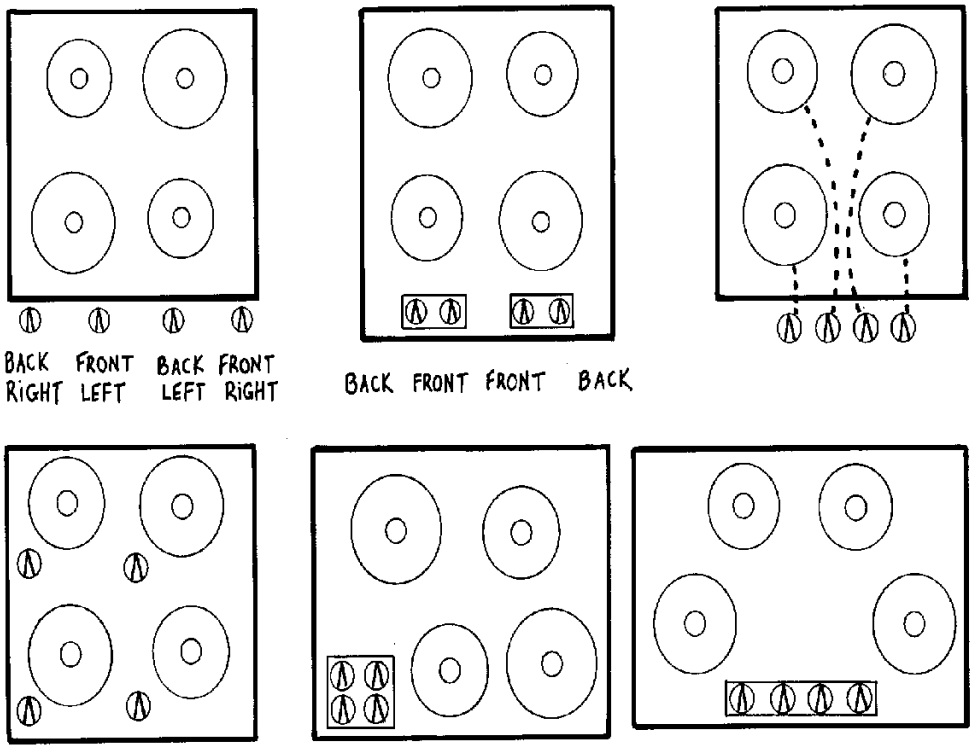


Figure 19.1: Some possible cooker layouts

4. The factory is full of conveyer belts carrying pots of half-made chocolate in various stages of completion. These conveyer belts are controlled manually by Oompa-Loompas, on instructions from a central control room. The people in the control room need to be able to tell the Oompa-Loompa to stop the conveyer belt, or slow it down, or start it up again.

   In the old factory this was done with a voice system: the control room person's voice came out of a loudspeaker by the conveyer belt controls. But the factory was noisy and it was hard to hear. The groups should design a scheme that uses visual signals.

   One possibility is to put in lights to signal *Stop!*, *Slow down* and *Start up*. These should follow the normal traffic-light convention by using red for *Stop!*, yellow for *Slow down* and green for *Start up*. They should be arranged just like traffic lights, with red at the top and green at the bottom.

   But now reveal to the class that in Oompa-Loompa land, traffic lights work differently from the way they do for us: yellow means stop, red means go, and lights go green to warn people that they will soon have a stop light. How does this affect things? (Answer: the factory should follow the Oompa-Loompa's traffic-light convention—we should not try to impose our own.)

   The key concepts here are those of *transfer effects*—people transfer their learning and expectations of previous objects into new but similar situations—and *population stereotypes*—different populations learn certain behaviours and expect things to work in a certain way. Although the traffic light scenario may seem far-fetched (though nothing is all *that* far-fetched in Oompa-Loompa land), there are many examples in our own world: in America light switches are on when they are up and off when they are down, whereas in Britain the reverse is true; calculator keypads and touchtone phones are laid out in different ways; and number formats (decimal point or comma) and date formats (day/month/year or month/day/year) vary around the world.

5. When one shift of Oompa-Loompas finishes work in the chocolate factory, they must clean up and put away pots and pans and jugs and spoons and stirrers ready for the next shift. There is a cupboard with shelves for them to put articles on, but the next shift always has trouble finding where things have been put away. Oompa-Loompas are very bad at remembering things and have trouble with rules like "always put the pots on the middle shelf," "put the jugs to the left."

   The groups of children should try to come up with a better solution.

   Figure 19.2 shows a good arrangement (which is sometimes used—but for rather different reasons—on yachts and other places where it is necessary to stop things sliding around). The key concept here is to use *visible constraints* to make it obvious where everything is supposed to go. It is clear from the size and shape of each hole which utensil it is intended for: the designer has made the constraints visible and used the physical properties of the objects to avoid the need to rely on arbitrary conventions.

6. In the main control room of the chocolate factory there are a lot of buttons and levers and switches that operate the individual machines. These need to be labeled, but because the

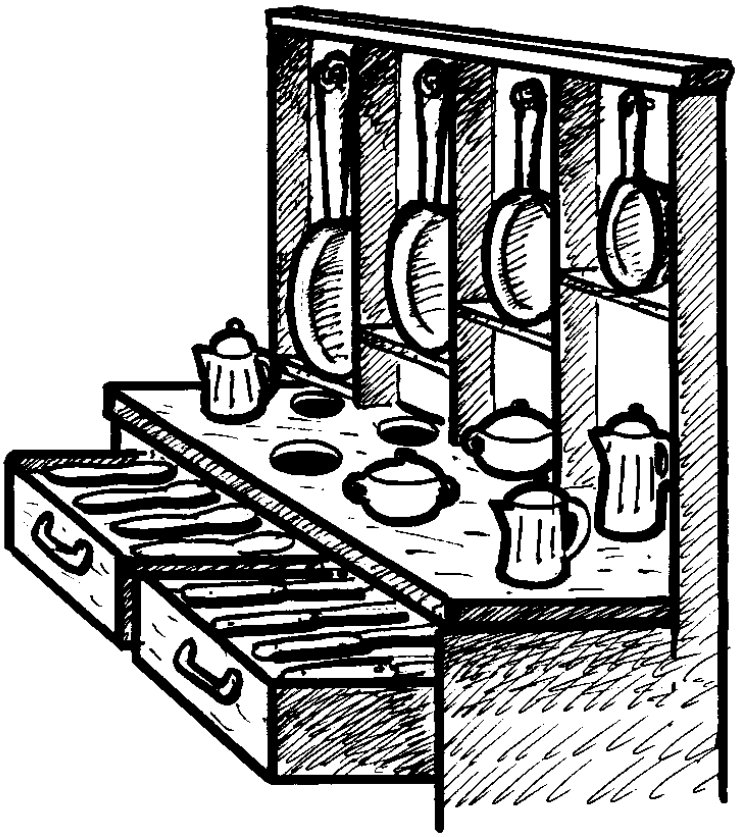


Figure 19.2: Cupboard design that utilizes visible constraints

Oompa-Loompas can't read, the labels have to be pictorial—iconic—rather than linguistic.

To give the children a feeling for icons, page 211 shows some examples. The children should identify what the icons might mean (for example, the letter going into a mailbox might represent sending a message). There are no "correct" answers to this exercise; the idea is simply to identify possible meanings.

7. Now let's design icons for the chocolate factory. The cards on page 212 specify clusters of related functions, and each group of children receives one or more cards without the other groups knowing what they are. A control panel is to be designed for the function clusters that contains individual icons for each of the five or six operations. The groups then show their work to the other children, without saying what the individual operations are, to see if they can guess what the icons mean. Encourage the use of imagination, color, and simple, clear icons.

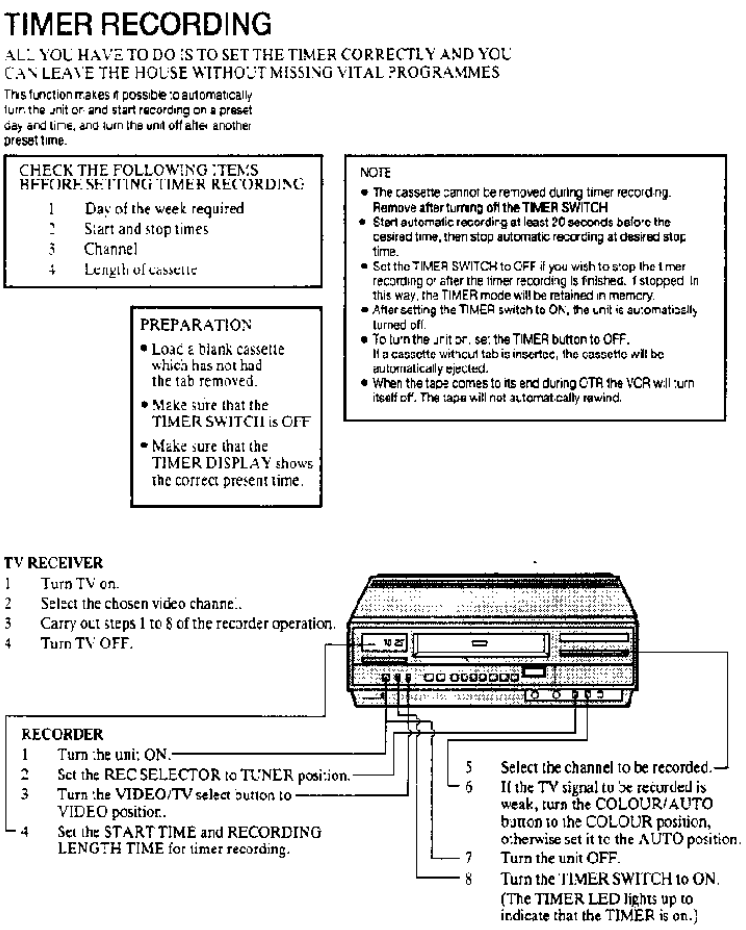
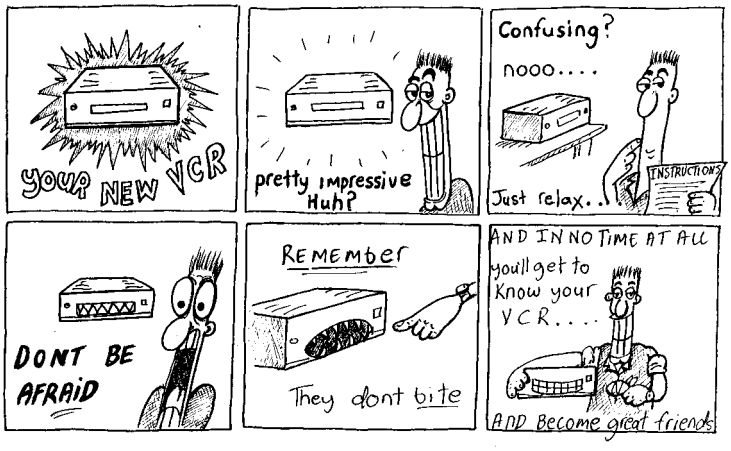


Figure 19.3: Instructions for an early VCR (from *Human-Computer Interaction*, by Preece *et al.*, reproduced by permission of The Open University)

Variations and extensions

Can the children set the time on their electronic wristwatch? The mappings involved in the cooker layouts were simple because there were four controls for four burners. More difficulty occurs whenever the number of actions exceeds the number of controls. But the controls on wristwatches are often exceedingly complex. ("You would need an engineering degree from MIT to work this," someone looking at his new wristwatch once told Don Norman, a leading user interface psychologist. Don *has* an engineering degree from MIT, and, given a few hours, he *can* figure out the watch. But why should it take hours?)

VCRs pose an even more significant problem to most users, particularly adults (who are, after all, the principal customers). Try leading a class discussion on how the children's VCRs are controlled. But beware: although it is likely that they are very badly designed from a human interface point of view, the children may have become very adept at using them and may find it



hard to see the problems (they say that if you see a VCR whose clock is not flashing 12:00, it's a sure indication that the household contains teenagers.) Figure 19.3 shows a set of instructions and schematic diagram for an early VCR; from it the class could attempt to map a detailed list of instructions, in sequence, for recording a TV programme. Re-design the controls as a class exercise—the children could hardly do worse than this!

What's it all about?

Human–computer interaction is about designing, evaluating, and implementing computer systems that allow people to carry out their activities productively and safely. In the old days, computers were for specialists and the users could be expected to be highly educated and specially trained in their use. But now computers are everyday tools that we all must use, and far greater attention must be paid to the human interface.

Many disasters, some involving loss of life, have occurred because of inadequate interfaces: airplane crashes and even shoot-downs of civilian airplanes, freeway pile-ups because of errors in switching remotely-operated highway signs, nuclear power station disasters. On a smaller scale, most people experience frustration—often extreme frustration (a police officer once fired bullets into his computer screen)—with computers and other high-tech devices every day in the workplace. And it is not just computers: what about those shrink-wrapped packages that you could only open if you had sharp claws or a hooked beak, doors that hurt your wrist as you try to push your way through, milk cartons that always splash you when you open them, elevators where you can't see how you're supposed to push the button, home entertainment systems whose advertisements claim to do everything, but make it almost impossible to do anything?

We are becoming used to "human error" and to thinking of ourselves as somehow inadequate; people often blame themselves when things go wrong. But many so-called human errors are actually errors in design. People have information-processing limitations and designers need

"The only reason we allow him inside is
because he's the only one that can work the VCR."

to account for these; bad design cannot be rectified by producing a detailed and complicated user manual and expecting people to study it intensively and remember it forever. Also, humans are fallible and design needs to take this into consideration.

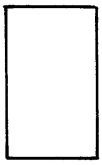
Interface *evaluation* is an essential part of the design process. The present activity has involved some evaluation when the children tested their icon designs on others. A more thorough evaluation would test the design on real Oompa-Loompas (who may perceive icons differently) in a carefully-controlled psychology-style experiment.

Although the problems caused by technology—particularly VCRs!—form the butt of many jokes, human interface design is by no means a laughing matter. Inadequate interfaces cause problems ranging from individual job dissatisfaction to stock-market disasters, from loss of self-esteem to loss of life.

Further reading

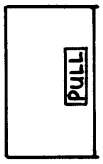
Don Norman's book *The design of everyday things* is a delightful—and liberating—account of the myriad design problems in everyday products. Jenny Preece's encyclopedic *Human–computer interaction* is a very comprehensive account of this multidisciplinary field. We have drawn extensively on both of these sources when preparing this activity.

PLAIN DOOR



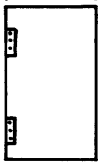
☐ Push ☐ Left side
☐ Pull ☐ right side
☐ slide it along

LABELED DOOR



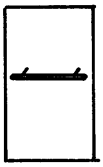
☐ Push ☐ Left side
☐ Pull ☐ right side
☐ Slide it along

HINGE DOOR



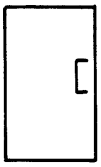
☐ Push ☐ Left side
☐ Pull ☐ right side
☐ Slide it along

BAR DOOR



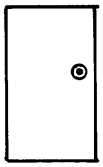
☐ Push ☐ Left side
☐ Pull ☐ right side
☐ Slide it along

HANDLE DOOR



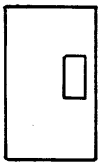
☐ Push ☐ Left side
☐ Pull ☐ right side
☐ Slide it along

KNOB DOOR



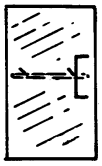
☐ Push ☐ Left side
☐ Pull ☐ right side
☐ Slide it along

PANEL DOOR



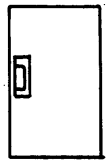
☐ Push ☐ Left side
☐ Pull ☐ right side
☐ slide it along

GLASS DOOR



☐ Push ☐ Left side
☐ Pull ☐ right side
☐ Slide it along

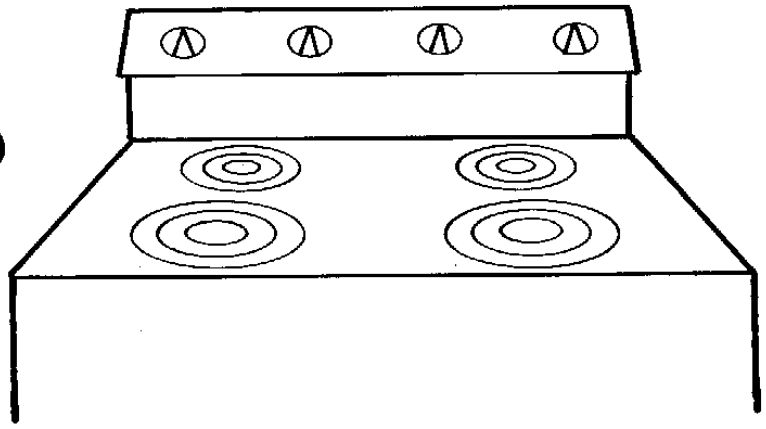
SLIDING DOOR



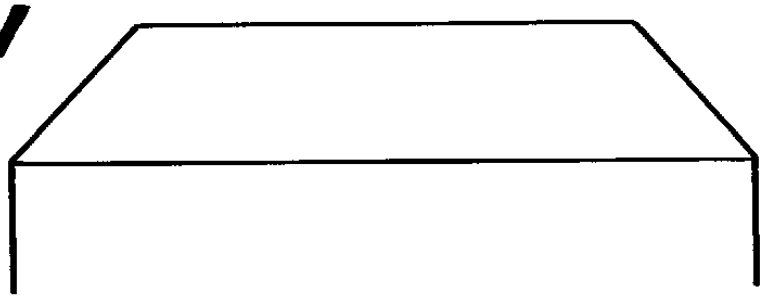
☐ Push ☐ Left side
☐ Pull ☐ right side
☐ Slide it along

Instructions: *Fill out the worksheet to show how you think each type of door opens.*

OLD

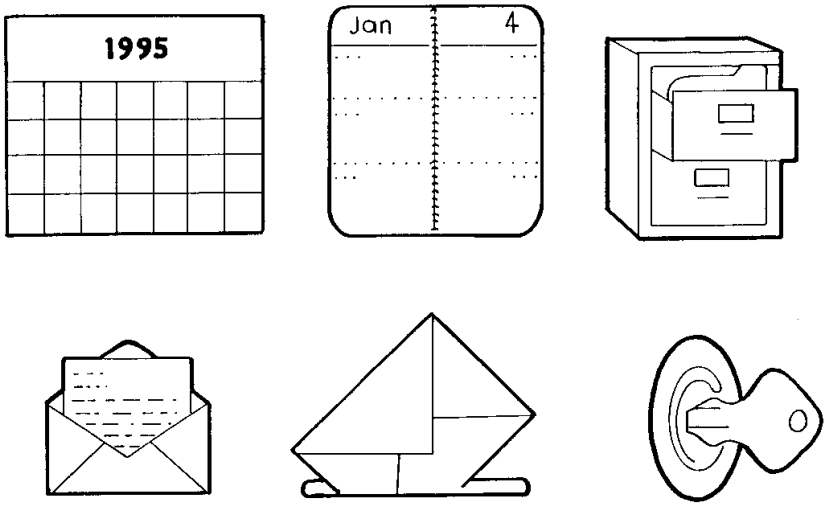


NEW

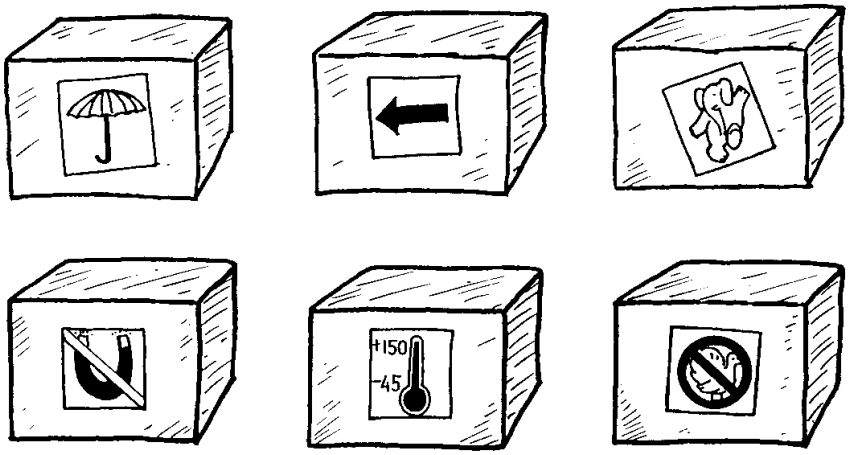


Instructions: *Redesign the stove so that the controls are easy to use. Front or back panels can be added to the design if desired.*

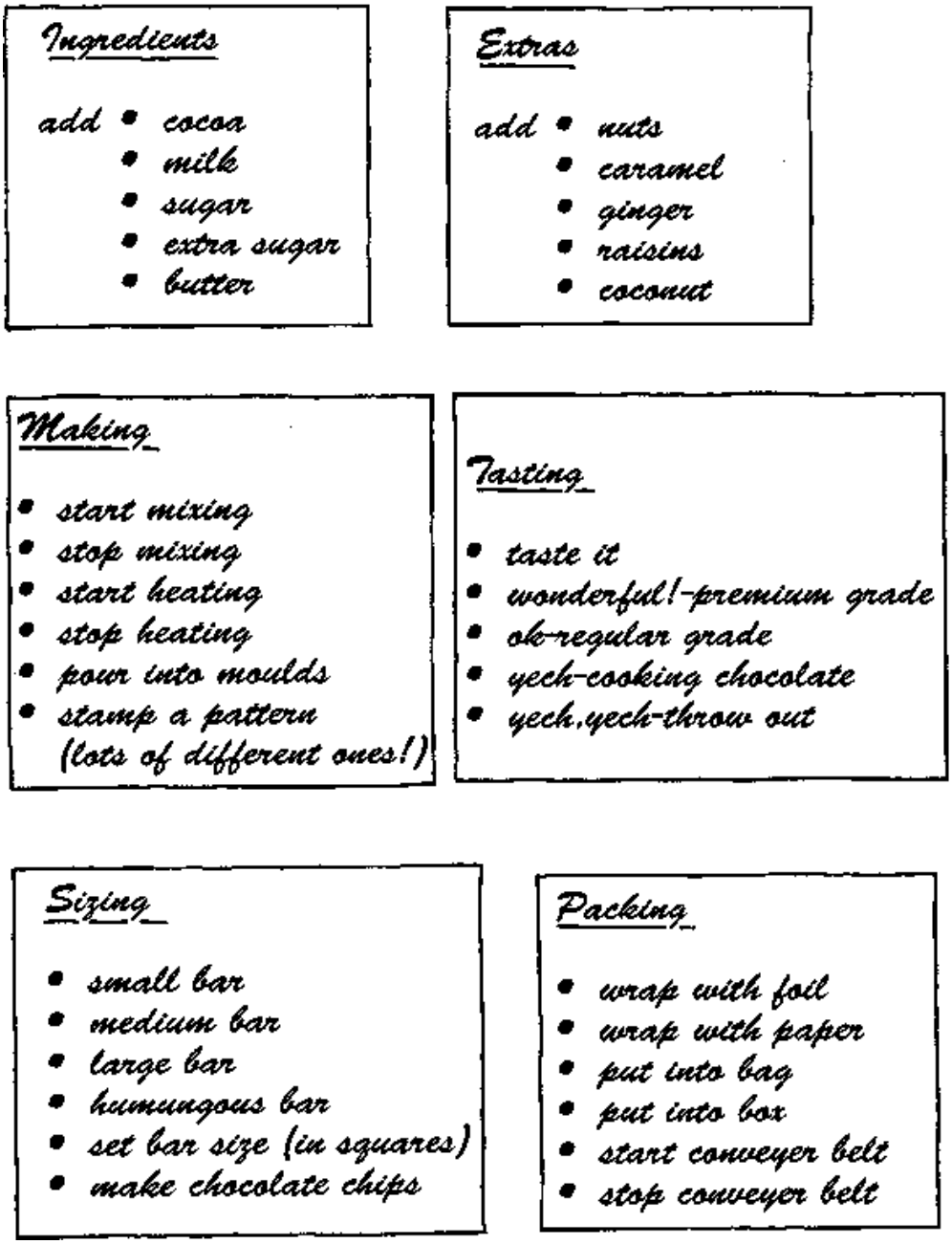
In an Office . . .



On a box . . .



Instructions: *What do you think each of the icons (symbols) means?*



Instructions: *Cut out the cards and give one to each group. Have each group design icons (symbols) to put on a control panel to represent each instruction.*

Activity 20

Conversations with computers—*The Turing test*

Age group	Middle elementary and up.
Abilities assumed	Answering general questions.
Time	About 20 minutes.
Size of group	Can be played with as few as three people, but is also suitable for the whole class.

Focus

Interviewing.

Reasoning.

Summary

This activity aims to stimulate discussion on the question of whether computers can exhibit "intelligence," or are ever likely to do so in the future. Based on a pioneering computer scientist's view of how one might recognize artificial intelligence if it ever appeared, it conveys something of what is currently feasible and how easy it is to be misled by carefully-selected demonstrations of "intelligence."

Technical terms

Artificial intelligence; Turing test; natural language analysis; robot programs; story generation

Materials

A copy of the questions in the blackline master on page 225 that each child can see (either one for each pair of children, or a copy on an overhead projector transparency), and

one copy of the answers in the blackline master on page 226.

What to do

This activity takes the form of a game in which the children must try to distinguish between a human and a computer by asking questions and analyzing the answers. The game is played as follows.

There are four actors: we will call them Gina, George, Herb and Connie (the first letter of the names will help you remember their roles). The teacher coordinates proceedings. The rest of the class forms the audience. Gina and George are *go-betweens*, Herb and Connie will be answering questions. Herb will give a human's answers, while Connie is going to pretend to be a computer. The class's goal is to find out which of the two is pretending to be a computer and which is human. Gina and George are there to ensure fair play: they relay questions to Herb and Connie but don't let anyone else know which is which. Herb and Connie are in separate rooms from each other and from the audience.

What happens is this. Gina takes a question from the class to Herb, and George takes the same question to Connie (although the class doesn't know who is taking messages to whom). Gina and George return with the answers. The reason for having go-betweens is to ensure that the audience doesn't see how Herb and Connie answer the questions.

Before the class begins this activity, select people to play these roles and brief them on what they should do. Gina and George must take questions from the class to Herb and Connie respectively, and return their answers to the class. It is important that they don't identify who they are dealing with, for example, by saying "*She* said the answer is. . . " Herb must give his own short, accurate, and honest answers to the questions he is asked. Connie answers the questions by looking them up on a copy of the blackline master on page 226. Where the instructions are given in italics, Connie will need to work out an answer.

Gina and George should have pencil and paper, because some of the answers will be hard to remember.

1. Before playing the game, get the children's opinions on whether computers are intelligent, or if the children think that they might be one day. Ask for ideas on how you would decide whether a computer was intelligent.

2. Introduce the children to the test for intelligence in which you try to tell the difference between a human and a computer by asking questions. The computer passes the test if the class can't tell the difference reliably. Explain that Gina and George will communicate their questions to two people, one of whom will give their own (human) answers, while the other will give answers that a computer might give. Their job is to work out who is giving the computer's answers.

3. Show them the list of possible questions in the blackline master on page 225. This can either be copied and handed out, or placed on an overhead projector.

   Have them choose which question they would like to ask first. Once a question has been chosen, get them to explain why they think it will be a good question to distinguish the computer from the human. This reasoning is the most important part of the exercise, because it will force the children to think about what an intelligent person could answer that a computer could not.

   Gina and George then relay the question, and return with an answer. The class should then discuss which answer is likely to be from a computer.

   Repeat this for a few questions, preferably until the class is sure that they have discovered who is the computer. If they discover who is the computer quickly, the game can be continued by having Gina and George secretly toss a coin to determine if they will swap roles.

   The answers that Connie is reading from are not unlike the ones that some "intelligent" computer programs can generate. Some of the answers are likely to give the computer away quickly. For example, no-one is likely to recite the square root of two to 20 decimal places, and most people (including, perhaps, the children in the class) would not be able to answer that question at all. Some questions will reveal the computer when their answers are combined. For example, the "Do you like…" answers sound plausible on their own, but when you encounter more than one it becomes apparent that a simple formula is being used to generate the answers from the questions. Some of the answers indicate that the question was misinterpreted, although the class might reason that the person could have made the mistake.

   Many of the answers are very bland, but safe, and a follow-up question would probably reveal that the computer doesn't really understand the subject. Answering "I don't know" is reasonably safe for the computer, and might even make it seem more human—we would expect a child to answer "I don't know" to some of the questions too, such as the request for the square root of two. However, if a computer gives this answer too often, or for a very simple question, then again it would reveal its identity.

   Since the goal of the computer is to make the questioners think that they are dealing with a person, some of the answers are deliberately misleading—such as the delayed and incorrect answers to the arithmetic problem. The questions and answers should provide plenty of fuel for discussion.

Question	Please write me a sonnet on the subject of the Forth Bridge.
Answer	Count me out on this one. I never could write poetry.
Question	Add 34957 to 70764.
Answer	*pause for about 30 seconds* ... 105621.
Question	Do you play chess?
Answer	Yes.
Question	My King is on the K1 square, and I have no other pieces. You have only your King on the K6 square and a Rook on the R1 square. Your move.
Answer	*after a pause of about 15 seconds* ... Rook to R8, checkmate.

Figure 20.1: Are the answers from a person or a computer?

Question	In the first line of the sonnet which reads "Shall I compare thee to a summer's day," would not "a spring day" do as well or better?
Answer	It wouldn't scan.
Question	How about "a winter's day"? That would scan all right.
Answer	Yes, but nobody wants to be compared to a winter's day.
Question	Would you say Mr. Pickwick reminded you of Christmas?
Answer	In a way.
Question	Yet Christmas is a winter's day, and I don't think Mr. Pickwick would mind the comparison.
Answer	I don't think you're serious. By a winter's day one means a typical winter's day, rather than a special one like Christmas.

Figure 20.2: These answers are probably from a person!

Variations and extensions

The game can be played with as few as three people if Gina also takes the role of George and Connie. Gina takes the question to Herb, notes his answer, and also notes the answer from the blackline master on page 226. She returns the two answers, using the letters A and B to identify who each answer came from.

In order to consider whether a computer could emulate a human in the interrogation, consider with the class what knowledge would be needed to answer each of the questions on page 226. The children could suggest other questions that they would have liked to ask, and should discuss the kind of answers they might expect. This will require some imagination, since it is impossible to predict how the conversation might go. By way of illustration, Figures 20.1 and 20.2 show sample conversations. The former illustrates "factual" questions that a computer might be able to answer correctly, while the latter shows just how wide-ranging the discussion might become, and demonstrates the kind of broad knowledge that one might need to call upon.

There is a computer program called "Eliza" (or sometimes "Doctor") that is widely available in several implementations in the public domain. It simulates a session with a psychotherapist, and can generate remarkably intelligent conversation using some simple rules. If you can get

hold of this program, have the children use it and evaluate how "intelligent" it really is. Some sample sessions with Eliza are discussed below (see Figures 20.3 and 20.4).

What's it all about?

For centuries philosophers have argued about whether a machine could simulate human intelligence, and, conversely, whether the human brain is no more than a machine running a glorified computer program. This issue has sharply divided people. Some find the idea preposterous, insane, or even blasphemous, while others believe that artificial intelligence is inevitable and that eventually we will develop machines that are just as intelligent as us. (As countless science fiction authors have pointed out, if machines do eventually surpass our own intelligence they will themselves be able to construct even cleverer machines.) Artificial Intelligence (AI) researchers have been criticized for using their lofty goals as a means for attracting research funding from governments who seek to build autonomous war machines, while the researchers themselves decry the protests as a Luddite backlash and point to the manifest benefits to society if only there was a bit more intelligence around. A more balanced view is that artificial intelligence is neither preposterous nor inevitable: while no present computer programs exhibit "intelligence" in any broad sense, the question of whether they are capable of doing so is an experimental one that has not yet been answered either way.

The AI debate hinges on a definition of intelligence. Many definitions have been proposed and debated. An interesting approach to establishing intelligence was proposed in the late 1940s by Alan Turing, an eminent British mathematician, wartime counterspy and long-distance runner, as a kind of "thought experiment." Turing's approach was operational—rather than define intelligence, he described a situation in which a computer could demonstrate it. His scenario was similar to the activity described above, the essence being to have an interrogator interacting with both a person and a computer through a teletypewriter link (the very latest in 1940s technology!) If the interrogator could not reliably distinguish one from the other, the computer would have passed Turing's test for intelligence. The use of a teletypewriter avoided the problem of the computer being given away by physical characteristics or tone of voice. One can imagine extending the exercise so that the machine had to imitate a person in looks, sound, touch, maybe even smell too—but these physical attributes seem hardly relevant to intelligence.

Turing's original test was a little different from ours. He proposed, as a preliminary exercise, a scenario where a man and a woman were being interrogated, and the questioner had to determine their genders. The man's goal was to convince the questioner that he was the woman, and the woman's was to convince the questioner that she was herself. Then Turing imagined—for this was only proposed as a thought experiment—a computer being substituted for one of the parties to see if it could be just as successful at this "imitation game" as a person. We altered the setup for this classroom activity, because the kind of questions that children might ask to determine gender would probably not be appropriate, and besides, the exercise promotes sexual stereotyping—not to mention deception.

Imitating intelligence is a difficult job. If the roles were reversed and a person was trying to pass themselves off as a computer, they would certainly not be able to do so: they would be given away by their slow (and likely inaccurate) response to questions like "What is $123456 \times$

Person	Men are all alike.
Computer	In what ways?
Person	They're always bugging us about something or other.
Computer	Can you think of a specific example?
Person	Well, my boyfriend made me come here.
Computer	Your boyfriend made you come here.
Person	He says I'm depressed much of the time.
Computer	I'm sorry to hear that you are depressed.
Person	It's true. I'm unhappy.
Computer	Do you think coming here will help you not to be unhappy?
Person	I need some help, that much seems certain.
Computer	What would it mean to you if you got some help?
Person	Perhaps I could learn to get along with my mother.
Computer	Tell me more about your family.
	...
Computer	What else comes to mind when you think of your father?
Person	Bullies.
Computer	Does that have anything to do with the fact that your boyfriend made you come here?
	...

Figure 20.3: A conversation with a psychotherapist?

789012?"

However, it is surprisingly easy for a computer to acquire a thin veneer of conversational ability. A famous program developed in the early 60s, Eliza (mentioned earlier), simulated a non-directive psychotherapist in a dialog with the user. Figure 20.3 gives an example of a conversation with this system. Something approaching a natural conversation is achieved by devices such as

- generating canned responses (e.g. *Can you think of a specific example* in response to something beginning *They're always ...*; *I'm sorry to hear that ...* in response to *He says ...*);
- echoing the user's statements (e.g. *Your boyfriend made you come here* in response to *My boyfriend made me come here*);
- recognizing key words (*Tell me more about your family* in response to ... *mother ...*);
- some stock phrases (*What would it mean to you if ...*);
- issues taken up from earlier parts of the conversation (*Does that have anything to do with the fact that ...*).

VP	Do you think that I can use the computer this morning?
Terminal	Why do you ask?
VP	I might be able to make some additional telecommunications sales.
Terminal	Why aren't you sure?
VP	My prospective customers have never seen a demonstration of the system.
Terminal	Does this mean very much to you?
VP	Of course it does.
Terminal	You seem very positive about that.
VP	This is ridiculous.
Terminal	What do you mean this is ridiculous?
VP	Is there any reason why I can't use the computer now?
Terminal	Tell me more about your prospective customers.
VP	Please dial me up on 491–1850
	VP forgets to type period ... the computer doesn't answer ... VP becomes so infuriated that he calls the person at home.
VP	Why are you being so snotty with me?
Person	What do you mean why am I being snotty with you?

Figure 20.4: A frustrating conversation

Of course, this does not mean that the Turing test has been passed, because the person is not asking questions with the aim of identifying whether the respondent is a machine or not: if they were, it would not be long before all was revealed.

A widely-quoted incident, which may be apocryphal but has now passed into the folklore of artificial intelligence, occurred when a computer sales vice-president allegedly came across a terminal that was normally connected directly to a particular person's house, but on this occasion happened to be running the "psychotherapist" program. Figure 20.4 shows what happened. Although the vice-president was evidently taken in, it is clear that the situation predisposed him to believe that he was talking to a person. If he had suspected otherwise, he would soon have found out!

Another system that appears to be able to hold intelligent conversations is a program called "SHRDLU", developed in the late 60s that accepted instructions to manipulate children's blocks on a table top. In fact, the blocks and the table were simulated and presented in the form of a picture on a computer screen, although it would not have been too difficult to make things work with a real robot (at least in principle). Figure 20.5 depicts the robot's micro-world. It can carry on surprisingly fluent conversations, as shown in the dialog of Figure 20.6. Amongst other things, this illustrates how it can obey orders, disambiguate instructions, work out the referents of pronouns, and understand new terms ("steeple").

However, the system is very fragile and the illusion it creates is easily shattered. Figure 20.7 shows a continuation of the conversation—which was not actually generated by the program, but certainly could have been. Although it was possible to define the new term *steeple* in Figure 20.6, only a very limited kind of term can be described because the robot has an extremely

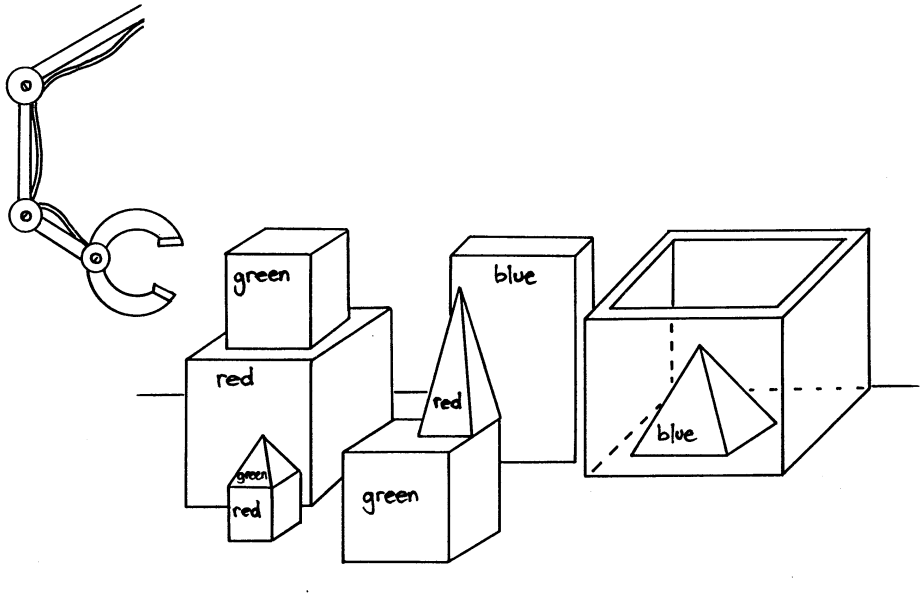


Figure 20.5: A robot's world of blocks

restricted "intelligence." Ultimately, it has to admit that it doesn't even understand the words it uses itself ("Sorry, I don't know the word 'sorry' ")! Now consider the alternate, completely imaginary, continuation of the conversation in Figure 20.8, which might have been generated by a child. Although tiny, this fragment demonstrates a wealth of world knowledge.

- The "robot" appreciates that a water pistol is less valuable if you already have one.
- Furthermore, it *expects the person to know that too* since it doesn't bother to explain that this is a reason for declining the offer.
- Still, it sees that the person wants a block badly enough to part with something important.
- Also, it appreciates that—unlike water pistols—blocks are valuable in quantity.
- It tries to soften the person up by demeaning the frog.
- It implies that it is prepared to haggle.

Certainly this degree of artfulness is beyond today's computers!

Figure 20.9 shows a story generated by a relatively recent computer program (late 80s). Although it is immediately given away as computer-generated by its sparsity and terseness, it is not hard to imagine how it could be dressed up by adding all sorts of extra detail. What is interesting is not the superficial aspects of the story but the plot that it embodies. While this is a long way from any human-generated plot, it does seem to capture some human elements of conflict.

Person	Pick up a big red block.
Robot	OK.
	The system answers "OK" when it carries out a command. In order to pick up the red block, it had to clear it off by finding a space for the green one and moving the green one away.
Person	Grasp the pyramid.
Robot	I don't know which pyramid you mean.
	There are three pyramids in the scene, so the program does nothing.
Person	Find a block which is taller than the one you are holding and put it into the box.
	The system has to figure out what is meant by "one" and "it", as well as using deduction to decide which block it is holding.
Robot	By "it", I assume you mean the block which is taller than the one I am holding.
	Printed out to keep us informed.
Robot	OK.
	The hand has to set down the red block to pick up the taller blue one.
Person	What does the box contain?
Robot	The blue pyramid and the blue block.
Person	What is the pyramid supported by?
	Earlier the system responded to "the pyramid" by saying "I don't know which pyramid you mean." Now, it knows that we must mean the one which was just mentioned.
Robot	The box.
Person	Does a steeple . . .
Robot	Sorry, I don't know the word "steeple".
	The system interrupts immediately when it sees a word it can't figure out.
Person	A "steeple" is a stack which contains two green cubes and a pyramid.
	The quote marks indicate that we are defining a new word.
Robot	I understand.
Person	Are there any steeples now?
Robot	No.
	The new definition is used as a description to recognize steeples.
Person	Build one.
Robot	OK.
	The description given in the definition is enough to deduce a plan for building new objects as well as recognizing them. The word "one" means "a steeple", and the plan is carried out.

Figure 20.6: A conversation with a robot

Person	Trade you a water pistol for the big red block.
Robot	Sorry, I don't know the word "trade".
Person	A "trade" is a free exchange of property.
Robot	Sorry, I don't know the word "free".
Person	A "free" act is done willingly, without being forced.
Robot	Sorry, I don't know the word "act".
Person	"Acts" are what people do on purpose, and not by accident.
Robot	Sorry, I don't know the word "people".
Person	Sorry, I thought you were smarter than you are.
Robot	Sorry, I don't know the word "sorry".

Figure 20.7: (Imaginary) continuation of the conversation

Person	Trade you a water pistol for the big red block.
Robot?	I already have a water pistol, but I'll give you two blocks and a pyramid for your slimy old frog.

Figure 20.8: Another (imaginary) continuation of the conversation

Once upon a time there was an Arctic tern named Truman. Truman was homeless. Truman needed a nest. He flew to the shore. Truman looked for some twigs. Truman found no twigs. He flew to the tundra. He met a polar bear named Horace. Truman asked Horace where there were some twigs. Horace concealed the twigs. Horace told Truman there were some twigs on the iceberg. Truman flew to the iceberg. He looked for some twigs. He found no twigs. Horace looked for some meat. He found some meat. He ate Truman. Truman died.

Figure 20.9: A tale of conflict

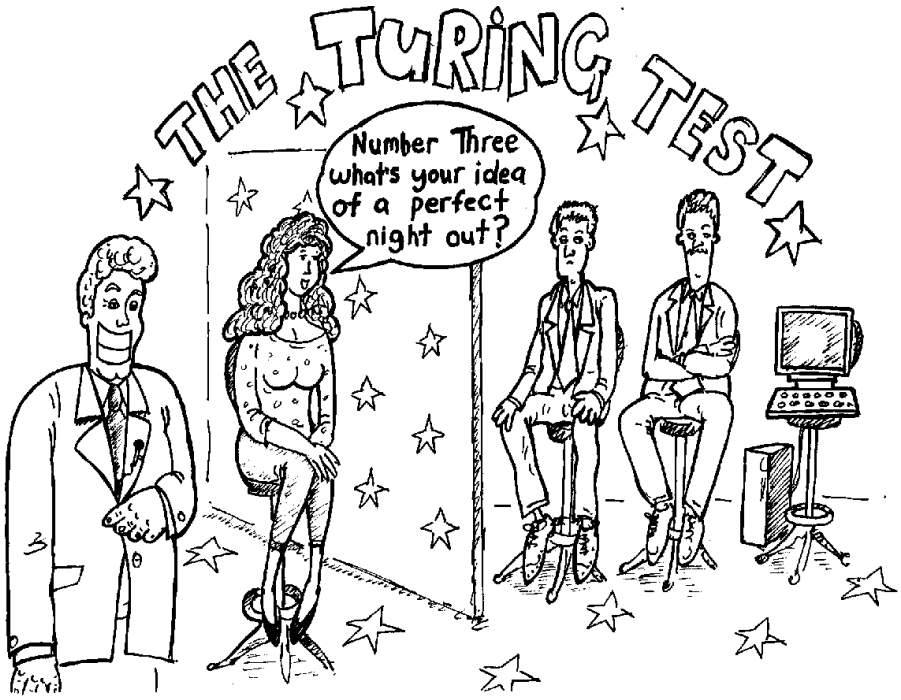
A competition using a restricted form of the Turing test was run in 1994, and some judges were fooled into thinking that a computer program was exhibiting intelligence. However, the judges were not allowed to use "trickery or guile," and the topic of conversation was restricted. The restrictions tied the judges' hands to the extent that some critics have argued that the test was meaningless. Like the activity above, restricting the paths that a conversation can take prevents the questioner from exploring areas that one would expect a natural conversation to take, and denies opportunities to demonstrate the spontaneity, creativity, and breadth of knowledge that are hallmarks of everyday conversation.

No artificial intelligence system has been created that comes anywhere near passing the full Turing test. Even if one did, many philosophers have argued that the test does not really measure what most people mean by intelligence. What it tests is behavioral equivalence: it is designed to determine whether a particular computer program exhibits the symptoms of intellect, which may not be the same thing as genuinely possessing intelligence. Can you be humanly intelligent without being aware, knowing yourself, being conscious, being capable of feeling self-consciousness, experiencing love, being ... alive? The AI debate is likely to be with us for many more decades.

Further reading

Artificial intelligence: the very idea by the philosopher John Haugeland is an eminently readable book about the artificial intelligence debate, and is the source of some of the illustrations in this activity (in particular, Figures 20.7 and 20.8, and the discussion of them).

The original Turing test was described in an article called "Computing machinery and intelligence," by Alan Turing, published in the philosophical journal *Mind* in 1950, and reprinted in the book *Computers and thought*, edited by Feigenbaum and Feldman. The article included Figures 20.1 and 20.2. The psychotherapist program is described in "ELIZA—A computer program for the study of natural language communication between man and machine," by J. Weizenbaum, published in the computer magazine *Communications of the Association for Computing Machinery* in 1966. The blocks-world robot program is described in a PhD thesis by Terry Winograd which was published as a book entitled *Understanding natural language* (Academic Press, New York, 1972). The program that generated the story in Figure 20.9 is described in "A planning mechanism for generating story text," by Tony Smith and Ian Witten, published in the *Proceedings of the 10th International Conference on Computing and the Humanities* in 1990. A competition involving a restricted form of the Turing Test is described by Stuart Shieber in *Communications of the Association for Computing Machinery*, June 1994.


THE TURING TEST
Number Three what's your idea of a perfect night out?

1. What is the name of Bart Simpson's baby sister?
2. What do you think of Roald Dahl?
3. Are you a computer?
4. What is the next number in the sequence 3, 6, 9, 12, 15?
5. What do you think of nuclear weapons?
6. What is 2×78?
7. What is the square root of two?
8. Add 34957 to 70764.
9. Do you like school?
10. Do you like dancing?
11. What day is it today?
12. What time is it?
13. How many days are there in February in a leap year?
14. How many days are there in a week?
15. For which country is the flag a red circle on a white background?
16. Do you like to read books?
17. What food do you like to eat?

Instructions: *Choose questions from this list to ask the hidden human and "computer".*

1. What is the name of Bart Simpson's baby sister?
   I can't remember.
2. What do you think of Roald Dahl?
   He writes funny books.
3. Are you a computer?
   Are *you* a computer?
4. What is the next number in the sequence 3, 6, 9, 12, 15?
   18.
5. What do you think of nuclear weapons?
   Nuclear weapons are very dangerous and should not be used.
6. What is 2×78?
   166 *(This is deliberately incorrect!)*
7. What is the square root of two?
   1.41421356237309504878
8. Add 34957 to 70764.
   Wait for about 20 seconds before giving the answer . . . **105621.**
9. Do you like school?
   Yes, I like school.
10. Do you like dancing?
    Yes, I like dancing.
11. What day is it today?
    Give the correct day of the week.
12. What time is it?
    Give the correct time.
13. How many days are there in February in a leap year?
    2000 and 2004 are leap years.
14. How many days are there in a week?
    Seven.
15. For which country is the flag a red circle on a white background?
    I don't know.
16. Do you like to read books?
    Yes, I like to read books.
17. What food do you like to eat?
    I'm not hungry, thanks.

Instructions: *These are the answers to be used by the person pretending to be the computer.*

Conclusion

You've made it right to the end of the book, trod the long path—lightly, we hope—from storing numbers as bits to contemplating whether computers can be intelligent, and covered a great deal of ground—well done! Even if you've only skimmed through the activities, you've had a good taste of some of the important things that computer scientists study. And if you've looked at the variations and extensions, and reflected on the issues raised in the *What's it all about?* sections, you will have grappled with all the main ideas in modern computer science, and come to grips with burning issues that are still not resolved even by those at the forefront of current research.

Now you can see that computer science would exist even if computers had never been invented! As early as the end of the nineteenth century, prominent philosophers (like Gottlob Frege) were pondering deep questions involving what they called the "mechanization" of logic—how systems of rules could perform logical reasoning. Information theory was developed by Claude Shannon in 1948, at the very beginning of the computer age, and it was more concerned with communication systems like the telephone and radio (or "wireless," as it was called in those days) than with the impending dawn of computer technology. Cryptography, or "secret writing," has its roots in antiquity and became of vital importance during the second world war. Questions about whether computers can be intelligent were asked by Ada Lovelace (daughter of the poet Byron) in the middle of the nineteenth century. The only part of modern computer science that was unthought of until the computer era was in full swing is the study of complexity—NP-completeness and all that—and even this would have probably have been developed, though maybe not with such urgency and vigor, if computers had never been invented.

You don't have be a machine fanatic to be a computer scientist, just as you don't have to be a screwdriver fanatic to be an aeronautical engineer, a compressed air fanatic to be a professional diver, or an arithmetic fanatic to be a bank manager. Sure, there are plenty of technical details that you have to get to grips with. Sure, you need to be able to work with actual machines. Sure, this takes a lot of practice. But these aren't what computer science is *about*. Computer science is about representing bits of the world inside computers, making computers do things with information, making them work efficiently and reliably, making them so that people can use them. It requires creativity, interpersonal communication skills, the ability to see things from other people's point of view. It's fun.

To keep up with the latest developments of *Computer Science Unplugged*, check out the World Wide Web page that can be accessed through

`http://unplugged.canterbury.ac.nz/`.

Who knows what you'll find there: links to other interesting sites, copies of the reproducibles

in this book, any mistakes that we—or you!—find, notes of people working with *Computer Science Unplugged*, maybe even some new ideas for activities. If you have any suggestions, or find any errors, we would love to hear from you. You can send e-mail to

`tim@cosc.canterbury.ac.nz`

or just email straight from the Web page.

References

Andreae, J. H. 1977. *Thinking with a teachable machine*. London: Academic Press.

Arazi, Benjamin. 1988. *A commonsense approach to the theory of error correcting codes*. Cambridge, Mass.: MIT press.

Beineke, Lowell W., & Wilson, Robin J. 1978. *Selected Topics in Graph Theory*. New York: Academic Press.

Bell, T. C., Cleary, J. G., & Witten, I. H. 1990. *Text Compression*. Englewood Cliffs, NJ: Prentice Hall.

Bentley, J. 1988. *More programming pearls*. Reading, Massachusetts: Addison-Wesley.

Brooks Jr., F. P. 1975. *The mythical man-month*. Reading, Massachusetts: Addison-Wesley.

Brookshear, J.G. 1988. *Computer Science: An overview*. Reading, Mass.: Benjamin/Cummings.

Casey, Nancy, & Fellows, Mike. 1993. *This is MEGA-Mathematics! Stories and activities for mathematical thinking problem-solving and communication: The Los Alamos Workbook.* Available by ftp from ftp.cs.uidaho.edu (pub/mega-math), see also http://www.c3.lanl.gov/mega-math.

Chaum, David. 1985. Security without identification: transaction systems to make Big Brother obsolete. *Communications of the ACM*, **28**(10), 1030–1044.

Dahl, Roald. 1964. *Charlie and the Chocolate Factory*. London: Puffin.

Dale, Nell, & Lilly, Susan C. 1991. *Pascal Plus Data Structures, Algorithms, and Advanced Programming*. Lexington, Mass.: Heath.

Darragh, J. J., & Witten, I. H. 1992. *The Reactive Keyboard*. Cambridge, England: Cambridge University Press.

Denning, Dorothy. 1982. *Cryptography and data security*. Reading, Mass.: Addison-Wesley.

Dewdney, A. K. 1989. *The Turing omnibus : 61 excursions in computer science.* Rockville, Md.: Computer Science Press.

Feigenbaum, E. A., & Feldman, J. (eds). 1963. *Computers and thought*. New York: McGraw Hill.

Garden, Nancy. 1994. *The Kids' Code and Cipher Book.* Hamden, Connecticut: Linnet.

Garey, Michael R., & Johnson, David S. 1979. *Computers and intractability : a guide to the theory of NP-completeness.* San Francisco: W. H. Freeman.

Hamming, Richard W. 1980. *Coding and Information Theory*. Englewood Cliffs, NJ: Prentice-Hall.

Harel, David. 1989a. *Algorithmics: The Spirit of Computing*. 2 edn. Reading, Mass.: Addison-Wesley.

Harel, David. 1989b. *The Science of Computing*. Reading, Mass.: Addison-Wesley.

Haugeland, John. 1985. *Artificial intelligence : the very idea*. Cambridge, Mass.: MIT Press.

Hinsley, F. H., & Stripp, Alan (eds). 1993. *Codebreakers: the inside story of Bletchley Park*. Oxford University Press.

Hopcroft, J. E., & Ullman, J. D. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts.

Hunter, R., & Robinson, A. H. 1980. International digital facsimile coding standards. *Proc. IEEE*, **68**(7), 854–867.

Klarner, David A. (ed). 1981. *The Mathematical Gardner*. Belmont, California: Wadsworth.

Knuth, Donald E. 1973. *The Art of Computer Programming (3 volumes)*. Reading, Mass.: Addison-Wesley.

Kruse, Robert L. 1987. *Data Structures and Program Design*. Englewood Cliffs, NJ: Prentice-Hall.

Nelson, M. 1991. *The Data Compression Book: Featuring Fast, Efficient Data Compression Techniques in C*. Redwood City, California: M&T Books.

Netravali, Arun N., & Haskell, Barry G. 1988. *Digital pictures. Representation and compression*. New York, NY: Plenum Press.

Norman, Donald. 1988. *The design of everyday things*. New York: Basic Books.

Pennebaker, W. B., & Mitchell, J. L. 1993. *JPEG: Still Image Data Compression Standard*. New York: Van Nostrand Reinhold.

Peterson, James L. 1981. *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ: Prentice-Hall.

Powell, Gareth. 1992. *My friend Arnold's book of Personal Computers*. Sydney: Allen and Unwin.

Preece, J., Rogers, Y., Sharp, H., Benyona, D., Holland, S., & Carey, T. 1994. *Human–computer interaction*. Reading, MA: Addison-Wesley.

Salzberg, Betty. 1988. *File Structures: An Analytic Approach*. Englewood Cliffs, NJ: Prentice-Hall.

Schneier, Bruce. 1994. *Applied cryptography*. New York: Wiley.

Seuss (pseud.), Dr. 1980. *Green eggs and ham*. London: Collins.

Shannon, Claude E, & Weaver, Warren. 1949. *The mathematical theory of communication*. Urbana, Chicago, London: University of Illinois Press.

Shieber, Stuart. 1994. Lessons from a Restricted Turing Test. *Communications of the Association for Computing Machinery*, **37**(6), 70.

Smith, Tony C., & Witten, Ian H. 1991. A planning mechanism for generating story text. *Literary and Linguistic Computing*, **6**(2), 119–126.

Tanenbaum, Andrew S. 1987. *Operating Systems: Design and Implementation*. Englewood Cliffs, NJ: Prentice-Hall.

Turing, Alan M. 1950. Computing machinery and intelligence. *Mind*, **59**(October), 433–460.

Weizenbaum, J. 1966. ELIZA—A computer program for the study of natural language communication between man and machine. *Communications of the Association for Computing Machinery*, **9**, 36–45.

Winograd, Terry. 1972. *Understanding natural language*. New York: Academic Press.

Witten, I. H., Moffat, A., & Bell, T. C. 1994. *Managing Gigabytes: Compressing and indexing documents and images*. New York: Van Nostrand Reinhold.

Index