

OXFORD

COMPUTERS LTD

WHAT THEY REALLY CAN'T DO

DAVID HAREL

Computers Ltd.

What they *really* can't do

David Harel

*Faculty of Mathematics and Computer Science
The Weizmann Institute of Science, Rehovot, Israel*

OXFORD
UNIVERSITY PRESS

This One



W4Q1-C4D-41CG

Copyrighted material

OXFORD

UNIVERSITY PRESS

Great Clarendon Street, Oxford OX2 6DP

Oxford University Press is a department of the University of Oxford.
It furthers the University's objective of excellence in research, scholarship,
and education by publishing worldwide in

Oxford New York

Auckland Bangkok Buenos Aires Cape Town
Chennai Dar es Salaam Delhi Hong Kong Istanbul Karachi
Kolkata Kuala Lumpur Madrid Melbourne Mexico City Mumbai
Nairobi São Paulo Shanghai Taipei Tokyo Toronto

Oxford is a registered trade mark of Oxford University Press
in the UK and in certain other countries

Published in the United States by
Oxford University Press Inc., New York

© D. Harel 2000

The moral rights of the author have been asserted

Database right Oxford University Press (maker)

First published 2000

First published as an

Oxford University Press paperback 2003

Reprinted with corrections 2004

All rights reserved. No part of this publication may be reproduced,
stored in a retrieval system, or transmitted, in any form or by any means,
without the prior permission in writing of Oxford University Press,
or as expressly permitted by law, or under terms agreed with the appropriate
reprographics rights organization. Enquiries concerning reproduction
outside the scope of the above should be sent to the Rights Department,
Oxford University Press, at the address above

You must not circulate this book in any other binding or cover
and you must impose this same condition on any acquirer

British Library Cataloguing in Publication Data

Data available

Library of Congress Cataloguing in Publication Data

Data available

ISBN 0 19 860 442 4

2

Typeset by Footnote Graphics Limited, Warminster, Wilts

Printed in Great Britain by

Clays Ltd., St. Ives plc

Contents

- 1 What's it all about? 1
 - Algorithms 2
 - Basic instructions 5
 - The text vs. the process 7
 - Inputs 9
 - What do algorithms solve? 10
 - Isn't our setup too simplistic? 15
 - Solving algorithmic problems 16
 - Programming 18
 - Errors and correctness 21
 - Termination 25

- 2 Sometimes we can't do it 27
 - Finite problems are solvable 29
 - The tiling problem 30
 - Do we really mean it? 33
 - Elementary computing devices 36
 - The Church–Turing thesis 40
 - Computability is robust 42
 - Domino snakes 46
 - Program verification 48
 - The halting problem 50
 - Nothing about computation can be computed! 53
 - Some problems are even worse 54

- 3 Sometimes we can't afford to do it 59
 - Resources: time and memory space 60
 - Improving running time 61
 - Upper and lower bounds 65
 - So what? 69
 - The towers of Hanoi 69
 - The good, the bad, and the ugly 73
 - Intractability 77
 - Roadblocks and chess 81
 - Problems that are even harder 85
 - Unreasonable memory requirements 88

- 4 Sometimes we just don't know 91
 - The monkey puzzle 92
 - NP-complete problems 95
 - Finding short paths 97
 - Scheduling and matching 100
 - More on puzzles 103
 - Coloring networks 105
 - Magic coins 107
 - Standing of falling together 109
 - The great mystery: is P equal to NP? 111
 - Can we come close? 113
 - Sometimes we succeed 115

- 5 Trying to ease the pain 119
 - Parallelism, or joining forces 121
 - Can parallelism eliminate the bad news? 125
 - Randomization, or tossing coins 129
 - More on Monte Carlo algorithms 132
 - Testing for primality 134
 - Randomized primality testing 137
 - Can randomization eliminate the bad news? 141
 - Can computers simulate true randomness? 142
 - Quantum computing 143

	Quantum algorithms	147
	Can there be a quantum computer?	151
	Molecular computing	153
6	Turning bad into good	157
	Classical cryptography	158
	Public-key cryptography	161
	Signing messages	165
	Can this be made to work?	168
	The RSA cryptosystem	171
	Interactive proofs	173
	Zero-knowledge proofs	178
	I can 3-color a network	181
	On millionaires, ballots, and more	187
7	Can we ourselves do any better?	191
	Algorithmic intelligence?	193
	The Turing test	194
	ELIZA and zupchoks	198
	Heuristics	201
	What is knowledge?	206
	Understanding natural language	210
	Postamble	215
	Index	217

Chapter 1

What's it all about?

Computers are amazing. They seem to have it all. They fly aircraft and spaceships, and control power stations and hazardous chemical plants. Companies cannot be run without them, and many medical procedures cannot be performed in their absence. They serve lawyers and judges who seek judicial precedents, and help scientists and engineers to perform immensely involved mathematical computations. They route and control millions of simultaneous telephone calls and manage the remarkable movement of Internet data in enormous global networks. They execute tasks with great precision – from map-reading and typesetting to image processing, robot-aided manufacturing and integrated circuit design. They help individuals in many boring daily chores and at the same time provide entertainment through computer games or the delight of surfing the Web. Moreover, the computers of today are hard at work helping design the even more powerful computers of tomorrow.

It is all the more remarkable, therefore, that the digital computer – even the most modern and complex one – is merely a large collection of switches, called **bits**, each of which can be on or off. On is denoted by 1 and off by 0. Typically, the value

of a bit is determined by some electronic characteristic, such as whether a certain point has a positive or negative charge. In a technical sense, a computer can really execute only a small number of extremely simple operations on bits, like flipping a bit's value, zeroing it, or testing it (that is, doing one thing if the bit is on and another if it is off).

Computers may differ in size, i.e. in the number of bits available, and in internal organization, as well as in the types of elementary operations allowed and the speed at which they are performed. They can also differ in outward appearance and in their connections with the external world. However, appearances are peripheral when compared to the bits and their internal arrangement. It is the bits that 'sense' the input stimuli arriving from the outside world, and it is the bits that 'decide' how to react to them by output stimuli. The inputs can arrive via keyboards, touch screens, control panels, electronic communication lines, or even microphones, cameras, and chemical sensors. The outputs are fed to the outside world via display screens, communication lines, printers, loudspeakers, beepers, robot arms, or whatever.

How do they do it? What is it that transforms simple operations like flipping zeros and ones into the incredible feats computers perform? The answer lies in the concepts that underlie the science of computing: the computational process, and the algorithm, or program, that causes it to take place.

Algorithms

Imagine a kitchen, containing a supply of ingredients, an array of baking utensils, an oven, and a (human) baker. Baking is a process that *produces* a cake, *from* the ingredients, *by* the baker, *aided by* the oven, and, most significantly, *according to*

the recipe. The ingredients are the **input** to the process, the cake is its **output**, and the recipe is the **algorithm**. In the world of electronic computation, the recipes, or algorithms, are embodied in **software**, whereas the utensils and oven represent the **hardware**. See Fig. 1.1.

Just like computers carrying out bit operations, the baker with his or her oven and utensils has very limited direct abilities. This cake-baking hardware can pour, mix, spread, drip, knead, light the oven, open the oven door, measure time, measure quantities, etc. It cannot directly bake cakes. The recipes – those magical prescriptions that convert the limited abilities of novice bakers and kitchen hardware into cakes – are at the heart of the matter; not the ovens or the bakers.

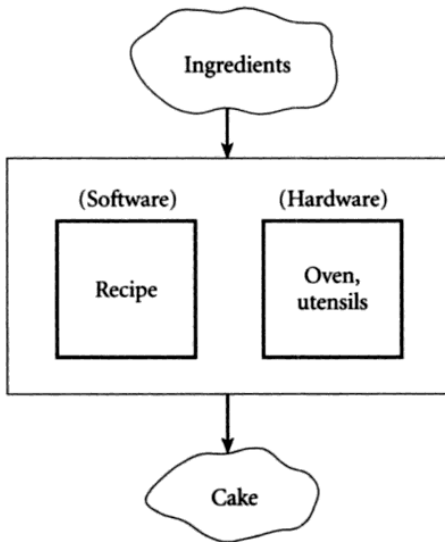


Fig. 1.1. Baking a cake.

In our world, recipes are called algorithms, and the study, knowledge, and expertise that concerns algorithms has been termed **algorithmics**.¹

The analogy with cooking can be understood as follows: the recipe, which is an abstract entity, is the algorithm; the formal written version of the recipe, such as is found in a particular cookbook, is analogous to a **computer program** – the precise representation of an algorithm, written in a special computer-readable formalism called a **programming language**. It is important to realize that, just as a recipe remains the same whether written in English, French, or Latin, and regardless of where and by whom it is carried out, so does an algorithm remain the same whether written in Fortran, C, Cobol, or Java, and regardless of the computer it runs on, be it an ultra-light laptop or a room-size main-frame. The generic term **software** actually refers to programs rather than to the abstract notion of algorithms, since software is written for real computers. However,

¹ The word 'algorithm' is derived from the name of the Arabic/Persian mathematician of the ninth century, Mohammed al-Khowârizmî, who is credited with providing the step-by-step rules for carrying out the fundamental operations of decimal arithmetic. In Latin the name became *Algorismus*, from which 'algorithm' is derived. Historically, the first nontrivial algorithm was invented somewhere between 400 and 300 BC by the great Greek mathematician Euclid. The **Euclidian algorithm**, as it is called, finds the greatest common divisor (gcd) of two positive integers, i.e. the largest integer that exactly divides them both. For example, the gcd of 80 and 32 is 16. The word 'algorithmics' was apparently coined by J. F. Traub (1964). *Iterative Methods for the Solution of Equations*, Prentice Hall. It was proposed as the name for the relevant field of study by D. E. Knuth (1985). 'Algorithmic Thinking and Mathematical Thinking', *American Math. Monthly* 92, 170–181, and by the present author in *Algorithmics: The Spirit of Computing*, Addison-Wesley (1987).

we shall blur the distinction, since the story told in the following chapters applies just as well to both.

Basic instructions

Let us take the gastronomical analogy a little further. Here is a recipe for chocolate mousse.² The ingredients – that is, the recipe's input – include 8 ounces of semi-sweet chocolate pieces, 2 tablespoons of water, a 1/4 cup of powdered sugar, 6 separated eggs, and so on. The output is described as six to eight servings of delicious *mousseline au chocolat*:

Melt chocolate and 2 tablespoons water in double boiler. When melted, stir in powdered sugar; add butter bit by bit. Set aside. Beat egg yolks until thick and lemon-colored, about 5 minutes. Gently fold in chocolate. Reheat slightly to melt chocolate, if necessary. Stir in rum and vanilla. Beat egg whites until foamy. Beat in 2 tablespoons sugar; beat until stiff peaks form. Gently fold whites into chocolate-yolk mixture. Pour into individual serving dishes. Chill at least 4 hours. Serve with whipped cream, if desired. Makes 6 to 8 servings.

This is the 'software' relevant to the preparation of the mousse; it is the algorithm that prescribes the process that produces mousse from the ingredients. The process itself is carried out by the person preparing the mousse, together with the 'hardware', in this case the various utensils: double boiler, heating apparatus, beater, spoons, timer, and so on.

One of the basic instructions, or basic actions, present in this recipe is '*stir in powdered sugar*'. Why does the recipe not say '*take a little powdered sugar, pour it into the melted chocolate,*

² From Sinclair and Malinowski (1978). *French Cooking*. Weathervane Books, p. 73.

stir it in, take a little more, pour, stir, . . .? Even more specifically, why does it not say *'take 2,365 grains of powdered sugar, pour them into the melted chocolate, pick up a spoon and use circular movements to stir it in, . . .*'? Or, to be even more precise, why not *'move your arm towards the ingredients at an angle of 14°, at an approximate velocity of 18 inches per second, . . .*'? The answer, of course, is obvious. The 'hardware' knows how to stir powdered sugar into melted chocolate, and does not need further details.

This begs the question of whether the hardware knows how to prepare sugared and buttered chocolate mixture, in which case the entire first part of the recipe could be replaced by the simple instruction *'prepare chocolate mixture'*. Taking this to the extreme, perhaps the hardware knows how to do the whole thing. This would make it possible to replace the entire recipe by *'prepare chocolate mousse'*, indeed a perfect recipe for producing the chocolate mousse; it is clear and precise, contains no mistakes, and is guaranteed to produce the desired output just as required.

Obviously, the level of detail is very important when it comes to an algorithm's elementary instructions. The actions that the algorithm asks to be carried out must be tailored to fit the capabilities of the hardware that does this carrying out. Moreover, the actions should also match the comprehension level of a human. This is because humans construct algorithms, humans must become convinced that they operate correctly, and humans are the ones who maintain those algorithms and possibly modify them for future use.

Consider another example, which is closer to conventional computation: multiplying integers manually. Suppose we are asked to multiply 528 by 46. The usual 'recipe' for this is to first

multiply the 8 by the 6, yielding 48, to write down the units digit of the result, 8, and to remember the tens digit, 4. The 2 is then multiplied by the 6, and the 4 is added, yielding 16. The units digit 6 is then written down to the left of the 8 and the tens digit 1 is remembered. And so on.

The same questions can be asked here too. Why '*multiply the 8 by the 6*'? Why not '*look up the entry appearing in the eighth row and sixth column of a multiplication table*', or '*add 6 to itself 8 times*'? Similarly, why can't we solve the entire problem in one stroke by the simple and satisfactory algorithm '*multiply 528 by 46*'? This last question is rather subtle: we are allowed to multiply 8 by 6 directly, but not 528 by 46. Why?

Again, the level of detail plays a crucial part in our acceptance of the multiplication algorithm. We assume that the relevant hardware (in this case, ourselves) is capable of carrying out 8 times 6 directly, but not 528 times 46, so that the former can be used as a basic instruction in an algorithm for carrying out the latter.

Another point illustrated by these examples is that different problems are naturally associated with different kinds of basic actions. Recipes entail stirring, mixing, pouring, and heating; multiplying numbers entails addition, digit multiplication, and remembering a digit; looking up a telephone number might entail turning a page, moving a finger down a list, and comparing a given name to the one being pointed out. Interestingly, we shall see later that when it comes to algorithms intended for computers these differences are inessential.

The text vs. the process

Suppose we are given a list of personnel records, one for each employee in the company. Each record contains an employee's

name, some other details, and his or her salary. We are interested in the total sum of the salaries of all employees. Here is an algorithm for this:

1. make a note of the number 0;
2. proceed through the list, adding the current employee's salary to the noted number;
3. having reached the end of the list, produce the noted number as output.

Clearly, the algorithm does the job. The 'noted' number can be thought of as a sort of box containing a single number, whose value can change. Such an object is often called a **variable**. In our case, the noted number starts out with the value zero. After the addition in line 2 is carried out for the first employee, its value is that employee's salary. After the addition for the second employee, its value is the sum of the salaries of the first two employees, and so on. At the end, the value of the noted number is the sum of all salaries (see Fig. 1.2).

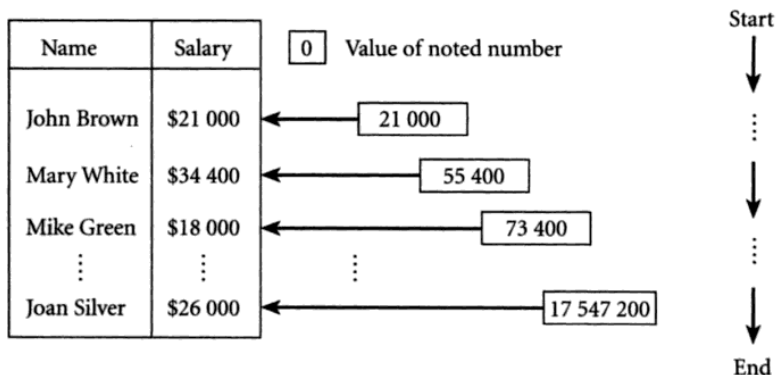


Fig. 1.2. Summing salaries.

It is interesting that the *text* of this algorithm is short and is fixed in length, but the *process* it describes varies with the size of the employee list, and can be very, very long. Two companies, the first with 10 employees and the second with a million, can both use the very same algorithm to sum their respective employees' salaries. The process, though, will be much faster for the first company than for the second. Moreover, not only is the text of the algorithm short and of fixed size, but both companies require only a single variable (the noted number) to do the job. So the quantity of 'utensils' is also small and fixed. Of course, the *value* of the noted number will be larger for a larger company, but only a single number is required to be 'noted' all along.

Thus we have a fixed algorithm, that requires no change in order to be used in different situations (i.e. for each and every different input list), but the processes it prescribes can differ in length and duration for different input situations.

Inputs

Even the simple example of salary summation shows a variety of possible inputs: small companies, large companies, companies in which some salaries are zero, or ones in which all salaries are equal. The algorithm might also have to deal with unusual or even bizarre inputs, such as companies with no employees at all or with employees who receive negative salaries (that is, the employee pays the company for the pleasure of working for it).

Actually, the salary algorithm is supposed to perform satisfactorily for an *infinite* number of perfectly acceptable lists of employees. This is an extreme way of appreciating the 'short-algorithm-for-lengthy-process' principle. Not only the contrast

in duration, or length, is interesting; the very *number* of processes prescribed by a single algorithm of fixed length can be large, and most often is infinite.³

An algorithm's inputs must be **legal** relative to its purpose. This means that the *New York Times* list of bestsellers would be unacceptable as an input to the salary summation algorithm, just as peanut butter and jelly are unacceptable as ingredients for the mousse recipe. This means that we need a **specification** of the allowed inputs. Someone must decide precisely which employee lists are legal and which ones are not, where an employee record ends and another begins, where exactly in each record the salary is to be found and whether it is given in longhand (for example, \$32 000) or in some abbreviated form (e.g. \$32K), and so on.

What do algorithms solve?

All this leads us to the central notion underlying the world of algorithmics and computation – the **algorithmic problem**, which is what an algorithm is designed to solve. The description of an algorithmic problem must include two items (see Fig. 1.3):

- a precise definition of the set of legal inputs;
- a precise characterization of the required output as a function of the input.

³ This issue of an infinite number of potential inputs doesn't quite fit the recipe analogy, since although a recipe should work perfectly well no matter how many times it is used, ingredients are usually described in fixed quantities. Hence, the recipe really has only one potential input (at least as quantities go; clearly the molecules and atoms will be different each time). However, the chocolate mousse recipe could have been made generic, to fit varying but proportional quantities of ingredients.

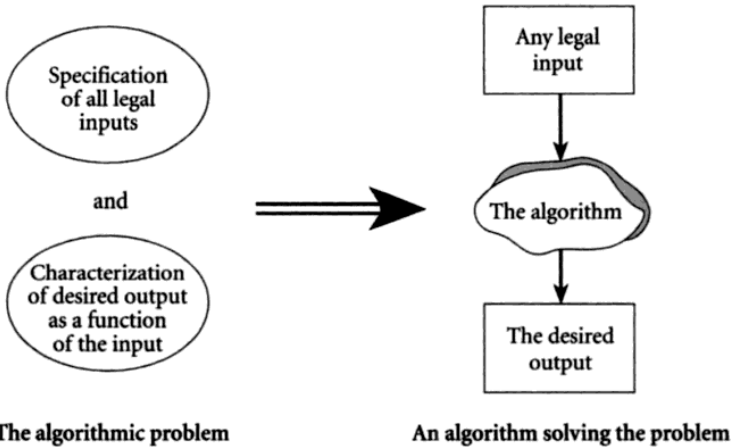


Fig. 1.3. The algorithmic problem and its solution

When we discuss an algorithmic problem as applied to a particular input (like the salary summation problem applied to some concrete list of employees), we call it an **instance** of the problem.

Here now are some additional examples of algorithmic problems. Each one is defined, as is proper, by its set of legal inputs and a description of the desired output. They are numbered, and we will refer to them at various points in the following chapters.

Problem 1

Input: Two integers, J and K .

Output: The number $J^2 + 3K$.

This is a simple problem that calls for an arithmetic calculation on two input numbers.

Problem 2

Input: A positive integer K .

Output: The sum of the integers from 1 to K .

This problem also involves arithmetic, but the number of elements it deals with varies, and itself depends on the input.

Problem 3

Input: A positive integer K .

Output: 'Yes' if K is prime and 'No' if it isn't.

This is what we shall be referring to as a decision problem. It calls for deciding the status of its input number. (Recall that a **prime number** is a positive integer greater than 1 that can be divided without a remainder only by 1 and itself. For example, 2, 17, and 113 are primes, whereas 6, 91, and 133 are not. Non-primes are termed **composite**.) Solving this problem will surely involve arithmetic, but it does not provide a numeric output, only a 'Yes' or a 'No'.

Problem 4

Input: A list L of words in English.

Output: The list L sorted in alphabetic (lexicographic) order.

This is a non-arithmetical problem, but like Problem 2 it has to deal with a varying number of elements; in this case words.

Problem 5

Input: Two texts in English.

Output: A list of the words common to the two texts.

This too involves words, rather than numbers. We assume that texts have been defined appropriately, say, as a string of symbols consisting of letters, spaces, and punctuation marks. A word in a text would be a string of letters enclosed by spaces or punctuation marks.

Problem 6

Input: A road map of cities with distances attached to road segments, and two designated cities therein, *A* and *B*.

Output: A description of the shortest possible path (trip) between *A* and *B*.

This is a search problem, involving points and distances between them. It calls for some kind of optimization process to find the shortest path.

Problem 7

Input: A road map of cities, with distances attached to road segments, and a number *K*.

Output: 'Yes' if it is possible to take a trip that passes through all the cities, and whose total length is no greater than *K* miles, and 'No' if such a trip is impossible.

This too asks to search for a short path, not between two points but, rather, a path that traverses *all* points. Also, this problem is not phrased as requiring an optimization (i.e. find the 'best' path), but as a decision problem that asks just whether there is some path shorter than the given limit.

Problem 8

Input: A program P written in Java, with integer input variable X and output variable Y , and a number K .

Output: The number $2K$ if the program P always sets Y 's value to be equal to X^2 , and $3K$ if not.

This problem is about algorithms, in their formal attire as programs. It wants to know something about the behavior of a given program in general; not of a particular input.

So algorithmic problems have all kinds of inputs: numbers, words, texts, maps, and even other algorithms or programs. Also, some problems are truly computational in nature, some involve rearrangements (sorting), some require information retrieval (finding common words), some are optimization problems (shortest path), and some are decision problems (primality testing and all-point trips). Thus, a **decision problem** is a yes/no algorithmic problem. Decision problems appear not to compute, retrieve or optimize, only to *decide*, determining whether some property is true or false. Some algorithmic problems are hybrids: Problem 8, for example, combines decision with computation; its output is the result of one of two simple computations, but which of these it will be depends on a property of the input that has to be decided.

All these sample problems have infinite sets of legal inputs. To solve them, we have to be able to deal with arithmetic on *all* numbers, with sorting *all* lists of words, with finding the shortest trip in *all* city maps, etc. Put another way, each problem requires that we devise a method, a common procedure or recipe, that will solve *any* given instance of the problem. The number of potential instances is infinite. Such a method constitutes an algorithm.

Many algorithmic problems in the real everyday world are not so easy to define. Sometimes the difficulty is in specifying the required output, as when asking for the best move for a legal board position in chess (what exactly is 'best?'). In other cases, describing the inputs can be complicated. Suppose 20 000 newspapers are to be distributed to 1000 delivery points in 100 towns using 50 trucks. The input contains the road distances between the towns and between the delivery points in each town, the number of newspapers required at each point, the present location of each truck, details of available drivers including their present whereabouts, and each truck's newspaper carrying ability, gasoline capacity, and miles-per-gallon performance. The output is to be a list, matching drivers and destinations to trucks in a way that minimizes the total cost to the distributing company. Actually, the problem calls for an algorithm that works for any number of newspapers, towns, delivery points, and trucks.

Some problems have hard-to-pin-down inputs as well as hard-to-specify outputs, such as the ones required to predict the weather or to evaluate stock market investments.

In this book, we shall stick to simple-looking algorithmic problems, usually with easy to describe inputs and outputs. In fact, for the most part, we will concentrate on decision problems. So describing our problems will be easy, and the outputs will usually be just 'Yes's and 'No's.

Isn't our setup too simplistic?

Aren't we oversimplifying things? Computers are busy struggling with tasks far more complicated than merely reading a simple input, doing some work, producing a 'Yes' or a 'No', and quitting. Aren't we greatly weakening our presentation by avoiding modern real-world computational frameworks, such as

interactive computing, distributed systems, real-time embedded systems, graphics-intensive applications, multimedia, and the entire world of the Internet?

To me, the author, you might be saying under your breath 'Are you just another one of those stuffy academics? Don't you know *anything* about computing? Stop giving us this chit-chat about simple input/work/output computations. Just get real, will you?'

The answer is: indeed, yes. We *are* simplifying things, and in fact quite radically. But for a very good reason. Remember that we are dealing with the *bad* news. This book is not about making things better, smaller, stronger, or faster. It is about showing that very often things *cannot* be improved in these ways. That things can become very, very nasty. That certain tasks are simply impossible. Now, given that we are after bad news here, our arguments and claims become *stronger*, not weaker, by considering a simpler class of problems! We will be showing that even in a simple computational framework things can be devastatingly bad; all the more so in an intricate and seemingly more powerful one. The fact that computers are hopelessly limited is *more* striking with a simple input-output paradigm for computation than with a more complex one. Moreover, since the book is devoted almost exclusively to decision problems, we are also implying that the bad news has nothing to do with the need for complicated and lengthy outputs. The desire to generate even a simple 'Yes' or 'No' is enough to yield real nightmares.

Solving algorithmic problems

An algorithmic problem is **solved** when an appropriate algorithm has been found. What is 'appropriate'? Well, the

algorithm must provide correct outputs for all legal inputs: if the algorithm is **executed**, or **run**, on any one of the legal inputs defined in the problem, it must produce the output specified in the problem for that input. A solution algorithm that works well for *some* of the inputs is not good enough.

Finding solutions to most of the sample problems described earlier is easy. Computing $J^2 + 3K$ is trivial (assuming, of course, that we have basic operations for addition and multiplication), and likewise summing the integers from 1 to K . In the latter case, of course, we must use a **counter** to keep track of how far we have gone and to stop the process when we have dealt with K itself.

To test whether a number K is prime (Problem 3), we divide it by all the integers from 2 to $K - 1$, stopping and saying 'No' if one of them is found to divide K without a remainder, and stopping and saying 'Yes' only when all the divisions have been completed and they have all yielded a remainder.⁴

Problem 4 can be solved by numerous different sorting algorithms. A simple one involves repeatedly searching for the smallest element in the input list L , removing it from L and adding it to the accumulating output list. The process stops when the

⁴ Of course, this algorithm can be improved: we can stop the process of testing for divisors at \sqrt{K} , the square root of K , rather than at $K - 1$. The reason is that if K has a clean divisor that is larger than \sqrt{K} it must also have one that is smaller. We can also avoid testing multiples of the numbers already tested, thus further expediting the process. Some of the other problems can also be solved more efficiently than the ways we mention. However, efficiency and practicality of algorithms are not addressed until later in the book, so we shall not dwell on these issues right now. Here we impose only the minimal requirement – that the algorithm does, in fact, solve the problem, providing correct outputs for all legal inputs, even though it might do so inefficiently.

original list is empty. Problems 6 and 7 can both be solved by considering all possible paths between cities (that is, one-way paths between *A* and *B* in Problem 6, and round-trip paths that traverse all the cities in Problem 7), and computing their lengths. Since the number of cities is finite, the number of paths is finite too, so that an algorithm can be set up to run through them all. This has to be done with care, however, so as not to miss any paths.

As mentioned, we shall return to several of these sample problems in the following chapters.

Programming

An important issue that we should address, although it is not really critical to the central concerns of the book, is the way algorithms are executed by real computers. How do computers bridge the gap between their extremely modest capability to carry out operations on bits and the high-level actions humans use to describe algorithms? For example, how can bit manipulation be made to accomplish even such a simple-looking task as *'proceed through the list, adding the current employee's salary to the noted number'*? What list? Where does the computer find the list? How does it proceed through the list? Where exactly is the salary to be found? How is the 'noted number' accessed? And so on.

We have already mentioned that algorithms must be presented to the computer in a rigorous, unambiguous fashion, since when it comes to precision and unambiguity, *'proceed through the list'* is not much better than *'beat egg whites until foamy'*. This rigor is achieved by presenting the computer with a **program**, which is a carefully formulated version of the algorithm, suitable for computer execution. It is written in a **pro-**

programming language, which provides the notation and rules by which one writes programs for the computer.

A programming language must have a rigid **syntax**, allowing the use of only special words and symbols. Any attempt to stretch this syntax might turn out to be disastrous. For example, if **'input K '** is written in a language whose input commands are of the form **'read K '**, chances are that the result will be something like **'SYNTAX ERROR E4514 IN LINE 108'**. And of course, we cannot hope to address the computer with the like of *'please read a value for K from the input'*, or *'how about getting me a value for K '*. These might result in a long string of obscure error messages. It is true that nice, talkative instructions, such as the ones we find in recipes, are more pleasant and perhaps less ambiguous to a human reader than their terse and impersonal equivalents. It is also true that we strive to make computers as user-friendly as possible. But since we are still far from computers that can understand free-flowing natural language like English (see Chapter 7), a formal, concise, and rigid set of syntactic rules is essential.

An algorithm for summing the numbers from 1 to K might be written in a typical programming language as follows:

```
input  $K$ 
 $X$ : = 0
for  $Y$  from 1 to  $K$  do
     $X$ : =  $X$  +  $Y$ 
end
output  $X$ 
```

The intended meaning of this program is as follows. First, K is received as an input and the variable X (a 'noted number') is assigned an initial value of zero. Its role will be to accumulate the running sum we are calculating. Next, a loop is carried out,

calling for its **body** – in our case the $X := X + Y$ that appears between the **for** command and the **end** – to be executed again and again. The loop is controlled by the variable Y , which starts out with the value 1 and increases repeatedly by 1 until it reaches K , which is the last time the $X := X + Y$ is executed. This causes the computer to consider all the integers from 1 to K , in that order, and in each iteration through the loop the integer considered is added to the current value of X . In this way X accumulates the required sum. When the loop is completed, the final sum is output.

Of course, this is what we *intend* the program to mean, which is not enough. The computer must somehow be told about the intended meaning of programs. This is done by a carefully devised **semantics** that assigns an unambiguous meaning to each syntactically allowed phrase in the programming language. Without this, the syntax is worthless. If meanings for instructions in the language have not been provided and somehow ‘explained’ to the computer, the program segment ‘**for** Y **from** 1 **to** K **do**’ might, for all we know, mean ‘subtract Y from 1 and store the result in K ’, instead of it being the controlling command of the loop, as we intended. Worse still, who says that the keywords **from**, **to**, **do**, for example, have anything at all to do with their meaning in English? Maybe the very same program segment means ‘erase the computer’s entire memory, change the values of all variables to zero, output “TO HELL WITH PROGRAMMING LANGUAGES”, and stop!’. Who says that ‘:=’ stands for ‘assign to’, and that ‘+’ denotes addition? And on and on. We might be able to *guess* what is meant, since the language designer probably chose keywords and special symbols intending their meaning to be similar to some accepted norm. But a computer cannot be made to act on such assumptions.

To summarize, a programming language comes complete with rigid rules that prescribe the allowed *form* of a legal program, and also with rules, just as rigid, that prescribe its *meaning*. We can now phrase, or **code** our algorithms in the language, and they will be unambiguous not only to a human observer, but to the computer too.

Once the program is read in by the computer, it undergoes a number of computerized transformations, aimed at bringing it down to the bit-manipulation level that the computer really 'understands'. At this point the program (or, rather, its low-level equivalent) can be run, or executed, on a given input (see Fig. 1.4).⁵

Errors and correctness

Coming up with a bright idea for an algorithm, constructing the algorithm itself carefully and then writing it up formally as a program, doesn't mean we are done. Consider the following:

- Several years ago, around her 107th birthday, an elderly lady received a computerized letter from the local school authority in a Danish county, with registration forms for first grade in elementary school. It turned out that only two digits were allotted to the 'age' field in the population database.
- In January 1990, one of AT&T's switching systems in New York City failed, causing a major crash of the national AT&T telephone system. For nine hours, almost half of the calls made through AT&T failed to connect. As a result, the company lost more than \$60 million, not to mention the

⁵ The main transformation among these is called **compilation**. The **compiler**, which is itself a piece of software, transforms the high-level program into a functionally equivalent program written in a low-level format called **assembly language**, which is much closer to the **machine language** of bit manipulation.

- three-color problem 106
 - NP-completeness 106
- $3x+1$ problem 51–2
- tic-tac-toe 203
- tiling problem 30–3
 - equivalence to halting problem 54–5
- highly undecidable variant 57
- unboundedness does not imply
 - undecidability 46–7
 - undecidability 32–3
 - variants 33
- time, *see* running time
- time complexity 65–9
 - dependence on language and compiler 66
 - not determined by length of output 72–3
 - practical issues 66
 - upper and lower bounds 65–9
 - see also* exponential time; linear time; NP; PTIME; QP; quadratic time; RP
- timetable problem 91, 100–2
 - NP-completeness 101
 - practical compromise solutions 101–2
 - tractable variants 102
- Towers of Hanoi 69–73
 - algorithm for solution 71
 - time complexity 72
- tractable problem 78, 91
 - see also* P; PTIME
- trapdoor function 164, 169–70
- traveling salesman problem [Problem 7]
 - 13, 17–18, 97–100, 154
 - approximation algorithm 113
 - NP-completeness 100
 - relevance to applications 99
- Turing machine 36–40
 - multiplication algorithm 38–40
 - slowness not unreasonable 80
 - solvability using 40
 - status as universal machine 40–2
- Turing test 194–8
- Twinkle 173
- undecidable problem 28, 32–6, 47–58
 - degrees of hardness 54–8
- unlimited intractability 87–8
- unsolvable problem 27
 - see also* undecidable problem
- variable 8
- verification of programs 48–50
- verification problem 53–4
 - more undecidable than halting problem 55–6
- voting 187–8
- Waldo 179
- witness 137
 - see also* certificate
- worst case 62
- WS1S 86–8
 - unlimited intractability 87
- Y2K problem 23, 49–50
- year 2000 bug 23, 49–50
- zero-knowledge protocol 178–86
 - cryptographic applications 179–80
 - for three-coloring a network 181–6
 - P vs. NP question 186