1

A Science of Computing

COPYRIGHTED MATERIAL

The hallmark of a science is the avoidance of error.

J. Robert Oppenheimer

The electronic, digital computer is a marvel of modern technology. Within a lifetime, it has developed from nothing to a tool whose use is so widespread that we are often unaware that it is there. The first computers, built in the 1940s, were monstrous. As recently as the 1960s and 1970s, it was common for new employees in a company to be shown 'the computer'—housed in a special-purpose, air-conditioned room, to which admission was restricted to a select few. Nowadays, computers are used in many household appliances, they are also used in cars, trains and aircraft, and we even carry them around with us—laptops, palm-tops, what would we do without them?

The developments in computer *hardware* have been truly phenomenal. But developments in computer *software* have not kept pace. The programming languages that are in use today have changed little from the programming languages that were developed in the 1950s, and programming remains a highly skilled activity. It is the nature of the task that is the problem. The hardware designer must build a dumb machine, whose sole purpose is to slavishly execute the instructions given to it by the programmer; the programmer has to design general-purpose systems and programs, at a level of detail at which they can be faithfully executed, without error, by even the dumbest of machines.

1.1 Debugging

There is a story about the current state of computer software that is widely reported on the Internet. The chief executive of a multi-billion dollar software company compared the computer industry with the car industry.

> 'If the automobile industry had kept up with technology like the computer industry has,'

he is reported to have said,

> 'we would all be driving $25 cars that get 1000 to the gallon'.

> 'That may be true,'

was the swift response of the president of one automobile company,

> 'but who would want to drive a car that crashes twice a day?'

This story is, most likely, just a joke[1]. But, like all good jokes, it succeeds because it reflects profoundly on modern-day reality.

At the time of writing (December 2002), it is still the case that computer software is liable to spontaneously 'crash', due to simple programming errors (or 'bugs' as they are called in the software industry). In contrast, the car industry is often held up as an exemplar of excellence in standards of safety and reliability.

Memories are short. Cars may be (relatively) safe and reliable today but, in the 1950s, as highlighted by Ralph Nader in his acclaimed book *Unsafe At Any Speed*, cars were knowingly made and sold that were liable to spontaneous crashes, and car manufacturers were guilty of deliberately undermining efforts to invest in improved standards. ('Safety and sales strategies do not mix' was the argument used at the time.)

The computer industry of today is very much like the car industry of the 1950s. Computers are still relatively new, and the novelty factor has not worn off, so that guarantees of reliability and fitness-for-purpose are, for many, a low priority. Few programmers are trained in scientific methods of constructing programs and, consequently, they waste substantial amounts of effort 'debugging' their programs, rarely with complete success. ('Debugging' is the process of repeatedly testing, followed by patching, the program, in an attempt to remove discovered errors.)

The need for alternative, mathematically rigorous, program construction techniques was recognized in the late 1960s when the large computer manufacturers first began to realize that the costs of producing computer software were

[1] At least, I think it is. It may be true, but it is difficult to verify the authenticity of material on the Internet. For this reason, names have been omitted in this account.

outstripping by far the costs of producing computer hardware. They spoke of a 'software crisis'. The problems of producing reliable computer software were aired at two conferences on Software Engineering, held in 1968 and 1969 and sponsored by the NATO Science Committee. Typical of the sort of facts laid bare at these conferences was the following statement by Hopkins of the IBM Corporation.

> We face a fantastic problem in big systems. For instance, in OS/360[2] we have about 1000 errors per release.

Tellingly, he added:

> Programmers call their errors 'bugs' to preserve their sanity; that number of 'mistakes' would not be psychologically acceptable.

The process of debugging has several drawbacks: it is useless as a methodology for constructing programs, it can never be used to establish the correctness of a correct program, and it cannot be relied upon to establish the incorrectness of an incorrect program.

Let us look in detail at these drawbacks. Here are two examples, each illustrating a different aspect.

1.2 Testing a Correct Program

One well-known method of computing n^2, for some positive integer n, without performing a multiplication is to sum the first n odd numbers. This is based on the property that

$$n^2 = 1+3+5+\ldots+(2n-1) .$$

Not so well known is that a similar method can be used to compute n^3, n^4, n^5, etc., without performing a multiplication. To see how this is done let us re-express the computation of n^2 as follows.

First, write down all the positive integers up to (and including) $2n-1$. For $n=6$, this means the numbers 1 through 11.

1 2 3 4 5 6 7 8 9 10 11

Cross out every second number:

1 3 5 7 9 11

Finally, add these together to form a running total:

1 4 9 16 25 36

To compute n^3, begin as before by writing down all the positive numbers, but this time up to $3n-2$. For $n=6$, this means the numbers 1 through 16.

[2] At that time, OS/360 was a widely used operating system for IBM computers.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Cross out every third number:															
1	2		4	5		7	8		10	11		13	14		16
Add these together to form a running total:															
1	3		7	12		19	27		37	48		61	75		91
Cross out every second number:															
1			7			19			37			61			91
Finally, form a running total:															
1			8			27			64			125			216

The general algorithm for computing n^m is to write down all the positive numbers up to $mn - m + 1$. Then we iterate $m-1$ times the process of crossing out numbers followed by forming a running total. On the kth iteration every $(m-k+1)$th number is crossed out. (In this way, the set of numbers crossed out changes from every mth number on the first iteration to every second number on the last iteration.)

Now, we can test this algorithm in two ways. We can extend one of the existing tables to the right; for example, the table for n^2 can be extended to calculate 7^2 and 8^2:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1		3		5		7		9		11		13		15
1		4		9		16		25		36		49		64

Alternatively, we can add new tables to the ones we already have; for example, the table for 3^4:

1	2	3	4	5	6	7	8	9
1	2	3		5	6	7		9
1	3	6		11	17	24		33
1	3			11	17			33
1	4			15	32			65
1				15				65
1				16				81

We can continue this testing as much as we like (indeed, for ever and a day). Each time we add more entries and verify the correct result, our confidence in the algorithm grows. But, this process will never make us totally confident. Can we be sure that it will correctly compute 21^5 or 6^{12}? On the evidence presented so far would you be willing to gamble on its correctness?

Edsger W. Dijkstra, an eminent computer scientist, has summarized the flaw in testing in a now-famous quotation.

> Program testing can be used to show the presence of bugs, but never to show their absence.

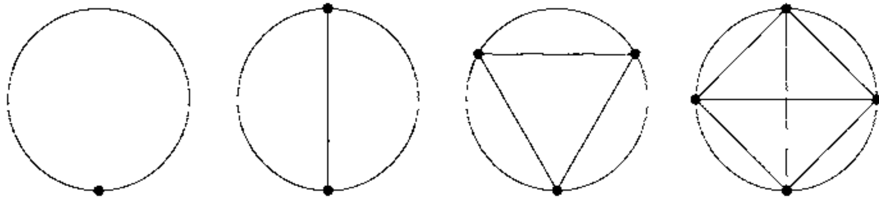

Figure 1.1 Cutting the cake.

Exercise 1.1 (Cutting the cake). This exercise demonstrates the fallacy of using 'poor man's induction', i.e. extrapolating from a few cases to a more general claim.

Suppose n points are marked on the circumference of a circular cake and then the cake is cut along the chords joining them. The points are chosen in such a way that all intersection points of pairs of chords are distinct. The question is: in how many portions does this cut the cake?

Figure 1.1 shows the case when n is 1, 2, 3 or 4.

Suppose $n = 5$. Determine the number of portions. On the basis of these five cases make a conjecture about the number of portions for arbitrary n. Now, suppose $n = 0$. Does this support or refute the conjecture? Next, suppose $n = 6$. Determine the number of portions. What do you discover? □

1.3 Testing an Incorrect Program

Testing cannot be used to establish the absence of errors in a program. Nor can testing be relied upon to show the presence of errors.

I once had a very graphic illustration of this when I had to mark a programming assignment. The problem the students had been set was to write a program that would compare two strings for equality. One student's solution was to assign the value true or false to a boolean *equal* as follows[3]:

```
equal := (string1.length = string2.length);
if equal
then for i := 1 to string1.length
     do equal := (string1.character[i] = string2.character[i])
```

The problem with this code is that it returns the value true whenever the two strings have equal length and their last characters are identical. For example, the two strings 'cat' and 'mat' would be declared equal because they both have length three and end in 't'.

The student was quite taken aback when I demonstrated the error. Indeed, upon further questioning, it emerged that the program had been tested quite system-

[3]The program is coded in Pascal. The characters of a string s are assumed to be stored in the array *s.character*, indexed from 1 to *s.length*.

atically. First, it had been tested on several pairs of identical strings, and then on several pairs of strings of unequal length. Both these tests had produced satisfactory results. The final test had been to input several pairs of equal length but unequal strings, such as 'cat' and 'dog', or 'house' and 'river'.

This final test is interesting because it is possible to use simple probability theory to make a rough estimate of the chances of discovering the programming error. The details are not relevant here. Suffice it to say that, assuming pairs of words of equal length are generated with equal letter frequencies (that is, each letter in the alphabet is chosen with a probability of 1/26), there is only a one in three chance of discovering the error after ten tests; increasing the number of tests to 20 would still only yield a one in two chance, and one would need to perform over 50 tests to achieve a 90% chance of discovering the error. Finally, and most importantly, there is no certainty that one would ever discover the error, no matter how many tests have been performed.

So, you see that program testing is never-ending. We can never be sure that all avenues have been tried; we can never be sure that there is not one more error lurking unseen, just waiting for the most crucial opportunity to show itself.

(Needless to say, I observed the error by reading the code, not by testing. The student was blameless. The error was the responsibility of the student's teacher for having failed to teach proper design principles and suggesting that testing was adequate.)

1.4 Correct by Construction

That debugging is not fail-safe is a drawback, but not its main limitation. The main problem with debugging is that it is useless as the basis for program *design*, of which fact even a small acquaintance with programming will convince you. An alternative to debugging is the development of a *science* of programming. Such a science should provide the techniques to enable the verification of a program against its specification. But it should do more than that; it should provide a discipline for the *construction* of programs that guarantees their correctness.

Of course, the science guarantees correctness only if it is used correctly, and people will continue to make mistakes. So testing is still wise, and debugging will occasionally be necessary. But now, rather than chance, it is our own skill in applying the science on which we rely. The aim of this book is to impart that skill and to enable you to take a pride in your programming ability.

Bibliographic Remarks

The Internet is awash with accounts of bugs and other programming errors that have made headline news over the years. For a particularly tragic case, search on the word 'Therac' for details of a software error that resulted in at least three

deaths by an overdose of radiation in 1986. Searching on 'Risks' (short for 'Forum on Risks to the Public in Computers and Related Systems') yields details of many other examples.

The summary of the state of the car industry in the 1950s (see Section 1.1) is based on *Unsafe At Any Speed* by Nader (1965). The reports on the two NATO Science Committee-sponsored conferences (Naur and Randell, 1969; Buxton and Randell, 1970) are available on the Internet.

A very large part of the text has been directly influenced by the work of Edsger W. Dijkstra. The quotation attributed to Dijkstra in Section 1.2 appears in Buxton and Randell (1970), which is also the source of Hopkins's remarks.

I first saw the algorithm for computing the powers of n, discussed in Section 1.2, in Polya (1954); it was originally discovered by Mössner (1951) and verified by Perron (1951).

The case $n = 0$ in the cutting-the-cake problem (Exercise 1.1) was pointed out to me by Diethard Michaelis. A formula giving the number of portions is quite complicated. See Dijkstra (1990) for a derivation of the correct formula.