# Chapter 1
# Why Mathematics?

How are programming and computing machines related to mathematics? Does one really need mathematics to understand computer science concepts?

1. In the sciences, mathematics is used to build a model of the observed reality. In physics, for example, scientists propose mathematical systems describing physical objects and phenomena pertaining to them. After accumulating data about planetary motion, Kepler formulated three mathematical laws which planetary motion satisfies. Using these, he could simulate planets moving around a sun, predict the actual position of planets and verify properties such as "Using these laws of planetary motion, there is no way in which Mars will collide with Earth in the next million years."

   Unlike these natural sciences, in computer science, we study artificial objects: computers, programs, programming languages, etc. Despite this there is no fundamental difference, in the sense that we would still like to have a model of these systems to reason mathematically about them. Will this program terminate? Does it give the right answer? Is this computer capable of performing this complex operation?
2. In programming, very frequently we have to represent complex structures. Imagine writing a program which has an internal representation of the map of the London Underground, chess game configurations or rules of English grammar. Describing these objects adequately requires the use of complex data structures. Fortunately, objects such as the ones mentioned have been very well studied by mathematicians. Or rather, no mathematician studied the London Underground map, but many have studied *graph theory,* using which it is straightforward to describe the map of the London Underground. The mathematical way of describing such systems can then be readily translated into a data structure to be used in our programs. What is even more interesting is that mathematicians have proved various things about these classes of objects, properties which would then also be true of our data structures, allowing us to analyse and optimise our programs.
3. Just as we can use mathematics to guide the design of data structures, computer designers and computer scientists use mathematics to build circuits and computers, and design programming languages. A circuit is nothing but a physical

G.J. Pace, *Mathematics of Discrete Structures for Computer Science*,
DOI 10.1007/978-3-642-29840-0_1, 

instantiation of mathematical propositions, and a computer is nothing but a physical instance of a Turing machine (which is not a machine, but a mathematical model). Whereas in the natural sciences one usually uses mathematics to model reality, in computer science our objects of study are concrete instances of mathematical abstractions.

## 1.1 What Is Mathematics?

The layman usually associates mathematics with numbers. Numbers just happen to be one application of one particular field of mathematics. However, mathematics can be seen as the study of ideal, regular and formal structures. *Ideal* in the sense that it is detached from reality. No matter if in the real world no perfect circle exists, mathematicians in the field of geometry have been studying perfect circles for over two thousand years. *Regular* in the sense that the objects can be described in a compact way. Any circle on a plane can be described in terms of a point (the position of its centre) and a distance (its radius). Using this compact (and general) description, we can compare circles and study how they interact. *Formal* in the sense that the objects have a well-defined meaning and can be reasoned about in an unambiguous way. Once we agree on what a perfect circle is and the definition of the intersection of two circles, the deductions made cannot be disputed. If, based on the formal definitions, one proves that two particular circles do not intersect, the conclusion is final and cannot be further argued about.

The best way to understand what characteristics distinguish mathematical reasoning from other forms of reasoning is to take a brief look at the history of mathematics and see how mathematical thought was shaped and became what it is today.

## 1.2 A Historical Perspective

From the birth of the comparison of quantities and counting, it must have been inevitable that certain patterns were observed. If Paul has more sheep than Mary, and Mary has more sheep than Kevin, then it is inevitable that Paul has more sheep than Kevin, without any need to compare the flocks of Paul and Kevin directly. If Paul gives Mary some sheep, then Mary will still have more sheep than Kevin. It is typical human nature to try to identify patterns in observations, and in the same way that our ancestors observed the repetition and order of the seasons, they also observed patterns about quantities.

From the earliest numeric systems recorded on clay tablets, it is clear that the scribes were aware of properties about the numbers they were inscribing. For example, on a number of Babylonian tablets describing problem and solution, some sums were performed by adding the first quantity to the second, while in others, they added the second to the first. This implies that they were implicitly aware of the rule that $x + y = y + x$.

Numbers and geometry have various everyday applications: calculating areas for landlords to charge rent, working out profits and in architecture. The Ancient Greek civilisation, however, also gave great importance to abstract and mental pursuits, from which arose the study of numbers and other abstract notions for their own sake rather than simply as tools to calculate results. Sects such as the Pythagoreans gave an almost divine standing to the perfection of whole numbers. They identified classes of numbers: square numbers, perfect numbers, triangular numbers, prime numbers, etc., and asked questions about them: Is there an infinite number of prime numbers? Are there numbers which can be written as the sum of two square numbers in more than one way? Can the sum of two perfect numbers be itself perfect? They presented arguments which supported their claims. Unlike most philosophical claims which most fellow philosophers made, and which one could argue about for days and present arguments both for and against (Is the universe finite? Is there a basic building block out of which everything in the universe is made?), arguments about mathematical notions were irrefutable.

This gave rise to the question of whether such forms of argumentation can also be applied to other fields of philosophy and law. What was special about mathematical arguments that made them irrefutable? A mathematical argument always took the form of a sequence of statements, with each statement being either (i) a truth upon which the debating parties agreed, or (ii) built upon previous statements already proved and combined, once again in an agreed way. Therefore, one had to identify two types of shared truths to be able to build irrefutable arguments: a number of basic truths everyone agrees on, and a number of rules which dictate how one is allowed to conclude a new true statement from already known ones. For example, one basic truth may be that every circle has the same radius as itself, while a rule may say that, if two circles have the same radius, then they have the same area. If these truths are not agreed on, then one must seek even simpler truths upon which to agree.

Around 300 BC, Euclid constructed a mathematical system using this approach to reason about geometry. He proved various complex statements based on a small number of basic truths and rules. This mode of reasoning, usually called *formal* or *axiomatic reasoning*, is the basis of mathematics. Armed with a small number of obvious truths, the mathematician seeks out to prove complex properties of his or her objects of study.

But can these basic truths and rules be shown to be correct? One way of doing so is to give another group of rules and show the two to be equivalent. But that is just relegating the truth of one set of rules to another. What does it take to ensure that our basic truths and rules are correct? It means that they faithfully describe the objects they set out to describe. This is more of an empirical, or observational matter. What our proofs really do tell us, is "Give me a system which obeys these rules and this is what I can conclude about it ...". Nothing more, but more importantly, nothing less.

## 1.3 On the Superiority of Formal Reasoning

We have already argued that one of the main advantages of using formal reasoning is *incontrovertibility*—one can check an argument or proof for correctness quite easily. If the rules are adhered to, then the conclusions must be correct. This means that, as long as the basic truths remain undisputed, there can be no revisions of the mathematical conclusions.

Another advantage is that of *precision*. When we state something mathematically we are stating it in an unambiguous, exact manner. This is not so in English as in, for example, the following sentence: "Rugby is a sport played by men with oddly shaped balls". When one is designing a computer program, ambiguities and underlying assumptions in specifications can be very costly: "Yes, I just said that we wanted to be able to search the records, but why does your program search them by name? We always search by identity card numbers here!" In mathematics, any implicit assumptions can be discovered through analysis. If your sorting routine only works when the items to sort are positive numbers, it may lie hidden for years, until someone tries to sort a list of bank account balances. A mathematical analysis of the routine would have identified this hidden assumption immediately.

*Abstraction* is the act of ignoring irrelevant detail. Imagine someone is called in to check whether a sheep can go through all the doorways in a certain house. It is allowed to take measurements of the sheep but not physically push the sheep through the doors. Furthermore, justification of the conclusion is also required. One solution is to model the complex shape of the sheep mathematically using millions of points, and then build a computer program which rotates the sheep in different ways so as to assess whether it actually passes through all the doorways. A smarter solution, however, would be to physically check that the sheep fits inside a two metre by one metre by one metre box. Then, using pencil and paper, it can easily be shown that, since all the doorways are at least one metre wide and one metre high, the sheep can go through them all. What is so strong about this approach is that these mathematicians started by reasoning, 'if we know that the sheep will fit in such a box, then we can prove that it passes through all the doorways.' The act of using a box rather than the actual sheep gives a much simpler object to reason about, and is called *abstraction*.

One act of abstraction we are very familiar with is when we apply equalities. We take an equality $x = y$ and proceed to prove things by replacing $x$ by $y$ or $y$ by $x$ in a huge expression. This hides a very strong abstraction: $x$ and $y$ are equal no matter the context in which they occur. Whether $x$ appears in a huge equation or a small one does not really matter.

We will be illustrating some applications of mathematics to computing in the coming section.

## 1.4 The Mathematics of Computing

Mathematics is a vast subject although, obviously, some topics are more important to computing than others. In this book we will be focussing on the mathematics of

discrete structures—objects that vary in large steps, as opposed to small continuous changes. To mathematically model computers or programs, we can thus only use discrete structures made up of ones and zeros.[1]

Although the focus and aim of this book is to expose the mathematical foundations which are applied in various fields of computer science, the applications are exposed through examples throughout the text. As initial motivation, and a taste of where mathematics and computing overlap, we will look at a number of application areas.

### 1.4.1 The Analysis of Programs

Faced with the same specification to implement, different programmers may come up with different solutions. Consider the task of sorting a collection of numbers in ascending order. One person may propose the following algorithm:

> *Go through the whole list, choose the smallest, and put it in the first position. Then repeat the procedure with the other numbers until no numbers are left to sort.*

For example, faced with the sequence of numbers $\langle 2, 4, 1, 3\rangle$, this approach will identify 1 as the smallest number and swap it with the number in the first position, resulting in $\langle 1, 2, 4, 3\rangle$. The procedure is then reapplied to the sequence disregarding the first item $\langle 2, 4, 3\rangle$. Eventually, one obtains the sorted list.

However, another person may come up with a different solution:

> *A list with no more than one number is already sorted. If the list is longer, remove the first item in the list, and separate the other numbers into two: those not larger than the first number, and those larger. These two collections of numbers are sorted (using this same procedure) and then just catenated, starting with the sorted list of smaller numbers, followed by the number which was the first item in the original list and finally the sorted list of larger numbers.*

If we consider sorting the sequence $\langle 2, 4, 1, 3\rangle$, we would remove 2 (the first item in the list) and create two lists—those not larger than 2 and those larger than 2. This gives $\langle 1\rangle$ and $\langle 4, 3\rangle$. Using the same approach, these lists are sorted, giving $\langle 1\rangle$ and $\langle 3, 4\rangle$, and concatenated with the item 2 in between: $\langle 1, 2, 3, 4\rangle$.

The two sorting procedures, although giving the same result, seem very different. Are they? By running the routines on some sample data, it is easy to see that they

[1] Some analogue devices can calculate values in a continuous manner. For example, one can build a device that, given two numbers encoded as voltages, outputs their sum as a voltage. This is an analogue device since its inputs and outputs can be varied in an infinitude of ways in arbitrarily small steps. Normal computers calculate things by first expressing them in terms of ones and zeros, thus allowing us to only change things in steps by changing ones to zeros and vice versa.

sort the numbers in different ways and take different times to sort the same list of numbers. But is one better than the other? If so, in what way is it better, and by what measure? Two important measures for computer programs are memory usage and execution time. We would like to be able to analyse these algorithms mathematically to be able to compare their relative efficiency. Wouldn't it be sufficient to test them together to see which runs faster? Although one could, such an analysis may not be easily generalised, and would not help us answer questions such as whether there are procedures which are even more efficient than these.

It can be mathematically proved that both algorithms work correctly. However, given a list of $n$ numbers, on average the first algorithm (called *Selection Sort*) makes a number of comparisons proportional to $n^2$ to sort, while the second algorithm (called *Quicksort*) takes a number of steps proportional to $n \log n$. Since $\log n$ is smaller than $n$, the latter algorithm is preferred.

### *1.4.2 Formal Specification of Requirements*

Every program is written to solve a problem. To judge whether or not a program works as intended, one must also know precisely what problem it intends to solve. Do the sorting routines above work? Well, it depends. If the task is to reverse a list of numbers, they certainly do not. However, if the specification is to sort the numbers in ascending order, they work correctly.

When software or hardware producers start on a project to design a system, the first step is to identify and document the requirements of the client or user. Most projects use informal notation (such as English), which means that ambiguity is inevitable, and formal analysis impossible. However, another school of thought proposes that mathematics be used to document the requirements. We have already discussed how effective mathematics is in describing things in a precise yet abstract fashion. We will see various examples of this later on in the book.

### *1.4.3 Reasoning About Programs*

Once we have a specification of the requirements and the program itself, we would like to know whether or not the program works as expected. One way to do this is to feed the program with a large number of inputs and check that the outputs are as expected. However, to quote the computer scientist Edsger Dijkstra, "Testing shows the presence, not the absence of bugs". Even if it is usually not possible to test with every possible input, various techniques have been developed to intelligently choose values to test for or to quantify how much of the system's potential behaviour has been tested. But if testing can only detect the presence, not absence of bugs, can one ever be sure that a program is bug-free, or at least that it conforms to its specification?

Take an engineer trying to verify whether a particular bridge can take the weight of 50 cars without collapsing. One approach would be to test the statement by placing 50 cars in various configurations on the bridge: well distributed, all at one end, all at the other end, etc. Apart from being dangerous, this testing would only show us that, as long as the cars are in the positions we have tried, the bridge will not collapse. What a smart engineer would do, however, is to use the laws of physics to prove mathematically that no 50-car load distribution over the bridge would cause it to collapse. Similarly, since the 1960s, computer scientists have tried to develop analogous techniques for doing the same with computer programs.

The problem is that computer programs are not easily describable using the laws of physics.[2] To address this, computer scientists have developed and used various logics and mathematical tools to give meaning to computer programs. The translation of a program into a logic statement which describes its behaviour is called the *formal semantics* of the program. To check that a particular program satisfies a certain requirement, it suffices to translate the program into logic and prove that the requirements always follow from this logic description. Obviously, the requirements would also have to be written in a mathematical notation.

This is easier said than done. The proofs are usually very long, difficult and tedious to complete. However, when lives are on the line, it may still be desirable to verify critical sections of the code. Over recent decades, techniques have been developed to perform such verification in an automatic or semi-automatic manner.

### *1.4.4 The Limits of Computing*

Is the computer omnipotent? Given a mathematical specification, can one always write a program to solve it? In the 1930s, Alan Turing proved that certain problems can never be solved using an algorithm. Surprisingly, it was not a difficult mathematical problem that he identified as incomputable, but one that most new programmers would think difficult, yet possible. Consider the problem of writing a program which decides if a given algorithm terminates. Obviously, we can write a program which can tell us whether particular programs terminate, however it will not work on all programs. For instance, if a program contains no loops or recursive calls, one can deduce that it will terminate. However, this does not help us answer the question for the sorting programs we saw earlier, both of which always terminate. Even simulating the algorithm is not sufficient. If the algorithm terminates, the program will correctly announce that it terminates, but it will never say anything about programs which do not terminate. Turing showed that, no matter how smart a programmer is, it is impossible to come up with a program which always works.

[2] Well, of course, if we model the transistors which make up the computer, the program would just be an instance of the transistors being set with particular voltages. However, we would be committing the cardinal sin of not abstracting away unnecessary detail, resulting in a precise description of the behaviour of a computer, but one too big to be of any practical use.

But couldn't Turing have been wrong? How could he have shown that something is impossible? Later on in this book, we will look at a proof of Turing's result, and you can judge for yourself. Interestingly, his proof, published in the 1930s, is still applicable to today's computing devices.

So, next time you are working with a compiler company, and you are asked to add an option to the compiler which reports whether the program being compiled terminates, just shrug your shoulders as you cite Turing's paper.

### *1.4.5 A Physical Embodiment of Mathematics*

Computing was originally an offspring of mathematics. In the first half of the 20th century, mathematicians built all sorts of mathematical computation models. These were embodied into the first computers, and, conceptually, the computer has not changed much since then.

Mathematics is the reduction of domains into a finite description (the rules and basic truths) which can be used to construct complex truths. In a similar fashion, using a programming language is to give a finite description which can be used to solve complex problems. The similarity is not coincidental. Computers are a direct physical embodiment of mathematics. Computing and mathematics are intimately entwined, as the rest of the book will reveal.

## 1.5 Terminology

Finally, a short note about some terminology used in this (and other) books. When one makes a mathematical statement with no justification but which is believed to be true, it is called a *conjecture*. Once proved, it is no longer called a conjecture, but a theorem, lemma, proposition or corollary. The distinction between these names is not very important, but they are usually used to identify the importance of the result. Fundamental and general results are called *theorems*. In the process of proving a theorem, one sometimes needs to prove intermediate results which are interesting but are more of a stepping stone towards the theorem. These are called *lemmata* (singular *lemma*). When something follows almost immediately from a theorem, it is called a *corollary*. Finally, *propositions* are statements which follow directly from definitions. Apart from the term *conjecture,* the choice of which of the other terms to use may be rather arbitrary and not very important. No one will protest that a lemma should be a proposition or a theorem. As long as it carries a proof with the statement no one really minds. However, these terms give a structure to the mathematical results which may help their understanding.

It will all seem confusing at first, but as you read through this text you will get more familiar with these terms, and this primer is meant more for later reference than to be understood at this stage. When you read through the first few chapters and start wondering 'But why is this result called a theorem, whereas the other is called a lemma?' just refer back to this brief outline.