Introduction

Computer Science is a diverse subject: it covers topics that range from the design of computer hardware through to programming that hardware so it can do useful things, and encompasses applications in fields such as Science, Engineering and the Arts. As a result, a modern Computer Science degree will often see students study topics that could be described as applied Mathematics (e.g., cryptography), applied Physics (e.g., quantum computing), Psychology (e.g., human-computer interaction), Philosophy (e.g., artificial intelligence), Biology (e.g., bio-informatics) and Linguistics (e.g., speech synthesis); such a list could go on and on.

On one hand, this diversity means trying to capture what Computer Science *is* can be quite hard and the subject can be misunderstood as a result. A prime example is that ICT, a subject often taught in schools, can give a false impression of what Computer Science would be about at University: if ICT gives the impression that a Computer Science degree would be about *using* a word processor for example, the reality is it relates more closely to *creating* the word processor itself. The knock-on effect of this potential misunderstanding is that recruiting students to do Computer Science degrees is much harder work than you might think.

But on the other hand, the same diversity is *tremendously* exciting. Bjarne Stroustrup, inventor of the C++ programming language, is quoted as stating that "our civilisation runs on software" and therefore, by implication, the Computer Scientists who create it do likewise. From the perspective of other fields this might seem like a wild claim; for example, an Engineer might argue various wonders of the ancient world were realised without assistance from computers! However, the increasing ubiquity of computing in our lives somewhat justifies the statement. Beyond software running on desktop and laptop computers, we are increasingly dependent on huge numbers of devices that have permeated society in less obvious ways. Examples include software running on embedded computers within consumer products such as automobiles and televisions, and software running on chip-and-pin credit cards. All of these examples form systems on which we routinely depend. From domestic tasks to communication, from banking to travel: Computer Science underpins almost *everything* we do, even if you cannot see it on the surface.

Beyond the challenge this presents, it has tangible advantages in terms of employment: Computer Science graduates are regularly snapped up by industries as diverse as finance, media and aerospace for example. The reason is obvious: of course they are highly skilled in the use and creation of technology, but more importantly they have been trained to understand and solve problems, and to challenge what is possible. These skills in particular turn out to be the "gold dust" that distinguishes graduates in Computer Science from other subjects; they also represent the main reason why such a high proportion of those graduates become entrepreneurs by developing their own ideas into their own businesses.

So our goal here is to answer questions about, and encourage you to be interested in, the subject of Computer Science. More specifically, we want to answer questions such as "what is Computer Science" and "why should I study it at University". Of course, to some extent the answers will differ for everyone; different topics within the subject excite different people for example. But rather than give a very brief overview of many *topics*, our approach is to showcase specific topics all roughly relating to one *theme*: information security. This is a slightly subjective choice, but it turns out to allow a very natural connection between lots of things you are already familiar with (e.g., how compact discs work, or the problem of computer viruses) and theory in Computer Science that can help explain them. In each case, we try to demonstrate how the topic relates to other subjects (e.g., Mathematics) and how it allows us to solve *real* problems. You can think of each one as a "mini" lecture course: if you find one or more of them interesting, the chances are you would find a Computer Science degree interesting as well.

Intended Audience

This book attempts a careful balancing act between two intended types of reader, who have related but somewhat different needs. Clearly the book can still be useful if you fall outside this remit, but by focusing on these types we can make some assumptions about what you already know and hence the level of detail presented.

Students Our primary audience are students looking for an introduction to Computer Science. Two examples stand out:

1. A student studying Mathematics (perhaps ICT or even Computer Science) in later years of secondary or further education.

   In this case our goal is to give a taste of what Computer Science is about, and to provide a connection to your existing studies. Either way, we view you as a target to recruit into a Computer Science degree at University!
2. A student in early years of a Computer Science degree at University, or taking flavours of such a degree from within another subject (e.g., as an optional or elective course).

   In this case our goal is to (re)introduce topics and challenges you may already be faced with; offering a new, and hopefully accessible perspective can help understand basic concepts and motivate further study by illustrating concrete applications.

Teachers Our secondary audience are teachers. Anecdotally at least, we have found two use-cases that seem important:

1. Where there is a lack of specialist staff, non-specialists or early career staff members are often tasked with teaching Computer Science. In some cases, for example with content in Mathematics, there is clear synergy or even overlap. Either way however, this task can be very challenging.

   In this case the book can be used more or less as a student would. Although you might absorb the material quicker or more easily, it still offers an good introduction to the subject as a whole.
2. Even *with* specialist staff, it can be hard to find and/or develop suitable material; there are an increasing number of good resources online, but still few that focus on fundamentals of the subject and hence form a link to University-level study.

   In this case, the book can act as a useful way to enrich existing lesson plans, or as further reading where appropriate. The tasks embedded in the material, in particular, offer potential points for discussion or work outside the classroom.

Overview of Content

Some books are intended to be read from front-to-back in one go; others are like reference books, where you can dive into a specific part if or when you need to. This book sits somewhere between these types. It is comprised of various parts as explained below, each with specific goals and hence a suggested approach to reading the associated material.

Core Material

The first part is concerned with the fundamentals of Computer Science, and outlines concepts used variously elsewhere in the book. As a result, each chapter in this part should ideally be read in order, and before starting any other part:

Chapter 1 introduces the idea that numbers can be represented in different ways, and uses this to discuss the concepts of data compression, and error detection and correction. The example context is CDs and DVDs. Both store vast amounts of data and are fairly robust to damage; the question is, how do they do this?

Chapter 2 is a brief introduction to the study of algorithms, i.e., formal sets of directions which describe how to perform a task. The idea is to demonstrate that algorithms are fundamental tools in Computer Science, but no more complicated than a set of driving directions or cookery instructions. By studying how algorithms behave, the chapter shows that we can compare them against each other and select the best one for a particular task.

Chapter 3 uses the example of computer viruses to show how computers actually work, i.e., how they are able to execute the programs (or software) we use every day. The idea is to show that there is no magic involved: even modern computers are based on fairly simple principles which everyone can understand.

Chapter 4 highlights the role of data structures, the natural companion to algorithms. By using the example of strings (which are sequences of characters) the chapter shows why data structures are needed, and how their design can have a profound influence on algorithms that operate on them.

Chapter 5 deals with one of the most ubiquitous and influential information systems available on the Internet: Google web-search. Using only fairly introductory Mathematics, it explains how the system seems able to "understand" the web and hence produce such good results for a given search query.

The second part focuses specifically on examples drawn from cryptography and information security. After completing the first part, the idea is that each chapter in the second part can be read more or less independently from the rest:

Chapter 6 gives a brief introduction to what we now regard as historical schemes for encrypting messages. The goal is to demonstrate the two sides of cryptography: the constructive side where new schemes are designed and used, and the destructive side where said schemes are attacked and broken. By using simple programs available on every UNIX-based computer, the chapter shows how historical cryptanalysts were able to decrypt messages their sender thought were secure.

Chapter 7 discusses the idea of randomness: what does random even mean, and when can we describe a number as random? The idea is that random numbers often play a crucial role in cryptography, and using just some simple experiments one can demonstrate the difference between "good" and "bad" randomness.

Chapter 8 shifts the focus from history to the present day. By giving a more complete overview of a specific area in Mathematics (i.e., the idea of modular arithmetic) it describes two modern cryptographic schemes that more or less all of us rely on every day.

Chapter 9 introduces the concept of steganography which relates to hiding secret data within non-secret data. This allows a discussion of how computers represent and process images (e.g., those taken with a digital camera) and how simple steganographic techniques can embed secret messages in them.

Chapter 10 approaches the idea of security from a different perspective than the description in Chap. 8. As well as reasoning in theory about how secure a system is, we *also* need to worry about whether the physical system leaks information or not. The example scenario is guessing passwords: can you break into a computer without having to try every possible password, i.e., use any leaked information to help you out?

Supplementary Material

In addition to electronic copies of each core chapter, various supplementary material is available online at

http://www.cs.bris.ac.uk/home/page/teaching/wics.html

This includes additional chapter(s) extending the range of topics covered, plus Appendices providing extra introduction and explanation for topics we think might need it.

Perhaps the single most important instance is an Appendix supporting the use of BASH within examples and tasks. If you have no experience with BASH, which is common, the examples can be confusing. As such, we have written a short tutorial that includes a high-level overview of BASH itself, plus lower-level explanations of every command used in the book. The best approach is arguably to use it for reference as you read through each chapter: whenever you encounter something unfamiliar or confusing, take some time to look through the Appendix which *should* provide an explanation.

Embedded Tasks

To ensure the content is as practically oriented as possible, various tasks are embedded into the material alongside the fixed examples. These fall into various categories:

Implement (task #1)

These tasks represent implementation challenges, usually with a clear or fixed answer or outcome. In some cases the task might act as a prompt to reproduce or extend an example; in other cases the task might ask you to design something, e.g., an algorithm based on an information description of something in the material.

Research (task #2)

These tasks outline topics that represent a good next step on from the material presented: you are interested in the material but want to learn more, or about a specific aspect in more depth, these give some suggestions of what to look at.

They often present open-ended challenges or questions, and as a result often make good discussion points. Either way, the idea is that you stop reading through the material, and attempt to solve the task yourself (rather than rely on the resources provided).

Some tasks of this type will be harder than others, but none are designed to represent a significant amount of work: if you get stuck on one, there is no problem with just skipping it and moving on.

Notation

Throughout the book we have tried to make the notation used as simple and familiar as possible. On the other, hand some notation is inevitable: we need a way to express sets and sequences for instance.

Ranges

When we write $a \ldots b$ for a starting point a and a finishing point b, we are describing a **range** that includes all numbers between (and including) a and b. So writing $0 \ldots 7$ is basically the same as writing 0, 1, 2, 3, 4, 5, 6, 7. If we say c is *in* the range $0 \ldots 7$ we mean that

$$0 \leq c \leq 7$$

i.e., c is one of 0, 1, 2, 3, 4, 5, 6 and 7.

Sequences

We write a **sequence** of **elements** called A, which you can think of as like a list, as follows

$$A = \langle 0, 3, 1, 2 \rangle.$$

This sequence contains elements which are numbers, but it is important to keep in mind that elements can be *any* objects we want. For example we could write a sequence of characters such as

$$B = \langle \text{‘a’}, \text{‘b’}, \text{‘c’}, \text{‘d’}, \text{‘e’} \rangle.$$

Either way, we know the size of A and B, i.e., the number of elements they contain; in the case of A we write this as $|A|$ so that $|A| = 4$ for instance.

In a sequence, the order of the elements is important, and we can refer to each one using an **index**. When we want to refer to the i-th element in A for example (where i is the index) we write A_i. Reading the elements left-to-right, within A we have that $A_0 = 0$, $A_1 = 3$, $A_2 = 1$ and $A_3 = 2$. Note that we count from zero, so the first element is A_0, and that referring to the element A_4 is invalid (because there is no element in A with index 4). Using $\perp$ to mean invalid, we write $A_4 = \perp$ to make this more obvious.

Sometimes it makes sense to save space by not writing all the elements in a given sequence. For example we might rewrite B as

$$B = \langle \text{‘a’}, \text{‘b’}, \ldots, \text{‘e’} \rangle$$

where the continuation dots written as $\ldots$ represent elements ‘c’ and ‘d’ which have been left out: we assume whoever reads the sequence can fill in the $\ldots$ part appropriately. This means it should always be clear and unambiguous what $\ldots$ means. This way of writing B still means we know what $|B|$ is, and also that $B_5 = \perp$ for example. Another example is the sequence C written as

$$C = \langle 0, 3, 1, 2, \ldots \rangle.$$

When we used continuation dots in B, there was a well defined start and end to the sequence so they were just a short-hand to describe elements we did not want to write down. However, with C the continuation dots now represent elements either we do not know, or do not matter: since there is no end to the sequence we cannot necessarily fill in the ... part appropriately as before. This also means we might not know what $|C|$ is, or whether $C_4 = \perp$ or not.

It is possible to join together, or **concatenate**, two sequences. For example, imagine we start with two 4-element sequences

$$\begin{aligned} D &= \langle 0, 1, 2, 3 \rangle \\ E &= \langle 4, 5, 6, 7 \rangle \end{aligned}$$

and want to join them together; we would write

$$F = D \parallel E = \langle 0, 1, 2, 3 \rangle \parallel \langle 4, 5, 6, 7 \rangle = \langle 0, 1, 2, 3, 4, 5, 6, 7 \rangle.$$

Notice that the result F is an 8-element sequence, where $F_{0...3}$ are those from D and $F_{4...7}$ are those from E.

Sets

The concept of a **set** and the theory behind such structures is fundamental to Mathematics. A set is an unordered collection of **elements**; as with a sequence, the elements can be anything you want. We can write a set called A by listing the elements between a pair of braces as follows

$$A = \{2, 3, 4, 5, 6, 7, 8\}.$$

This set contains the whole numbers between two and eight inclusive. The size of a set is the number of elements it contains. For the set A this is written $|A|$, so we have that $|A| = 7$. If the element a is in the set A, we say a is a **member** of A or write

$$a \in A.$$

We know for example that $2 \in A$ but $9 \notin A$, i.e., 2 is a member of the set A, but 9 is not. Unlike a sequence, the ordering of the elements in a set does *not* matter, only their membership or non-membership. This means we *cannot* refer to elements in A as A_i. However, if we define another set

$$B = \{8, 7, 6, 5, 4, 3, 2\},$$

we can be safe in the knowledge that $A = B$. Note that elements cannot occur in a set more than once.

As with sequences, it sometimes makes sense to save space by not writing all the elements in a set. For example we might rewrite the set A as

$$A = \{2, 3, \ldots, 7, 8\}.$$

Sometimes we might want to write a set with unknown size such as

$$C = \{2, 4, 6, 8, \ldots\}.$$

This set is infinite in size in the sense there is no end: it represents all even whole numbers starting at two and continuing to infinity. In this case, the continuation dots are a necessity; if we did not use them, we could not write down the set at all.

Frequently Asked Questions (FAQs)

I have a question/comment/complaint for you. Any (positive *or* negative) feedback, experience or comment is very welcome; this helps us to improve and extend the material in the most useful way. To get in contact, email

page@cs.bris.ac.uk

or

nigel@cs.bris.ac.uk

We are not perfect, so mistakes are of course possible (although hopefully rare). Some cases are hard for us to check, and make your feedback even more valuable: for instance

1. minor variation in software versions can produce subtle differences in how some commands and hence examples work, and
2. some examples download and use online resources, but web-sites change over time (or even might differ depending on where you access them from) so might cause the example to fail.

Either way, if you spot a problem then let us know: we will try to explain and/or fix things as fast as we can!

Why are all your references to Wikipedia? Our goal is to give an easily accessible overview, so it made no sense to reference lots of research papers. There are basically two reasons why: research papers are often written in a way that makes them hard to read (even when their intellectual content is not difficult to understand), and although many research papers are available on the Internet, many are not (or have to be paid for). So although some valid criticisms of Wikipedia exist, for introductory material on Computer Science it certainly represents a good place to start.

I like programming; why do the examples include so little programming? We want to focus on interesting topics rather than the mechanics of programming. So even when we include example programs, the idea is to do so in a way where their meaning is fairly clear. For example it makes more sense to use pseudocode algorithms or reuse existing software tools than complicate a description of something by including pages and pages of program listings.

If programming really is your sole interest, you might prefer

S.S. Skiena and M.A. Revilla.
Programming Challenges: The Programming Contest Training Manual.
Springer, 2003. ISBN: 978-0387001630.

which offers a range of programming challenges; the excellent online resource

http://projecteuler.net/

is similar, although with greater emphasis on problems grounded in Mathematics.

But you need to be able to program to do Computer Science, right? Yes! But only in the same way as you need to be able to read and write to study English. Put another way, reading and writing, or grammar and vocabulary, are just tools: they simply allow us to study topics such as English literature. Computer Science is the same. Although it *is* possible to study programming as a topic in itself, we are more interested in what can be achieved *using* programs: we treat programming itself as another tool.

Are there any other things like this I can read? There are many books about specific topics in Computer Science, but somewhat fewer which overview the subject itself. Amongst these, some excellent examples are the following:

A.K. Dewdney.
The New Turing Omnibus.
Palgrave-Macmillan, 2003. ISBN: 978-0805071665.

B. Vöcking, H. Alt, M. Dietzfelbinger, R. Reischuk, C. Scheideler, H. Vollmer and D. Wagner.
Algorithms Unplugged.
Springer, 2011. ISBN: 978-3642153273.

J. MacCormick.
Nine Algorithms That Changed the Future: The Ingenious Ideas that Drive Today's Computers.
Princeton University Press, 2011. ISBN: 978-0691147147.

There are of course innumerable web-site, blog and wiki style resources online. Some structured examples include the CS4FN (or "Computer Science for fun") series from Queen Mary, University of London, UK

http://www.dcs.qmul.ac.uk/cs4fn/

and Computer Science Unplugged series from the University of Canterbury, New Zealand

http://csunplugged.org/

the latter of which now also offers downloadable and printable books ideal for use in earlier stages of school.

Acknowledgements

This book was typeset with LaTeX, originally developed by Leslie Lamport and based on TeX by Donald Knuth; among the numerous packages used, some important examples include adjustbox by Martin Scharrer, algorithm2e by Christophe Fiorio, listings by Carsten Heinz, PGF and TiKZ by Till Tantau, and pxfonts by Young Ryu. The embedded examples make heavy use of the BASH shell by Brian Fox, and numerous individual commands developed by members of the GNU project housed at

http://www.gnu.org

Throughout the book, images from sources other than the authors have been carefully reproduced under permissive licenses only; each image of this type notes both the source and license in question.

We owe a general debt of gratitude to everyone who has offered feedback, ideas or help with production and publication of this book. We have, in particular, benefited massively from tireless proof reading by Valentina Banciu, David Bernhard, Jake Longo Galea, Simon Hoerder, Daniel Martin, Luke Mather, Jim Page, and Carolyn Whitnall; any remaining mistakes are, of course, our own doing. We would like to acknowledge the support of the EPSRC (specifically via grant EP/H001689/1), whose model for outreach and public engagement was instrumental in allowing development of the material. All royalties resulting from printed versions of the book are donated to the Computing At School (CAS) group, whose ongoing work can be followed at

http://www.computingatschool.org.uk/

We thank Simon Rees and Wayne Wheeler (Springer) for making publication such a smooth process, and additionally Simon Humphreys (CAS) and Jeremy Barlow (BCS) for dealing ably with non-standard administrative overhead of the royalties arrangement.

Bristol, UK

Daniel Page
Nigel Smart

Springer

http://www.springer.com/978-3-319-04041-7

What Is Computer Science?
An Information Security Perspective
Page, D.; Smart, N.
2014, XVIII, 232 p. 84 illus., Softcover
ISBN: 978-3-319-04041-7