

Preface

This book is an introduction to the analysis of algorithms, from the point of view of proving algorithm correctness. Our theme is the following: how do we argue mathematically, without a burden of excessive formalism, that a given algorithm does what it is supposed to do? And why is this important? In the words of C.A.R. Hoare:

As far as the fundamental science is concerned, we still certainly do not know how to prove programs correct. We need a lot of steady progress in this area, which one can foresee, and a lot of breakthroughs where people suddenly find there's a simple way to do something that everybody hitherto has thought to be far too difficult¹.

Software engineers know many examples of things going terribly wrong because of program errors; their particular favorites are the following two². The blackout in the American North-East during the summer of 2003 was due to a software bug in an energy management system; an alarm that should have been triggered never went off, leading to a chain of events that climaxed in a cascading blackout. The Ariane 5, flight 501, the maiden flight of the rocket in June 4, 1996, ended with an explosion 40 seconds into the flight; this \$500 million loss was caused by an overflow in the conversion from a 64-bit floating point number to a 16-bit signed integer.

While the goal of absolute certainty in program correctness is elusive, we can develop methods and techniques for reducing errors. The aim of this book is modest: we want to present an introduction to the analysis of algorithms—the “ideas” behind programs, and show how to prove their correctness.

¹From *An Interview with C.A.R. Hoare*, in [Shustek (2009)].

²These two examples come from [van Vliet (2000)], where many more instances of spectacular failures may be found.

The algorithm may be correct, but the implementation itself might be flawed. Some syntactical errors in the program implementation may be uncovered by a compiler or translator—which in turn could also be buggy—but there might be other hidden errors. The hardware itself might be faulty; the libraries on which the program relies at run time might be unreliable, etc. It is the task of the software engineer to write code that works given such a delicate environment, prone to errors. Finally, the algorithmic content of a piece of software might be very small; the majority of the lines of code could be the “menial” task of interface programming. Thus, the ability to argue correctly about the soundness of an algorithm is only one of many facets of the task at hand, yet an important one, if only for the pedagogical reason of learning to argue rigorously about algorithms.

We begin this book with a chapter of preliminaries, containing the key ideas of induction and invariance, and the framework of pre/post-conditions and loop invariants.

The remaining, purely algorithmic, contents of the book are as follows. We present three standard algorithm design techniques in eponymous chapters: greedy algorithms, dynamic programming and the divide and conquer paradigm. We are concerned with correctness of algorithms, rather than, say, efficiency or the underlying data structures. For example, in the chapter on the greedy paradigm we explore in depth the idea of a *promising partial solution*, a powerful technique for proving the correctness of greedy algorithms. We also include online algorithms and the idea of *competitive analysis*, and the last chapter is an introduction to randomized algorithms, with a section on cryptography.

The intended audience for this book consists of undergraduate students in computer science and mathematics. The book is very self-contained: the first chapter, Preliminaries, reviews induction and the invariance principle. It also introduces the aforementioned ideas of pre/post-conditions, loop invariants and termination—in other words, it sets the mathematical stage for the rest of the book. Not much mathematics is assumed (besides some tame forays into linear algebra and number theory), but a certain penchant for discrete mathematics is considered helpful.

Algorithms solve problems, and many of the problems in this book fall under the category of *optimization problems*, whether cost minimization, such as Kruskal’s algorithm for computing minimum cost spanning trees—section 2.1, or profit maximization, such as selecting the most profitable subset of activities—section 4.5.

This book draws on many sources. First of all, [Cormen *et al.* (2001)] is a fantastic reference for anyone who is learning algorithms. I have also used as reference the elegantly written [Kleinberg and Tardos (2006)]. A classic in the field is [Knuth (1997)], and I base my presentation of online algorithms on the material in [Borodin and El-Yaniv (1998)]. I have learned greedy algorithms and dynamic programming from Stephen A. Cook at the University of Toronto. Appendix B, on relations, is based on hand-written lectures slides of Ryszard Janicki.

No book on algorithms is complete without a short introduction to the “big-Oh” notation. We say that $g(n) \in O(f(n))$ if there exist constants c, n_0 such that for all $n \geq n_0$, $g(n) \leq cf(n)$, and the *little-oh* notation, $g(n) \in o(f(n))$, which denotes that $\lim_{n \rightarrow \infty} g(n)/f(n) = 0$. We also say that $g(n) \in \Omega(f(n))$ if there exist constants c, n_0 such that for all $n \geq n_0$, $g(n) \geq cf(n)$. Finally, we say that $g(n) \in \Theta(f(n))$ if it is the case that both $g(n) \in O(f(n))$ and $g(n) \in \Omega(f(n))$.

The ubiquitous *floor* and *ceil* functions are defined, respectively, as follows: $\lfloor x \rfloor = \max\{n \in \mathbb{Z} | n \leq x\}$ and $\lceil x \rceil = \min\{n \in \mathbb{Z} | n \geq x\}$.