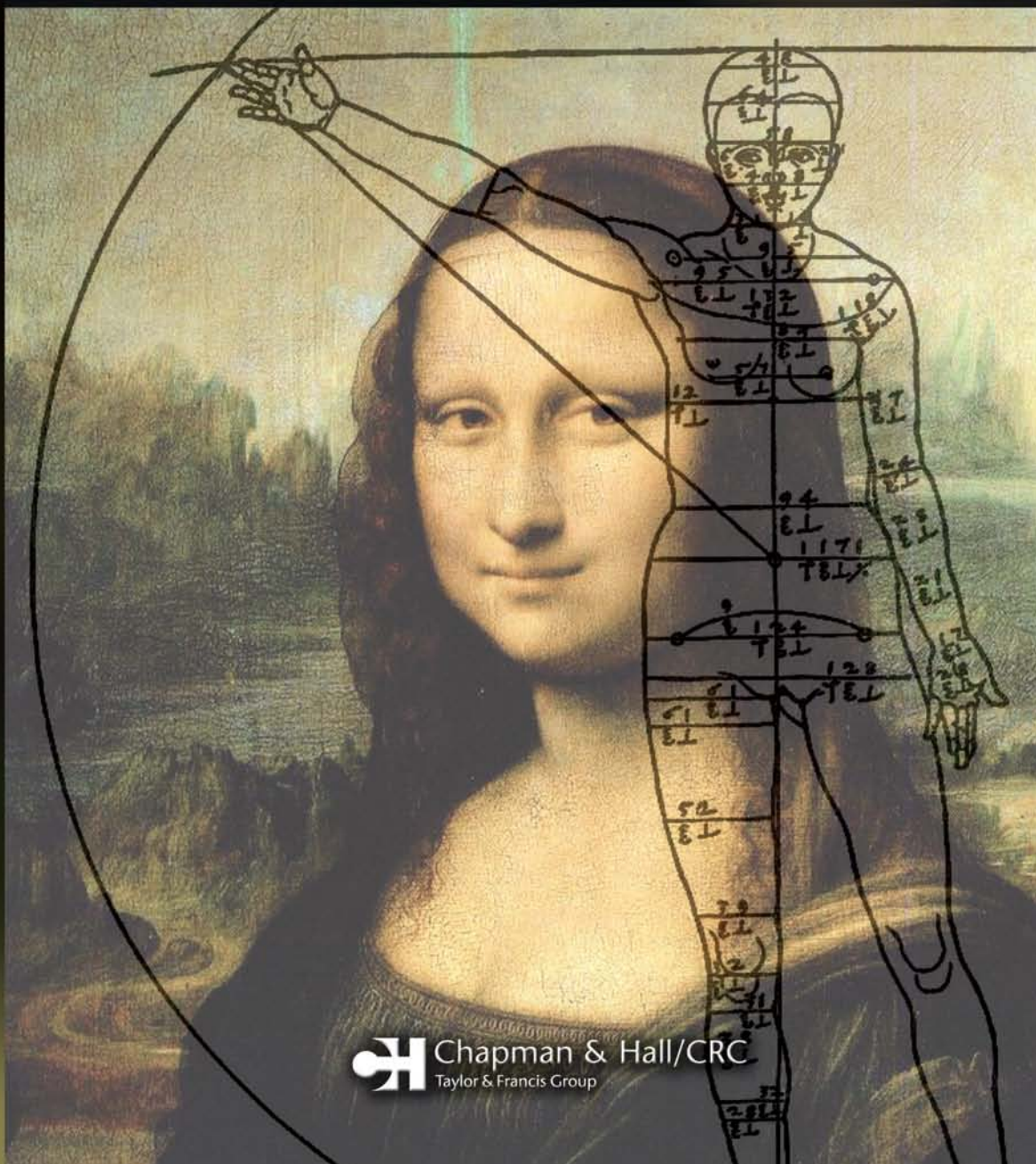
A PROGRAMMER'S COMPANION TO ALGORITHM ANALYSIS

Ernst L. Leiss



Chapman & Hall/CRC
Taylor & Francis Group

A PROGRAMMER'S COMPANION TO ALGORITHM ANALYSIS

A PROGRAMMER'S COMPANION TO ALGORITHM ANALYSIS

Ernst L. Leiss
University of Houston,
Texas, U.S.A.

Chapman & Hall/CRC
Taylor & Francis Group
Boca Raton London New York

Chapman & Hall/CRC is an imprint of the
Taylor & Francis Group, an informa business

Chapman & Hall/CRC
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number-10: 1-58488-673-0 (Softcover)
International Standard Book Number-13: 978-1-58488-673-0 (Softcover)

Library of Congress Cataloging-in-Publication Data

Leiss, Ernst L., 1952-
A programmer's companion to algorithm analysis / Ernst L. Leiss.
p. cm.
Includes bibliographical references and index.
ISBN 1-58488-673-0 (acid-free paper)
1. Programming (Mathematics) 2. Algorithms--Data processing. I. Title.

QA402.5.L398 2006
005.1--dc22 2006044552

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Preface

The primary emphasis of this book is the transition from an algorithm to a program. Given a problem to solve, the typical first step is the design of an algorithm; this algorithm is then translated into software. We will look carefully at the interface between the design and analysis of algorithms on the one hand and the resulting program solving the problem on the other. This approach is motivated by the fact that algorithms for standard problems are readily available in textbooks and literature and are frequently used as building blocks for more complex designs. Thus, the correctness of the algorithm is much less a concern than its adaptation to a working program.

Many textbooks, several excellent, are dedicated to algorithms, their design, their analysis, the techniques involved in creating them, and how to determine their time and space complexities. They provide the building blocks of the overall design. These books are usually considered part of the theoretical side of computing. There are also numerous books dedicated to designing software, from those concentrating on programming in the small (designing and debugging individual programs) to programming in the large (looking at large systems in their totality). These books are usually viewed as belonging to software engineering. However, there are no books that look systematically at the gap separating the theory of algorithms and software engineering, even though many things can go wrong in taking several algorithms and producing a software product derived from them.

This book is intended to fill this gap. It is not intended to teach algorithms from scratch; indeed, I assume the reader has already been exposed to the ordinary machinery of algorithm design, including the standard algorithms for sorting and searching and techniques for analyzing the correctness and complexity of algorithms (although the most important ones will be reviewed). Nor is this book meant to teach software design; I assume that the reader has already gained experience in designing reasonably complex software systems. Ideally, the readers' interest in this book's topic was prompted by the uncomfortable realization that the path from algorithm to software was much more arduous than anticipated, and, indeed, results obtained on the theory side of the development process, be they results derived by readers or acquired from textbooks, did not translate satisfactorily to corresponding results, that is, performance, for the developed software. Even if the reader has never encountered a situation where the performance predicted by the complexity analysis of a specific algorithm did not correspond to the performance observed by running the resulting software, I argue that such occurrences are increasingly more likely, given

the overall development of our emerging hardware platforms and software environments.

In many cases, the problems I will address are rooted in the different way memory is viewed. For the designer of an algorithm, memory is inexhaustible, has uniform access properties, and generally behaves *nicely* (I will be more specific later about the meaning of *niceness*). Programmers, however, have to deal with memory hierarchies, limits on the availability of each class of memory, and the distinct nonuniformity of access characteristics, all of which imply a definite absence of niceness. Additionally, algorithm designers assume to have complete control over their memory, while software designers must deal with several agents that are placed between them and the actual memory — to mention the most important ones, compilers and operating systems, each of which has its own idiosyncrasies. All of these conspire against the software designer who has the naïve and often seriously disappointed expectation that properties of algorithms easily translate into properties of programs.

The book is intended for software developers with some exposure to the design and analysis of algorithms and data structures. The emphasis is clearly on practical issues, but the book is naturally dependent on some knowledge of standard algorithms — hence the notion that it is a companion book. It can be used either in conjunction with a standard algorithm text, in which case it would most likely be within the context of a course setting, or it can be used for independent study, presumably by practitioners of the software development process who have suffered disappointments in applying the theory of algorithms to the production of efficient software.

Contents

Foreword

The foremost goal for (most) computer scientists is the creation of efficient and effective programs. This premise dictates a disciplined approach to software development. Typically, the process involves the use of one or more suitable algorithms; these may be standard algorithms taken from textbooks or literature, or they may be custom algorithms that are developed during the process. A well-developed body of theory is related to the question of what constitutes a good algorithm. Apart from the obvious requirement that it must be correct, the most important quality of an algorithm is its efficiency. Computational complexity provides the tools for determining the efficiency of an algorithm; in many cases, it is relatively easy to capture the efficiency of an algorithm in this way. However, for the software developer the ultimate goal is efficient software, not efficient algorithms. Here is where things get a bit tricky — it is often not well understood how to go from a good algorithm to good software. It is this transition that we will focus on.

This book consists of two complementary parts. In the first part we describe the idealized universe that algorithm designers inhabit; in the second part we outline how this ideal can be adapted to the real world in which programmers must dwell. While the algorithm designer's world is idealized, it nevertheless is not without its own distinct problems, some having significance for programmers and others having little practical relevance. We describe them so that it becomes clear which are important in practice and which are not. For the most part, the way in which the algorithm designer's world is idealized becomes clear only once it is contrasted with the programmer's.

In Chapter 1 we sketch a taxonomy of algorithmic complexity. While *complexity* is generally used as a measure of the performance of a program, it is important to understand that there are several different aspects of complexity, all of which are related to performance but reflect it from very different points of view. In Chapter 2 we describe precisely in what way the algorithm designer's universe is idealized; specifically, we explore the assumptions that fundamentally underlie the various concepts of algorithmic complexity. This is crucially important since it will allow us to understand how disappointments may arise when we translate an algorithm into a program.

This is the concern of the second part of this book. In Chapter 4 we explore a variety of ways in which things can go wrong. While there are many causes of software behaving in unexpected ways, we are concerned only with those where a significant conceptual gap may occur between what the algorithm analysis indicates and what the eventual observations of the resulting

program demonstrate. Specifically, in this chapter we look at ways in which slight variations in the (implied) semantics of algorithms and software may cause the software to be incorrect, perform much worse than predicted by algorithmic analysis, or perform unpredictably. We also touch upon occasions where a small change in the goal, a seemingly innocuous generalization, results in (quite literally) impossible software. In order for this discussion to develop in some useful context, Part 1 ends (in Chapter 3) with a discussion of analysis techniques and sample algorithms together with their worked-out analyses. In Chapter 5 we discuss extensively the rather significant implications of the memory hierarchies that typically are encountered in modern programming environments, whether they are under the direct control of the programmer (e.g., out-of-core programming) or not (e.g., virtual memory management). Chapter 6 focuses on issues that typically are never under the direct control of the programmer; these are related to actions performed by the compiling system and the operating system, ostensibly in support of the programmer's intentions. That this help comes at a sometimes steep price (in the efficiency of the resulting programs) must be clearly understood. Many of the disappointments are rooted in memory issues; others arise because of compiler- or language-related issues.

The next three chapters of Part 2 are devoted to somewhat less central issues, which may or may not be of concern in specific situations. Chapter 7 examines implicit assumptions made by algorithm designers and their implications for software; in particular, the case is made that exceptions must be addressed in programs and that explicit tests for assumptions must be incorporated in the code. Chapter 8 considers the implications of the way numbers are represented in modern computers; while this is mainly of interest when dealing with numerical algorithms (where one typically devotes a good deal of attention to error analysis and related topics), occasionally questions related to the validity of mathematical identities and similar topics arise in distinctly nonnumerical areas. Chapter 9 addresses the issue of constant factors that are generally hidden in the asymptotic complexity derivation of algorithms but that matter for practical software performance. Here, we pay particular attention to the notion of crossover points. Finally, in Chapter 10 we look at the meaning of undecidability for software development; specifically, we pose the question of what to do when the algorithm text tells us that the question we would like to solve is undecidable. Also examined in this chapter are problems arising from excessively high computational complexities of solution methods.

Four appendices round out the material. Appendix I briefly outlines which basic algorithms should be familiar to all programmers. Appendix II presents a short overview of some systems that are implicated in the disappointments addressed in Part 2. In particular, these are the memory hierarchy, virtual memory management, optimizing compilers, and garbage collection. Since each of them can have dramatic effects on the performance of software, it is sensible for the programmer to have at least a rudimentary appreciation of them. Appendix III gives a quick review of NP-completeness, a concept that

for many programmers appears rather nebulous. This appendix also looks at higher-complexity classes and indicates what their practical significance is. Finally, Appendix IV sketches undecidability, both the halting problem for Turing machines and the Post's Correspondence Problem. Since undecidability has rather undesirable consequences for software development, programmers may want to have a short synopsis of the two fundamental problems in undecidability.

Throughout, we attempt to be precise when talking about algorithms; however, our emphasis is clearly on the practical aspects of taking an algorithm, together with its complexity analysis, and translating it into software that is expected to perform as close as possible to the performance predicted by the algorithm's complexity. Thus, for us the ultimate goal of designing algorithms is the production of efficient software; if, for whatever reason, the resulting software is not efficient (or, even worse, not correct), the initial design of the algorithm, no matter how elegant or brilliant, was decidedly an exercise in futility.

A Note on the Footnotes

The footnotes are designed to permit reading this book at two levels. The straight text is intended to dispense with some of the technicalities that are not directly relevant to the narrative and are therefore relegated to the footnotes. Thus, we may occasionally trade precision for ease of understanding in the text; readers interested in the details or in complete precision are encouraged to consult the footnotes, which are used to qualify some of the statements, provide proofs or justifications for our assertions, or expand on some of the more esoteric aspects of the discussion.

Bibliographical Notes

The two (occasionally antagonistic) sides depicted in this book are analysis of algorithms and software engineering. While numerous other fields of computer science and software production turn out to be relevant to our discussion and will be mentioned when they arise, we want to make at least some reference to representative works of these two sides. On the algorithm front, Knuth's *The Art of Computer Programming* is the classical work on algorithm design and analysis; in spite of the title's emphasis on programming, most practical aspects of modern computing environments, and especially the interplay of their different components, hardly figure in the coverage. Another influential work is Aho, Hopcroft, and Ullman's *The*

Design and Analysis of Computer Algorithms. More references are given at the end of Chapter 1.

While books on algorithms have hewn to a fairly uniform worldview over the decades, the software side is considerably more peripatetic; it has traditionally been significantly more trendy, prone to fads and fashions, perhaps reflecting the absence of a universally accepted body of theory that forms the backbone of the discipline (something clearly present for algorithms). The list below reflects some of this.

Early influential works on software development are Dijkstra, Dahl, et al.: *Structured Programming*; Aron: *The Program Development Process*; and Brooks: *The Mythical Man Month*. A historical perspective of some aspects of software engineering is provided by Brooks: *No Silver Bullet: Essence and Accidents of Software Engineering* and by Larman and Basili: *Iterative and Incremental Development: A Brief History*. The persistence of failure in developing software is discussed in Jones: *Software Project Management Practices: Failure Versus Success*; this is clearly a concern that has no counterpart in algorithm design. Software testing is covered in Bezier: *Software Testing Techniques*; Kit: *Software Testing in the Real World: Improving the Process*; and Beck: *Test Driven Development: By Example*. Various techniques for and approaches to producing code are discussed in numerous works; we give, essentially in chronological order, the following list, which provides a bit of the flavors that have animated the field over the years: Liskov and Guttag: *Abstraction and Specification in Program Development*; Booch: *Object-Oriented Analysis and Design with Applications*; Arthur: *Software Evolution*; Rumbaugh, Blaha, et al.: *Object-Oriented Modeling and Design*; Neilsen, *Usability Engineering*; Gamma, Helm, et al.: *Design Patterns: Elements of Reusable Object-Oriented Software*; Yourdon: *When Good-Enough Software Is Best*; Hunt and Thomas: *The Pragmatic Programmer: From Journeyman to Master*; Jacobson, Booch, and Rumbaugh: *The Unified Software Development Process*; Krutchen: *The Rational Unified Process: An Introduction*; Beck and Fowler: *Planning Extreme Programming*; and Larman: *Agile and Iterative Development: A Manager's Guide*.

Quite a number of years ago, Jon Bentley wrote a series of interesting columns on a variety of topics, all related to practical aspects of programming and the difficulties programmers encounter; these were collected in two volumes that appeared under the titles *Programming Pearls* and *More Programming Pearls: Confessions of a Coder*. These two collections are probably closest, in goals and objectives as well as in emphasis, to this book.

Part 1

The Algorithm Side: Regularity, Predictability, and Asymptotics

This part presents the view of the designer of algorithms. It first outlines the various categories of complexity. Then it describes in considerable detail the assumptions that are fundamental in the process of determining the algorithmic complexity of algorithms. The goal is to establish the conceptual as well as the mathematical framework required for the discussion of the practical aspects involved in taking an algorithm, presumably a good or perhaps even the best (defined in some fashion), and translating it into a good piece of software.[1]

The general approach in Chapter 1 will be to assume that an algorithm is given. In order to obtain a measure of its goodness, we want to determine its complexity. However, before we can do this, it is necessary to define what we mean by *goodness* since in different situations, different measures of quality might be applicable. Thus, we first discuss a taxonomy of complexity analysis. We concentrate mainly on the standard categories, namely time and space, as well as average-case and worst-case computational complexities. Also in this group of standard classifications falls the distinction

[1] It is revealing that optimal algorithms are often a (very legitimate) goal of algorithm design, but nobody would ever refer to optimal software.

between word and bit complexity, as does the differentiation between on-line and off-line algorithms. Less standard perhaps is the review of parallel complexity measures; here our focus is on the EREW model. (While other models have been studied, they are irrelevant from a practical point of view.) Also, in preparation of what is more extensively covered in Part 2, we introduce the notion of I/O complexity. Finally, we return to the fundamental question of the complexity analysis of algorithms, namely what is a good algorithm, and establish the importance of lower bounds in any effort directed at answering this question.

In Chapter 2 we examine the methodological background that enables the process of determining the computational complexity of an algorithm. In particular, we review the fundamental notion of statement counts and discuss in some detail the implications of the assumption that statement counts reflect execution time. This involves a detailed examination of the memory model assumed in algorithmic analysis. We also belabor a seemingly obvious point, namely that mathematical identities hold at this level. (Why we do this will become clear in Part 2, where we establish why they do not necessarily hold in programs.) We also discuss the asymptotic nature of complexity analysis, which is essentially a consequence of the assumptions underlying the statement count paradigm.

Chapter 3 is dedicated to amplifying these points by working out the complexity analysis of several standard algorithms. We first describe several general techniques for determining the time complexity of algorithms; then we show how these are applied to the algorithms covered in this chapter. We concentrate on the essential aspects of each algorithm and indicate how they affect the complexity analysis.

Most of the points we make in these three chapters (and all of the ones we make in Chapter 2) will be extensively revisited in Part 2 because many of the assumptions that underlie the process of complexity analysis of algorithms are violated in some fashion by the programming and execution environment that is utilized when designing and running software. As such, it is the discrepancies between the model assumed in algorithm design, and in particular in the analysis of algorithms, and the model used for software development that are the root of the disappointments to be discussed in Part 2, which frequently sneak up on programmers. This is why we spend considerable time and effort explaining these aspects of algorithmic complexity.

1

A Taxonomy of Algorithmic Complexity

About This Chapter

This chapter presents various widely used measures of the performance of algorithms. Specifically, we review time and space complexity; average, worst, and best complexity; amortized analysis; bit versus word complexity; various incarnations of parallel complexity; and the implications for the complexity of whether the given algorithm is on-line or off-line. We also introduce the input/output (I/O) complexity of an algorithm, even though this is a topic of much more interest in Part 2. We conclude the chapter with an examination of the significance of lower bounds for good algorithms.

1.1 Introduction

Suppose someone presents us with an algorithm and asks whether it is good. How are we to answer this question? Upon reflection, it should be obvious that we must first agree upon some criterion by which we judge the quality of the algorithm. Different contexts of this question may imply different criteria.

At the most basic level, the algorithm should be correct. Absent this quality, all other qualities are irrelevant. While it is by no means easy to ascertain the correctness of an algorithm,[1] we will assume here that it is given. Thus, our focus throughout this book is on performance aspects of the given (correct) algorithm. This approach is reasonable since in practice we are most likely to use algorithms from the literature as building blocks of the ultimate solution we are designing. Therefore, it is sensible to assume that these algorithms are correct. What we must, however, derive ourselves is the

[1] There are different aspects of correctness, the most important one relating to the question of whether the algorithm does in fact solve the problem that is to be solved. While techniques exist for demonstrating formally that an algorithm is correct, this approach is fundamentally predicated upon a formal definition of what the algorithm is supposed to do. The difficulty here is that problems in the real world are rarely defined formally.

complexity of these algorithms. While the literature may contain a complexity analysis of an algorithm, it is our contention that complexity analysis offers many more potential pitfalls when transitioning to software than correctness. As a result, it is imperative that the software designer have a good grasp of the principles and assumptions involved in algorithm analysis.

An important aspect of the performance of an algorithm is its dependence on (some measure of) the input. If we have a program and want to determine some aspect of its behavior, we can run it with a specific input set and *observe* its behavior on that input set. This avenue is closed to us when it comes to algorithms — there is no execution and therefore no observation. Instead, we desire a much more universal description of the behavior of interest, namely a description that holds for any input set. This is achieved by abstracting the input set and using that abstraction as a parameter; usually, the size of the input set plays this role. Consequently, the description of the behavior of the algorithm has now become a function of this parameter. In this way, we hope to obtain a universal description of the behavior because we get an answer for any input set. Of course, in this process of abstracting we have most likely lost information that would allow us to give more precise answers. Thus, there is a tension between the information loss that occurs when we attempt to provide a global picture of performance through abstraction and the loss of precision in the eventual answer.

For example, suppose we are interested in the number of instructions necessary to sort a given input set using algorithm A. If we are sorting a set S of 100 numbers, it stands to reason that we should be able to determine accurately how many instructions will have to be executed. However, the question of how many instructions are necessary to sort any set with 100 elements is likely to be much less precise; we might be able to say that we must use at least this many and at most that many instructions. In other words, we could give a range of values, with the property that no matter how the set of 100 elements looks, the actual number of instructions would always be within the given range. Of course, now we could carry out this exercise for sets with 101 elements, 102, 103, and so on, thereby using the size n of the set as a parameter with the property that for each value of n, there is a range $F(n)$ of values so that any set with n numbers is sorted by A using a number of instructions that falls in the range $F(n)$.

Note, however, that knowing the range of the statement counts for an algorithm may still not be particularly illuminating since it reveals little about the likelihood of a value in the range to occur. Clearly, the two extremes, the smallest value and the largest value in the range $F(n)$ for a specific value of n have significance (they correspond to the best- and the worst-case complexity), but as we will discuss in more detail below, how often a particular value in the range may occur is related to the average complexity, which is a significantly more complex topic.

While the approach to determining explicitly the range $F(n)$ for every value of n is of course prohibitively tedious, it is nevertheless the conceptual basis for determining the computational complexity of a given algorithm. Most

importantly, the determination of the number of statements for solving a problem is also abstracted, so that it typically is carried out by examining the syntactic components, that is, the statements, of the given algorithm.

Counting statements is probably the most important aspect of the behavior of an algorithm because it captures the notion of execution time quite accurately, but there are other aspects. In the following sections, we examine these qualities of algorithms.

1.2 The Time and Space Complexities of an Algorithm

The most burning question about a (correct) program is probably, "How long does it take to execute?" The analogous question for an algorithm is, "What is its time complexity?" Essentially, we are asking the same question ("How long does it take?"), but within different contexts. Programs can be executed, so we can simply run the program, admittedly with a specific data set, and measure the time required; algorithms cannot be run and therefore we have to resort to a different approach. This approach is the statement count. Before we describe it and show how statement counts reflect time, we must mention that time is not the only aspect that may be of interest; space is also of concern in some instances, although given the ever-increasing memory sizes of today's computers, space considerations are of decreasing import. Still, we may want to know how much memory is required by a given algorithm to solve a problem.

Given algorithm A (assumed to be correct) and a measure n of the input set (usually the size of all the input sets involved), the *time complexity* of algorithm A is defined to be the number $f(n)$ of atomic instructions or operations that must be executed when applying A to any input set of measure n. (More specifically, this is the worst-case time complexity; see the discussion below in Section 1.3.) The *space complexity* of algorithm A is the amount of space, again as a function of the measure of the input set, that A requires to carry out its computations, over and above the space that is needed to store the given input (and possibly the output, namely if it is presented in memory space different from that allocated for the input).

To illustrate this, consider a vector **V** of n elements (of type integer; **V** is of type [1:n] and $n \geq 1$) and assume that the algorithm solves the problem of finding the maximum of these n numbers using the following approach:

Algorithm Max to find the largest integer in the vector **V**[1:n]:

1. Initialize TempMax to **V**[1].
2. Compare TempMax with all other elements of **V** and update TempMax if TempMax is smaller.

Let us count the number of atomic operations[2] that occur when applying the algorithm Max to a vector with n integers. Statement 1 is one simple assignment. Statement 2 involves $n - 1$ integers, and each is compared to TempMax; furthermore, if the current value of TempMax is smaller than the vector element examined, that integer must be assigned to TempMax. It is important to note that no specific order is implied in this formulation; as long as all elements of **V** are examined, the algorithm works. At this point, our statement count stands at n, the 1 assignment from statement 1 and the $n - 1$ comparisons in statement 2 that must always be carried out. The updating operation is a bit trickier, since it only arises if TempMax is smaller. Without knowing the specific integers, we cannot say how many times we have to update, but we can give a range; if we are lucky (if V[1] happens to be the largest element), no updates of TempMax are required. If we are unlucky, we must make an update after every comparison. This clearly is the range from best to worst case. Consequently, we will carry out between 0 and $n - 1$ updates, each of which consists of one assignment. Adding all this up, it follows that the number of operations necessary to solve the problem ranges from n to $2n - 1$. It is important to note that this process does not require any execution; our answer is independent of the size of n. More bluntly, if $n = 10^{10}$ (10 billion), our analysis tells us that we need between 10 and 20 billion operations; this analysis can be carried out much faster than it would take to run a program derived from this algorithm.

We note as an aside that the algorithm corresponds in a fairly natural way to the following pseudo code[3]:

```
TempMax := V[1];
for i:=2 to n do
  { if TempMax < V[i] then TempMax := V[i] };
Max := TempMax
```

However, in contrast to the algorithm, the language requirements impose on us a much greater specificity. While the algorithm simply referred to examining all elements of **V** other than **V**[1], the program stipulates a (quite unnecessarily) specific order. While any order would do, the fact that the language constructs typically require us to specify one has implications that we will comment on in Part 2 in more detail.

We conclude that the algorithm Max for finding the maximum of n integers has a time complexity of between n and $2n - 1$. To determine the space complexity, we must look at the instructions again and figure out how much additional space is needed for them. Clearly, TempMax requires space (one

[2] We will explain in Chapter 2 in much more detail what we mean by *atomic operations*. Here, it suffices to assume that these operations are arithmetic operations, comparisons, and assignments involving basic types such as integers.

[3] We use a notation that should be fairly self-explanatory. It is a compromise between C notation and Pascal notation; however, for the time being we sidestep more complex issues such as the method used in passing parameters.

unit[4] of it), and from the algorithm, it appears that this is all that is needed. This is, however, a bit misleading, because we will have to carry out an enumeration of all elements of **V**, and this will cost us at least one more memory unit (for example for an index variable, such as the variable *i* in our program). Thus, the space complexity of algorithm Max is 2, independent of the size of the input set (the number of elements in vector **V**). We assume that *n*, and therefore the space to hold it, was given.

It is important to note that the time complexity of any algorithm should never be smaller than its space complexity. Recall that the space complexity determines the additional memory needed; thus, it stands to reason that this is memory space that should be used in some way (otherwise, what is the point in allocating it?). Since doing anything with a memory unit will require at least one operation, that is, one time unit, the time complexity should never be inferior to the space complexity.[5]

It appears that we are losing quite a bit of precision during the process of calculating the operation or statement count, even in this very trivial example. However, it is important to understand that the notion of complexity is predominantly concerned with the *long-term behavior* of an algorithm. By this, we mean that we want to know the growth in execution time as *n* grows. This is also called the *asymptotic behavior* of the complexity of the algorithm. Furthermore, in order to permit easy comparison of different algorithms according to their complexities (time or space), it is advantageous to lose precision, since the loss of precision allows us to come up with a relatively small number of categories into which we may classify our algorithms. While these two issues, asymptotic behavior and comparing different algorithms, seem to be different, they turn out to be closely related.

To develop this point properly requires a bit of mathematical notation. Assume we have obtained the (time or space) complexities $f_1(n)$ and $f_2(n)$ of two different algorithms, A_1 and A_2 (presumably both solving the same problem correctly, with *n* being the same measure of the input set). We say that the function $f_1(n)$ is *on the order of* the function $f_2(n)$, and write

$$f_1(n) = O(f_2(n)),$$

or briefly $f_1 = O(f_2)$ if *n* is understood, if and only if there exists an integer $n_0 \geq 1$ and a constant $c > 0$ such that

$$f_1(n) \leq c \cdot f_2(n) \text{ for all } n \geq n_0.$$

[4] We assume here that one number requires one unit of memory. We discuss the question of what one unit really is in much greater detail in Chapter 2 (see also the discussion of bit and word complexity in Section 1.4).

[5] Later we will see an example where, owing to incorrect passing of parameters, this assertion is violated.

Intuitively, $f_1 = O(f_2)$ means that f_1 does not grow faster asymptotically than f_2; it is asymptotic growth because we are only interested in the behavior from n_0 onward. Finally, the constant c simply reflects the loss of precision we have referred to earlier. As long as f_1 stays "close to" f_2 (namely within that constant c), this is fine.

Example: Let $f(n) = 5 \cdot n \cdot \log_2(n)$ and $g(n) = n^2/100 - 32n$. We claim that $f = O(g)$. To show this, we have to find n_0 and c such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. There are many (in fact, infinitely many) such pairs (n_0, c). For example, $n_0 = 10{,}000$, $c = 1$, or $n_0 = 100{,}000$, $c = 1$, or $n_0 = 3{,}260$, $c = 100$.

In each case, one can verify that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. More interesting may be the fact that $g \neq O(f)$; in other words, one can verify that there do not exist n_0 and c such that $g(n) \leq c \cdot f(n)$ for all $n \geq n_0$.

It is possible that both $f_1 = O(f_2)$ and $f_2 = O(f_1)$ hold; in this case we say that f_1 and f_2 are equivalent and write $f_1 \equiv f_2$.

Let us now return to our two algorithms A_1 and A_2 with their time complexities $f_1(n)$ and $f_2(n)$; we want to know which algorithm is faster. In general, this is a bit tricky, but if we are willing to settle for asymptotic behavior, the answer is simple: if $f_1 = O(f_2)$, then A_1 is no worse than A_2, and if $f_1 \equiv f_2$, then A_1 and A_2 behave identically.[6]

Note that the notion of asymptotic behavior hides a constant factor; clearly if $f(n) = n^2$ and $g(n) = 5 \cdot n^2$, then $f \equiv g$, so the two algorithms behave identically, but obviously the algorithm with time complexity f is five times faster than that with time complexity g.

However, the hidden constant factors are just what we need to establish a classification of complexities that has proven very useful in characterizing algorithms. Consider the following eight categories:

$$\varphi_1 = 1,\ \varphi_2 = \log_2(n),\ \varphi_3 = \sqrt[2]{n},\ \varphi_4 = n,\ \varphi_5 = n \cdot \log_2(n),\ \varphi_6 = n^2,\ \varphi_7 = n^3,\ \varphi_8 = 2^n.$$

(While one could define arbitrarily many categories between any two of these, those listed are of the greatest practical importance.) Characterizing a given function $f(n)$ consists of finding the most appropriate category φ_i for the function f. This means determining φ_i so that $f = O(\varphi_i)$ but $f \neq O(\varphi_{i-1})$.[7] For example, a complexity $n^2/\log_2(n)$ would be classified as n^2, as would be $(n^2 - 3n + 10) \cdot (n^4 - n^3)/(n^4 + n^2 + n + 5)$; in both cases, the function is $O(n^2)$, but not $O(n \cdot \log_2(n))$.

We say that a complexity of φ_1 is *constant*, of φ_2 is *logarithmic* (note that the base is irrelevant because $\log_a(x)$ and $\log_b(x)$ for two different bases a and b

[6] One can develop a calculus based on these notions. For example, if $f_1 \equiv g_1$ and $f_2 \equiv g_2$, then $f_1 + f_2 \equiv g_1 + g_2$, $f_1 - f_2 \equiv g_1 - g_2$ (under some conditions), and $f_1 * f_2 \equiv g_1 * g_2$. Moreover, if f_2 and g_2 are different from 0 for all argument values, then $f_1/f_2 \equiv g_1/g_2$. A similar calculus holds for functions f and g such that $f = O(g)$: $f_i = O(g_i)$ for $i = 1,2$ implies $f_1 \circ f_2 = O(g_1 \circ g_2)$ for $\circ$ any of the four basic arithmetic operations (with the obvious restriction about division by zero).

[7] Note that if $f = O(\varphi_i)$, then $f = O(\varphi_{i+j})$ for all $j > 0$; thus, it is important to find the best category for a function.

are related by a constant factor, which of course is hidden when we talk about the asymptotic behavior of complexity[8]), of φ_4 is *linear*, of φ_6 is *quadratic*, of φ_7 is *cubic*, and of φ_8 is *exponential*.[9] It should be clear that of all functions in a category, the function that represents it should be the simplest one. Thus, from now on, we will place a given complexity into one of these eight categories, even though the actual complexity may be more complicated.

So far in our discussion of asymptotic behavior, we have carefully avoided addressing the question of the range of the operation counts. However, revisiting our algorithm Max, it should be now clear that the time complexity, which we originally derived as a range from n to $2n - 1$, is simply linear. This is because the constant factor involved (which is 1 for the smallest value in the range and 2 for the largest) is hidden in the asymptotic function that we obtain as final answer.

In general, the range may not be as conveniently described as for our algorithm Max. Specifically, it is quite possible that the largest value in the range is not a constant factor of the smallest value, for all n. This then leads to the question of best-case, average-case, and worst-case complexity, which we take up in the next section.

Today, the quality of most algorithms is measured by their speed. For this reason, the computational complexity of an algorithm usually refers to its time complexity. Space complexity has become much less important; as we will see, typically, it attracts attention only when something goes wrong.

1.3 The Worst-, Average-, and Best-Case Complexities of an Algorithm

Recall that we talked about the range of the number of operations that corresponds to a specific value of (the measure of the input set) n. The *worst-case complexity* of an algorithm is thus the largest value of this range, which is of course a function of n. Thus, for our algorithm Max, the worst-case complexity is $2n - 1$, which is linear in n. Similarly, the *best-case complexity* is the smallest value of the range for each value of n. For the algorithm Max, this was n (also linear in n).

Before we turn our attention to the average complexity (which is quite a bit more complicated to define than best- or worst-case complexity), it is useful to relate these concepts to practical concerns. Worst-case complexity is easiest to motivate: it simply gives us an upper bound (in the number of statements to be executed) on how long it can possibly take to complete a task. This is of course a very common concern; in many cases, we would

[8] Specifically, $\log_a(x) = c \cdot \log_b(x)$ for $c = \log_a(b)$ for all $a, b > 1$.

[9] In contrast to logarithms, exponentials are not within a constant of each other: specifically, for $a > b > 1$, $a^n \neq O(b^n)$. However, from a practical point of view, exponential complexities are usually so bad that it is not really necessary to differentiate them much further.

like to be able to assert that under no circumstances will it take longer than this amount of time to complete a certain task. Typical examples are real-time applications such as algorithms used in air-traffic control or power-plant operations. Even in less dramatic situations, programmers want to be able to guarantee at what time completion of a task is assured. Thus, even if everything conspires against earlier completion, the worst-case time complexity provides a measure that will not fail. Similarly, allocating an amount of memory equal to (or no less than) the worst-case space complexity assures that the task will never run out of memory, no matter what happens.

Average complexity reflects the (optimistic) expectation that things will usually not turn out for the worst. Thus, if one has to perform a specific task many times (for different input sets), it probably makes more sense to be interested in the average behavior, for example the average time it takes to complete the task, than the worst-case complexity. While this is a very sensible approach (more so for time than for space), defining what one might view as average turns out to be rather complicated, as we will see below.

The best-case complexity is in practice less important, unless you are an inveterate gambler who expects to be always lucky. Nevertheless, there are instances where it is useful. One such situation is in cryptography. Suppose we know about a certain encryption scheme, that there exists an algorithm for breaking this scheme whose worst-case time complexity and average time complexity are both exponential in the length of the message to be decrypted. We might conclude from this information that this encryption scheme is very safe — and we might be very wrong. Here is how this could happen. Assume that for 50% of all encryptions (that usually would mean for 50% of all encryption keys), decryption (without knowledge of the key, that is, breaking the code) takes time 2^n, where n is the length of the message to be decrypted. Also assume that for the other 50%, breaking the code takes time n. If we compute the average time complexity of breaking the code as the average of n and 2^n (since both cases are equally likely), we obviously obtain again approximately 2^n (we have $(n + 2^n)/2 > 2^{n-1}$, and clearly $2^{n-1} = O(2^n)$). So, both the worst-case and average time complexities are 2^n, but in half of all cases the encryption scheme can be broken with minimal effort. Therefore, the overall encryption scheme is absolutely worthless. However, this becomes clear only when one looks at the best-case time complexity of the algorithm.

Worst- and best-case complexities are very specific and do not depend on any particular assumptions; in contrast, average complexity depends crucially on a precise notion of what constitutes the average case of a particular problem. To gain some appreciation of this, consider the task of locating an element x in a linear list containing n elements. Let us determine how many probes are necessary to find the location of x in that linear list. Note that the number of operations per probe is a (very small) constant; essentially, we must do a comparison. Then we must follow a link in the list, unless the comparison was the last one (determining this requires an additional simple test). Thus, the number of probes is the number of operations up to a constant

factor — providing additional justification for our systematic hiding of constant factors when determining the asymptotic complexity of algorithms. It should be clear what are the best and worst cases in our situation. The best case occurs if the first element of the linear list contains x, resulting in one probe, while for the worst case we have two possibilities: either it is the last element of the linear list that contains x or x is not in the list at all. In both of these worst cases, we need n probes since x must be compared with each of the n elements in the linear list. Thus, the best-case time complexity is $O(1)$ and the worst case complexity is $O(n)$, but what is the average time complexity?

The answer to this question depends heavily on the probability distribution of the elements. Specifically, we must know what is the likelihood for x to be in the element of the linear list with number i, for $i = 1, \ldots, n$. Also, we must know what is the probability of x not being in the linear list. Without all this information, it is impossible to determine the average time complexity of our algorithm, although it is true that, no matter what our assumptions are, the average complexity will always lie between the best- and worst-case complexity. Since in this case the best-case and worst-case time complexities are quite different (there is no constant factor relating the two measures, in contrast to the situation for Max), one should not be surprised that different distributions may result in different answers. Let us work out two scenarios.

1.3.1 Scenario 1

The probability p_{not} of x not being in the list is 0.50; that is, the likelihood that x is in the linear list is equal to it not being there. The likelihood p_i of x to occur in position i is $0.5/n$; that is, each position is equally likely to contain x. Using this information, the average number of probes is determined as follows:

To encounter x in position i requires i probes; this occurs with probability $p_i = 0.5/n$. With probability 0.5, we need n probes to account for the case that x is not in the linear list. Thus, on average we have

$$1 \cdot p_1 + 2 \cdot p_2 + 3 \cdot p_3 + \ldots + (n-1) \cdot p_{n-1} + n \cdot p_n + n \cdot 0.5 =$$
$$(1 + 2 + 3 + \ldots + n) \cdot 0.5/n + n \cdot 0.5 =$$
$$(n + 1)/4 + n/2 = (3n + 1)/4.$$[10]

Thus, the average number of probes is $(3n + 1)/4$.

[10] In this computation, we used the mathematical formula $\Sigma_{i = 1, \ldots, n} i = n \cdot (n + 1)/2$. It can be proven by induction on n.