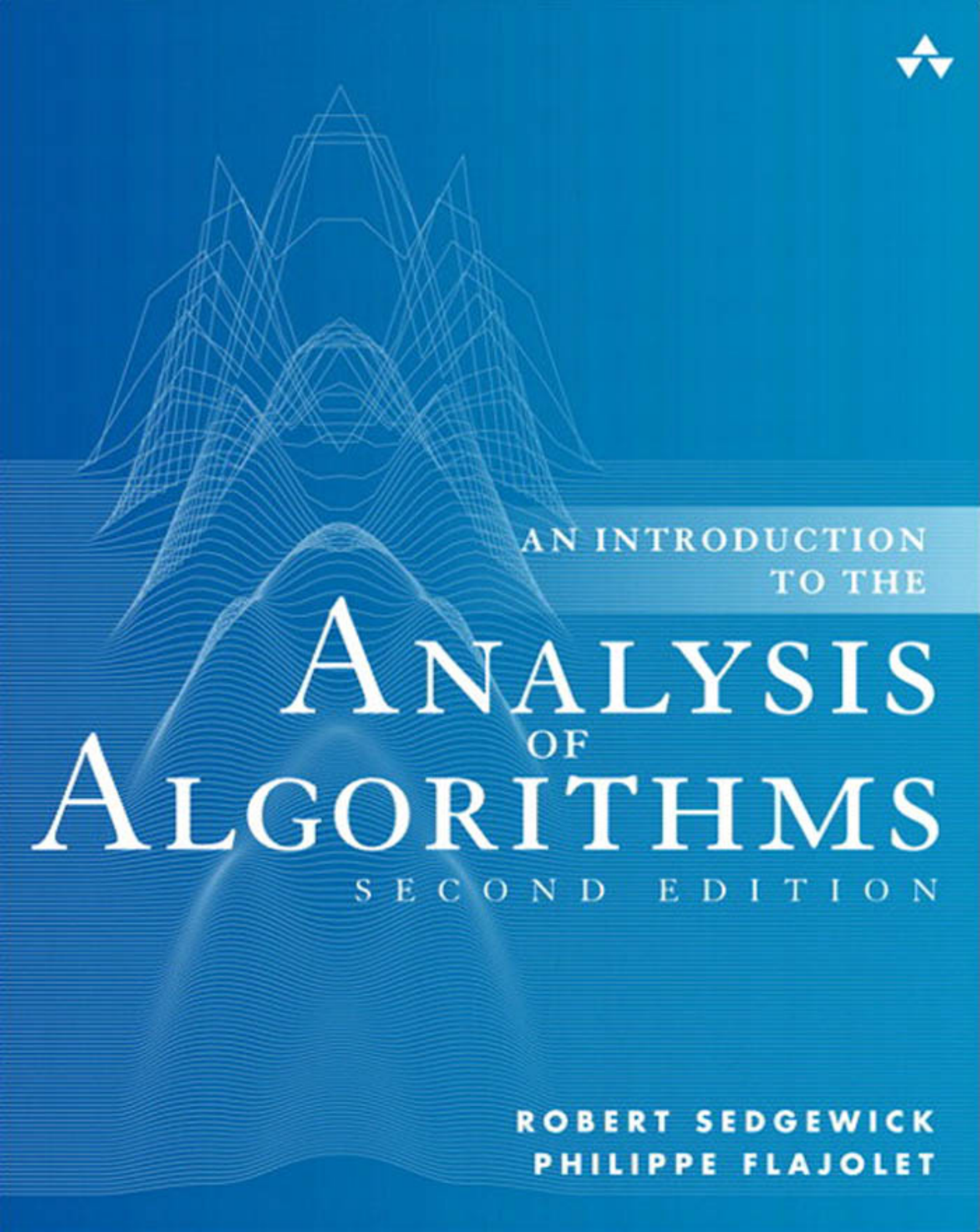

AN INTRODUCTION TO THE

ANALYSIS OF ALGORITHMS

SECOND EDITION

ROBERT SEDGEWICK
PHILIPPE FLAJOLET

AN INTRODUCTION TO THE ANALYSIS OF ALGORITHMS

Second Edition

This page intentionally left blank

AN INTRODUCTION TO THE ANALYSIS OF ALGORITHMS

Second Edition

Robert Sedgewick
Princeton University

Philippe Flajolet
INRIA Rocquencourt

Addison-Wesley
Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Control Number: 2012955493

ISBN-13: 978-0-321-90575-8
ISBN-10: 0-321-90575-X

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
First printing, January 2013

FOREWORD

PEOPLE who analyze algorithms have double happiness. First of all they experience the sheer beauty of elegant mathematical patterns that surround elegant computational procedures. Then they receive a practical payoff when their theories make it possible to get other jobs done more quickly and more economically.

Mathematical models have been a crucial inspiration for all scientific activity, even though they are only approximate idealizations of real-world phenomena. Inside a computer, such models are more relevant than ever before, because computer programs create artificial worlds in which mathematical models often apply precisely. I think that's why I got hooked on analysis of algorithms when I was a graduate student, and why the subject has been my main life's work ever since.

Until recently, however, analysis of algorithms has largely remained the preserve of graduate students and post-graduate researchers. Its concepts are not really esoteric or difficult, but they are relatively new, so it has taken awhile to sort out the best ways of learning them and using them.

Now, after more than 40 years of development, algorithmic analysis has matured to the point where it is ready to take its place in the standard computer science curriculum. The appearance of this long-awaited textbook by Sedgewick and Flajolet is therefore most welcome. Its authors are not only worldwide leaders of the field, they also are masters of exposition. I am sure that every serious computer scientist will find this book rewarding in many ways.

D. E. Knuth

This page intentionally left blank

PREFACE

THIS book is intended to be a thorough overview of the primary techniques used in the mathematical analysis of algorithms. The material covered draws from classical mathematical topics, including discrete mathematics, elementary real analysis, and combinatorics, as well as from classical computer science topics, including algorithms and data structures. The focus is on "average-case" or "probabilistic" analysis, though the basic mathematical tools required for "worst-case" or "complexity" analysis are covered as well.

We assume that the reader has some familiarity with basic concepts in both computer science and real analysis. In a nutshell, the reader should be able to both write programs and prove theorems. Otherwise, the book is intended to be self-contained.

The book is meant to be used as a textbook in an upper-level course on analysis of algorithms. It can also be used in a course in discrete mathematics for computer scientists, since it covers basic techniques in discrete mathematics as well as combinatorics and basic properties of important discrete structures within a familiar context for computer science students. It is traditional to have somewhat broader coverage in such courses, but many instructors may find the approach here to be a useful way to engage students in a substantial portion of the material. The book also can be used to introduce students in mathematics and applied mathematics to principles from computer science related to algorithms and data structures.

Despite the large amount of literature on the mathematical analysis of algorithms, basic information on methods and models in widespread use has not been directly accessible to students and researchers in the field. This book aims to address this situation, bringing together a body of material intended to provide readers with both an appreciation for the challenges of the field and the background needed to learn the advanced tools being developed to meet these challenges. Supplemented by papers from the literature, the book can serve as the basis for an introductory graduate course on the analysis of algorithms, or as a reference or basis for self-study by researchers in mathematics or computer science who want access to the literature in this field.

Preparation. Mathematical maturity equivalent to one or two years' study at the college level is assumed. Basic courses in combinatorics and discrete mathematics may provide useful background (and may overlap with some

material in the book), as would courses in real analysis, numerical methods, or elementary number theory. We draw on all of these areas, but summarize the necessary material here, with reference to standard texts for people who want more information.

Programming experience equivalent to one or two semesters' study at the college level, including elementary data structures, is assumed. We do not dwell on programming and implementation issues, but algorithms and data structures are the central object of our studies. Again, our treatment is complete in the sense that we summarize basic information, with reference to standard texts and primary sources.

Related books. Related texts include *The Art of Computer Programming* by Knuth; *Algorithms, Fourth Edition*, by Sedgewick and Wayne; *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein; and our own *Analytic Combinatorics*. This book could be considered supplementary to each of these.

In spirit, this book is closest to the pioneering books by Knuth. Our focus is on mathematical techniques of analysis, though, whereas Knuth's books are broad and encyclopedic in scope, with properties of algorithms playing a primary role and methods of analysis a secondary role. This book can serve as basic preparation for the advanced results covered and referred to in Knuth's books. We also cover approaches and results in the analysis of algorithms that have been developed since publication of Knuth's books.

We also strive to keep the focus on covering algorithms of fundamental importance and interest, such as those described in Sedgewick's *Algorithms* (now in its fourth edition, coauthored by K. Wayne). That book surveys classic algorithms for sorting and searching, and for processing graphs and strings. Our emphasis is on mathematics needed to support scientific studies that can serve as the basis of predicting performance of such algorithms and for comparing different algorithms on the basis of performance.

Cormen, Leiserson, Rivest, and Stein's *Introduction to Algorithms* has emerged as the standard textbook that provides access to the research literature on algorithm design. The book (and related literature) focuses on *design* and the *theory* of algorithms, usually on the basis of worst-case performance bounds. In this book, we complement this approach by focusing on the *analysis* of algorithms, especially on techniques that can be used as the basis for scientific studies (as opposed to theoretical studies). Chapter 1 is devoted entirely to developing this context.

This book also lays the groundwork for our *Analytic Combinatorics*, a general treatment that places the material here in a broader perspective and develops advanced methods and models that can serve as the basis for new research, not only in the analysis of algorithms but also in combinatorics and scientific applications more broadly. A higher level of mathematical maturity is assumed for that volume, perhaps at the senior or beginning graduate student level. Of course, careful study of this book is adequate preparation. It certainly has been our goal to make it sufficiently interesting that some readers will be inspired to tackle more advanced material!

How to use this book. Readers of this book are likely to have rather diverse backgrounds in discrete mathematics and computer science. With this in mind, it is useful to be aware of the implicit structure of the book: nine chapters in all, an introductory chapter followed by four chapters emphasizing mathematical methods, then four chapters emphasizing combinatorial structures with applications in the analysis of algorithms, as follows:

Chapter 1 puts the material in the book into perspective, and will help all readers understand the basic objectives of the book and the role of the remaining chapters in meeting those objectives. Chapters 2 through 4 cover

methods from classical discrete mathematics, with a primary focus on developing basic concepts and techniques. They set the stage for Chapter 5, which is pivotal, as it covers *analytic combinatorics*, a calculus for the study of large discrete structures that has emerged from these classical methods to help solve the modern problems that now face researchers because of the emergence of computers and computational models. Chapters 6 through 9 move the focus back toward computer science, as they cover properties of combinatorial structures, their relationships to fundamental algorithms, and analytic results.

Though the book is intended to be self-contained, this structure supports differences in emphasis when teaching the material, depending on the background and experience of students and instructor. One approach, more mathematically oriented, would be to emphasize the theorems and proofs in the first part of the book, with applications drawn from Chapters 6 through 9. Another approach, more oriented towards computer science, would be to briefly cover the major mathematical tools in Chapters 2 through 5 and emphasize the algorithmic material in the second half of the book. But our primary intention is that most students should be able to learn new material from both mathematics and computer science in an interesting context by working carefully all the way through the book.

Supplementing the text are lists of references and several hundred exercises, to encourage readers to examine original sources and to consider the material in the text in more depth.

Our experience in teaching this material has shown that there are numerous opportunities for instructors to supplement lecture and reading material with computation-based laboratories and homework assignments. The material covered here is an ideal framework for students to develop expertise in a symbolic manipulation system such as Mathematica, MAPLE, or SAGE. More important, the experience of validating the mathematical studies by comparing them against empirical studies is an opportunity to provide valuable insights for students that should not be missed.

Booksite. An important feature of the book is its relationship to the booksite `aofa.cs.princeton.edu`. This site is freely available and contains supplementary material about the analysis of algorithms, including a complete set of lecture slides and links to related material, including similar sites for *Algorithms* and *Analytic Combinatorics*. These resources are suitable both for use by any instructor teaching the material and for self-study.

Acknowledgments. We are very grateful to INRIA, Princeton University, and the National Science Foundation, which provided the primary support for us to work on this book. Other support has been provided by Brown University, European Community (Alcom Project), Institute for Defense Analyses, Ministère de la Recherche et de la Technologie, Stanford University, Université Libre de Bruxelles, and Xerox Palo Alto Research Center. This book has been many years in the making, so a comprehensive list of people and organizations that have contributed support would be prohibitively long, and we apologize for any omissions.

Don Knuth's influence on our work has been extremely important, as is obvious from the text.

Students in Princeton, Paris, and Providence provided helpful feedback in courses taught from this material over the years, and students and teachers all over the world provided feedback on the first edition. We would like to specifically thank Philippe Dumas, Mordecai Golin, Helmut Prodinger, Michele Soria, Mark Daniel Ward, and Mark Wilson for their help.

Corfu, September 1995 Ph. F. and R. S.
Paris, December 2012 R. S.

This page intentionally left blank

NOTE ON THE SECOND EDITION

IN March 2011, I was traveling with my wife Linda in a beautiful but somewhat remote area of the world. Catching up with my mail after a few days offline, I found the shocking news that my friend and colleague Philippe had passed away, suddenly, unexpectedly, and far too early. Unable to travel to Paris in time for the funeral, Linda and I composed a eulogy for our dear friend that I would now like to share with readers of this book.

> *Sadly, I am writing from a distant part of the world to pay my respects to my longtime friend and colleague, Philippe Flajolet. I am very sorry not to be there in person, but I know that there will be many opportunities to honor Philippe in the future and expect to be fully and personally involved on these occasions.*
>
> *Brilliant, creative, inquisitive, and indefatigable, yet generous and charming, Philippe's approach to life was contagious. He changed many lives, including my own. As our research papers led to a survey paper, then to a monograph, then to a book, then to two books, then to a life's work, I learned, as many students and collaborators around the world have learned, that working with Philippe was based on a genuine and heartfelt camaraderie. We met and worked together in cafes, bars, lunchrooms, and lounges all around the world. Philippe's routine was always the same. We would discuss something amusing that happened to one friend or another and then get to work. After a wink, a hearty but quick laugh, a puff of smoke, another sip of a beer, a few bites of steak frites, and a drawn out "Well..." we could proceed to solve the problem or prove the theorem. For so many of us, these moments are frozen in time.*
>
> *The world has lost a brilliant and productive mathematician. Philippe's untimely passing means that many things may never be known. But his legacy is a coterie of followers passionately devoted to Philippe and his mathematics who will carry on. Our conferences will include a toast to him, our research will build upon his work, our papers will include the inscription "Dedicated to the memory of Philippe Flajolet," and we will teach generations to come. Dear friend, we miss you so very much, but rest assured that your spirit will live on in our work.*

This second edition of our book *An Introduction to the Analysis of Algorithms* was prepared with these thoughts in mind. It is dedicated to the memory of Philippe Flajolet, and is intended to teach generations to come.

Jamestown RI, October 2012 R. S.

This page intentionally left blank

TABLE OF CONTENTS

This page intentionally left blank

NOTATION

$\lfloor x \rfloor$ *floor function*
largest integer less than or equal to x

$\lceil x \rceil$ *ceiling function*
smallest integer greater than or equal to x

$\{x\}$ *fractional part*
$x - \lfloor x \rfloor$

$\lg N$ *binary logarithm*
$\log_2 N$

$\ln N$ *natural logarithm*
$\log_e N$

$\binom{n}{k}$ *binomial coefficient*
number of ways to choose k out of n items

$\begin{bmatrix} n \\ k \end{bmatrix}$ *Stirling number of the first kind*
number of permutations of n elements that have k cycles

$\left\{ {n \atop k} \right\}$ *Stirling number of the second kind*
number of ways to partition n elements into k nonempty subsets

ϕ *golden ratio*
$(1 + \sqrt{5})/2 = 1.61803\cdots$

γ *Euler's constant*
$.57721\cdots$

σ *Stirling's constant*
$\sqrt{2\pi} = 2.50662\cdots$

This page intentionally left blank

CHAPTER ONE

ANALYSIS OF ALGORITHMS

MATHEMATICAL studies of the properties of computer algorithms have spanned a broad spectrum, from general complexity studies to specific analytic results. In this chapter, our intent is to provide perspective on various approaches to studying algorithms, to place our field of study into context among related fields and to set the stage for the rest of the book. To this end, we illustrate concepts within a fundamental and representative problem domain: the study of sorting algorithms.

First, we will consider the general motivations for algorithmic analysis. Why analyze an algorithm? What are the benefits of doing so? How can we simplify the process? Next, we discuss the theory of algorithms and consider as an example mergesort, an "optimal" algorithm for sorting. Following that, we examine the major components of a full analysis for a sorting algorithm of fundamental practical importance, quicksort. This includes the study of various improvements to the basic quicksort algorithm, as well as some examples illustrating how the analysis can help one adjust parameters to improve performance.

These examples illustrate a clear need for a background in certain areas of discrete mathematics. In Chapters 2 through 4, we introduce recurrences, generating functions, and asymptotics—basic mathematical concepts needed for the analysis of algorithms. In Chapter 5, we introduce the *symbolic method*, a formal treatment that ties together much of this book's content. In Chapters 6 through 9, we consider basic combinatorial properties of fundamental algorithms and data structures. Since there is a close relationship between fundamental methods used in computer science and classical mathematical analysis, we simultaneously consider some introductory material from both areas in this book.

1.1 Why Analyze an Algorithm? There are several answers to this basic question, depending on one's frame of reference: the intended use of the algorithm, the importance of the algorithm in relationship to others from both practical and theoretical standpoints, the difficulty of analysis, and the accuracy and precision of the required answer.

The most straightforward reason for analyzing an algorithm is to discover its characteristics in order to evaluate its suitability for various applications or compare it with other algorithms for the same application. The characteristics of interest are most often the primary resources of time and space, particularly time. Put simply, we want to know how long an implementation of a particular algorithm will run on a particular computer, and how much space it will require. We generally strive to keep the analysis independent of particular implementations—we concentrate instead on obtaining results for essential characteristics of the algorithm that can be used to derive precise estimates of true resource requirements on various actual machines.

In practice, achieving independence between an algorithm and characteristics of its implementation can be difficult to arrange. The quality of the implementation and properties of compilers, machine architecture, and other major facets of the programming environment have dramatic effects on performance. We must be cognizant of such effects to be sure the results of analysis are useful. On the other hand, in some cases, analysis of an algorithm can help identify ways for it to take full advantage of the programming environment.

Occasionally, some property other than time or space is of interest, and the focus of the analysis changes accordingly. For example, an algorithm on a mobile device might be studied to determine the effect upon battery life, or an algorithm for a numerical problem might be studied to determine how accurate an answer it can provide. Also, it is sometimes appropriate to address multiple resources in the analysis. For example, an algorithm that uses a large amount of memory may use much less time than an algorithm that gets by with very little memory. Indeed, one prime motivation for doing a careful analysis is to provide accurate information to help in making proper tradeoff decisions in such situations.

The term *analysis of algorithms* has been used to describe two quite different general approaches to putting the study of the performance of computer programs on a scientific basis. We consider these two in turn.

The first, popularized by Aho, Hopcroft, and Ullman [2] and Cormen, Leiserson, Rivest, and Stein [6], concentrates on determining the growth of the worst-case performance of the algorithm (an "upper bound"). A prime goal in such analyses is to determine which algorithms are optimal in the sense that a matching "lower bound" can be proved on the worst-case performance of any algorithm for the same problem. We use the term *theory of algorithms*

to refer to this type of analysis. It is a special case of *computational complexity*, the general study of relationships between problems, algorithms, languages, and machines. The emergence of the theory of algorithms unleashed an Age of Design where multitudes of new algorithms with ever-improving worst-case performance bounds have been developed for multitudes of important problems. To establish the practical utility of such algorithms, however, more detailed analysis is needed, perhaps using the tools described in this book.

The second approach to the analysis of algorithms, popularized by Knuth [17][18][19][20][22], concentrates on precise characterizations of the best-case, worst-case, and average-case performance of algorithms, using a methodology that can be refined to produce increasingly precise answers when desired. A prime goal in such analyses is to be able to accurately predict the performance characteristics of particular algorithms when run on particular computers, in order to be able to predict resource usage, set parameters, and compare algorithms. This approach is *scientific*: we build mathematical models to describe the performance of real-world algorithm implementations, then use these models to develop hypotheses that we validate through experimentation.

We may view both these approaches as necessary stages in the design and analysis of efficient algorithms. When faced with a new algorithm to solve a new problem, we are interested in developing a rough idea of how well it might be expected to perform and how it might compare to other algorithms for the same problem, even the best possible. The theory of algorithms can provide this. However, so much precision is typically sacrificed in such an analysis that it provides little specific information that would allow us to predict performance for an actual implementation or to properly compare one algorithm to another. To be able to do so, we need details on the implementation, the computer to be used, and, as we see in this book, mathematical properties of the structures manipulated by the algorithm. The theory of algorithms may be viewed as the first step in an ongoing process of developing a more refined, more accurate analysis; we prefer to use the term *analysis of algorithms* to refer to the whole process, with the goal of providing answers with as much accuracy as necessary.

The analysis of an algorithm can help us understand it better, and can suggest informed improvements. The more complicated the algorithm, the more difficult the analysis. But it is not unusual for an algorithm to become simpler and more elegant during the analysis process. More important, the

careful scrutiny required for proper analysis often leads to better and more efficient *implementation* on particular computers. Analysis requires a far more complete understanding of an algorithm that can inform the process of producing a working implementation. Indeed, when the results of analytic and empirical studies agree, we become strongly convinced of the validity of the algorithm as well as of the correctness of the process of analysis.

Some algorithms are worth analyzing because their analyses can add to the body of mathematical tools available. Such algorithms may be of limited practical interest but may have properties similar to algorithms of practical interest so that understanding them may help to understand more important methods in the future. Other algorithms (some of intense practical interest, some of little or no such value) have a complex performance structure with properties of independent mathematical interest. The dynamic element brought to combinatorial problems by the analysis of algorithms leads to challenging, interesting mathematical problems that extend the reach of classical combinatorics to help shed light on properties of computer programs.

To bring these ideas into clearer focus, we next consider in detail some classical results first from the viewpoint of the theory of algorithms and then from the scientific viewpoint that we develop in this book. As a running example to illustrate the different perspectives, we study *sorting algorithms*, which rearrange a list to put it in numerical, alphabetic, or other order. Sorting is an important practical problem that remains the object of widespread study because it plays a central role in many applications.

1.2 Theory of Algorithms. The prime goal of the theory of algorithms is to classify algorithms according to their performance characteristics. The following mathematical notations are convenient for doing so:

Definition Given a function $f(N)$,

$O(f(N))$ denotes the set of all $g(N)$ such that $|g(N)/f(N)|$ is bounded from above as $N \to \infty$.

$\Omega(f(N))$ denotes the set of all $g(N)$ such that $|g(N)/f(N)|$ is bounded from below by a (strictly) positive number as $N \to \infty$.

$\Theta(f(N))$ denotes the set of all $g(N)$ such that $|g(N)/f(N)|$ is bounded from both above and below as $N \to \infty$.

These notations, adapted from classical analysis, were advocated for use in the analysis of algorithms in a paper by Knuth in 1976 [21]. They have come

into widespread use for making mathematical statements about bounds on the performance of algorithms. The O-notation provides a way to express an upper bound; the Ω-notation provides a way to express a lower bound; and the Θ-notation provides a way to express matching upper and lower bounds.

In mathematics, the most common use of the O-notation is in the context of asymptotic series. We will consider this usage in detail in Chapter 4. In the theory of algorithms, the O-notation is typically used for three purposes: to hide constants that might be irrelevant or inconvenient to compute, to express a relatively small "error" term in an expression describing the running time of an algorithm, and to bound the worst case. Nowadays, the Ω- and Θ- notations are directly associated with the theory of algorithms, though similar notations are used in mathematics (see [21]).

Since constant factors are being ignored, derivation of mathematical results using these notations is simpler than if more precise answers are sought. For example, both the "natural" logarithm $\ln N \equiv \log_e N$ and the "binary" logarithm $\lg N \equiv \log_2 N$ often arise, but they are related by a constant factor, so we can refer to either as being $O(\log N)$ if we are not interested in more precision. More to the point, we might say that the running time of an algorithm is $\Theta(N \log N)$ seconds just based on an analysis of the frequency of execution of fundamental operations and an assumption that each operation takes a constant number of seconds on a given computer, without working out the precise value of the constant.

Exercise 1.1 Show that $f(N) = N\lg N + O(N)$ implies that $f(N) = \Theta(N \log N)$.

As an illustration of the use of these notations to study the performance characteristics of algorithms, we consider methods for sorting a set of numbers in an array. The input is the numbers in the array, in arbitrary and unknown order; the output is the same numbers in the array, rearranged in ascending order. This is a well-studied and fundamental problem: we will consider an algorithm for solving it, then show that algorithm to be "optimal" in a precise technical sense.

First, we will show that it is possible to solve the sorting problem efficiently, using a well-known recursive algorithm called mergesort. Mergesort and nearly all of the algorithms treated in this book are described in detail in Sedgewick and Wayne [30], so we give only a brief description here. Readers interested in further details on variants of the algorithms, implementations, and applications are also encouraged to consult the books by Cor-

men, Leiserson, Rivest, and Stein [6], Gonnet and Baeza-Yates [11], Knuth [17][18][19][20], Sedgewick [26], and other sources.

Mergesort divides the array in the middle, sorts the two halves (recursively), and then merges the resulting sorted halves together to produce the sorted result, as shown in the Java implementation in Program 1.1. Mergesort is prototypical of the well-known *divide-and-conquer* algorithm design paradigm, where a problem is solved by (recursively) solving smaller subproblems and using the solutions to solve the original problem. We will analyze a number of such algorithms in this book. The recursive structure of algorithms like mergesort leads immediately to mathematical descriptions of their performance characteristics.

To accomplish the merge, Program 1.1 uses two auxiliary arrays `b` and `c` to hold the subarrays (for the sake of efficiency, it is best to declare these arrays external to the recursive method). Invoking this method with the call `mergesort(0, N-1)` will sort the array `a[0...N-1]`. After the recursive

```
private void mergesort(int[] a, int lo, int hi)
{
   if (hi <= lo) return;
   int mid = lo + (hi - lo) / 2;
   mergesort(a, lo, mid);
   mergesort(a, mid + 1, hi);
   for (int k = lo; k <= mid; k++)
      b[k-lo] = a[k];
   for (int k = mid+1; k <= hi; k++)
      c[k-mid-1] = a[k];
   b[mid-lo+1] = INFTY; c[hi - mid] = INFTY;
   int i = 0, j = 0;
   for (int k = lo; k <= hi; k++)
      if (c[j] < b[i]) a[k] = c[j++];
      else             a[k] = b[i++];
}
```

Program 1.1 Mergesort

calls, the two halves of the array are sorted. Then we move the first half of `a[]` to an auxiliary array `b[]` and the second half of `a[]` to another auxiliary array `c[]`. We add a "sentinel" `INFTY` that is assumed to be larger than all the elements to the end of each of the auxiliary arrays, to help accomplish the task of moving the remainder of one of the auxiliary arrays back to `a` after the other one has been exhausted. With these preparations, the merge is easily accomplished: for each `k`, move the smaller of the elements `b[i]` and `c[j]` to `a[k]`, then increment `k` and `i` or `j` accordingly.

Exercise 1.2 In some situations, defining a sentinel value may be inconvenient or impractical. Implement a mergesort that avoids doing so (see Sedgewick [26] for various strategies).

Exercise 1.3 Implement a mergesort that divides the array into *three* equal parts, sorts them, and does a three-way merge. Empirically compare its running time with standard mergesort.

In the present context, mergesort is significant because it is guaranteed to be as efficient as any sorting method can be. To make this claim more precise, we begin by analyzing the dominant factor in the running time of mergesort, the number of compares that it uses.

Theorem 1.1 (Mergesort compares). Mergesort uses $N\lg N + O(N)$ compares to sort an array of N elements.

Proof. If C_N is the number of compares that the Program 1.1 uses to sort N elements, then the number of compares to sort the first half is $C_{\lfloor N/2 \rfloor}$, the number of compares to sort the second half is $C_{\lceil N/2 \rceil}$, and the number of compares for the merge is N (one for each value of the index `k`). In other words, the number of compares for mergesort is precisely described by the recurrence relation

$$C_N = C_{\lfloor N/2 \rfloor} + C_{\lceil N/2 \rceil} + N \qquad \text{for } N \geq 2 \text{ with } C_1 = 0. \tag{1}$$

To get an indication for the nature of the solution to this recurrence, we consider the case when N is a power of 2:

$$C_{2^n} = 2C_{2^{n-1}} + 2^n \qquad \text{for } n \geq 1 \text{ with } C_1 = 0.$$

Dividing both sides of this equation by 2^n, we find that

$$\frac{C_{2^n}}{2^n} = \frac{C_{2^{n-1}}}{2^{n-1}} + 1 = \frac{C_{2^{n-2}}}{2^{n-2}} + 2 = \frac{C_{2^{n-3}}}{2^{n-3}} + 3 = \ldots = \frac{C_{2^0}}{2^0} + n = n.$$

This proves that $C_N = N\lg N$ when $N = 2^n$; the theorem for general N follows from (1) by induction. The exact solution turns out to be rather complicated, depending on properties of the binary representation of N. In Chapter 2 we will examine how to solve such recurrences in detail. ▪

Exercise 1.4 Develop a recurrence describing the quantity $C_{N+1} - C_N$ and use this to prove that

$$C_N = \sum_{1 \le k < N} (\lfloor \lg k \rfloor + 2).$$

Exercise 1.5 Prove that $C_N = N\lceil \lg N \rceil + N - 2^{\lceil \lg N \rceil}$.

Exercise 1.6 Analyze the number of compares used by the three-way mergesort proposed in Exercise 1.2.

For most computers, the relative costs of the elementary operations used Program 1.1 will be related by a constant factor, as they are all integer multiples of the cost of a basic instruction cycle. Furthermore, the total running time of the program will be within a constant factor of the number of compares. Therefore, a reasonable hypothesis is that the running time of mergesort will be within a constant factor of $N\lg N$.

From a theoretical standpoint, mergesort demonstrates that $N\log N$ is an "upper bound" on the intrinsic difficulty of the sorting problem:

There exists an algorithm that can sort any
N-element file in time proportional to $N\log N$.

A full proof of this requires a careful model of the computer to be used in terms of the operations involved and the time they take, but the result holds under rather generous assumptions. We say that the "time complexity of sorting is $O(N\log N)$."

Exercise 1.7 Assume that the running time of mergesort is $cN\lg N + dN$, where c and d are machine-dependent constants. Show that if we implement the program on a particular machine and observe a running time t_N for some value of N, then we can accurately estimate the running time for $2N$ by $2t_N(1 + 1/\lg N)$, *independent of the machine*.

Exercise 1.8 Implement mergesort on one or more computers, observe the running time for $N = 1{,}000{,}000$, and predict the running time for $N = 10{,}000{,}000$ as in the previous exercise. Then observe the running time for $N = 10{,}000{,}000$ and calculate the percentage accuracy of the prediction.

The running time of mergesort as implemented here depends only on the number of elements in the array being sorted, not on the way they are arranged. For many other sorting methods, the running time may vary substantially as a function of the initial ordering of the input. Typically, in the theory of algorithms, we are most interested in worst-case performance, since it can provide a guarantee on the performance characteristics of the algorithm no matter what the input is; in the analysis of particular algorithms, we are most interested in average-case performance for a reasonable input model, since that can provide a path to predict performance on "typical" input.

We always seek better algorithms, and a natural question that arises is whether there might be a sorting algorithm with asymptotically better performance than mergesort. The following classical result from the theory of algorithms says, in essence, that there is not.

Theorem 1.2 (Complexity of sorting). Every compare-based sorting program uses at least $\lceil \lg N! \rceil > N \lg N - N/(\ln 2)$ compares for some input.

Proof. A full proof of this fact may be found in [30] or [19]. Intuitively the result follows from the observation that each compare can cut down the number of possible arrangements of the elements to be considered by, at most, only a factor of 2. Since there are $N!$ possible arrangements before the sort and the goal is to have just one possible arrangement (the sorted one) after the sort, the number of compares must be at least the number of times $N!$ can be divided by 2 before reaching a number less than unity—that is, $\lceil \lg N! \rceil$. The theorem follows from Stirling's approximation to the factorial function (see the second corollary to Theorem 4.3). ∎

From a theoretical standpoint, this result demonstrates that $N \log N$ is a "lower bound" on the intrinsic difficulty of the sorting problem:

All compare-based sorting algorithms require time proportional to $N \log N$ to sort some N-element input file.

This is a general statement about an entire class of algorithms. We say that the "time complexity of sorting is $\Omega(N \log N)$." This lower bound is significant because it matches the upper bound of Theorem 1.1, thus showing that mergesort is optimal in the sense that no algorithm can have a better asymptotic running time. We say that the "time complexity of sorting is $\Theta(N \log N)$." From a theoretical standpoint, this completes the "solution" of the sorting "problem:" matching upper and lower bounds have been proved.

Again, these results hold under rather generous assumptions, though they are perhaps not as general as it might seem. For example, the results say nothing about sorting algorithms that do not use compares. Indeed, there exist sorting methods based on index calculation techniques (such as those discussed in Chapter 9) that run in linear time on average.

Exercise 1.9 Suppose that it is known that each of the items in an N-item array has one of two distinct values. Give a sorting method that takes time proportional to N.

Exercise 1.10 Answer the previous exercise for *three* distinct values.

We have omitted many details that relate to proper modeling of computers and programs in the proofs of Theorem 1.1 and Theorem 1.2. The essence of the theory of algorithms is the development of complete models within which the intrinsic difficulty of important problems can be assessed and "efficient" algorithms representing upper bounds matching these lower bounds can be developed. For many important problem domains there is still a significant gap between the lower and upper bounds on asymptotic worst-case performance. The theory of algorithms provides guidance in the development of new algorithms for such problems. We want algorithms that can lower known upper bounds, but there is no point in searching for an algorithm that performs better than known lower bounds (except perhaps by looking for one that violates conditions of the model upon which a lower bound is based!).

Thus, the theory of algorithms provides a way to classify algorithms according to their asymptotic performance. However, the very process of approximate analysis ("within a constant factor") that extends the applicability of theoretical results often limits our ability to accurately predict the performance characteristics of any particular algorithm. More important, the theory of algorithms is usually based on worst-case analysis, which can be overly pessimistic and not as helpful in predicting actual performance as an average-case analysis. This is not relevant for algorithms like mergesort (where the running time is not so dependent on the input), but average-case analysis can help us discover that nonoptimal algorithms are sometimes faster in practice, as we will see. The theory of algorithms can help us to identify good algorithms, but then it is of interest to refine the analysis to be able to more intelligently compare and improve them. To do so, we need precise knowledge about the performance characteristics of the particular computer being used and mathematical techniques for accurately determining the frequency of execution of fundamental operations. In this book, we concentrate on such techniques.

1.3 Analysis of Algorithms. Though the analysis of sorting and mergesort that we considered in §1.2 demonstrates the intrinsic "difficulty" of the sorting problem, there are many important questions related to sorting (and to mergesort) that it does not address at all. How long might an implementation of mergesort be expected to run on a particular computer? How might its running time compare to other $O(N \log N)$ methods? (There are many.) How does it compare to sorting methods that are fast on average, but perhaps not in the worst case? How does it compare to sorting methods that are not based on compares among elements? To answer such questions, a more detailed analysis is required. In this section we briefly describe the process of doing such an analysis.

To analyze an algorithm, we must first identify the resources of primary interest so that the detailed analysis may be properly focused. We describe the process in terms of studying the running time since it is the resource most relevant here. A complete analysis of the running time of an algorithm involves the following steps:

- Implement the algorithm completely.
- Determine the time required for each basic operation.
- Identify unknown quantities that can be used to describe the frequency of execution of the basic operations.
- Develop a realistic model for the input to the program.
- Analyze the unknown quantities, assuming the modeled input.
- Calculate the total running time by multiplying the time by the frequency for each operation, then adding all the products.

The first step in the analysis is to carefully implement the algorithm on a particular computer. We reserve the term *program* to describe such an implementation. One algorithm corresponds to many programs. A particular implementation not only provides a concrete object to study, but also can give useful empirical data to aid in or to check the analysis. Presumably the implementation is designed to make efficient use of resources, but it is a mistake to overemphasize efficiency too early in the process. Indeed, a primary application for the analysis is to provide informed guidance toward better implementations.

The next step is to estimate the time required by each component instruction of the program. In principle and in practice, we can often do so with great precision, but the process is very dependent on the characteristics

of the computer system being studied. Another approach is to simply run the program for small input sizes to "estimate" the values of the constants, or to do so indirectly in the aggregate, as described in Exercise 1.7. We do not consider this process in detail; rather we focus on the "machine-independent" parts of the analysis in this book.

Indeed, to determine the total running time of the program, it is necessary to study the branching structure of the program in order to express the frequency of execution of the component instructions in terms of unknown mathematical quantities. If the values of these quantities are known, then we can derive the running time of the entire program simply by multiplying the frequency and time requirements of each component instruction and adding these products. Many programming environments have tools that can simplify this task. At the first level of analysis, we concentrate on quantities that have large frequency values or that correspond to large costs; in principle the analysis can be refined to produce a fully detailed answer. We often refer to the "cost" of an algorithm as shorthand for the "value of the quantity in question" when the context allows.

The next step is to model the input to the program, to form a basis for the mathematical analysis of the instruction frequencies. The values of the unknown frequencies are dependent on the input to the algorithm: the problem size (usually we name that N) is normally the primary parameter used to express our results, but the order or value of input data items ordinarily affects the running time as well. By "model," we mean a precise description of typical inputs to the algorithm. For example, for sorting algorithms, it is normally convenient to assume that the inputs are randomly ordered and distinct, though the programs normally work even when the inputs are not distinct. Another possibility for sorting algorithms is to assume that the inputs are random numbers taken from a relatively large range. These two models can be shown to be nearly equivalent. Most often, we use the simplest available model of "random" inputs, which is often realistic. Several different models can be used for the same algorithm: one model might be chosen to make the analysis as simple as possible; another model might better reflect the actual situation in which the program is to be used.

The last step is to analyze the unknown quantities, assuming the modeled input. For average-case analysis, we analyze the quantities individually, then multiply the averages by instruction times and add them to find the running time of the whole program. For worst-case analysis, it is usually difficult

to get an exact result for the whole program, so we can only derive an upper bound, by multiplying worst-case values of the individual quantities by instruction times and summing the results.

This general scenario can successfully provide exact models in many situations. Knuth's books [17][18][19][20] are based on this precept. Unfortunately, the details in such an exact analysis are often daunting. Accordingly, we typically seek *approximate* models that we can use to estimate costs.

The first reason to approximate is that determining the cost details of all individual operations can be daunting in the context of the complex architectures and operating systems on modern computers. Accordingly, we typically study just a few quantities in the "inner loop" of our programs, implicitly hypothesizing that total cost is well estimated by analyzing just those quantities. Experienced programmers regularly "profile" their implementations to identify "bottlenecks," which is a systematic way to identify such quantities. For example, we typically analyze compare-based sorting algorithms by just counting compares. Such an approach has the important side benefit that it is *machine independent*. Carefully analyzing the number of compares used by a sorting algorithm can enable us to predict performance on many different computers. Associated hypotheses are easily tested by experimentation, and we can refine them, in principle, when appropriate. For example, we might refine comparison-based models for sorting to include data movement, which may require taking caching effects into account.

Exercise 1.11 Run experiments on two different computers to test the hypothesis that the running time of mergesort divided by the number of compares that it uses approaches a constant as the problem size increases.

Approximation is also effective for mathematical models. The second reason to approximate is to avoid unnecessary complications in the mathematical formulae that we develop to describe the performance of algorithms. A major theme of this book is the development of classical approximation methods for this purpose, and we shall consider many examples. Beyond these, a major thrust of modern research in the analysis of algorithms is methods of developing mathematical analyses that are simple, sufficiently precise that they can be used to accurately predict performance and to compare algorithms, and able to be refined, in principle, to the precision needed for the application at hand. Such techniques primarily involve complex analysis and are fully developed in our book [10].

1.4 Average-Case Analysis. The mathematical techniques that we consider in this book are not just applicable to solving problems related to the performance of algorithms, but also to mathematical models for all manner of scientific applications, from genomics to statistical physics. Accordingly, we often consider structures and techniques that are broadly applicable. Still, our prime motivation is to consider mathematical tools that we need in order to be able to make precise statements about resource usage of important algorithms in practical applications.

Our focus is on *average-case* analysis of algorithms: we formulate a reasonable input model and analyze the expected running time of a program given an input drawn from that model. This approach is effective for two primary reasons.

The first reason that average-case analysis is important and effective in modern applications is that straightforward models of randomness are often extremely accurate. The following are just a few representative examples from sorting applications:

- Sorting is a fundamental process in *cryptanalysis*, where the adversary has gone to great lengths to make the data indistinguishable from random data.
- *Commercial data processing* systems routinely sort huge files where keys typically are account numbers or other identification numbers that are well modeled by uniformly random numbers in an appropriate range.
- Implementations of *computer networks* depend on sorts that again involve keys that are well modeled by random ones.
- Sorting is widely used in *computational biology*, where significant deviations from randomness are cause for further investigation by scientists trying to understand fundamental biological and physical processes.

As these examples indicate, simple models of randomness are effective, not just for sorting applications, but also for a wide variety of uses of fundamental algorithms in practice. Broadly speaking, when large data sets are created by humans, they typically are based on arbitrary choices that are well modeled by random ones. Random models also are often effective when working with scientific data. We might interpret Einstein's oft-repeated admonition that "God does not play dice" in this context as meaning that random models are effective, because if we discover significant deviations from randomness, we have learned something significant about the natural world.

The second reason that average-case analysis is important and effective in modern applications is that we can often manage to inject randomness into a problem instance so that it appears to the algorithm (and to the analyst) to be random. This is an effective approach to developing efficient algorithms with predictable performance, which are known as *randomized algorithms*. M. O. Rabin [25] was among the first to articulate this approach, and it has been developed by many other researchers in the years since. The book by Motwani and Raghavan [23] is a thorough introduction to the topic.

Thus, we begin by analyzing random models, and we typically start with the challenge of computing the mean—the average value of some quantity of interest for N instances drawn at random. Now, elementary probability theory gives a number of different (though closely related) ways to compute the average value of a quantity. In this book, it will be convenient for us to explicitly identify two different approaches to doing so.

Distributional. Let Π_N be the number of possible inputs of size N and Π_{Nk} be the number of inputs of size N that cause the algorithm to have cost k, so that $\Pi_N = \sum_k \Pi_{Nk}$. Then the probability that the cost is k is Π_{Nk}/Π_N and the expected cost is

$$\frac{1}{\Pi_N} \sum_k k\Pi_{Nk}.$$

The analysis depends on "counting." How many inputs are there of size N and how many inputs of size N cause the algorithm to have cost k? These are the steps to compute the probability that the cost is k, so this approach is perhaps the most direct from elementary probability theory.

Cumulative. Let Σ_N be the total (or cumulated) cost of the algorithm on all inputs of size N. (That is, $\Sigma_N = \sum_k k\Pi_{Nk}$, but the point is that it is not necessary to compute Σ_N in that way.) Then the average cost is simply Σ_N/Π_N. The analysis depends on a less specific counting problem: what is the total cost of the algorithm, on all inputs? We will be using general tools that make this approach very attractive.

The distributional approach gives complete information, which can be used directly to compute the standard deviation and other moments. Indirect (often simpler) methods are also available for computing moments when using the cumulative approach, as we will see. In this book, we consider both approaches, though our tendency will be toward the cumulative method,

which ultimately allows us to consider the analysis of algorithms in terms of combinatorial properties of basic data structures.

Many algorithms solve a problem by recursively solving smaller subproblems and are thus amenable to the derivation of a recurrence relationship that the average cost or the total cost must satisfy. A direct derivation of a recurrence from the algorithm is often a natural way to proceed, as shown in the example in the next section.

No matter how they are derived, we are interested in average-case results because, in the large number of situations where random input is a reasonable model, an accurate analysis can help us:

- Compare different algorithms for the same task.
- Predict time and space requirements for specific applications.
- Compare different computers that are to run the same algorithm.
- Adjust algorithm parameters to optimize performance.

The average-case results can be compared with empirical data to validate the implementation, the model, and the analysis. The end goal is to gain enough confidence in these that they can be used to predict how the algorithm will perform under whatever circumstances present themselves in particular applications. If we wish to evaluate the possible impact of a new machine architecture on the performance of an important algorithm, we can do so through analysis, perhaps before the new architecture comes into existence. The success of this approach has been validated over the past several decades: the sorting algorithms that we consider in the section were first analyzed more than 50 years ago, and those analytic results are still useful in helping us evaluate their performance on today's computers.

1.5 Example: Analysis of Quicksort. To illustrate the basic method just sketched, we examine next a particular algorithm of considerable importance, the quicksort sorting method. This method was invented in 1962 by C. A. R. Hoare, whose paper [15] is an early and outstanding example in the analysis of algorithms. The analysis is also covered in great detail in Sedgewick [27] (see also [29]); we give highlights here. It is worthwhile to study this analysis in detail not just because this sorting method is widely used and the analytic results are directly relevant to practice, but also because the analysis itself is illustrative of many things that we will encounter later in the book. In particular, it turns out that the same analysis applies to the study of basic properties of tree structures, which are of broad interest and applicability. More gen-

erally, our analysis of quicksort is indicative of how we go about analyzing a broad class of recursive programs.

Program 1.2 is an implementation of quicksort in Java. It is a recursive program that sorts the numbers in an array by partitioning it into two independent (smaller) parts, then sorting those parts. Obviously, the recursion should terminate when empty subarrays are encountered, but our implementation also stops with subarrays of size 1. This detail might seem inconsequential at first blush, but, as we will see, the very nature of recursion ensures that the program will be used for a large number of small files, and substantial performance gains can be achieved with simple improvements of this sort.

The partitioning process puts the element that was in the last position in the array (the *partitioning element*) into its correct position, with all smaller elements before it and all larger elements after it. The program accomplishes this by maintaining two pointers: one scanning from the left, one from the right. The left pointer is incremented until an element larger than the parti-

```
private void quicksort(int[] a, int lo, int hi)
{
   if (hi <= lo) return;
   int i = lo-1, j = hi;
   int t, v = a[hi];
   while (true)
   {
      while (a[++i] < v) ;
      while (v < a[--j]) if (j == lo) break;
      if (i >= j) break;
      t = a[i]; a[i] = a[j]; a[j] = t;
   }
   t = a[i]; a[i] = a[hi]; a[hi] = t;
   quicksort(a, lo, i-1);
   quicksort(a, i+1, hi);
}
```

Program 1.2 Quicksort

tioning element is found; the right pointer is decremented until an element smaller than the partitioning element is found. These two elements are exchanged, and the process continues until the pointers meet, which defines where the partitioning element is put. After partitioning, the program exchanges `a[i]` with `a[hi]` to put the partitioning element into position. The call `quicksort(a, 0, N-1)` will sort the array.

There are several ways to implement the general recursive strategy just outlined; the implementation described above is taken from Sedgewick and Wayne [30] (see also [27]). For the purposes of analysis, we will be assuming that the array `a` contains randomly ordered, distinct numbers, but note that this code works properly for all inputs, including equal numbers. It is also possible to study this program under perhaps more realistic models allowing equal numbers (see [28]), long string keys (see [4]), and many other situations.

Once we have an implementation, the first step in the analysis is to estimate the resource requirements of individual instructions for this program. This depends on characteristics of a particular computer, so we sketch the details. For example, the "inner loop" instruction

```
while (a[++i] < v) ;
```

might translate, on a typical computer, to assembly language instructions such as the following:

```
LOOP  INC   I,1         # increment i
      CMP   V,A(I)      # compare v with A(i)
      BL    LOOP        # branch if less
```

To start, we might say that one iteration of this loop might require four time units (one for each memory reference). On modern computers, the precise costs are more complicated to evaluate because of caching, pipelines, and other effects. The other instruction in the inner loop (that decrements `j`) is similar, but involves an extra test of whether `j` goes out of bounds. Since this extra test can be removed via sentinels (see [26]), we will ignore the extra complication it presents.

The next step in the analysis is to assign variable names to the frequency of execution of the instructions in the program. Normally there are only a few true variables involved: the frequencies of execution of all the instructions can be expressed in terms of these few. Also, it is desirable to relate the variables to

the algorithm itself, not any particular program. For quicksort, three natural quantities are involved:

A – the number of partitioning stages
B – the number of exchanges
C – the number of compares

On a typical computer, the total running time of quicksort might be expressed with a formula, such as

$$4C + 11B + 35A. \tag{2}$$

The exact values of these coefficients depend on the machine language program produced by the compiler as well as the properties of the machine being used; the values given above are typical. Such expressions are quite useful in comparing different algorithms implemented on the same machine. Indeed, the reason that quicksort is of practical interest even though mergesort is "optimal" is that the cost per compare (the coefficient of C) is likely to be significantly lower for quicksort than for mergesort, which leads to significantly shorter running times in typical practical applications.

Theorem 1.3 (Quicksort analysis). Quicksort uses, on the average,

$$\begin{aligned} &(N-1)/2 \quad \text{partitioning stages,} \\ &2(N+1)\left(H_{N+1}-3/2\right) \approx 2N\ln N - 1.846N \quad \text{compares, and} \\ &(N+1)\left(H_{N+1}-3\right)/3+1 \approx .333N\ln N - .865N \quad \text{exchanges} \end{aligned}$$

to sort an array of N randomly ordered distinct elements.

Proof. The exact answers here are expressed in terms of the *harmonic numbers*

$$H_N = \sum_{1\le k\le N} 1/k,$$

the first of many well-known "special" number sequences that we will encounter in the analysis of algorithms.

As with mergesort, the analysis of quicksort involves defining and solving recurrence relations that mirror directly the recursive nature of the algorithm. But, in this case, the recurrences must be based on probabilistic

statements about the inputs. If C_N is the average number of compares to sort N elements, we have $C_0 = C_1 = 0$ and

$$C_N = N + 1 + \frac{1}{N} \sum_{1 \le j \le N} (C_{j-1} + C_{N-j}), \qquad \text{for } N > 1. \tag{3}$$

To get the total average number of compares, we add the number of compares for the first partitioning stage ($N+1$) to the number of compares used for the subarrays after partitioning. When the partitioning element is the jth largest (which occurs with probability $1/N$ for each $1 \le j \le N$), the subarrays after partitioning are of size $j-1$ and $N-j$.

Now the analysis has been reduced to a mathematical problem (3) that does not depend on properties of the program or the algorithm. This recurrence relation is somewhat more complicated than (1) because the right-hand side depends directly on the history of all the previous values, not just a few. Still, (3) is not difficult to solve: first change j to $N-j+1$ in the second part of the sum to get

$$C_N = N + 1 + \frac{2}{N} \sum_{1 \le j \le N} C_{j-1} \qquad \text{for } N > 0.$$

Then multiply by N and subtract the same formula for $N-1$ to eliminate the sum:

$$NC_N - (N-1)C_{N-1} = 2N + 2C_{N-1} \qquad \text{for } N > 1.$$

Now rearrange terms to get a simple recurrence

$$NC_N = (N+1)C_{N-1} + 2N \qquad \text{for } N > 1.$$

This can be solved by dividing both sides by $N(N+1)$:

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1} \qquad \text{for } N > 1.$$

Iterating, we are left with the sum

$$\frac{C_N}{N+1} = \frac{C_1}{2} + 2 \sum_{3 \le k \le N+1} 1/k$$

which completes the proof, since $C_1 = 0$.

As implemented earlier, every element is used for partitioning exactly once, so the number of stages is always N; the average number of exchanges can be found from these results by first calculating the average number of exchanges on the first partitioning stage.

The stated approximations follow from the well-known approximation to the harmonic number $H_N \approx \ln N + .57721\cdots$. We consider such approximations below and in detail in Chapter 4. ■

Exercise 1.12 Give the recurrence for the total number of compares used by quicksort on all $N!$ permutations of N elements.

Exercise 1.13 Prove that the subarrays left after partitioning a random permutation are themselves both random permutations. Then prove that this is *not* the case if, for example, the right pointer is initialized at `j:=r+1` for partitioning.

Exercise 1.14 Follow through the steps above to solve the recurrence

$$A_N = 1 + \frac{2}{N} \sum_{1 \le j \le N} A_{j-1} \qquad \text{for } N > 0.$$

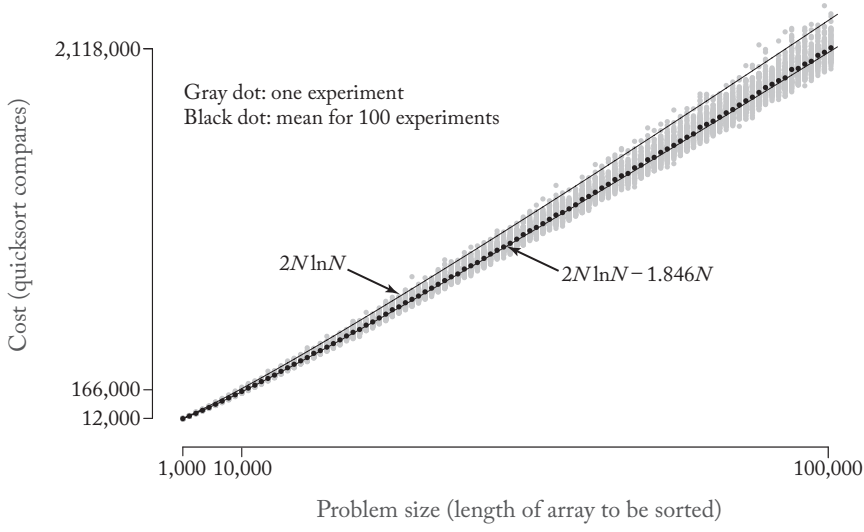

Figure 1.1 Quicksort compare counts: empirical and analytic

Exercise 1.15 Show that the average number of exchanges used during the first partitioning stage (before the pointers cross) is $(N-2)/6$. (Thus, by linearity of the recurrences, $B_N = \frac{1}{6}C_N - \frac{1}{2}A_N$.)

Figure 1.1 shows how the analytic result of Theorem 1.3 compares to empirical results computed by generating random inputs to the program and counting the compares used. The empirical results (100 trials for each value of N shown) are depicted with a gray dot for each experiment and a black dot at the mean for each N. The analytic result is a smooth curve fitting the formula given in Theorem 1.3. As expected, the fit is extremely good.

Theorem 1.3 and (2) imply, for example, that quicksort should take about $11.667N\ln N - .601N$ steps to sort a random permutation of N elements for the particular machine described previously, and similar formulae for other machines can be derived through an investigation of the properties of the machine as in the discussion preceding (2) and Theorem 1.3. Such formulae can be used to predict (with great accuracy) the running time of quicksort on a particular machine. More important, they can be used to evaluate and compare variations of the algorithm and provide a quantitative testimony to their effectiveness.

Secure in the knowledge that machine dependencies can be handled with suitable attention to detail, we will generally concentrate on analyzing generic algorithm-dependent quantities, such as "compares" and "exchanges," in this book. Not only does this keep our focus on major techniques of analysis, but it also can extend the applicability of the results. For example, a slightly broader characterization of the sorting problem is to consider the items to be sorted as *records* containing other information besides the sort *key*, so that accessing a record might be much more expensive (depending on the size of the record) than doing a compare (depending on the relative size of records and keys). Then we know from Theorem 1.3 that quicksort compares keys about $2N\ln N$ times and moves records about $.667N\ln N$ times, and we can compute more precise estimates of costs or compare with other algorithms as appropriate.

Quicksort can be improved in several ways to make it the sorting method of choice in many computing environments. We can even analyze complicated improved versions and derive expressions for the average running time that match closely observed empirical times [29]. Of course, the more intricate and complicated the proposed improvement, the more intricate and com-

plicated the analysis. Some improvements can be handled by extending the argument given previously, but others require more powerful analytic tools.

Small subarrays. The simplest variant of quicksort is based on the observation that it is not very efficient for very small files (for example, a file of size 2 can be sorted with one compare and possibly one exchange), so that a simpler method should be used for smaller subarrays. The following exercises show how the earlier analysis can be extended to study a hybrid algorithm where "insertion sort" (see §7.6) is used for files of size less than M. Then, this analysis can be used to help choose the best value of the parameter M.

Exercise 1.16 How many subarrays of size 2 or less are encountered, on the average, when sorting a random file of size N with quicksort?

Exercise 1.17 If we change the first line in the quicksort implementation above to

```
if r-l<=M then insertionsort(l,r) else
```

(see §7.6), then the total number of compares to sort N elements is described by the recurrence

$$C_N = \begin{cases} N+1+\dfrac{1}{N}\displaystyle\sum_{1\le j\le N}(C_{j-1}+C_{N-j}) & \text{for } N > M; \\ \frac{1}{4}N(N-1) & \text{for } N \le M. \end{cases}$$

Solve this exactly as in the proof of Theorem 1.3.

Exercise 1.18 Ignoring small terms (those significantly less than N) in the answer to the previous exercise, find a function $f(M)$ so that the number of compares is approximately

$$2N\ln N + f(M)N.$$

Plot the function $f(M)$, and find the value of M that minimizes the function.

Exercise 1.19 As M gets larger, the number of compares increases again from the minimum just derived. How large must M get before the number of compares exceeds the original number (at $M = 0$)?

Median-of-three quicksort. A natural improvement to quicksort is to use sampling: estimate a partitioning element more likely to be near the middle of the file by taking a small sample, then using the median of the sample. For example, if we use just three elements for the sample, then the average number

of compares required by this "median-of-three" quicksort is described by the recurrence

$$C_N = N+1+\sum_{1\le k\le N}\frac{(N-k)(k-1)}{\binom{N}{3}}(C_{k-1}+C_{N-k}) \qquad \text{for } N>3 \quad (4)$$

where $\binom{N}{3}$ is the binomial coefficient that counts the number of ways to choose 3 out of N items. This is true because the probability that the kth smallest element is the partitioning element is now $(N-k)(k-1)/\binom{N}{3}$ (as opposed to $1/N$ for regular quicksort). We would like to be able to solve recurrences of this nature to be able to determine how large a sample to use and when to switch to insertion sort. However, such recurrences require more sophisticated techniques than the simple ones used so far. In Chapters 2 and 3, we will see methods for developing precise solutions to such recurrences, which allow us to determine the best values for parameters such as the sample size and the cutoff for small subarrays. Extensive studies along these lines have led to the conclusion that median-of-three quicksort with a cutoff point in the range 10 to 20 achieves close to optimal performance for typical implementations.

Radix-exchange sort. Another variant of quicksort involves taking advantage of the fact that the keys may be viewed as binary strings. Rather than comparing against a key from the file for partitioning, we partition the file so that all keys with a leading 0 bit precede all those with a leading 1 bit. Then these subarrays can be independently subdivided in the same way using the second bit, and so forth. This variation is referred to as "radix-exchange sort" or "radix quicksort." How does this variation compare with the basic algorithm? To answer this question, we first have to note that a different mathematical model is required, since keys composed of random bits are essentially different from random permutations. The "random bitstring" model is perhaps more realistic, as it reflects the actual representation, but the models can be proved to be roughly equivalent. We will discuss this issue in more detail in Chapter 8. Using a similar argument to the one given above, we can show that the average number of bit compares required by this method is described by the recurrence

$$C_N = N+\frac{1}{2^N}\sum_k \binom{N}{k}(C_k+C_{N-k}) \qquad \text{for } N>1 \text{ with } C_0=C_1=0.$$

This turns out to be a rather more difficult recurrence to solve than the one given earlier—we will see in Chapter 3 how generating functions can be used to transform the recurrence into an explicit formula for C_N, and in Chapters 4 and 8, we will see how to develop an approximate solution.

One limitation to the applicability of this kind of analysis is that all of the preceding recurrence relations depend on the "randomness preservation" property of the algorithm: if the original file is randomly ordered, it can be shown that the subarrays after partitioning are also randomly ordered. The implementor is not so restricted, and many widely used variants of the algorithm do not have this property. Such variants appear to be extremely difficult to analyze. Fortunately (from the point of view of the analyst), empirical studies show that they also perform poorly. Thus, though it has not been analytically quantified, the requirement for randomness preservation seems to produce more elegant and efficient quicksort implementations. More important, the versions that preserve randomness do admit to performance improvements that can be fully quantified mathematically, as described earlier.

Mathematical analysis has played an important role in the development of practical variants of quicksort, and we will see that there is no shortage of other problems to consider where detailed mathematical analysis is an important part of the algorithm design process.

1.6 Asymptotic Approximations. The derivation of the average running time of quicksort given earlier yields an exact result, but we also gave a more concise approximate expression in terms of well-known functions that still can be used to compute accurate numerical estimates. As we will see, it is often the case that an exact result is not available, or at least an approximation is far easier to derive and interpret. Ideally, our goal in the analysis of an algorithm should be to derive exact results; from a pragmatic point of view, it is perhaps more in line with our general goal of being able to make useful performance predications to strive to derive concise but precise approximate answers.

To do so, we will need to use classical techniques for manipulating such approximations. In Chapter 4, we will examine the Euler-Maclaurin summation formula, which provides a way to estimate sums with integrals. Thus, we can approximate the harmonic numbers by the calculation

$$H_N = \sum_{1\le k\le N} \frac{1}{k} \approx \int_1^N \frac{1}{x} dx = \ln N.$$

But we can be much more precise about the meaning of $\approx$, and we can conclude (for example) that $H_N = \ln N + \gamma + 1/(2N) + O(1/N^2)$ where $\gamma = .57721\cdots$ is a constant known in analysis as Euler's constant. Though the constants implicit in the O-notation are not specified, this formula provides a way to estimate the value of H_N with increasingly improving accuracy as N increases. Moreover, if we want even better accuracy, we can derive a formula for H_N that is accurate to within $O(N^{-3})$ or indeed to within $O(N^{-k})$ for any constant k. Such approximations, called *asymptotic expansions*, are at the heart of the analysis of algorithms, and are the subject of Chapter 4.

The use of asymptotic expansions may be viewed as a compromise between the ideal goal of providing an exact result and the practical requirement of providing a concise approximation. It turns out that we are normally in the situation of, on the one hand, having the ability to derive a more accurate expression if desired, but, on the other hand, not having the desire, because expansions with only a few terms (like the one for H_N above) allow us to compute answers to within several decimal places. We typically drop back to using the $\approx$ notation to summarize results without naming irrational constants, as, for example, in Theorem 1.3.

Moreover, exact results and asymptotic approximations are both subject to inaccuracies inherent in the probabilistic model (usually an idealization of reality) and to stochastic fluctuations. Table 1.1 shows exact, approximate, and empirical values for number of compares used by quicksort on random files of various sizes. The exact and approximate values are computed from the formulae given in Theorem 1.3; the "empirical" is a measured average, taken over 100 files consisting of random positive integers less than 10^6; this tests not only the asymptotic approximation that we have discussed, but also the "approximation" inherent in our use of the random permutation model, ignoring equal keys. The analysis of quicksort when equal keys are present is treated in Sedgewick [28].

Exercise 1.20 How many keys in a file of 10^4 random integers less than 10^6 are likely to be equal to some other key in the file? Run simulations, or do a mathematical analysis (with the help of a system for mathematical calculations), or do both.

Exercise 1.21 Experiment with files consisting of random positive integers less than M for $M = 10{,}000, 1000, 100$ and other values. Compare the performance of quicksort on such files with its performance on random permutations of the same size. Characterize situations where the random permutation model is inaccurate.

Exercise 1.22 Discuss the idea of having a table similar to Table 1.1 for mergesort.

In the theory of algorithms, O-notation is used to suppress detail of all sorts: the statement that mergesort requires $O(N \log N)$ compares hides everything but the most fundamental characteristics of the algorithm, implementation, and computer. In the analysis of algorithms, asymptotic expansions provide us with a controlled way to suppress irrelevant details, while preserving the most important information, especially the constant factors involved. The most powerful and general analytic tools produce asymptotic expansions directly, thus often providing simple direct derivations of concise but accurate expressions describing properties of algorithms. We are sometimes able to use asymptotic estimates to provide *more* accurate descriptions of program performance than might otherwise be available.

file size	exact solution	approximate	empirical
10,000	175,771	175,746	176,354
20,000	379,250	379,219	374,746
30,000	593,188	593,157	583,473
40,000	813,921	813,890	794,560
50,000	1,039,713	1,039,677	1,010,657
60,000	1,269,564	1,269,492	1,231,246
70,000	1,502,729	1,502,655	1,451,576
80,000	1,738,777	1,738,685	1,672,616
90,000	1,977,300	1,977,221	1,901,726
100,000	2,218,033	2,217,985	2,126,160

Table 1.1 Average number of compares used by quicksort

1.7 Distributions. In general, probability theory tells us that other facts about the distribution Π_{Nk} of costs are also relevant to our understanding of performance characteristics of an algorithm. Fortunately, for virtually all of the examples that we study in the analysis of algorithms, it turns out that knowing an asymptotic estimate for the average is enough to be able to make reliable predictions. We review a few basic ideas here. Readers not familiar with probability theory are referred to any standard text—for example, [9].

The full distribution for the number of compares used by quicksort for small N is shown in Figure 1.2. For each value of N, the points $C_{Nk}/N!$ are plotted: the proportion of the inputs for which quicksort uses k compares. Each curve, being a full probability distribution, has area 1. The curves move to the right, since the average $2N\ln N + O(N)$ increases with N. A slightly different view of the same data is shown in Figure 1.3, where the horizontal axes for each curve are scaled to put the mean approximately at the center and shifted slightly to separate the curves. This illustrates that the distribution converges to a "limiting distribution."

For many of the problems that we study in this book, not only do limiting distributions like this exist, but also we are able to precisely characterize them. For many other problems, including quicksort, that is a significant challenge. However, it is very clear that the distribution is *concentrated near*

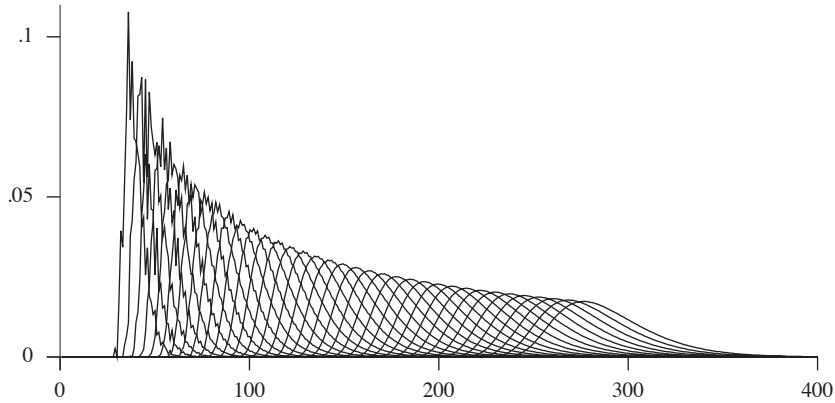

Figure 1.2 Distributions for compares in quicksort, $15 \leq N \leq 50$

the mean. This is commonly the case, and it turns out that we can make precise statements to this effect, and do not need to learn more details about the distribution.

As discussed earlier, if Π_N is the number of inputs of size N and Π_{Nk} is the number of inputs of size N that cause the algorithm to have cost k, the average cost is given by

$$\mu = \sum_k k\Pi_{Nk}/\Pi_N.$$

The *variance* is defined to be

$$\sigma^2 = \sum_k (k-\mu)^2\Pi_{Nk}/\Pi_N = \sum_k k^2\Pi_{Nk}/\Pi_N - \mu^2.$$

The *standard deviation* σ is the square root of the variance. Knowing the average and standard deviation ordinarily allows us to predict performance

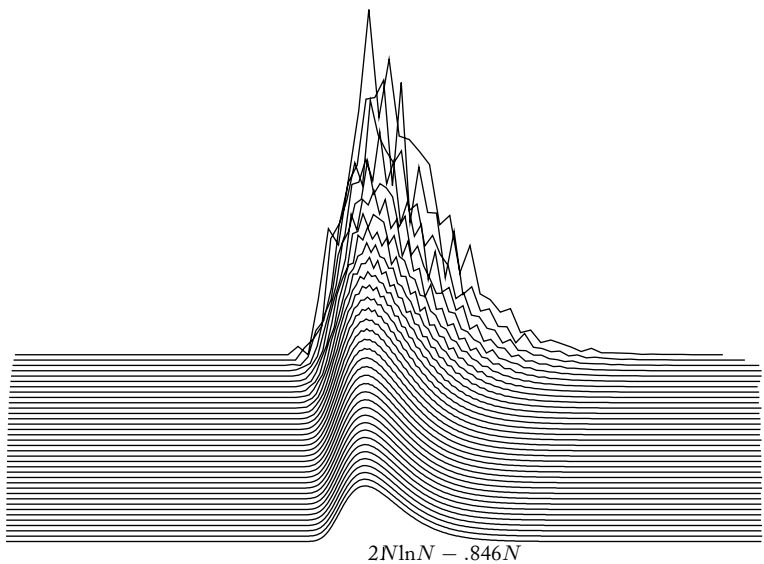

Figure 1.3 Distributions for compares in quicksort, $15 \leq N \leq 50$ (scaled and translated to center and separate curves)