AN INTRODUCTION TO THE ANALYSIS OF ALGORITHMS

SECOND EDITION

ROBERT SEDGEWICK
PHILIPPE FLAJOLET

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

AN INTRODUCTION TO THE ANALYSIS OF ALGORITHMS

Second Edition

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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
`corpsales@pearsontechgroup.com`

For sales outside the United States, please contact:

International Sales
`international@pearsoned.com`

Visit us on the Web: `informit.com/aw`

Library of Congress Control Number: 2012955493

Copyright © 2013 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

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

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:

INTRODUCTION

ONE ANALYSIS OF ALGORITHMS

DISCRETE MATHEMATICAL METHODS

TWO RECURRENCE RELATIONS

THREE GENERATING FUNCTIONS

FOUR ASYMPTOTIC APPROXIMATIONS

FIVE ANALYTIC COMBINATORICS

ALGORITHMS AND COMBINATORIAL STRUCTURES

SIX TREES

SEVEN PERMUTATIONS

EIGHT STRINGS AND TRIES

NINE WORDS AND MAPPINGS

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.

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

Chapter One: Analysis of Algorithms **3**

1.1 Why Analyze an Algorithm? 3
1.2 Theory of Algorithms 6
1.3 Analysis of Algorithms 13
1.4 Average-Case Analysis 16
1.5 Example: Analysis of Quicksort 18
1.6 Asymptotic Approximations 27
1.7 Distributions 30
1.8 Randomized Algorithms 33

Chapter Two: Recurrence Relations **41**

2.1 Basic Properties 43
2.2 First-Order Recurrences 48
2.3 Nonlinear First-Order Recurrences 52
2.4 Higher-Order Recurrences 55
2.5 Methods for Solving Recurrences 61
2.6 Binary Divide-and-Conquer Recurrences and Binary Numbers 70
2.7 General Divide-and-Conquer Recurrences 80

Chapter Three: Generating Functions **91**

3.1 Ordinary Generating Functions 92
3.2 Exponential Generating Functions 97
3.3 Generating Function Solution of Recurrences 101
3.4 Expanding Generating Functions 111
3.5 Transformations with Generating Functions 114
3.6 Functional Equations on Generating Functions 117
3.7 Solving the Quicksort Median-of-Three Recurrence with OGFs 120
3.8 Counting with Generating Functions 123
3.9 Probability Generating Functions 129
3.10 Bivariate Generating Functions 132
3.11 Special Functions 140

CHAPTER FOUR: ASYMPTOTIC APPROXIMATIONS 151

4.1 Notation for Asymptotic Approximations 153
4.2 Asymptotic Expansions 160
4.3 Manipulating Asymptotic Expansions 169
4.4 Asymptotic Approximations of Finite Sums 176
4.5 Euler-Maclaurin Summation 179
4.6 Bivariate Asymptotics 187
4.7 Laplace Method 203
4.8 "Normal" Examples from the Analysis of Algorithms 207
4.9 "Poisson" Examples from the Analysis of Algorithms 211

CHAPTER FIVE: ANALYTIC COMBINATORICS 219

5.1 Formal Basis 220
5.2 Symbolic Method for Unlabelled Classes 221
5.3 Symbolic Method for Labelled Classes 229
5.4 Symbolic Method for Parameters 241
5.5 Generating Function Coefficient Asymptotics 247

CHAPTER SIX: TREES 257

6.1 Binary Trees 258
6.2 Forests and Trees 261
6.3 Combinatorial Equivalences to Trees and Binary Trees 264
6.4 Properties of Trees 272
6.5 Examples of Tree Algorithms 277
6.6 Binary Search Trees 281
6.7 Average Path Length in Catalan Trees 287
6.8 Path Length in Binary Search Trees 293
6.9 Additive Parameters of Random Trees 297
6.10 Height 302
6.11 Summary of Average-Case Results on Properties of Trees 310
6.12 Lagrange Inversion 312
6.13 Rooted Unordered Trees 315
6.14 Labelled Trees 327
6.15 Other Types of Trees 331

Chapter Seven: Permutations **345**

7.1 Basic Properties of Permutations 347
7.2 Algorithms on Permutations 355
7.3 Representations of Permutations 358
7.4 Enumeration Problems 366
7.5 Analyzing Properties of Permutations with CGFs 372
7.6 Inversions and Insertion Sorts 384
7.7 Left-to-Right Minima and Selection Sort 393
7.8 Cycles and In Situ Permutation 401
7.9 Extremal Parameters 406

Chapter Eight: Strings and Tries **415**

8.1 String Searching 416
8.2 Combinatorial Properties of Bitstrings 420
8.3 Regular Expressions 432
8.4 Finite-State Automata and the Knuth-Morris-Pratt Algorithm 437
8.5 Context-Free Grammars 441
8.6 Tries 448
8.7 Trie Algorithms 453
8.8 Combinatorial Properties of Tries 459
8.9 Larger Alphabets 465

Chapter Nine: Words and Mappings **473**

9.1 Hashing with Separate Chaining 474
9.2 The Balls-and-Urns Model and Properties of Words 476
9.3 Birthday Paradox and Coupon Collector Problem 485
9.4 Occupancy Restrictions and Extremal Parameters 495
9.5 Occupancy Distributions 501
9.6 Open Addressing Hashing 509
9.7 Mappings 519
9.8 Integer Factorization and Mappings 532

List of Theorems 543
List of Tables 545
List of Figures 547
Index 551

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

$\left[{n \atop k} \right]$ *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

$$4C + 11B + 35A. \qquad (2)$$

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,

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

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

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

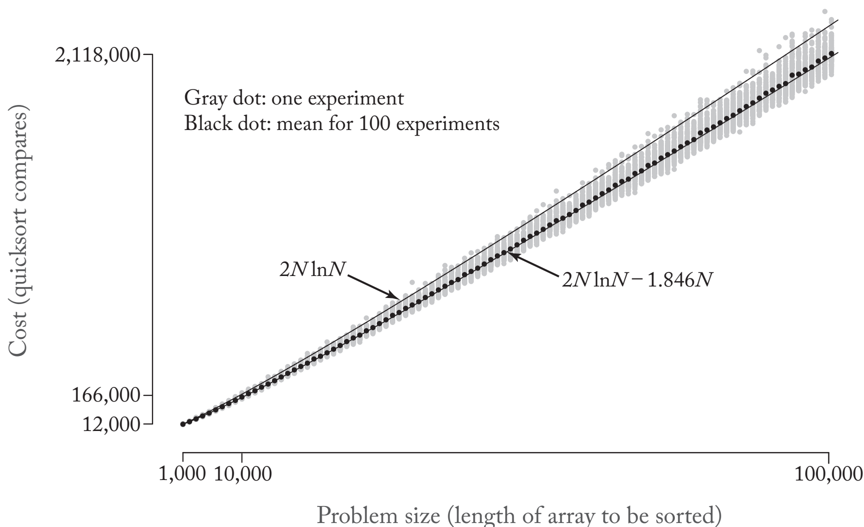

Figure 1.1 Quicksort compare counts: empirical and analytic

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*

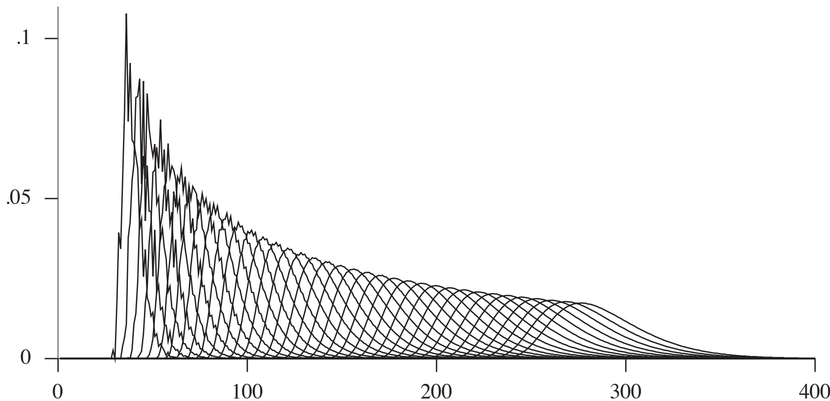

Figure 1.2 Distributions for compares in quicksort, $15 \leq N \leq 50$

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

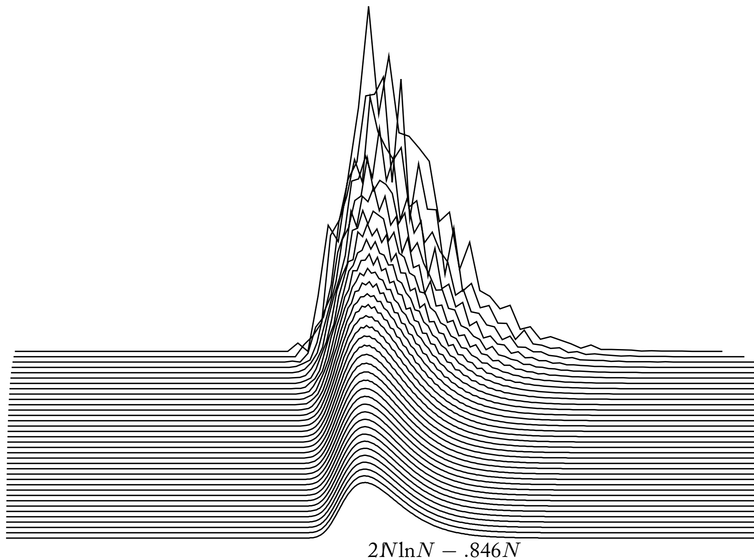

Figure 1.3 Distributions for compares in quicksort, $15 \le N \le 50$ (scaled and translated to center and separate curves)

reliably. The classical analytic tool that allows this is the *Chebyshev inequality*: the probability that an observation will be more than c multiples of the standard deviation away from the mean is less than $1/c^2$. If the standard deviation is significantly smaller than the mean, then, as N gets large, an observed value is very likely to be quite close to the mean. This is often the case in the analysis of algorithms.

Exercise 1.23 What is the standard deviation of the number of compares for the mergesort implementation given earlier in this chapter?

The standard deviation of the number of compares used by quicksort is

$$\sqrt{(21 - 2\pi^2)/3}\,N \approx .6482776N$$

(see §3.9) so, for example, referring to Table 1.1 and taking $c = \sqrt{10}$ in Chebyshev's inequality, we conclude that there is more than a 90% chance that the number of compares when $N = 100{,}000$ is within 205,004 (9.2%) of 2,218,033. Such accuracy is certainly adequate for predicting performance.

As N increases, the relative accuracy also increases: for example, the distribution becomes more localized near the peak in Figure 1.3 as N increases. Indeed, Chebyshev's inequality underestimates the accuracy in this situation, as shown in Figure 1.4. This figure plots a histogram showing the number of compares used by quicksort on 10,000 different random files of 1000 elements. The shaded area shows that more than 94% of the trials fell within *one* standard deviation of the mean for this experiment.

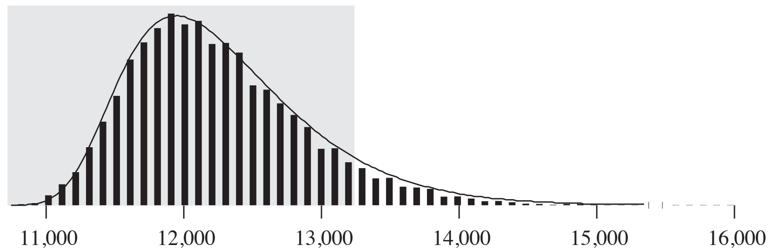

Figure 1.4 Empirical histogram for quicksort compare counts (10,000 trials with N=1000)

For the total running time, we can sum averages (multiplied by costs) of individual quantities, but computing the variance is an intricate calculation that we do not bother to do because the variance of the total is asymptotically the same as the largest variance. The fact that the standard deviation is small relative to the average for large N explains the observed accuracy of Table 1.1 and Figure 1.1. Cases in the analysis of algorithms where this does not happen are rare, and we normally consider an algorithm "fully analyzed" if we have a precise asymptotic estimate for the average cost and knowledge that the standard deviation is asymptotically smaller.

1.8 Randomized Algorithms.

The analysis of the average-case performance of quicksort depends on the input being randomly ordered. This assumption is not likely to be strictly valid in many practical situations. In general, this situation reflects one of the most serious challenges in the analysis of algorithms: the need to properly formulate models of inputs that might appear in practice.

Fortunately, there is often a way to circumvent this difficulty: "randomize" the inputs before using the algorithm. For sorting algorithms, this simply amounts to randomly permuting the input file before the sort. (See Chapter 7 for a specific implementation of an algorithm for this purpose.) If this is done, then probabilistic statements about performance such as those made earlier are completely valid and will accurately predict performance in practice, no matter what the input.

Often, it is possible to achieve the same result with less work, by making a random choice (as opposed to a specific arbitrary choice) whenever the algorithm could take one of several actions. For quicksort, this principle amounts to choosing the element to be used as the partitioning element at random, rather than using the element at the end of the array each time. If this is implemented with care (preserving randomness in the subarrays) then, again, it validates the probabilistic analysis given earlier. (Also, the cutoff for small subarrays should be used, since it cuts down the number of random numbers to generate by a factor of about M.) Many other examples of randomized algorithms may be found in [23] and [25]. Such algorithms are of interest in practice because they take advantage of randomness to gain efficiency and to avoid worst-case performance with high probability. Moreover, we can make precise probabilistic statements about performance, further motivating the study of advanced techniques for deriving such results.

THE example of the analysis of quicksort that we have been considering perhaps illustrates an idealized methodology: not all algorithms can be as smoothly dealt with as this. A full analysis like this one requires a fair amount of effort that should be reserved only for our most important algorithms. Fortunately, as we will see, there are many fundamental methods that do share the basic ingredients that make analysis worthwhile, where we can

- Specify realistic input models.
- Derive mathematical models that describe costs.
- Develop concise, accurate solutions.
- Use the solutions to compare variants and compare with other algorithms, and help adjust values of algorithm parameters.

In this book, we consider a wide variety of such methods, concentrating on mathematical techniques validating the second and third of these points.

Most often, we skip the parts of the methodology outlined above that are program-specific (dependent on the implementation), to concentrate either on algorithm design, where rough estimates of the running time may suffice, or on the mathematical analysis, where the formulation and solution of the mathematical problem involved are of most interest. These are the areas involving the most significant intellectual challenge, and deserve the attention that they get.

As we have already mentioned, one important challenge in analysis of algorithms in common use on computers today is to formulate models that realistically represent the input and that lead to manageable analysis problems. We do not dwell on this problem because there is a large class of *combinatorial* algorithms for which the models are natural. In this book, we consider examples of such algorithms and the fundamental structures upon which they operate in some detail. We study permutations, trees, strings, tries, words, and mappings because they are all both widely studied combinatorial structures and widely used data structures *and* because "random" structures are both straightforward and realistic.

In Chapters 2 through 5, we concentrate on techniques of mathematical analysis that are applicable to the study of algorithm performance. This material is important in many applications beyond the analysis of algorithms, but our coverage is developed as preparation for applications later in the book. Then, in Chapters 6 through 9 we apply these techniques to the analysis of some fundamental combinatorial algorithms, including several of practical interest. Many of these algorithms are of basic importance in a wide variety

of computer applications, and so are deserving of the effort involved for detailed analysis. In some cases, algorithms that seem to be quite simple can lead to quite intricate mathematical analyses; in other cases, algorithms that are apparently rather complicated can be dealt with in a straightforward manner. In both situations, analyses can uncover significant differences between algorithms that have direct bearing on the way they are used in practice.

It is important to note that we teach and present mathematical derivations in the classical style, even though modern computer algebra systems such as Maple, Mathematica, or Sage are indispensable nowadays to check and develop results. The material that we present here may be viewed as preparation for learning to make effective use of such systems.

Much of our focus is on effective methods for determining performance characteristics of algorithm implementations. Therefore, we present programs in a widely used programming language (Java). One advantage of this approach is that the programs are complete and unambiguous descriptions of the algorithms. Another is that readers may run empirical tests to validate mathematical results. Generally our programs are stripped-down versions of the full Java implementations in the Sedgewick and Wayne *Algorithms* text [30]. To the extent possible, we use standard language mechanisms, so people familiar with other programming environments may translate them. More information about many of the programs we cover may be found in [30].

The basic methods that we cover are, of course, applicable to a much wider class of algorithms and structures than we are able to discuss in this introductory treatment. We cover only a few of the large number of combinatorial algorithms that have been developed since the advent of computers in mid-20th century. We do not touch on the scores of applications areas, from image processing to bioinformatics, where algorithms have proved effective and have been investigated in depth. We mention only briefly approaches such as amortized analysis and the probabilistic method, which have been successfully applied to the analysis of a number of important algorithms. Still, it is our hope that mastery of the introductory material in this book is good preparation for appreciating such material in the research literature in the analysis of algorithms. Beyond the books by Knuth, Sedgewick and Wayne, and Cormen, Leiserson, Rivest, and Stein cited earlier, other sources of information about the analysis of algorithms and the theory of algorithms are the books by Gonnet and Baeza-Yates [11], by Dasgupta, Papadimitriou, and Vazirani [7], and by Kleinberg and Tardos [16].

Equally important, we are led to analytic problems of a combinatorial nature that allow us to develop general mechanisms that may help to analyze future, as yet undiscovered, algorithms. The methods that we use are drawn from the classical fields of combinatorics and asymptotic analysis, and we are able to apply classical methods from these fields to treat a broad variety of problems in a uniform way. This process is described in full detail in our book *Analytic Combinatorics* [10]. Ultimately, we are not only able to directly formulate combinatorial enumeration problems from simple formal descriptions, but also we are able to directly derive asymptotic estimates of their solution from these formulations.

In this book, we cover the important fundamental concepts while at the same time developing a context for the more advanced treatment in [10] and in other books that study advanced methods, such as Szpankowski's study of algorithms on words [32] or Drmota' study of trees [8]. Graham, Knuth, and Patashnik [12] is a good source of more material relating to the mathematics that we use; standard references such as Comtet [5] (for combinatorics) and Henrici [14] (for analysis) also have relevant material. Generally, we use elementary combinatorics and real analysis in this book, while [10] is a more advanced treatment from a combinatorial point of view, and relies on complex analysis for asymptotics.

Properties of classical mathematical functions are an important part of our story. The classic *Handbook of Mathematical Functions* by Abramowitz and Stegun [1] was an indispensable reference for mathematicians for decades and was certainly a resource for the development of this book. A new reference that is intended to replace it was recently published, with associated online material [24]. Indeed, reference material of this sort is increasingly found online, in resources such as *Wikipedia* and *Mathworld* [35]. Another important resource is Sloane's *On-Line Encyclopedia of Integer Sequences* [31].

Our starting point is to study characteristics of fundamental algorithms that are in widespread use, but our primary purpose in this book is to provide a coherent treatment of the combinatorics and analytic methods that we encounter. When appropriate, we consider in detail the mathematical problems that arise naturally and may not apply to any (currently known!) algorithm. In taking such an approach we are led to problems of remarkable scope and diversity. Furthermore, in examples throughout the book we see that the problems we solve are directly relevant to many important applications.

References

1. M. Abramowitz and I. Stegun. *Handbook of Mathematical Functions*, Dover, New York, 1972.
2. A. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Algorithms*, Addison-Wesley, Reading, MA, 1975.
3. B. Char, K. Geddes, G. Gonnet, B. Leong, M. Monagan, and S. Watt. *Maple V Library Reference Manual*, Springer-Verlag, New York, 1991. Also *Maple User Manual*, Maplesoft, Waterloo, Ontario, 2012.
4. J. Clément, J. A. Fill, P. Flajolet, and B. Valée. "The number of symbol comparisons in quicksort and quickselect," *36th International Colloquium on Automata, Languages, and Programming*, 2009, 750–763.
5. L. Comtet. *Advanced Combinatorics*, Reidel, Dordrecht, 1974.
6. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, MIT Press, New York, 3rd edition, 2009.
7. S. Dasgupta, C. Papadimitriou, and U. Vazirani. *Algorithms*, McGraw-Hill, New York, 2008.
8. M. Drmota. *Random Trees: An Interplay Between Combinatorics and Probability*, Springer Wein, New York, 2009.
9. W. Feller. *An Introduction to Probability Theory and Its Applications*, John Wiley, New York, 1957.
10. P. Flajolet and R. Sedgewick. *Analytic Combinatorics*, Cambridge University Press, 2009.
11. G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures in Pascal and C*, 2nd edition, Addison-Wesley, Reading, MA, 1991.
12. R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*, 1st edition, Addison-Wesley, Reading, MA, 1989. Second edition, 1994.
13. D. H. Greene and D. E. Knuth. *Mathematics for the Analysis of Algorithms*, Birkhäuser, Boston, 3rd edition, 1991.
14. P. Henrici. *Applied and Computational Complex Analysis*, 3 volumes, John Wiley, New York, 1974 (volume 1), 1977 (volume 2), 1986 (volume 3).
15. C. A. R. Hoare. "Quicksort," *Computer Journal* **5**, 1962, 10–15.

16. J. Kleinberg and E. Tardos. *Algorithm Design*, Addison-Wesley, Boston, 2005.
17. D. E. Knuth. *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, 1st edition, Addison-Wesley, Reading, MA, 1968. Third edition, 1997.
18. D. E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*, 1st edition, Addison-Wesley, Reading, MA, 1969. Third edition, 1997.
19. D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*, 1st edition, Addison-Wesley, Reading, MA, 1973. Second edition, 1998.
20. D. E. Knuth. *The Art of Computer Programming. Volume 4A: Combinatorial Algorithms, Part 1*, Addison-Wesley, Boston, 2011.
21. D. E. Knuth. "Big omicron and big omega and big theta," *SIGACT News*, April-June 1976, 18–24.
22. D. E. Knuth. "Mathematical analysis of algorithms," *Information Processing 71*, Proceedings of the IFIP Congress, Ljubljana, 1971, 19–27.
23. R. Motwani and P. Raghavan. *Randomized Algorithms*, Cambridge University Press, 1995.
24. F. W. J. Olver, D. W. Lozier, R. F. Boisvert, and C. W. Clark, ed., *NIST Handbook of Mathematical Functions*, Cambridge University Press, 2010. Also accessible as *Digital Library of Mathematical Functions* `http://dlmf.nist.gov`.
25. M. O. Rabin. "Probabilistic algorithms," in *Algorithms and Complexity*, J. F. Traub, ed., Academic Press, New York, 1976, 21–39.
26. R. Sedgewick. *Algorithms (3rd edition) in Java: Parts 1-4: Fundamentals, Data Structures, Sorting, and Searching*, Addison-Wesley, Boston, 2003.
27. R. Sedgewick. *Quicksort*, Garland Publishing, New York, 1980.
28. R. Sedgewick. "Quicksort with equal keys," *SIAM Journal on Computing* **6**, 1977, 240–267.
29. R. Sedgewick. "Implementing quicksort programs," *Communications of the ACM* **21**, 1978, 847–856.
30. R. Sedgewick and K. Wayne. *Algorithms*, 4th edition, Addison-Wesley, Boston, 2011.

31. N. Sloane and S. Plouffe. *The Encyclopedia of Integer Sequences*, Academic Press, San Diego, 1995. Also accessible as *On-Line Encyclopedia of Integer Sequences*, `http://oeis.org`.
32. W. Szpankowski. *Average-Case Analysis of Algorithms on Sequences*, John Wiley and Sons, New York, 2001.
33. E. Tufte. *The Visual Display of Quantitative Information*, Graphics Press, Chesire, CT, 1987.
34. J. S. Vitter and P. Flajolet, "Analysis of algorithms and data structures," in *Handbook of Theoretical Computer Science A: Algorithms and Complexity*, J. van Leeuwen, ed., Elsevier, Amsterdam, 1990, 431–524.
35. E. W. Weisstein, ed., *MathWorld*, `mathworld.wolfram.com`.

This page intentionally left blank

INDEX

Abel's identity, 514
Absolute errors in asymptotics, 165–166
Acyclic graphs, 319
Additive parameters for random trees, 297–301
Aho-Corasick algorithm, 456–457
Alcohol modeling, 326
Algebraic functions, 442–445
Algebraic geometry, 522
Alphabets. *See* Strings; Words
Ambiguity
  in context-free languages, 443
  in regular expressions, 432
Analysis of algorithms, 3, 536
  asymptotic approximation, 27–29
  average-case analysis, 16–18
  distributions, 30–33
  linear probing, 512–513
  normal approximation, 207–211
  Poisson approximation, 211–214
  process, 13–15
  purpose, 3–6
  quicksort, 18–27
  randomized, 33
  summary, 34–36
  theory, 6–12
Analytic combinatorics, 36, 219–220
  binary trees, 228, 251, 260
  bitstrings, 226
  bytestrings, 478
  Catalan numbers, 228, 251, 260
  coefficient asymptotics, 247–251
  cumulative GF, 372
  derangements, 239–240, 367–368
  formal basis, 220–221
  generalized derangements, 239–240, 251
  generating function coefficient asymptotics, 247–253
  increasing subsequences, 380
  inversions, 386
  involutions, 369
  labelled trees, 341
  labelled objects, 229–240
  linear probing, 516–517
  parameters, 244–246
  permutations, 234–236, 369
  runs, 375–378
  summary, 253–254
  surjections, 492–493
  symbolic methods for parameters, 241–246
  tables, 254, 341, 383
  transfer theorems, 225, 233, 242, 249–250
  trees, 263
  unlabelled objects, 221–229
  unlabelled trees, 341
  words, 478
  2-ordered permutations, 443–445
Ancestor nodes in binary trees, 259
Approximations
  asymptotic. *See* Asymptotic approximations
  models for cost estimates, 15
Arbitrary patterns in strings, 428–431
Arithmetic expressions, 278–280
Arrangements

maximal occupancy, 496
minimal occupancy, 498
permutations, 355–357
Arrays
associative, 281
sorting. *See* Sorts
Assembly language instructions, 20–21
Associative arrays, 281
Asymptotic analysis
coefficient asymptotics, 113, 247–253, 324, 334
Darboux-Polya method, 326
Euler-Maclaurin summation. *See* Euler-Maclaurin summation
Laplace method, 153, 203–207, 369, 380
linear recurrences. *See* Linear recurrences
Stirling's formula. *See* Stirling's formula
Asymptotic approximations, 27–29
bivariate, 187–202
Euler-Maclaurin summation, 179–186
expansions. *See* Asymptotic expansions
exponentially small terms, 156–157
finite sums, 176–178
normal examples, 207–211
notation, 153–159
overview, 151–153
Poisson approximation, 211–214
summary, 214–215
Asymptotic expansions, 28
absolute errors, 165–166
definitions, 160–162
nonconvergence, 164
relative errors, 166–167
special numbers, 167–168
Stirling's formula, 164–165
Taylor, 162–163
Asymptotic notations
o, 153–157
O, 6–7, 153–157, 169–175
Ω, 6–7
Θ, 6–7
$\sim$, 153–157
Asymptotic notations, 6–7, 169–175
Asymptotic scales, 160
Asymptotic series, 160
Atoms in combinatorial classes, 221–223
Autocorrelation of strings, 428–430
Automata
finite-state, 416, 437–440, 456–457
and regular expressions, 433
and string searching, 437–440
trie-based finite-state, 456–457
Average. *See* Expected value
Average-case analysis, 16–18
AVL (Adel'son-Vel'skii and Landis) trees, 336–339

B-trees, 336–337
Bach's theorem on quadratic maps, 522
Balanced trees, 284, 336
Ballot problem, 268, 314, 445–447
Balls-and-urns model
occupancy distributions, 474, 501–510
Poisson approximation, 198–199
and word properties, 476–485
Batcher's odd-even merge, 208–209
Bell curve, 153, 168, 194–195
Bell numbers, 493
Bernoulli distribution. *See* Binomial distributions
Bernoulli numbers (B_N)

asymptotics, 168
definition, 140, 142–143
in summations, 179–182
values, 181

Bernoulli polynomials ($B_m(x)$)
definition, 143–144
in summations, 179–180

Bernoulli trials
binomial coefficients in, 142
bitstrings, 421
words, 473

BGF. *See* Bivariate generating functions (BGF)

Bijections
cycles and left-to-right minima in permutations, 359
inversion tables and permutations, 359
permutations and HOTs, 362
permutations and sets of cycles, 237, 402–403
tries and sets of strings, 448–452

Binary nodes in heap-ordered trees, 381

Binary number properties in divide-and-conquer recurrences, 75–77

Binary search, 72–75

Binary search trees, 257
additive parameters, 298–300
combinatorial constructions, 373–375
construction costs, 293–296
definition, 282
frequency, 363–364
heap-ordered, 361–365
insertion program, 283–286
leaves, 300–301
overview, 281
path length, 288–290, 293–297
permutations, 361
and quicksort, 294–295
search costs, 295–296
search program, 282

Binary trees
combinatorial equivalences, 264–272
counting, 123–124
definition, 123, 258–259, 321
enumerating, 260, 263
forests, 261–263
general, 261–262
generating functions, 125, 302–303, 441–442
height, 302–309
Lagrange inversion, 313
leaves, 244–246, 300–301
overview, 257–258
parenthesis systems, 265–267
path lengths, 272–276, 287–291
rotation correspondence, 264–265
table, 124
traversal representation, 267
and tries, 448–452
unlabelled objects, 222–223, 228

Binomial asymptotic expansion, 162

Binomial coefficients
asymptotics, 168, 194, 197
definition, 112, 140
special functions, 142

Binomial convolution, 99–100

Binomial distributions, 127–128
asymptotics, 168, 193–195
bivariate generating functions, 133–135
and hashing, 480
normal approximation, 195–198
occupancy problems, 501–505
Poisson approximation, 198–202, 474

probability generating functions, 131–132
strings, 415, 421
tails, 196–197
words, 473–474
Binomial theorem, 48, 111–112, 125
Binomial transforms, 115–116
Biology, computational, 16
Birthday problem, 187, 485–490, 509, 527
Bitstrings
combinatorial properties, 420–426
definition, 415
symbolic methods for parameters, 243
unlabelled objects, 222–223, 226–227
Bivariate asymptotics, 187
binomial distributions, 193–195
Ramanujan distributions, 187–193
Bivariate generating functions (BGF)
binomial distributions, 133–135
bitstrings, 243
Catalan trees, 287–292, 294
definition, 132–133
expansions, 134–138
exponential, 242
leaves, 245
ordinary, 241–242
permutations, 243–244, 373
quicksort distribution, 138–139
Bootstrapping method, 61, 67, 69, 175
Bounding tails in asymptotic approximations, 176–177
BST. *See* Binary search tree

Caching, 494
Carry propagation, 79, 426
Cartesian products, 224, 228
Catalan distributions, 287–288
Catalan models and trees
AVL trees, 338
binary. *See* Binary Catalan trees
general, 292–293, 323
random trees, 280, 287–291, 297–301
Catalan numbers ($T_N = G_{N+1}$)
asymptotics, 165–167, 172–173, 185
and binary trees, 125, 260–261, 263
definition, 140
expansions, 168
forests, 261, 263
generating functions for, 117, 141, 251
history, 269–270
planar subdivisions, 269
Catalan sums, 207–211
Cayley function ($C(z) = ze^{C(z)}$), 527–528
Cayley trees
enumerating, 329–331
exponential generating functions, 527–528
labelled classes, 341
Central limit theorem, 386
CFG (context-free grammars), 441–447
CGF. *See* Cumulative generating functions (CGF)
Change of variables method for recurrences, 61–64
Changing a dollar, 126–127
Characteristic polynomial of recurrences, 55, 106
Characters (letters). *See* Strings; Words
Chebyshev inequality, 32, 508

Child nodes in binary trees, 259
Chomsky and Schützenberger's theorem, 432, 442
Clustering in hashing, 510–513
Coalesced hashing, 509
Coefficient asymptotics
  generating functions, 113, 247–253
  and trees, 324, 334
Coefficient notation ($[z^n]f(z)$), 97
Coefficients in ordinary generating functions, 92
Coin flipping, 421, 457
Collisions
  hashing, 474, 486–488, 494, 509, 512
  occupancy, 495
Combinatorial constructions, 219, 224
  binary trees, 228, 251, 260
  bitstrings, 226, 420–426
  bytestrings, 478
  context-free grammars, 441–443
  cumulative generating function, 372
  derangements, 239–240, 367–368
  formal basis, 220–221
  generalized derangements, 239–240, 251
  increasing subsequences, 380
  inversions, 386
  involutions, 369
  labelled cycles, 527
  labelled objects, 229–240
  labelled trees, 341
  linear probing, 516–517
  multiset, 325
  for parameters, 241–246
  permutations, 234–236, 369
  regular expressions, 433
  rooted unordered trees, 318–320, 322–323
  runs, 375–378
  sets of cycles, 235–236
  summary, 253–254
  surjections, 492–493
  symbolic methods for parameters, 241–246
  tables, 254, 341, 383
  transfer theorems, 225, 233, 242, 249–250
  trees, 263
  unlabelled objects, 221–229
  unlabelled trees, 341
  words, 478
  2-ordered permutations, 443–445
Combinatorial construction operations
  Cartesian product, 223
  cycle, 232–233
  disjoint union, 223
  largest, 373, 395, 485
  last, 373, 395, 485
  min-rooting, 363
  sequence, 223, 232–233
  set, 232
  star product, 231
Combinatorial classes, 221. *See* Labelled classes, Unlabelled classes
Combinatorics, analytic. *See* Analytic combinatorics
Comparison-based sorting, 345
Complex analysis, 215
  generating functions, 113, 145
  rooted unordered trees, 326
  t-restricted trees, 335
Complex roots in linear recurrences, 107
Complexity of sorting, 11–12

Composition in asymptotic series, 174
Compositional functional equations, 118
Computational complexity, 5–13, 85
Computer algebra, 443–445
Connected components, mapping, 522–532
Connected graphs, 319
Construction costs in binary search trees, 293–296
Constructions. *See* Combinatorial constructions
Context-free grammars, 441–447
Continuant polynomials, 60
Continued fractions, 52, 60, 63–64
Convergence, 52
  asymptotic expansion, 164
  ordinary generating functions, 92
  quadratic, 52–53
  radius, 248–250
  simple, 52
  slow, 53–54
Convolutions
  binary trees, 125
  binomial, 99–100
  ordinary generating functions, 95–96
  Vandermonde, 114
Costs
  algorithms, 14–15
  binary search trees, 293–296
  bivariate generating functions, 133, 135–136
  cumulated, 17, 135–137
Counting sequence, 221, 223
Counting with generating functions, 123–128
Coupon collector problem, 488–495
Cryptanalysis, sorting in, 16
Cumulated costs, 17, 135–137
Cumulative analysis
  average-case analysis, 17–18
  string searching, 419–420
  trees, 287
  words, 503–505
Cumulative generating functions (CGF), 135, 137
  combinatorial constructions, 373–375
  increasing subsequences, 379–384
  left-to-right minima, 395–396
  peaks and valleys, 380–384
  permutation properties, 372–384
  random Catalan trees, 291
  runs and rises, 375–379
Cumulative population count function, 76
Cumulative ruler function, 76
Cycle detection in mapping, 532–534
Cycle distribution, 402–403
Cycle leaders, 349
Cycles in mapping, 522–534
Cycles in permutations
  definition, 229–232, 348
  in situ, 401–405
  length, 366–368
  longest and shortest, 409–410
  singleton, 369, 403–405
  structure, 358
  symbolic methods for parameters, 243–244

Darboux-Polya method, 326
Data compression with LZW, 466–467
Decompositions of permutations, 375
Derangements
  asymptotics, 176
  generating functions, 250–251, 370

minimal occupancy, 498
in permutations, 238–240, 348, 367
Descendant nodes in binary trees, 259
Dictionaries, tries for, 455
Dictionary problem, 281
Difference equations, 42, 50, 86
Differential equations
binary search trees, 299
Eulerian numbers, 376
generating functions, 117
heap-ordered trees, 363
increasing subsequences, 378–379
involutions, 370
maxima, 396
median-of-three quicksort, 120–123
quicksort, 109
Digital searching. *See* Tries
Dirichlet generating functions (DGF), 144–146
Discrete sums from recurrence, 48–49, 129
Disjoint union operations, 224
Distributed algorithms, 464
Distributed leader election, 457–458
Distributional analysis, 17
Distributions, 30–33
binomial. *See* Binomial distributions
Catalan, 287–288
normal, 153, 168, 194–195
occupancy, 198, 501–510
Poisson, 153, 168, 202, 405
quicksort, 30–32
Ramanujan, 187–193, 407
uniform discrete, 130–131
Divergent asymptotics series, 164–165
Divide-and-conquer recurrences, 70–72
algorithm, 8
binary number properties, 75–77
binary searches, 72–75
functions, 81–84
general, 80–85
non-binary methods, 78–80
periodicity, 71–72, 82
sequences, 84–85
theorems, 81–85
Division in asymptotic series, 172
Divisor function, 145–146
Dollar, changing a, 126–127
Double falls in permutations, 350–351
Double hashing, 511–512
Double rises in permutations, 350–351
Dynamic processes, 485–486

Elementary bounds in binary trees, 273–275
Elimination. *See* Gröbner basis
Empirical complexity in quicksort, 23
Empty combinatorial class, 221
Empty urns
in hashing, 510
image cardinality, 519
occupancy distribution, 503–505
Encoding tries, 454–455
End recursion removal, 309
Enumerating
binary trees, 260, 263
forests, 263
generating functions, 331
labelled trees, 327–331
pattern occurrences, 419–420
permutations, 366–371
rooted trees, 322, 325–327

surjections, 492–493
t-ary trees, 333–334
Equal length cycles in permutations, 366–367
Error terms in asymptotic expansions, 160
Euler and Segner on Catalan numbers, 269–270
Euler equation, 121
Euler-Maclaurin constants, 183–185
Euler-Maclaurin summation, 27, 153, 176
Bernoulli polynomials, 144
and Catalan sums, 209
discrete form, 183–186
general form, 179–182
and Laplace method, 204–205
overview, 179
and tree height, 307
Eulerian numbers, 376–379, 384
Euler's constant, 28
Exp-log transformation, 173, 188
Expansions
asymptotic. *See* Asymptotic expansions
generating functions, 111–114, 134–138
Expectation of discrete variables, 129–132
Exponential asymptotic expansion, 162
Exponential generating functions (EGF)
bivariate, 242
cumulative, 372
definition, 97
mapping, 527–528
operations, 99–101
permutation involutions, 369–371
symbolic methods for labelled classes, 233–234
table, 98–99
Exponential sequence, 111
Exponentially small terms
asymptotic approximations, 156–157
Ramanujan Q-function, 190, 204–205
Expressions
evaluation, 278–280
register allocation, 62, 280, 309
regular, 432–436, 440
External nodes
binary trees, 123, 258–259
tries, 448–456, 459
External path length for binary trees, 272–273
Extremal parameters for permutations, 406–410

Faà di Bruno's formula, 116
Factorials
asymptotics, 168
definition, 140
Factorization of integers, 532–536
Falls in permutations, 350–351
Fibonacci numbers (FN)
asymptotics, 168
definition, 44
generating functions, 103–104, 114–115, 140
golden ratio, 58
recurrences, 57–59, 64
and strings, 424
Fibonacci polynomials, 305
Find operations, union-find, 316
Finite asymptotic expansions, 161
Finite function. *See* Mapping
Finite-state automata (FSA)

description, 416
and string searching, 437–440
trie-based, 456–457
Finite sums in asymptotic approximations, 176–178
First constructions
cumulative generating functions, 373
occupancy problems, 485
First-order recurrences, 48–51
Floyd's cycle detection algorithm in mapping, 532–533
Foata's correspondence in permutations, 349, 358–359, 402
Footnote, 518
Forests
definition, 261–262
enumerating, 263
labelled trees, 330
Lagrange inversion, 314–315
parenthesis systems, 265
unordered, 315
Formal languages, 224
definitions, 441
and generating functions, 467
regular expressions, 432–433
Formal objects, 146
Formal power series, 92
Fractals, 71, 77, 86
Fractional part (x)
binary searches, 73
divide-and-conquer methods, 82
Euler-Maclaurin summation, 179
tries, 460
Free trees, 318–321, 323, 327–328
Frequency of instruction execution, 7, 20
Frequency of letters
table, 497–499
in words, 473
Fringe analysis, 51
FSA. *See* Finite-state automata (FSA)
Full tables in hashing, 510–511
Functional equations
binary Catalan trees, 287–291
binary search trees, 294, 303
binary trees, 125
context-free grammars, 442–444
expectations for trees, 310–311
generating functions, 117–119
in situ permutation, 405
labelled trees, 329–331
radix-exchange sort, 213
rooted unordered trees, 324
tries, 213
Functional inverse of Lagrange inversion, 312–313
Fundamental correspondence. *See* Foata's correspondence

Gambler's ruin
lattice paths, 268
regular expressions, 435–436
sequence of operations, 446
Gamma function, 186
General trees. *See* Trees
Generalized derangements, 239–240, 250–251
Generalized Fibonacci numbers and strings, 424
Generalized harmonic numbers ($H_N^{(2)}$), 96, 186
Generating functions (GF), 43, 91
bivariate. *See* Bivariate generating functions (BGF)
for Catalan trees, 302–303
coefficient asymptotics, 247–253
counting with, 123–128
cumulative. *See* Cumulative generating functions (CGF)

Dirichlet, 144–146
expansion, 111–114
exponential. *See* Exponential generating functions (EGF)
functional equations, 117–119
mapping, 527–531
ordinary. *See* Ordinary generating functions (OGF)
probability. *See* Probability generating functions (PGF)
recurrences, 101–110, 146
regular expression, 433–435
special functions, 141–146
summary, 146–147
transformations, 114–116
Geometric asymptotic expansion, 162
Geometric sequence, 111
GF. *See* Generating functions (GF)
Golden ratio ($\phi = (1 + \sqrt{5})/2$), 58
Grammars, context-free, 441–447
Graphs, 532
definitions, 318–320
permutations, 358
2-regular, 252
Gröbner basis algorithms, 442–445

Harmonic numbers, 21
approximating, 27–28
asymptotics, 168, 183–186
definition, 140
generalized, 96, 186
ordinary generating functions, 95–96
in permutations, 396
Hash functions, 474
Hashing algorithms, 473
birthday problem, 485–488
coalesced, 509
collisions, 474, 486–488, 494, 509, 512
coupon collector problem, 488–495
empty urns, 503–505, 510
linear probing, 509–518
longest list, 500
open addressing, 509–518
separate chaining, 474–476, 505–509
uniform hashing, 511–512
Heap-ordered trees (HOT)
construction, 375
node types, 380–384
permutations, 362–365
Height
expectations for trees, 310–312
in binary trees, 302–303
in binary search trees, 308–309
in general trees, 304–307
in random walk, 435–436
stack height, 308–309
Height-restricted trees, 336–340
Hierarchy of trees, 321–325
High-order linear recurrences, 104
Higher-order recurrences, 55–60
Homogeneous recurrences, 47
Horizontal expansion of BGFs, 134–136
Horse kicks in Prussian Army, 199
HOT. *See* Heap-ordered trees (HOT)
Huffman encoding, 455
Hydrocarbon modeling, 326

Image cardinality, 519–522
Implementation, analysis for, 6
In situ permutations, 401–405
Increasing subsequences of permutations, 351–352, 379–384
Infix expressions, 267
Information retrieval, 473
Inorder traversal of trees, 277
Input

models, 16, 33
random, 16–17
Insertion into binary search trees, 283–286
Insertion sort, 384–388
In situ permutation (rearrangement), 401–402
Integer factorization, 532–536
Integer partitions, 248
Integrals in asymptotic approximations, 177–178
Integration factor in differential equations, 109
Internal nodes
binary trees, 123, 259, 301
tries, 449–450, 459–462
Internal path length for binary trees, 272–274
Inversions
bubble sorts, 406
distributions, 386–388
Lagrange. *See* Lagrange inversion
permutations, 347, 350, 384–388, 391
tables, 347, 359, 394, 407–408
Involutions
minimal occupancy, 498
in permutations, 350, 369–371
Isomorphism of trees, 324
Iterations
functional equations, 118
in recurrences, 48, 63–64, 81

K-forests of binary trees, 314
Keys
binary search trees, 293
hashes, 474–476
search, 281
sort, 24, 355
Kleene's theorem, 433
Knuth, Donald
analysis of algorithms, 5, 512–513
hashing algorithms, 473
Knuth-Morris-Pratt algorithm (KMP), 420, 437–440, 456
Kraft equality, 275
Kruskal's algorithm, 320

Labelled cycle construction, 526
Labelled combinatorial classes, 229–240
Cayley trees, 329–331
derangements, 239–240, 367–368
generalized derangements, 239–240, 251
increasing subsequences, 380
cycles, 230–231, 527
trees, 327–331, 341
permutations, 234–236, 369
sets of cycles, 235–236
surjections, 492–493
unordered labelled trees, 329–331
urns, 229–231
words, 478
Labelled objects, 97, 229–240
Lagrange inversion theorem, 113, 312–313
binary trees, 313–315
labelled trees, 330–331
mappings, 528
t-ary trees, 333
ternary trees, 313–314
Lambert series, 145
Languages, 224
context-free grammars, 441–447
definitions, 441
and generating functions, 467
regular expressions, 432–436
strings. *See* Strings
words. *See* Words

Laplace method
  increasing subsequences, 380
  involutions, 369
  for sums, 153, 203–207
Laplace transform, 101
Largest constructions
  permutations, 373
  occupancy problems, 485
Last constructions
  permutations, 373, 395
  occupancy problems, 485
Lattice paths
  ballot problem, 445
  gambler's ruin, 268–269
  permutations, 390–392
Lattice representation for permutations, 360
Leader election, 464
Leaves
  binary search trees, 300–301
  binary trees, 244–246, 259, 261, 273
  heap-ordered trees, 382
Left-to-right maxima and minima in permutations, 348–349, 393–398
Lempel-Ziv-Welch (LZW) data compression, 466–467
Letters (characters). *See* Strings; Words
Level (of a node in a tree), 273
Level order traversal, 272, 278
L'Hôpital's rule, 158
Limiting distributions, 30–31
Linear functional equations, 117
Linear probing in hashing, 509–518
Linear recurrences
  asymptotics, 157–159
  constant coefficients, 55–56
  generating functions, 102, 104–108
  scaling, 46–47
Linear recurrences in applications
  fringe analysis, 51
  tree height, 305
Linked lists in hashing, 474–475, 500
Lists in hashing, 474–475, 500
Logarithmic asymptotic expansion, 162
Longest cycles in permutations, 409–410
Longest lists in hashing, 500
Longest runs in strings, 426–427
Lower bounds
  in theory of algorithms, 4, 12
  divide-and-conquer recurrences, 80, 85
  notation, 7
  for sorting, 11
  tree height, 302

M-ary strings, 415
Machine-independent algorithms, 15
Mappings, 474
  connected components, 522–532
  cycles in, 522–534
  definition, 519
  generating functions, 527–531
  image cardinality, 519–522
  path length, 522–527
  random, 519–522, 535–537
  and random number generators, 520–522
  summary, 536–538
  and trees, 523–531
Maxima in permutations, 348–349, 393–398
Maximal cycle lengths in permutations, 368
Maximal occupancy in words, 496–500

Maximum inversion table entry, 407–408
Means
  and probability generating functions, 129–132
  unnormalized, 135
Median-of-three quicksort, 25–26
  ordinary generating functions for, 120–123
  recurrences, 66
Mellin transform, 462
Mergesort algorithm, 7–11
  program, 9–10
  recurrences, 9–10, 43, 70–71, 73–74
  theorem, 74–75
Middle square generator method, 521
Minima in permutations, 348–349, 393–398
Minimal cycle lengths in permutations, 367–368
Minimal occupancy of words, 498–499
Minimal spanning trees, 320
Models
  balls-and-urns. *See* Balls-and-urns model
  Catalan. *See* Catalan models and trees
  costs, 15
  inputs, 16, 33
  random map, 531–532, 535–537
  random permutation, 345–346, 511
  random string, 415, 419–422
  random trie, 457–458
Moments of distributions, 17
  and probability generating functions, 130
  vertical computation, 136–138
Motzkin numbers, 334
Multiple roots in linear recurrences, 107–108
Multiple search patterns, 455–456
Multiplication in asymptotic series, 171–172
Multiset construction, 325
Multiset operations, 228
Multiway tries, 465

Natural numbers, 222–223
Neutral class ($\mathcal{E}$), 221
Neutral object (ϵ), 221
Newton series, 145
Newton's algorithm, 52–53
Newton's theorem, 111–112, 125
Nodes
  binary trees, 123, 258–259
  heap-ordered trees, 380–384
  rooted unordered trees, 322–323, 327–328
  tries, 448–456, 459–462
Nonconvergence in asymptotic series, 164
Nonlinear first-order recurrences, 52–54
Nonlinear functional equations, 117
Nonplane trees, 321
Nonterminal symbols, 441–447
Nonvoid trie nodes, 449–456
Normal approximation
  and analysis of algorithms, 207–211
  binomial distribution, 195–198, 474
  and hashing, 502–505
Normal distribution, 153, 168, 194–195
Notation of asymptotic approximations, 153–159

Number representations, 71–72, 86

o-notation (o), 153–159
O-notation (O), 6–7, 153–159, 169–175
Occupancy distributions, 198, 501–510
Occupancy problems, 474, 478–484, 495–500. *See also* Hashing algorithms; Words
Occurrences of string patterns, 416–420
Odd-even merge, 208–209
Omega-notation (Ω), 6–7
Open addressing hashing, 509–518
Ordered trees
  enumerating, 328–329
  heap-ordered. *See* Heap-ordered trees (HOT)
  hierarchy, 321
  labelled, 315, 327–328
  nodes, 322–323
Ordinary bivariate generating functions (OBGF), 241–242
Ordinary generating functions (OGF), 92
  birthday problem, 489–490
  context-free grammars, 442–443
  linear recurrences, 104–105
  median-of-three quicksort, 120–123
  operations, 95–97
  quicksort recurrences, 109–110
  table, 93–94
  unlabelled objects, 222–223, 225
Oriented trees, 321–322
Oscillation, 70–75, 82, 213, 340, 426, 462–464

Pachinko machine, 510
Page references (caching), 494
Paradox, birthday, 485–487, 509
Parameters
  additive, 297–301
  permutations, 406–410
  symbolic methods for, 241–246
Parent links in rooted unordered trees, 317
Parent nodes in binary trees, 259
Parenthesis systems for trees, 265–267
Parse trees of expressions, 278
Partial fractions, 103, 113
Partial mappings, 531
Partial sums, 95
Partitioning, 19–20, 23–24, 120–123, 139, 295, 454
Path length
  binary search trees, 293–297
  binary trees, 257–258, 272–276
  mapping, 522–527
  Catalan trees, 287–293
  table, 310
  tries, 459–462
Paths
  graphs, 319
  lattice, 268–269
  permutations, 390–392
Patricia tries, 454
Pattern-matching. *See* String searches
Patterns
  arbitrary, 428–431
  autocorrelation, 428–430
  multiple, 455–456
  occurrences, 416–420
Peaks in permutations, 350–351, 362, 380–384
Periodicities
  binary numbers, 70–75
  complex roots, 107

divide-and-conquer, 71–72, 82
mergesort, 70–75
tries, 460–464
Permutations
algorithms, 355–358
basic properties, 352–354
binary search trees, 284, 361
cumulative generating functions, 372–384
cycles. *See* Cycles in permutations
decompositions, 375
enumerating, 366–371
extremal parameters, 406–410
Foata's correspondence, 349
heap-ordered trees, 361–365
in situ, 401–405
increasing subsequences, 351–352, 379–384
inversion tables, 347, 359, 394
inversions in, 347, 350, 384–388, 407–408
labelled objects, 229–231, 234–235
lattice representation, 360
left-to-right minima, 348–349, 393–398
local properties, 382–384
overview, 345–346
peaks and valleys, 350–351, 362, 380–384
random, 23–24, 357–359
rearrangements, 347, 355–358, 401
representation, 358–365
rises and falls, 350–351
runs, 350
selection sorts, 397–400
shellsort, 389–393
summary, 410–411
symbolic methods for parameters, 243–244
table of properties, 383
two-line representation, 237
2-ordered, 208, 389–393, 443–444
Perturbation method for recurrences, 61, 68–69
PGF. *See* Probability generating functions (PGF)
Planar subdivisions, 269–270
Plane trees, 321
Poincaré series, 161
Poisson approximation
analysis of algorithms, 153
binomial distribution, 198–202, 474
and hashing, 502–505
Poisson distribution, 405
analysis of algorithms, 211–214
asymptotics, 168
binomial distribution, 201–202, 474
image cardinality, 519
Poisson law, 199
Poles of recurrences, 157–158
Pollard rho method, 522, 532–536
Polya, Darboux-Polya method, 326
Polygon triangulation, 269–271
Polynomials
Bernoulli, 143–144, 179–180
in context-free grammars, 442–444
Fibonacci, 305
Population count function, 76
Postfix tree traversal, 266–267, 277–279
Power series, 92
Prefix codes, 454
Prefix-free property, 449
Prefix tree traversal, 266–267, 277–278
Prefixes for strings, 419
Preservation of randomness, 27
Priority queues, 358, 362

Probabilistic algorithm, 33
Probability distributions. *See* Distributions
Probability generating functions (PGF)
  binary search trees, 296–297
  binomial, 131–132
  birthday problem, 489–490
  bivariate, 132–140
  mean and variance, 129–130
  and permutations, 386, 395
  uniform discrete distribution, 130–131
Probes
  in hashing, 476
  linear probing, 509–518
Prodinger's algorithm, 464
Product
  Cartesian (unlabelled), 224, 228
  Star (labelled), 231–235
Profiles for binary trees, 273
Program vs. algorithm, 13–14
Prussian Army, horse kicks in, 199
Pushdown stacks, 277, 308, 446

Q-function. *See* Ramanujan Q-function
Quad trees, 333
Quadratic convergence, 52–53
Quadratic mapping, 535
Quadratic random number generators, 521–522
Quadratic recurrences, 62
Queues, priority, 358, 362
Quicksort
  algorithm analysis, 18–27
  asymptotics table, 161
  and binary search trees, 294–295
  bivariate generating functions, 138–139
  compares in, 29
  distributions, 30–32
  empirical complexity, 23
  median-of-three, 25–26, 66, 120–123
  ordinary generating functions for, 109–110
  partitioning, 19–20, 23–24
  probability generating function, 131–132
  radix-exchange, 26–27, 211–213, 454, 459–460, 463
  recurrences, 21–22, 43, 66, 109–110
  subarrays, 25
  variance, 138–139

Radius of convergence bounds, 248–250
Radix-exchange sorts, 26–27
  analysis, 211–213
  and tries, 454, 459–460, 463
Ramanujan distributions (P, Q, R)
  bivariate asymptotics, 187–193
  maximum inversion tables, 407
Ramanujan-Knuth Q-function, 153
Ramanujan Q-function
  and birthday problem, 487
  LaPlace method for, 204–207
  and mapping, 527, 529
Ramanujan R-distribution, 191–193
Random bitstrings, 26
Random input, 16–17
Random mappings, 519–522, 531–532, 535–537
Random number generators, 520–522, 533–535
Random permutations, 23–24, 345–346, 357–359, 511
Random strings

alphabets, 431, 465
binomial distributions, 131
bitstrings, 420
leader election, 464
regular expressions, 432
Random trees
additive parameters, 297–301
binary search tree, 293–295
analysis of algorithms, 275–276
Catalan models, 280, 287–291
path length, 311–312
Random trie models, 457–458
Random variables, 129–132
Random walks, 435–436
Random words, 474, 478
Randomization in leader election, 464
Randomized algorithms, 33
Randomness preservation, 27
Rational functions, 104, 157
generating function coefficients, 247–248
and regular expression, 433
and runs in strings, 423
Rearrangement of permutations, 347, 355–358, 401
Records
in permutations, 348, 355–356
priority queues, 358
sorting, 24, 387, 397, 407
Recurrences, 18
asymptotics, 157–159
basic properties, 43–47
bootstrapping, 67
calculations, 45–46
change of variables method, 61–64
classification, 44–45
divide-and-conquer. *See* Divide-and-conquer recurrences
Fibonacci numbers, 57–59, 64
first-order, 48–51
fringe analysis, 51
generating functions, 101–110, 146
higher-order, 55–60
iteration, 81
linear. *See* Linear recurrences
linear constant coefficient, 55–56
median-of-three quicksort, 26
mergesort, 9–10, 43, 70–71, 73–74
nonlinear first-order, 52–54
overview, 41–43
perturbation, 61, 68–69
quadratic, 62
quicksort, 21–22, 43, 66
radix-exchange sort, 26–27
repertoire, 61, 65–66
scaling, 46–47
summary, 86–87
tree height, 303–305
Recursion, 18, 257, 295
binary trees, 123–126, 220, 228, 257–260, 273–275
binary search trees, 282–283, 361
context-free grammars, 443
divide-and-conquer, 80
distributed leader election, 457
expression evaluation, 278–279
forests, 261
heap-ordered trees, 362–364
mergesort, 7–9, 75–80
parenthesis systems, 265
quad trees, 333
quicksort, 19–21
radix-exchange sort, 454
and recurrences, 41, 45–46
rooted trees, 323
t-ary trees, 333
triangulated N-gons, 269–270
tree algorithms, 277–278
tree properties, 273–274, 290, 291, 297–312

trees, 261, 340
tries, 449–451
Register allocation, 62, 280, 309
Regular expressions, 432–436
and automata, 433, 440
gambler's ruin, 435–436
and generating function, 433–435
Relabelling objects, 231
Relative errors in asymptotics, 166–167
Repertoire method in recurrences, 61, 65–66
Representation of permutations, 358–365
Reversion in asymptotic series, 175
Rewriting rules, 442
Rho length, mapping, 522–527
Rho method, Pollard, 522, 532–536
Riemann sum, 179, 182
Riemann zeta function, 145
Right-left string searching, 466
Rises in permutations, 350–351, 375–379
Root nodes in binary trees, 259, 261
Rooted unordered trees, 315
definition, 315–316
enumerating, 325–327
graphs, 318–320
hierarchy, 321–325
Kruskal's algorithm, 320
nodes, 322–323, 327–328
overview, 315
representing, 324
sample application, 316–318
Rotation correspondence between trees, 264–265, 309
Ruler function, 76
Running time, 7
Runs
in permutations, 350, 375–379
in strings, 420–426, 434

Saddle point method, 499
Scales, asymptotic, 160
Scaling recurrences, 46
Search costs in binary search trees, 293, 295–296
Search problem, 281
Searching algorithms. *See* Binary search; Binary search trees; Hashing algorithms; String searches; Tries
Seeds for random number generators, 521
Selection sort, 397–400
Sentinels, 416–417
Separate chaining hashing algorithms, 474–476, 505–509
Sequence construction, 224, 228
Sequences, 95–97
ternary trees, 314
rooted unordered trees, 325
free trees, 327
ordered labelled trees, 329
unordered labelled trees, 330
runs and rises in permutations, 375
Stirling cycle numbers, 397
maximum inversion table entry, 406–407
3-words tieh restrictions, 495
Series, asymptotic, 160
Set construction, 228
Sets of cycles, 235–237, 527
Sets of strings, 416, 448–452
Shellsort, 389–393
Shifting recurrences, 46
Shortest cycles in permutations, 409–410
Sim-notation ($\sim$), 153–159
Simple convergence, 52

Simple paths in graphs, 319
Singleton cycles in permutations, 369, 403–405
Singularities of generating functions, 113
Singularity analysis, 252, 335
Size in combinatorial classes, 221, 223
Slow convergence, 53–54
Smallest construction, 373
Sorting
  algorithms, 6–12
  bubble, 406–407
  comparison-based, 345
  complexity, 11–12
  insertion, 384–388
  mergesort. *See* Mergesort algorithm
  permutations, 355–356, 397–400
  quicksort. *See* Quicksort
  radix-exchange, 26–27, 211–213, 454, 459–460, 463
  selection, 397–400
Spanning trees of graphs, 319
Special number sequences, 139
  asymptotics, 167–168
  Bernoulli numbers, 142–143
  Bernoulli polynomials, 143–144
  binomial coefficients, 142
  Dirichlet generating functions, 144–146
  harmonic numbers, 21
  overview, 141–142
  Stirling numbers, 142
  tables, 140
Stacks, 277
  ballot problem, 446–447
  height, 308–309
Standard deviation, 17
  bivariate generating functions, 138
  distributions, 31–32
  probability generating functions, 129–130
Star operations
  labelled classes, 231–232
  on languages, 432
Stirling numbers, overview, 142
Stirling numbers of the first kind ($\left[{n \atop k}\right]$), 140
  asymptotics, 168
  counting cycles, 402–403
  counting minima/maxima, 396–398
Stirling numbers of the second kind ($\left\{{n \atop k}\right\}$), 140
  asymptotics, 168
  and coupon collector, 491
  subset numbers, 491
  surjections, 492–493
Stirling's constant ($\sigma = \sqrt{2\pi}$), 183–184, 210
Stirling's formula
  asymptotic expansion, 164–165
  asymptotics, 173, 185
  and Laplace method, 207
  table, 166
  and trees, 168
String searches
  KMP algorithm, 437–440
  right-left, 466
  and tries, 416–420, 448, 455–458
Strings
  arbitrary patterns, 428–431
  autocorrelation, 428–430
  larger alphabets, 465–467
  overview, 415–416
  runs, 420–426, 434
  sets of. *See* Languages; Tries
  summary, 467–468
  words. *See* Words
Subset numbers, 491

Subtrees, 123, 258–259, 261
Successful searches, 295, 476, 508–509, 511, 515–518
Suffix tries, 455, 459
Summation factors, 50, 59
Sums
  asymptotic approximations, 176–178
  Euler-Maclaurin. *See* Euler-Maclaurin summation
  Laplace method for, 203–207
Superleaves, 228
Surjections
  enumerating, 492–493, 495
  image cardinality, 519
  maximal occupancy, 499
  minimal occupancy, 497–498
Symbol tables
  binary search trees, 281, 466
  hashing, 474–476
  rooted unordered trees, 317
  tries, 448, 453, 459, 465
Symbolic method, 219, 221, 229. *See* Combinatorial constructions.
t-ary trees, 331–333
  definition, 333
  enumerating, 333–334
t-restricted trees, 334–336
Tails
  asymptotic approximations, 176–177
  binomial distribution, 196–197
  Laplace method, 203–205
  in mapping, 523–524
Taylor expansions
  asymptotic, 162–163
  table, 162
Taylor theorem, 111–113, 220, 247
Telescoping recurrences, 86
Terminal symbols, 441–442
Ternary trees, 313–314
Ternary tries, 465–466
Text searching. *See* String searching
Theory of algorithms, 4–12
Theta notation (Θ), 6–7
Time complexity of sorting, 10–11
Toll function, 297–298
Transfer theorems, 219–220
  bitstrings, 228
  derangements, 251
  involutions, 369
  labelled objects, 232, 240
  Lagrange inversion, 312
  radius of convergence, 249–250
  Taylor's theorem, 247
  universal, 443
  unlabelled objects, 228–229
Transformations for generating functions, 114–116
Transitions
  finite-state automata, 437–439, 456
  state transition tables, 438–440
Traversal of trees
  algorithms, 277–278
  binary trees, 267, 278–280
  labelled trees, 328
  parenthesis system, 265
  preorder and postorder, 266–267
  stacks for, 308
Trees
  algorithm examples, 277–280
  average path length, 287–293
  binary. *See* Binary search trees; Binary trees
  Catalan. *See* Catalan models and trees
  combinatorial equivalences, 264–272
  enumerating, 322, 331
  expectations for trees, 310–312

expression evaluation, 278–280
heap-ordered trees, 362–365, 375, 380–384
height. *See* Height of trees
height-restricted, 336–340
hierarchy, 321–325
isomorphism, 324
labelled, 327–331
Lagrange inversion, 312–315
and mapping, 523–531
nomenclature, 321
ordered. *See* Ordered trees
parenthesis systems, 265–267
properties, 272–276
random. *See* Random trees
in random walk, 435–436
rooted unordered. *See* Rooted unordered trees
rotation correspondence, 264–265
summary, 340–341
t-ary, 331–334
t-restricted, 334–336
traversal. *See* Traversal of trees
unlabelled, 322, 328–329
unrooted, 318–321, 323, 327–328
Triangulation of polygons, 269–271
Tries
combinatorial properties, 459–464
context-free languages, 416
definitions, 449–451
encoding, 454–455
finite-state automata, 456–457
vs. hashing, 476
multiway, 465
nodes, 448–456, 459–462
overview, 448–449
path length and size, 459–462
Patricia, 454
pattern matching, 455–458
radix-exchange sorts, 211–214, 454, 459–460, 463
random, 457–458
string searching, 416–420
suffix, 455
sum, 211–214
summary, 467–468
ternary, 465–466
Trigonometric asymptotic expansion, 162
Two-line representation of permutations, 237
2-ordered permutations, 208, 389–393, 443–444
2-regular graphs, 252
2-3 trees, 336
fringe analysis, 51
functional equations, 118
2-3-4 trees, 336
2D-trees, 270

Unambiguous languages, 441–447
Unambiguous regular expressions, 432–433
Uniform discrete distributions, 130–131
Uniform hashing, 511–512
Union-find problem, 316, 324
Union operations, 224, 228
Unlabelled combinatorial classes, 221–229
AVL trees, 332, 336, 338
B-trees, 332, 336, 338
binary trees, 228, 251, 260
bitstrings, 226, 420–426
bytestrings, 478
context-free grammars, 441–443
Motzkin trees, 341
ordered trees, 328–329

rooted unordered trees, 318–320, 322–323
t-ary trees, 333–334, 341
t-restricted trees, 334–336, 341
trees, 263, 341
unrooted trees, 318–321, 323, 327–328
2-3 trees, 338
Unlabelled objects, 97, 221–229
Unnormalized mean (cumulated cost), 17, 135–137
Unordered trees
labelled, 329–331
rooted. *See* Rooted unordered trees
Unrooted trees, 318–321, 323, 327–328
Unsuccessful searches, 295, 476, 505, 508, 511–515, 517–518
Upper bounds
analysis of algorithms, 4
cycle length in permutations, 368
divide-and-conquer recurrences, 80, 85
notation, 7, 154
and performance, 12
sorts, 10–11
tree height, 302
Urns, 474
labelled objects, 229–230
occupancy distributions, 474, 501–510
Poisson approximation, 198–199
and word properties, 476–485

Valleys in permutations, 350–351, 362, 380–384
Vandermonde's convolution, 114
Variance, 31–33
binary search trees, 294, 296, 311
bivariate generating functions, 136–138
coupon collector problem, 490–491
inversions in permutations, 386
left-to-right minima in permutations, 394–395
occupancy distribution, 504–505
Poisson distribution, 202
probability generating functions, 129–130
runs in permutations, 378
singleton cycles in permutations, 404
selection sort, 399
quicksort, 138–139
Variations in unlabelled objects, 226–227
Vertical expansion of bivariate generating functions, 136–138
Void nodes in tries, 449–456

Words
balls-and-urns model, 476–485
birthday problem, 485–488
caching algorithms, 494
coupon collector problem, 488–495
frequency restrictions, 497–499
hashing algorithms, 474–476
and mappings. *See* Mappings
maximal occupancy, 496–500
minimal occupancy, 498–499
occupancy distributions, 501–509
occupancy problems, 478–484
overview, 473–474
Worst-case analysis, 78

Zeta function of Riemann, 145