



AN INTRODUCTION
TO THE

ANALYSIS OF ALGORITHMS

SECOND EDITION

ROBERT SEDGEWICK
PHILIPPE FLAJOLET

www.it-ebooks.info

AN INTRODUCTION TO THE ANALYSIS OF ALGORITHMS

Second Edition

This page intentionally left blank

AN INTRODUCTION TO THE ANALYSIS OF ALGORITHMS

Second Edition

Robert Sedgewick
Princeton University

Philippe Flajolet
INRIA Rocquencourt

◆ Addison-Wesley

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

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

This page intentionally left blank

P R E F A C E

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
Paris, December 2012*

Ph. F. and R. S.
R. S.

This page intentionally left blank

NOTE ON THE SECOND EDITION

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

Sadly, I am writing from a distant part of the world to pay my respects to my longtime friend and colleague, Philippe Flajolet. I am very sorry not to be there in person, but I know that there will be many opportunities to honor Philippe in the future and expect to be fully and personally involved on these occasions.

Brilliant, creative, inquisitive, and indefatigable, yet generous and charming, Philippe's approach to life was contagious. He changed many lives, including my own. As our research papers led to a survey paper, then to a monograph, then to a book, then to two books, then to a life's work, I learned, as many students and collaborators around the world have learned, that working with Philippe was based on a genuine and heartfelt camaraderie. We met and worked together in caf  s, 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

T A B L E O F C O N T E N T S

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

TABLE OF CONTENTS

xvii

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

This page intentionally left blank

NOTATION

$\lfloor x \rfloor$	<i>floor function</i> largest integer less than or equal to x
$\lceil x \rceil$	<i>ceiling function</i> smallest integer greater than or equal to x
$\{x\}$	<i>fractional part</i> $x - \lfloor x \rfloor$
$\lg N$	<i>binary logarithm</i> $\log_2 N$
$\ln N$	<i>natural logarithm</i> $\log_e N$
$\binom{n}{k}$	<i>binomial coefficient</i> number of ways to choose k out of n items
$\left[\begin{matrix} n \\ k \end{matrix} \right]$	<i>Stirling number of the first kind</i> number of permutations of n elements that have k cycles
$\left\{ \begin{matrix} n \\ k \end{matrix} \right\}$	<i>Stirling number of the second kind</i> number of ways to partition n elements into k nonempty subsets
ϕ	<i>golden ratio</i> $(1 + \sqrt{5})/2 = 1.61803 \dots$
γ	<i>Euler's constant</i> .57721 ...
σ	<i>Stirling's constant</i> $\sqrt{2\pi} = 2.50662 \dots$

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 $N \rightarrow \infty$.

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

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

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] = INFNY; c[hi - mid] = INFNY;
    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

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

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

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

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

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

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

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

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 $N\lg N$ 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 $N\lg N$.*

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 $O(N\lg N)$.”

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

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

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

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

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

to sort an array of N randomly ordered distinct elements.

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

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

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

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

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 j th largest (which occurs with probability $1/N$ for each $1 \leq j \leq N$), 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

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

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

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

Now rearrange terms to get a simple recurrence

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

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

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

Iterating, we are left with the sum

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

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 $H_N \approx \ln N + .57721 \dots$. 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

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

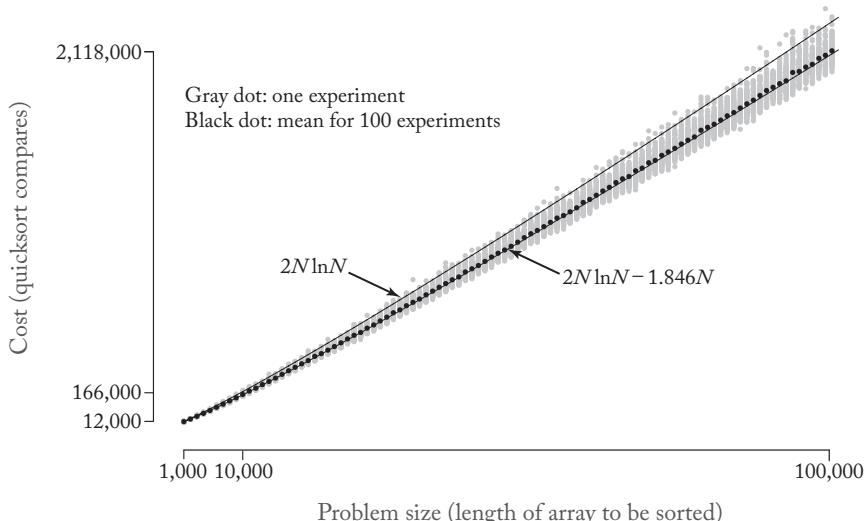


Figure 1.1 Quicksort compare counts: empirical and analytic

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

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

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

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

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 k th 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

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

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 predictions 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

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

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

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

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

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

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

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

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

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

Table 1.1 Average number of compares used by quicksort

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

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

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

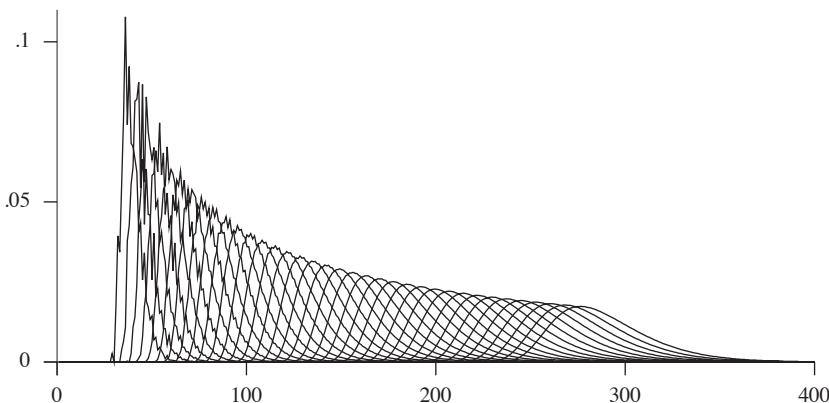


Figure 1.2 Distributions for compares in quicksort, $15 \leq N \leq 50$

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

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

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

The *variance* is defined to be

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

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

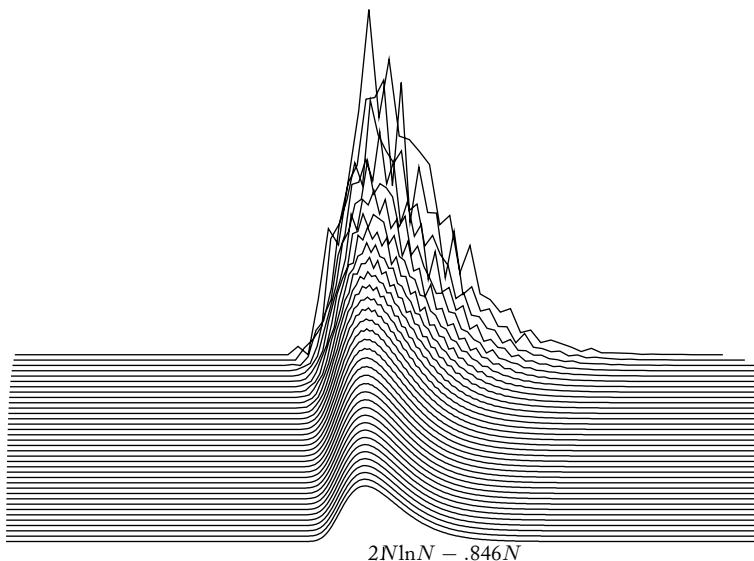


Figure 1.3 Distributions for compares in quicksort, $15 \leq N \leq 50$
(scaled and translated to center and separate curves)

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

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

(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.

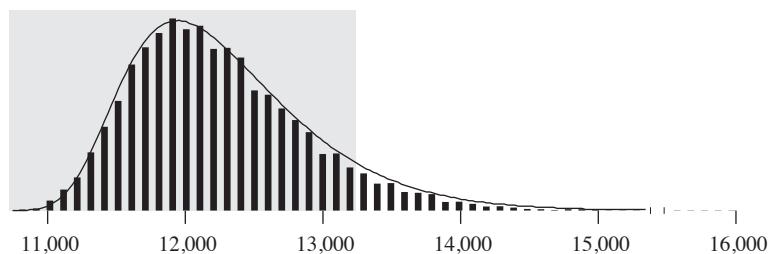


Figure 1.4 Empirical histogram for quicksort compare counts (10,000 trials with $N=1000$)

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's 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. PLLOUFFE. *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, Cheshire, 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

CHAPTER TWO

RECURRENCE RELATIONS

THE algorithms that we are interested in analyzing normally can be expressed as recursive or iterative procedures, which means that, typically, we can express the cost of solving a particular problem in terms of the cost of solving smaller problems. The most elementary approach to this situation mathematically is to use recurrence relations, as we saw in the quicksort and mergesort analyses in the previous chapter. This represents a way to realize a direct mapping from a recursive representation of a program to a recursive representation of a function describing its properties. There are several other ways to do so, though the same recursive decomposition is at the heart of the matter. As we will see in Chapter 3, this is also the basis for the application of generating function methods in the analysis of algorithms.

The development of a recurrence relation describing the performance of an algorithm is already a significant step forward in the analysis, since the recurrence itself carries a great deal of information. Specific properties of the algorithm as related to the input model are encapsulated in a relatively simple mathematical expression. Many algorithms may not be amenable to such a simple description; fortunately, many of our most important algorithms can be rather simply expressed in a recursive formulation, and their analysis leads to recurrences, either describing the average case or bounding the worst-case performance. This point is illustrated in Chapter 1 and in many examples in Chapters 6 through 9. In this chapter, we concentrate on fundamental mathematical properties of various recurrences without regard to their origin or derivation. We will encounter many of the types of recurrences seen in this chapter in the context of the study of particular algorithms, and we do revisit the recurrences discussed in Chapter 1, but our focus for the moment is on the recurrences themselves.

First, we examine some basic properties of recurrences and the ways in which they are classified. Then, we examine exact solutions to “first-order” recurrences, where a function of n is expressed in terms of the function evaluated at $n - 1$. We also look at exact solutions to higher-order linear recurrences with constant coefficients. Next, we look at a variety of other types

of recurrences and examine some methods for deriving approximate solutions to some nonlinear recurrences and recurrences with nonconstant coefficients. Following that, we examine solutions to a class of recurrence of particular importance in the analysis of algorithms: the “divide-and-conquer” class of recurrence. This includes the derivation of and exact solution to the merge-sort recurrence, which involves a connection with the binary representation of integers. We conclude the chapter by looking at general results that apply to the analysis of a broad class of divide-and-conquer algorithms.

All the recurrences that we have considered so far admit to exact solutions. Such recurrences arise frequently in the analysis of algorithms, especially when we use recurrences to do precise counting of discrete quantities. But exact answers may involve irrelevant detail: for example, working with an exact answer like $(2^n - (-1)^n)/3$ as opposed to the approximate answer $2^n/3$ is probably not worth the trouble. In this case, the $(-1)^n$ term serves to make the answer an integer and is negligible by comparison to 2^n ; on the other hand, we would not want to ignore the $(-1)^n$ term in an exact answer like $2^n(1 + (-1)^n)$. It is necessary to avoid the temptations of being overly careless in trading accuracy for simplicity and of being overzealous in trading simplicity for accuracy. We are interested in obtaining approximate expressions that are *both* simple and accurate (even when exact solutions may be available). In addition, we frequently encounter recurrences for which exact solutions simply are not available, but we can estimate the rate of growth of the solution, and, in many cases, derive accurate asymptotic estimates.

Recurrence relations are also commonly called *difference equations* because they may be expressed in terms of the discrete difference operator $\nabla f_n \equiv f_n - f_{n-1}$. They are the discrete analog of ordinary differential equations. Techniques for solving differential equations are relevant because similar techniques often can be used to solve analogous recurrences. In some cases, as we will see in the next chapter, there is an explicit correspondence that allows one to derive the solution to a recurrence from the solution to a differential equation.

There is a large literature on the properties of recurrences because they also arise directly in many areas of applied mathematics. For example, iterative numerical algorithms such as Newton’s method directly lead to recurrences, as described in detail in, for example, Bender and Orszag [3].

Our purpose in this chapter is to survey the types of recurrences that commonly arise in the analysis of algorithms and some elementary techniques

for deriving solutions. We can deal with many of these recurrence relations in a rigorous and systematic way using *generating functions*, as discussed in detail in the next chapter. We will also consider tools for developing asymptotic approximations in some detail in Chapter 4. In Chapters 6 through 9 we will encounter many different examples of recurrences that describe properties of basic algorithms.

Once we begin to study advanced tools in detail, we will see that recurrences may not necessarily be the most natural mathematical tool for the analysis of algorithms. They can introduce complications in the analysis that can be avoided by working at a higher level, using symbolic methods to derive relationships among generating functions, then using direct analysis on the generating functions. This theme is introduced in Chapter 5 and treated in detail in [12]. In many cases, it turns out that the simplest and most direct path to solution is to *avoid* recurrences. We point this out not to discourage the study of recurrences, which can be quite fruitful for many applications, but to assure the reader that advanced tools perhaps can provide simple solutions to problems that seem to lead to overly complicated recurrences.

In short, recurrences arise directly in natural approaches to algorithm analysis, and can provide easy solutions to many important problems. Because of our later emphasis on generating function techniques, we give only a brief introduction to techniques that have been developed in the literature for solving recurrences. More information about solving recurrences may be found in standard references, including [3], [4], [6], [14], [15], [16], [21], and [22].

2.1 Basic Properties. In Chapter 1, we encountered the following three recurrences when analyzing quicksort and mergesort:

$$C_N = \left(1 + \frac{1}{N}\right)C_{N-1} + 2 \quad \text{for } N > 1 \text{ with } C_1 = 2. \quad (1)$$

$$C_N = C_{\lfloor N/2 \rfloor} + C_{\lceil N/2 \rceil} + N \quad \text{for } N > 1 \text{ with } C_1 = 0. \quad (2)$$

$$C_N = N + 1 + \frac{1}{N} \sum_{1 \leq j \leq N} (C_{j-1} + C_{N-j}) \quad \text{for } N > 0 \text{ with } C_0 = 0. \quad (3)$$

Each of the equations presents special problems. We solved (1) by multiplying both sides by an appropriate factor; we developed an approximate solution

to (2) by solving for the special case $N = 2^n$, then proving a solution for general N by induction; and we transformed (3) to (1) by subtracting it from the same equation for $N - 1$.

Such ad hoc techniques are perhaps representative of the “bag of tricks” approach often required for the solution of recurrences, but the few tricks just mentioned do not apply, for example, to many recurrences that commonly arise, including perhaps the best-known linear recurrence

$$F_n = F_{n-1} + F_{n-2} \quad \text{for } n > 1 \text{ with } F_0 = 0 \text{ and } F_1 = 1,$$

which defines the Fibonacci sequence $\{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots\}$. Fibonacci numbers are well studied and actually arise explicitly in the design and analysis of a number of important algorithms. We consider a number of techniques for solving these and other recurrences in this chapter, and we consider other applicable systematic approaches in the next and later chapters.

Recurrences are classified by the way in which terms are combined, the nature of the coefficients involved, and the number and nature of previous

recurrence type	typical example
first-order	
linear	$a_n = na_{n-1} - 1$
nonlinear	$a_n = 1/(1 + a_{n-1})$
second-order	
linear	$a_n = a_{n-1} + 2a_{n-2}$
nonlinear	$a_n = a_{n-1}a_{n-2} + \sqrt{a_{n-2}}$
variable coefficients	$a_n = na_{n-1} + (n - 1)a_{n-2} + 1$
t th order	$a_n = f(a_{n-1}, a_{n-2}, \dots, a_{n-t})$
full-history	$a_n = n + a_{n-1} + a_{n-2} \dots + a_1$
divide-and-conquer	$a_n = a_{\lfloor n/2 \rfloor} + a_{\lceil n/2 \rceil} + n$

Table 2.1 Classification of recurrences

terms used. Table 2.1 lists some of the recurrences that we will be considering, along with representative examples.

Calculating values. Normally, a recurrence provides an efficient way to calculate the quantity in question. In particular, the very first step in attacking any recurrence is to use it to compute small values in order to get a feeling for how they are growing. This can be done by hand for small values, or it is easy to implement a program to compute larger values.

For example, Program 2.1 will compute the exact values for the average number of comparisons for quicksort for all N less than or equal to maxN , corresponding to the recurrence (3) (see Table 1.1). This program uses an array of size maxN to save previously computed values. The temptation to use a purely recursive program based directly on the recurrence should be avoided: computing C_N by computing all the values $C_{N-1}, C_{N-2}, \dots, C_1$ recursively would be extremely inefficient because many, many values would be unnecessarily recomputed.

We could avoid delving too deeply into the mathematics of the situation if something like Program 2.1 would suffice. We assume that succinct mathematical solutions are more desirable—indeed, one might view the analysis itself as a process that can make Program 2.1 more efficient! At any rate, such “solutions” can be used, for example, to validate analyses. At the other extreme on this continuum would be a brute-force (usually impractical) method for computing the average running time of a program by running it for all possible inputs.

```
C[0] = 0.0;
for (int N = 1; N <= maxN; N++)
{
    C[N] = N+1.0;
    for (int k = 0; k < N; k++)
        C[N] += (C[k] + C[N-1-k])/N;
}
```

Program 2.1 Calculating values (quicksort recurrence)

Exercise 2.1 Write recursive and nonrecursive programs to compute values for the Fibonacci recurrence and try to use each to compute F_{20} . Explain the behavior of each program in this case.

Exercise 2.2 How many arithmetic operations are used by Program 2.1, as a function of N_{\max} ?

Exercise 2.3 Write a recursive program to compute values using the recurrence (1) directly. How does the number of arithmetic operations used by this program compare with Program 2.1 (see the previous exercise)?

Exercise 2.4 Estimate how many operations would be required by both recursive and nonrecursive programs to compute values using the recurrences (2) and (3).

Exercise 2.5 Write a program to compare quicksort, its median-of-three variant, and radix-exchange sort, calculating values from the recurrences given in Chapter 1. For quicksort, check values against the known solution; for the others, make conjectures about properties of the solution.

Scaling and shifting. An essential property of recurrences is that they depend on their initial values: changing the initial condition in the linear recurrence

$$a_n = f(a_{n-1}) \quad \text{for } n > 0 \text{ with } a_0 = 1$$

from $a_0 = 1$ to $a_0 = 2$ will change the value of a_n for all n (if $f(0) = 0$, the value will be doubled). The “initial” value can be anywhere: if we have

$$b_n = f(b_{n-1}) \quad \text{for } n > t \text{ with } b_t = 1$$

then we must have $b_n = a_{n-t}$. Changing the initial values is referred to as *scaling* the recurrence; moving the initial values is referred to as *shifting* it. Our initial values are most often directly implied from a problem, but we often use scaling or shifting to simplify the path to the solution. Rather than state the most general form of the solution to a recurrence, we solve a natural form and presume that the solution can be scaled or shifted as appropriate.

Linearity. Linear recurrences with more than one initial value can be “scaled” by changing initial values independently and combining solutions. If $f(x, y)$ is a linear function with $f(0, 0) = 0$, then the solution to

$$a_n = f(a_{n-1}, a_{n-2}) \quad \text{for } n > 1$$

(a function of the initial values a_0 and a_1) is a_0 times the solution to

$$u_n = f(u_{n-1}, u_{n-2}) \quad \text{for } n > 1 \text{ with } u_0 = 1 \text{ and } u_1 = 0$$

plus a_1 times the solution to

$$v_n = f(v_{n-1}, v_{n-2}) \quad \text{for } n > 1 \text{ with } v_0 = 0 \text{ and } v_1 = 1.$$

The condition $f(0, 0) = 0$ makes the recurrence *homogeneous*: if there is a constant term in f , then that, as well as the initial values, has to be taken into account. This generalizes in a straightforward way to develop a general solution for any homogeneous linear t th-order recurrence (for any set of initial values) as a linear combination of t particular solutions. We used this procedure in Chapter 1 to solve the recurrence describing the number of exchanges taken by quicksort, in terms of the recurrences describing the number of comparisons and the number of stages.

Exercise 2.6 Solve the recurrence

$$a_n = a_{n-1} + a_{n-2} \quad \text{for } n > 1 \text{ with } a_0 = p \text{ and } a_1 = q,$$

expressing your answer in terms of the Fibonacci numbers.

Exercise 2.7 Solve the inhomogeneous recurrence

$$a_n = a_{n-1} + a_{n-2} + r \quad \text{for } n > 1 \text{ with } a_0 = p \text{ and } a_1 = q,$$

expressing your answer in terms of the Fibonacci numbers.

Exercise 2.8 For f linear, express the solution to the recurrence

$$a_n = f(a_{n-1}, a_{n-2}) \quad \text{for } n > 1$$

in terms of a_0 , a_1 , $f(0, 0)$ and the solutions to $a_n = f(a_{n-1}, a_{n-2}) - f(0, 0)$ for $a_1 = 1$, $a_0 = 0$ and $a_0 = 1$, $a_1 = 0$.

2.2 First-Order Recurrences. Perhaps the simplest type of recurrence reduces immediately to a product. The recurrence

$$a_n = x_n a_{n-1} \quad \text{for } n > 0 \text{ with } a_0 = 1$$

is equivalent to

$$a_n = \prod_{1 \leq k \leq n} x_k.$$

Thus, if $x_n = n!$ then $a_n = n!$, and if $x_n = 2^n$ then $a_n = 2^n$, and so on.

This transformation is a simple example of *iteration*: apply the recurrence to itself until only constants and initial values are left, then simplify. Iteration also applies directly to the next simplest type of recurrence, much

geometric series

$$\sum_{0 \leq k < n} x^k = \frac{1 - x^n}{1 - x}$$

arithmetic series

$$\sum_{0 \leq k < n} k = \frac{n(n-1)}{2} = \binom{n}{2}$$

binomial coefficients

$$\sum_{0 \leq k \leq n} \binom{k}{m} = \binom{n+1}{m+1}$$

binomial theorem

$$\sum_{0 \leq k \leq n} \binom{n}{k} x^k y^{n-k} = (x+y)^n$$

harmonic numbers

$$\sum_{1 \leq k \leq n} \frac{1}{k} = H_n$$

sum of harmonic numbers

$$\sum_{1 \leq k < n} H_k = nH_n - n$$

Vandermonde convolution

$$\sum_{0 \leq k \leq n} \binom{n}{k} \binom{m}{t-k} = \binom{n+m}{t}$$

Table 2.2 Elementary discrete sums

more commonly encountered, which reduces immediately to a sum:

$$a_n = a_{n-1} + y_n \quad \text{for } n > 0 \text{ with } a_0 = 0$$

is equivalent to

$$a_n = \sum_{1 \leq k \leq n} y_k.$$

Thus, if $y_n = 1$ then $a_n = n$, and if $y_n = n - 1$ then $a_n = n(n - 1)/2$, and so on.

Table 2.2 gives a number of commonly encountered discrete sums. A much more comprehensive list may be found in a standard reference such as Graham, Knuth, and Patashnik [14] or Riordan [23].

Exercise 2.9 Solve the recurrence

$$a_n = \frac{n}{n+2} a_{n-1} \quad \text{for } n > 0 \text{ with } a_0 = 1.$$

Exercise 2.10 Solve the recurrence

$$a_n = a_{n-1} + (-1)^n n \quad \text{for } n > 0 \text{ with } a_0 = 1.$$

If we have a recurrence that is not quite so simple, we can often simplify by multiplying both sides of the recurrence by an appropriate factor. We have seen examples of this already in Chapter 1. For example, we solved (1) by dividing both sides by $N + 1$, giving a simple recurrence in $C_N/(N + 1)$ that transformed directly into a sum when iterated.

Exercise 2.11 Solve the recurrence

$$na_n = (n - 2)a_{n-1} + 2 \quad \text{for } n > 1 \text{ with } a_1 = 1.$$

(Hint: Multiply both sides by $n - 1$.)

Exercise 2.12 Solve the recurrence

$$a_n = 2a_{n-1} + 1 \quad \text{for } n > 1 \text{ with } a_1 = 1.$$

(Hint: Divide both sides by 2^n .)

Solving recurrence relations (difference equations) in this way is analogous to solving differential equations by multiplying by an integrating factor and then integrating. The factor used for recurrence relations is sometimes called a *summation factor*. Proper choice of a summation factor makes it possible to solve many of the recurrences that arise in practice. For example, an exact solution to the recurrence describing the average number of comparisons used in median-of-three quicksort was developed by Knuth using such techniques [18] (see also [24]).

Theorem 2.1 (First-order linear recurrences). The recurrence

$$a_n = x_n a_{n-1} + y_n \quad \text{for } n > 0 \text{ with } a_0 = 0$$

has the explicit solution

$$a_n = y_n + \sum_{1 \leq j < n} y_j x_{j+1} x_{j+2} \dots x_n.$$

Proof. Dividing both sides by $x_n x_{n-1} \dots x_1$ and iterating, we have

$$\begin{aligned} a_n &= x_n x_{n-1} \dots x_1 \sum_{1 \leq j \leq n} \frac{y_j}{x_j x_{j-1} \dots x_1} \\ &= y_n + \sum_{1 \leq j < n} y_j x_{j+1} x_{j+2} \dots x_n. \end{aligned}$$

The same result can be derived by multiplying both sides by $x_{n+1} x_{n+2} \dots$ (provided it converges) and iterating. ■

For example, the proof of Theorem 2.1 says that we should solve the recurrence

$$C_N = \left(1 + \frac{1}{N}\right) C_{N-1} + 2 \quad \text{for } N > 1 \text{ with } C_1 = 2$$

by dividing both sides by

$$\frac{N+1}{N} \frac{N}{N-1} \frac{N-1}{N-2} \dots \frac{3}{2} \frac{2}{1} = N+1,$$

which is precisely what we did in §1.5. Alternatively, the solution

$$2(N+1)(H_{N+1} - 1)$$

follows directly from the explicit form of the solution given in the theorem statement.

Theorem 2.1 is a complete characterization of the transformation from first-order linear recurrences, with constant or nonconstant coefficients, to sums. The problem of solving the recurrence is reduced to the problem of evaluating the sum.

Exercise 2.13 Solve the recurrence

$$a_n = \frac{n}{n+1}a_{n-1} + 1 \quad \text{for } n > 0 \text{ with } a_0 = 1.$$

Exercise 2.14 Write down the solution to

$$a_n = x_n a_{n-1} + y_n \quad \text{for } n > t$$

in terms of the x 's, the y 's, and the initial value a_t .

Exercise 2.15 Solve the recurrence

$$na_n = (n+1)a_{n-1} + 2n \quad \text{for } n > 0 \text{ with } a_0 = 0.$$

Exercise 2.16 Solve the recurrence

$$na_n = (n-4)a_{n-1} + 12nH_n \quad \text{for } n > 4 \text{ with } a_n = 0 \text{ for } n \leq 4.$$

Exercise 2.17 [Yao] (“Fringe analysis of 2–3 trees [25]”) Solve the recurrence

$$A_N = A_{N-1} - \frac{2A_{N-1}}{N} + 2\left(1 - \frac{2A_{N-1}}{N}\right) \quad \text{for } N > 0 \text{ with } A_0 = 0.$$

This recurrence describes the following random process: A set of N elements is collected into “2-nodes” and “3-nodes.” At each step each 2-node is likely to turn into a 3-node with probability $2/N$ and each 3-node is likely to turn into two 2-nodes with probability $3/N$. What is the average number of 2-nodes after N steps?

2.3 Nonlinear First-Order Recurrences. When a recurrence consists of a nonlinear function relating a_n and a_{n-1} , a broad variety of situations arise, and we cannot expect to have a closed-form solution like Theorem 2.1. In this section we consider a number of interesting cases that do admit to solutions.

Simple convergence. One convincing reason to calculate initial values is that many recurrences with a complicated appearance simply converge to a constant. For example, consider the equation

$$a_n = 1/(1 + a_{n-1}) \quad \text{for } n > 0 \text{ with } a_0 = 1.$$

This is a so-called continued fraction equation, which is discussed in §2.5. By calculating initial values, we can guess that the recurrence converges to a constant:

n	a_n	$ a_n - (\sqrt{5} - 1)/2 $
1	0.5000000000000	0.118033988750
2	0.6666666666667	0.048632677917
3	0.6000000000000	0.018033988750
4	0.6250000000000	0.006966011250
5	0.615384615385	0.002649373365
6	0.619047619048	0.001013630298
7	0.617647058824	0.000386929926
8	0.618181818182	0.000147829432
9	0.617977528090	0.000056460660

Each iteration increases the number of significant digits available by a constant number of digits (about half a digit). This is known as *simple convergence*. If we assume that the recurrence does converge to a constant, we know that the constant must satisfy $\alpha = 1/(1 + \alpha)$, or $1 - \alpha - \alpha^2 = 0$, which leads to the solution $\alpha = (\sqrt{5} - 1)/2 \approx .6180334$.

Exercise 2.18 Define $b_n = a_n - \alpha$ with a_n and α defined as above. Find an approximate formula for b_n when n is large and a_0 is between 0 and 1.

Exercise 2.19 Show that $a_n = \cos(a_{n-1})$ converges when a_0 is between 0 and 1, and compute $\lim_{n \rightarrow \infty} a_n$ to five decimal places.

Quadratic convergence and Newton's method. This well-known iterative method for computing roots of functions can be viewed as a process of calculating an approximate solution to a first-order recurrence (see, for example,

[3]). For example, Newton's method to compute the square root of a positive number β is to iterate the formula

$$a_n = \frac{1}{2} \left(a_{n-1} + \frac{\beta}{a_{n-1}} \right) \quad \text{for } n > 0 \text{ with } a_0 = 1.$$

Changing variables in this recurrence, we can see why the method is so effective. Letting $b_n = a_n - \alpha$, we find by simple algebra that

$$b_n = \frac{1}{2} \frac{b_{n-1}^2 + \beta - \alpha^2}{b_{n-1} + \alpha},$$

so that if $\alpha = \sqrt{\beta}$ we have, roughly, $b_n \approx b_{n-1}^2$. For example, to compute the square root of 2, this iteration gives the following sequence:

n	a_n	$a_n - \sqrt{2}$
1	1.500000000000	0.085786437627
2	1.416666666667	0.002453104294
3	1.414215686275	0.000002123901
4	1.414213562375	0.000000000002
5	1.414213562373	0.000000000000

Each iteration approximately doubles the number of significant digits available. This is a case of so-called quadratic convergence.

Exercise 2.20 Discuss what happens when Newton's method is used to attempt computing $\sqrt{-1}$:

$$a_n = \frac{1}{2} \left(a_{n-1} - \frac{1}{a_{n-1}} \right) \quad \text{for } n > 0 \text{ with } a_0 \neq 0.$$

Slow convergence. Consider the recurrence

$$a_n = a_{n-1}(1 - a_{n-1}) \quad \text{for } n > 0 \text{ with } a_0 = \frac{1}{2}.$$

In §6.10, we will see that similar recurrences play a role in the analysis of the height of “random binary trees.” Since the terms in the recurrence decrease

and are positive, it is not hard to see that $\lim_{n \rightarrow \infty} a_n = 0$. To find the speed of convergence, it is natural to consider $1/a_n$. Substituting, we have

$$\begin{aligned}\frac{1}{a_n} &= \frac{1}{a_{n-1}} \left(\frac{1}{1 - a_{n-1}} \right) \\ &= \frac{1}{a_{n-1}} (1 + a_{n-1} + a_{n-1}^2 + \dots) \\ &> \frac{1}{a_{n-1}} + 1.\end{aligned}$$

This telescopes to give $1/a_n > n$, or $a_n < 1/n$. We have thus found that $a_n = O(1/n)$.

Exercise 2.21 Prove that $a_n = \Theta(1/n)$. Compute initial terms and try to guess a constant c such that a_n is approximated by c/n . Then find a rigorous proof that na_n tends to a constant.

Exercise 2.22 [De Bruijn] Show that the solution to the recurrence

$$a_n = \sin(a_{n-1}) \quad \text{for } n > 0 \text{ with } a_0 = 1$$

satisfies $\lim_{n \rightarrow \infty} a_n = 0$ and $a_n = O(1/\sqrt{n})$. (*Hint:* Consider the change of variable $b_n = 1/a_n$.)

The three cases just considered are particular cases of the form

$$a_n = f(a_{n-1})$$

for some continuous function f . If the a_n converge to a limit α , then necessarily α must be a fixed point of the function, with $\alpha = f(\alpha)$. The three cases above are representative of the general situation: if $0 < |f'(\alpha)| < 1$, the convergence is simple; if $f'(\alpha) = 0$, the convergence is quadratic; and if $|f'(\alpha)| = 1$, the convergence is “slow.”

Exercise 2.23 What happens when $f'(\alpha) > 1$?

Exercise 2.24 State sufficient criteria corresponding to the three cases above for local convergence (when a_0 is sufficiently close to α) and quantify the speed of convergence in terms of $f'(\alpha)$ and $f''(\alpha)$.

2.4 Higher-Order Recurrences. Next, we consider recurrences where the right-hand side of the equation for a_n is a linear combination of a_{n-2} , a_{n-3} , and so on, as well as a_{n-1} , and where the coefficients involved are constants. For a simple example, consider the recurrence

$$a_n = 3a_{n-1} - 2a_{n-2} \quad \text{for } n > 1 \text{ with } a_0 = 0 \text{ and } a_1 = 1.$$

This can be solved by first observing that $a_n - a_{n-1} = 2(a_{n-1} - a_{n-2})$, an elementary recurrence in the quantity $a_n - a_{n-1}$. Iterating this product gives the result $a_n - a_{n-1} = 2^{n-1}$; iterating the sum for this elementary recurrence gives the solution $a_n = 2^n - 1$. We could also solve this recurrence by observing that $a_n - 2a_{n-1} = a_{n-1} - 2a_{n-2}$. These manipulations correspond precisely to factoring the quadratic equation $1 - 3x + 2x^2 = (1 - 2x)(1 - x)$.

Similarly, we can find that the solution to

$$a_n = 5a_{n-1} - 6a_{n-2} \quad \text{for } n > 1 \text{ with } a_0 = 0 \text{ and } a_1 = 1$$

is $a_n = 3^n - 2^n$ by solving elementary recurrences on $a_n - 3a_{n-1}$ or $a_n - 2a_{n-1}$.

Exercise 2.25 Give a recurrence that has the solution $a_n = 4^n - 3^n + 2^n$.

These examples illustrate the general form of the solution, and recurrences of this type can be solved explicitly.

Theorem 2.2 (Linear recurrences with constant coefficients). All solutions to the recurrence

$$a_n = x_1a_{n-1} + x_2a_{n-2} + \dots + x_ta_{n-t} \quad \text{for } n \geq t$$

can be expressed as a linear combination (with coefficients depending on the initial conditions a_0, a_1, \dots, a_{t-1}) of terms of the form $n^j\beta^n$, where β is a root of the “characteristic polynomial”

$$q(z) \equiv z^t - x_1z^{t-1} - x_2z^{t-2} - \dots - x_t$$

and j is such that $0 \leq j < \nu$ if β has multiplicity ν .

Proof. It is natural to look for solutions of the form $a_n = \beta^n$. Substituting, any such solution must satisfy

$$\beta^n = x_1\beta^{n-1} + x_2\beta^{n-2} + \dots + x_t\beta^{n-t} \quad \text{for } n \geq t$$

or, equivalently,

$$\beta^{n-t} q(\beta) = 0.$$

That is, β^n is a solution to the recurrence for any root β of the characteristic polynomial.

Next, suppose that β is a double root of $q(z)$. We want to prove that $n\beta^n$ is a solution to the recurrence as well as β^n . Again, by substitution, we must have

$$n\beta^n = x_1(n-1)\beta^{n-1} + x_2(n-2)\beta^{n-2} + \dots + x_t(n-t)\beta^{n-t} \quad \text{for } n \geq t$$

or, equivalently,

$$\beta^{n-t}((n-t)q(\beta) + \beta q'(\beta)) = 0.$$

This is true, as desired, because $q(\beta) = q'(\beta) = 0$ when β is a double root. Higher multiplicities are treated in a similar manner.

This process provides as many solutions to the recurrence as there are roots of the characteristic polynomial, counting multiplicities. This is the same as the order t of the recurrence. Moreover, these solutions are linearly independent (they have different orders of growth at ∞). Since the solutions of a recurrence of order t form a vector space of dimension t , each solution of our recurrence must be expressible as a linear combination of the particular solutions of the form $n^j \beta^n$. ■

Finding the coefficients. An exact solution to any linear recurrence can be developed from Theorem 2.2 by using the initial values a_0, a_1, \dots, a_{t-1} to create a system of simultaneous equations that can be solved to yield the constants in the linear combination of the terms that comprise the solution. For example, consider the recurrence

$$a_n = 5a_{n-1} - 6a_{n-2} \quad \text{for } n \geq 2 \text{ with } a_0 = 0 \text{ and } a_1 = 1.$$

The characteristic equation is $z^2 - 5z + 6 = (z - 3)(z - 2)$ so

$$a_n = c_0 3^n + c_1 2^n.$$

Matching this formula against the values at $n = 0$ and $n = 1$, we have

$$a_0 = 0 = c_0 + c_1$$

$$a_1 = 1 = 3c_0 + 2c_1.$$

The solution to these simultaneous equations is $c_0 = 1$ and $c_1 = -1$, so $a_n = 3^n - 2^n$.

Degenerate cases. We have given a method for finding an exact solution for any linear recurrence. The process makes explicit the way in which the full solution is determined by the initial conditions. When the coefficients turn out to be zero and/or some roots have the same modulus, the result can be somewhat counterintuitive, though easily understood in this context. For example, consider the recurrence

$$a_n = 2a_{n-1} - a_{n-2} \quad \text{for } n \geq 2 \text{ with } a_0 = 1 \text{ and } a_1 = 2.$$

Since the characteristic equation is $z^2 - 2z + 1 = (z - 1)^2$ (with a single root, 1, of multiplicity 2), the solution is

$$a_n = c_0 1^n + c_1 n 1^n.$$

Applying the initial conditions

$$\begin{aligned} a_0 &= 1 = c_0 \\ a_1 &= 2 = c_0 + c_1 \end{aligned}$$

gives $c_0 = c_1 = 1$, so $a_n = n + 1$. But if the initial conditions were $a_0 = a_1 = 1$, the solution would be $a_n = 1$, meaning constant instead of linear growth. For a more dramatic example, consider the recurrence

$$a_n = 2a_{n-1} + a_{n-2} - 2a_{n-3} \quad \text{for } n > 3.$$

Here the solution is

$$a_n = c_0 1^n + c_1 (-1)^n + c_2 2^n,$$

and various choices of the initial conditions can make the solution constant, exponential in growth, or fluctuating in sign! This example points out that paying attention to details (initial conditions) is quite important when dealing with recurrences.

Fibonacci numbers. We have already mentioned the familiar *Fibonacci sequence* $\{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots\}$ that is defined by the prototypical second-order recurrence

$$F_n = F_{n-1} + F_{n-2} \quad \text{for } n > 1 \text{ with } F_0 = 0 \text{ and } F_1 = 1.$$

Since the roots of $u^2 - u - 1$ are $\phi = (1 + \sqrt{5})/2 = 1.61803 \dots$ and $\hat{\phi} = (1 - \sqrt{5})/2 = -.61803 \dots$, Theorem 2.2 says that the solution is

$$F_N = c_0\phi^N + c_1\hat{\phi}^N$$

for some constants c_0 and c_1 . Applying the initial conditions

$$F_0 = 0 = c_0 + c_1$$

$$F_1 = 1 = c_0\phi + c_1\hat{\phi}$$

yields the solution

$$F_N = \frac{1}{\sqrt{5}}(\phi^N - \hat{\phi}^N).$$

Since ϕ is larger than 1 and $\hat{\phi}$ is smaller than 1 in absolute value, the contribution of the $\hat{\phi}^N$ term in the above expression for F_N is negligible, and it turns out that F_N is always the nearest integer to $\phi^N/\sqrt{5}$. As N gets large, the ratio F_{N+1}/F_N approaches ϕ , which is well known in mathematics, art, architecture, and nature as the *golden ratio*.

While Theorem 2.2 provides a way to develop complete exact solutions to fixed-degree high-order linear recurrences, we will revisit this topic in Chapters 3 and 4 because the advanced tools there provide convenient ways to get useful results in practice. Theorem 3.3 gives an easy way to compute coefficients, and in particular identify those terms that vanish. Moreover, the phenomenon just observed for Fibonacci numbers generalizes: since the terms $n^j \beta^n$ are all exponential, the ones (among those with nonzero coefficient) with largest β will dominate all the others for large n , and among those, the one with largest j will dominate. Generating functions (Theorem 3.3) and asymptotic analysis (Theorem 4.1) provide us with convenient ways to identify the leading term explicitly and evaluate its coefficient for any linear recurrence. This can provide a shortcut to developing a good approximation to the solution in some cases, especially when t is large. For small t , the method described here for getting the exact solution is quite effective.

Exercise 2.26 Explain how to solve an inhomogeneous recurrence of the form

$$a_n = x_1 a_{n-1} + x_2 a_{n-2} + \dots + x_t a_{n-t} + r \quad \text{for } n \geq t.$$

Exercise 2.27 Give initial conditions a_0, a_1 for which the solution to

$$a_n = 5a_{n-1} - 6a_{n-2} \quad \text{for } n > 1$$

is $a_n = 2^n$. Are there initial conditions for which the solution is $a_n = 2^n - 1$?

Exercise 2.28 Give initial conditions a_0 , a_1 , and a_2 for which the growth rate of the solution to

$$a_n = 2a_{n-1} - a_{n-2} + 2a_{n-3} \quad \text{for } n > 2$$

is (i) constant, (ii) exponential, and (iii) fluctuating in sign.

Exercise 2.29 Solve the recurrence

$$a_n = 2a_{n-1} + 4a_{n-2} \quad \text{for } n > 1 \text{ with } a_1 = 2 \text{ and } a_0 = 1.$$

Exercise 2.30 Solve the recurrence

$$a_n = 2a_{n-1} - a_{n-2} \quad \text{for } n > 1 \text{ with } a_0 = 0 \text{ and } a_1 = 1.$$

Solve the same recurrence, but change the initial conditions to $a_0 = a_1 = 1$.

Exercise 2.31 Solve the recurrence

$$a_n = a_{n-1} - a_{n-2} \quad \text{for } n > 1 \text{ with } a_0 = 0 \text{ and } a_1 = 1.$$

Exercise 2.32 Solve the recurrence

$$2a_n = 3a_{n-1} - 3a_{n-2} + a_{n-3} \quad \text{for } n > 2 \text{ with } a_0 = 0, a_1 = 1 \text{ and } a_2 = 2.$$

Exercise 2.33 Find a recurrence describing a sequence for which the order of growth decreases exponentially for odd-numbered terms, but increases exponentially for even-numbered terms.

Exercise 2.34 Give an approximate solution for the “third-order” Fibonacci recurrence

$$F_N^{(3)} = F_{N-1}^{(3)} + F_{N-2}^{(3)} + F_{N-3}^{(3)} \quad \text{for } N > 2 \text{ with } F_0^{(3)} = F_1^{(3)} = 0 \text{ and } F_2^{(3)} = 1.$$

Compare your approximate result for $F_{20}^{(3)}$ with the exact value.

Nonconstant coefficients. If the coefficients are not constants, then more advanced techniques are needed because Theorem 2.2 does not apply. Typically, generating functions (see Chapter 3) or approximation methods (discussed later in this chapter) are called for, but some higher-order problems can be solved with summation factors. For example, the recurrence

$$a_n = na_{n-1} + n(n-1)a_{n-2} \quad \text{for } n > 1 \text{ with } a_1 = 1 \text{ and } a_0 = 0$$

can be solved by simply dividing both sides by $n!$, leaving the Fibonacci recurrence in $a_n/n!$, which shows that $a_n = n!F_n$.

Exercise 2.35 Solve the recurrence

$$n(n-1)a_n = (n-1)a_{n-1} + a_{n-2} \quad \text{for } n > 1 \text{ with } a_1 = 1 \text{ and } a_0 = 1.$$

Symbolic solution. Though no closed form like Theorem 2.2 is available for higher-order recurrences, the result of iterating the general form

$$a_n = s_{n-1}a_{n-1} + t_{n-2}a_{n-2} \quad \text{for } n > 1 \text{ with } a_1 = 1 \text{ and } a_0 = 0$$

has been studied in some detail. For sufficiently large n , we have

$$\begin{aligned} a_2 &= s_1, \\ a_3 &= s_2s_1 + t_1, \\ a_4 &= s_3s_2s_1 + s_3t_1 + t_2s_1, \\ a_5 &= s_4s_3s_2s_1 + s_4s_3t_1 + s_4t_2s_1 + t_3s_2s_1 + t_3t_1, \\ a_6 &= s_5s_4s_3s_2s_1 + s_5s_4s_3t_1 + s_5s_4t_2s_1 + s_5t_3s_2s_1 + s_5t_3t_1 \\ &\quad + t_4s_3s_2s_1 + t_4s_3t_1 + t_4t_2s_1, \end{aligned}$$

and so forth. The number of monomials in the expansion of a_n is exactly F_n , and the expansions have many other properties: they are related to the so-called continuant polynomials, which are themselves closely related to continued fractions (discussed later in this chapter). Details may be found in Graham, Knuth, and Patashnik [14].

Exercise 2.36 Give a simple algorithm to determine whether a given monomial $s_{i_1}s_{i_2}\dots s_{i_p}t_{j_1}t_{j_2}\dots t_{j_q}$ appears in the expansion of a_n . How many such monomials are there?

We argued earlier that for the case of constant coefficients, we are most interested in a derivation of the asymptotic behavior of the leading term because exact solutions, though available, are tedious to use. For the case of nonconstant coefficients, exact solutions are generally not available, so we must be content with approximate solutions for many applications. We now turn to techniques for developing such approximations.

2.5 Methods for Solving Recurrences. Nonlinear recurrences or recurrences with variable coefficients can normally be solved or approximated through one of a variety of approaches. We consider a number of such approaches and examples in this section.

We have been dealing primarily with recurrences that admit to exact solutions. While such problems do arise very frequently in the analysis of algorithms, one certainly can expect to encounter recurrences for which no method for finding an exact solution is known. It is premature to begin treating advanced techniques for working with such recurrences, but we give some guidelines on how to develop accurate approximate solutions and consider several examples.

We consider four general methods: *change of variable*, which involves simplifying a recurrence by recasting it in terms of another variable; *repertoire*, which involves working backward from a given recurrence to find a solution space; *bootstrapping*, which involves developing an approximate solution, then using the recurrence itself to find a more accurate solution, continuing until a sufficiently accurate answer is obtained or no further improvement seems likely; and *perturbation*, which involves studying the effects of transforming a recurrence into a similar, simpler, one with a known solution. The first two of these methods often lead to exact solutions of recurrences; the last two are more typically used to develop approximate solutions.

Change of Variables. Theorem 2.1 actually describes a change of variable: if we change variables to $b_n = a_n/(x_n x_{n-1} \dots x_1)$, then b_n satisfies a simple recurrence that reduces to a sum when iterated. We also used a change of variable in the previous section, and in other places earlier in the chapter. More complicated changes of variable can be used to derive exact solutions to formidable-looking recurrences. For instance, consider the nonlinear second-order recurrence

$$a_n = \sqrt{a_{n-1} a_{n-2}} \quad \text{for } n > 1 \text{ with } a_0 = 1 \text{ and } a_1 = 2.$$

If we take the logarithm of both sides of this equation and make the change of variable $b_n = \lg a_n$, then we find that b_n satisfies

$$b_n = \frac{1}{2}(b_{n-1} + b_{n-2}) \quad \text{for } n > 1 \text{ with } b_0 = 0 \text{ and } b_1 = 1,$$

a linear recurrence with constant coefficients.

Exercise 2.37 Give exact formulae for b_n and a_n .

Exercise 2.38 Solve the recurrence

$$a_n = \sqrt{1 + a_{n-1}^2} \quad \text{for } n > 0 \text{ with } a_0 = 0.$$

Our next example arises in the study of register allocation algorithms [11]:

$$a_n = a_{n-1}^2 - 2 \quad \text{for } n > 0.$$

For $a_0 = 0$ or $a_0 = 2$, the solution is $a_n = 2$ for $n > 1$, and for $a_0 = 1$, the solution is $a_n = -1$ for $n > 1$, but for larger a_0 the dependence on the initial value a_0 is more complicated for this recurrence than for other first-order recurrences that we have seen.

This is a so-called quadratic recurrence, and it is one of the few quadratic recurrences that can be solved explicitly, by change of variables. By setting $a_n = b_n + 1/b_n$, we have the recurrence

$$b_n + \frac{1}{b_n} = b_{n-1}^2 + \frac{1}{b_{n-1}^2} \quad \text{for } n > 0 \text{ with } b_0 + 1/b_0 = a_0.$$

But this implies that we can solve by making $b_n = b_{n-1}^2$, which iterates immediately to the solution

$$b_n = b_0^{2^n}.$$

By the quadratic equation, b_0 is easily calculated from a_0 :

$$b_0 = \frac{1}{2} \left(a_0 \pm \sqrt{a_0^2 - 4} \right).$$

Thus,

$$a_n = \left(\frac{1}{2} \left(a_0 + \sqrt{a_0^2 - 4} \right) \right)^{2^n} + \left(\frac{1}{2} \left(a_0 - \sqrt{a_0^2 - 4} \right) \right)^{2^n}.$$

For $a_0 > 2$, only the larger of the two roots predominates in this expression—the one with the plus sign.

Exercise 2.39 From the above discussion, solve the register allocation recurrence for $a_0 = 3, 4$. Discuss what happens for $a_0 = 3/2$.

Exercise 2.40 Solve the register allocation recurrence for $a_0 = 2 + \epsilon$, where ϵ is an arbitrary fixed positive constant. Give an accurate approximate answer.

Exercise 2.41 Find all values of the parameters α , β , and γ such that $a_n = \alpha a_{n-1}^2 + \beta a_{n-1} + \gamma$ reduces to $b_n = b_{n-1}^2 - 2$ by a linear transformation ($b_n = f(\alpha, \beta, \gamma)a_n + g(\alpha, \beta, \gamma)$). In particular, show that $a_n = a_{n-1}^2 + 1$ does not reduce to this form.

Exercise 2.42 [Melzak] Solve the recurrence

$$a_n = 2a_{n-1}\sqrt{1 - a_{n-1}^2} \quad \text{for } n > 0 \text{ with } a_0 = \frac{1}{2}$$

and with $a_0 = 1/3$. Plot a_6 as a function of a_0 and explain what you observe.

On the one hand, underlying linearity may be difficult to recognize, and finding a change of variable that solves a nonlinear recurrence is no easier than finding a change of variable that allows us to evaluate a definite integral (for example). Indeed, more advanced analysis (iteration theory) may be used to show that most nonlinear recurrences cannot be reduced in this way. On the other hand, a variable change that simplifies a recurrence that arises in practice may not be difficult to find, and a few such changes might lead to a linear form. As illustrated by the register allocation example, such recurrences do arise in the analysis of algorithms.

For another example, consider using change of variables to get an exact solution to a recurrence related to continued fractions.

$$a_n = 1/(1 + a_{n-1}) \quad \text{for } n > 0 \text{ with } a_0 = 1.$$

Iterating this recurrence gives the sequence

$$\begin{aligned} a_0 &= 1 \\ a_1 &= \frac{1}{1+1} = \frac{1}{2} \\ a_2 &= \frac{1}{1+\frac{1}{1+1}} = \frac{1}{1+\frac{1}{2}} = \frac{2}{3} \end{aligned}$$

Continuing, we have

$$a_3 = \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1+1}}}} = \frac{1}{1 + \frac{1}{2}} = \frac{3}{5}$$

$$a_4 = \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1+1}}}} = \frac{1}{1 + \frac{1}{3}} = \frac{5}{8}$$

and so on, which reveals the Fibonacci numbers. The form $a_n = b_{n-1}/b_n$ is certainly suggested: substituting this equation into the recurrence gives

$$\frac{b_{n-1}}{b_n} = 1 / \left(1 + \frac{b_{n-2}}{b_{n-1}}\right) \quad \text{for } n > 1 \text{ with } b_0 = b_1 = 1.$$

Dividing both sides by b_{n-1} gives

$$\frac{1}{b_n} = \frac{1}{b_{n-1} + b_{n-2}} \quad \text{for } n > 1 \text{ with } b_0 = b_1 = 1,$$

which implies that $b_n = F_{n+1}$, the Fibonacci sequence. This argument generalizes to give a way to express general classes of “continued fraction” representations as solutions to recurrences.

Exercise 2.43 Solve the recurrence

$$a_n = \frac{\alpha a_{n-1} + \beta}{\gamma a_{n-1} + \delta} \quad \text{for } n > 0 \text{ with } a_0 = 1.$$

Exercise 2.44 Consider the recurrence

$$a_n = 1/(s_n + t_n a_{n-1}) \quad \text{for } n > 0 \text{ with } a_0 = 1,$$

where $\{s_n\}$ and $\{t_n\}$ are arbitrary sequences. Express a_n as the ratio of two successive terms in a sequence defined by a linear recurrence.

Repertoire. Another path to exact solutions in some cases is the so-called repertoire method, where we use known functions to find a family of solutions similar to the one sought, which can be combined to give the answer. This method primarily applies to linear recurrences, involving the following steps:

- Relax the recurrence by adding an extra functional term.
- Substitute known functions into the recurrence to derive identities similar to the recurrence.
- Take linear combinations of such identities to derive an equation identical to the recurrence.

For example, consider the recurrence

$$a_n = (n - 1)a_{n-1} - na_{n-2} + n - 1 \quad \text{for } n > 1 \text{ with } a_0 = a_1 = 1.$$

We generalize this by introducing a quantity $f(n)$ to the right-hand side, so we want to solve

$$a_n = (n - 1)a_{n-1} - na_{n-2} + f(n)$$

for $n > 1$ and $a_0 = a_1 = 1$ with $f(n) = n - 1$. To do so, we inject various possibilities for a_n and look at the resulting $f(n)$ to get a “repertoire” of recurrences that we can solve (forgetting momentarily about initial conditions). For this example, we arrive at the table

a_n	$a_n - (n - 1)a_{n-1} + na_{n-2}$
1	2
n	$n - 1$
n^2	$n + 1$

The first row in this table says that $a_n = 1$ is a solution with $f(n) = 2$ (and initial conditions $a_0 = 1$ and $a_1 = 1$); the second row says that $a_n = n$ is a solution with $f(n) = n - 1$ (and initial conditions $a_0 = 0$ and $a_1 = 1$); and the third row says that $a_n = n^2$ is a solution with $f(n) = n + 1$ (and initial conditions $a_0 = 0$ and $a_1 = 1$). Now, linear combinations of these also give solutions. Subtracting the first row from the third gives the result that means that $a_n = n^2 - 1$ is a solution with $f(n) = n - 1$ (and initial conditions $a_0 = -1$ and $a_1 = 0$). Now we have two (linearly independent) solutions for $f(n) = n - 1$, which we combine to get the right initial values, yielding the result $a_n = n^2 - n + 1$.

The success of this method depends on being able to find a set of independent solutions, and on properly handling initial conditions. Intuition or knowledge about the form of the solution can be useful in determining the repertoire. The classic example of the use of this method is in the analysis of an equivalence algorithm by Knuth and Schönhage [20].

For the quicksort recurrence, we start with

$$a_n = f(n) + \frac{2}{n} \sum_{1 \leq j \leq n} a_{j-1}$$

for $n > 0$ with $a_0 = 0$. This leads to the following repertoire table:

a_n	$a_n - (2 \sum_{0 \leq j < n} a_j)/n$
1	-1
H_n	$-H_n + 2$
n	1
$\lambda(n+1)$	0
nH_n	$\frac{1}{2}(n-1) + H_n$
$n(n-1)$	$\frac{1}{3}(n^2 - 1) + n - 1$

Thus $2nH_n + 2H_n + \lambda(n+1) + 2$ is a solution with $f(n) = n+1$; resolving the initial value with $\lambda = -2$ gives the solution

$$2(n+1)H_n - 2n,$$

as expected (see Theorem 1.4). The solution depends on the fifth line in the table, which we are obliged to try because we might expect for other reasons that the solution might be $O(n\log n)$. Note that the repertoire table can conveniently also give the solution for other $f(n)$, as might be required by more detailed analysis of the algorithm.

Exercise 2.45 Solve the quicksort recurrence for $f(n) = n^3$.

Exercise 2.46 [Greene and Knuth] Solve the quicksort median-of-three recurrence (see equation (4) in Chapter 1) using the repertoire method. (See [18] or [24] for a direct solution to this recurrence using differencing and summation factors, and see Chapter 3 for a solution using generating functions.)

Bootstrapping. Often we are able to guess the approximate value of the solution to a recurrence. Then, the recurrence itself can be used to place constraints on the estimate that can be used to give a more accurate estimate. Informally, this method involves the following steps:

- Use the recurrence to calculate numerical values.
- Guess the approximate form of the solution.
- Substitute the approximate solution back into the recurrence.
- Prove tighter bounds on the solution, based on the guessed solution and the substitution.

For illustrative purposes, suppose that we apply this method to the Fibonacci recurrence:

$$a_n = a_{n-1} + a_{n-2} \quad \text{for } n > 1 \text{ with } a_0 = 0 \text{ and } a_1 = 1.$$

First, we note that a_n is increasing. Therefore, $a_{n-1} > a_{n-2}$ and $a_n > 2a_{n-2}$. Iterating this inequality implies that $a_n > 2^{n/2}$, so we know that a_n has at least an exponential rate of growth. On the other hand, $a_{n-2} < a_{n-1}$ implies that $a_n < 2a_{n-1}$, or (iterating) $a_n < 2^n$. Thus we have proved upper and lower exponentially growing bounds on a_n and we can feel justified in “guessing” a solution of the form $a_n \sim c_0\alpha^n$, with $\sqrt{2} < \alpha < 2$. From the recurrence, we can conclude that α must satisfy $\alpha^2 - \alpha - 1 = 0$, which leads to ϕ and $\hat{\phi}$. Having determined the value α , we can bootstrap and go back to the recurrence and the initial values to find the appropriate coefficients.

Exercise 2.47 Solve the recurrence

$$a_n = 2/(n + a_{n-1}) \quad \text{for } n > 0 \text{ with } a_0 = 1.$$

Exercise 2.48 Use bootstrapping to show that the number of compares used by median-of-three quicksort is $\alpha N \ln N + O(N)$. Then determine the value of α .

Exercise 2.49 [Greene and Knuth] Use bootstrapping to show that the solution to

$$a_n = \frac{1}{n} \sum_{0 \leq k < n} \frac{a_k}{n - k} \quad \text{for } n > 0 \text{ with } a_0 = 1$$

satisfies $n^2 a_n = O(1)$.

Perturbation. Another path to an approximate solution to a recurrence is to solve a simpler related recurrence. This is a general approach to solving recurrences that consists of first studying simplified recurrences obtained by extracting what seems to be dominant parts, then solving the simplified recurrence, and finally comparing solutions of the original recurrence to those of the simplified recurrence. This technique is akin to a class of methods familiar in numerical analysis, *perturbation methods*. Informally, this method involves the following steps:

- Modify the recurrence slightly to find a known recurrence.
- Change variables to pull out the known bounds and transform into a recurrence on the (smaller) unknown part of the solution.
- Bound the unknown “error” term.

For example, consider the recurrence

$$a_{n+1} = 2a_n + \frac{a_{n-1}}{n^2} \quad \text{for } n > 1 \text{ with } a_0 = 1 \text{ and } a_1 = 2.$$

It seems reasonable to assume that the last term, because of its coefficient $1/n^2$, makes only a small contribution to the recurrence, so that

$$a_{n+1} \approx 2a_n.$$

Thus a growth of the rough form $a_n \approx 2^n$ is anticipated. To make this precise, we thus consider the simpler sequence

$$b_{n+1} = 2b_n \quad \text{for } n > 0 \text{ with } b_0 = 1$$

(so that $b_n = 2^n$) and compare the two recurrences by forming the ratio

$$\rho_n = \frac{a_n}{b_n} = \frac{a_n}{2^n}.$$

From the recurrences, we have

$$\rho_{n+1} = \rho_n + \frac{1}{4n^2}\rho_{n-1} \quad \text{for } n > 0 \text{ with } \rho_0 = 1.$$

Clearly, the ρ_n are increasing. To prove they tend to a constant, note that

$$\rho_{n+1} \leq \rho_n \left(1 + \frac{1}{4n^2}\right) \quad \text{for } n \geq 1 \text{ so that} \quad \rho_{n+1} \leq \prod_{k=1}^n \left(1 + \frac{1}{4k^2}\right).$$

But the infinite product corresponding to the right-hand side converges monotonically to

$$\alpha_0 = \prod_{k=1}^{\infty} \left(1 + \frac{1}{4k^2}\right) = 1.46505 \dots$$

Thus, ρ_n is bounded from above by α_0 and, as it is increasing, it must converge to a constant. We have thus proved that

$$a_n \sim \alpha \cdot 2^n,$$

for some constant $\alpha < 1.46505 \dots$. (In addition, the bound is not too crude as, for instance, $\rho_{100} = 1.44130 \dots$)

The example above is only a simple one, meant to illustrate the approach. In general, the situation is likely to be more complex, and several steps of iteration of the method may be required, possibly introducing several intermediate recurrences. This relates to bootstrapping, which we have just discussed. Hardships may also occur if the simplified recurrence admits of no closed form expression. The perturbation method is nonetheless an important technique for the asymptotic solution of recurrences.

Exercise 2.50 Find the asymptotic growth of the solution to the “perturbed” Fibonacci recurrence

$$a_{n+1} = \left(1 + \frac{1}{n}\right)a_n + \left(1 - \frac{1}{n}\right)a_{n-1} \quad \text{for } n > 1 \text{ with } a_0 = 0 \text{ and } a_1 = 1.$$

Exercise 2.51 Solve the recurrence

$$a_n = na_{n-1} + n^2a_{n-2} \quad \text{for } n > 1 \text{ with } a_1 = 1 \text{ and } a_0 = 0.$$

Exercise 2.52 [Aho and Sloane] The recurrence

$$a_n = a_{n-1}^2 + 1 \quad \text{for } n > 0 \text{ with } a_0 = 1$$

satisfies $a_n \sim \lambda\alpha^{2^n}$ for some constants α and λ . Find a convergent series for α and determine α to 50 decimal digits. (*Hint:* Consider $b_n = \lg a_n$.)

Exercise 2.53 Solve the following perturbation of the Fibonacci recurrence:

$$a_n = \left(1 - \frac{1}{n}\right)(a_{n-1} + a_{n-2}) \quad \text{for } n > 1 \text{ with } a_0 = a_1 = 1.$$

Try a solution of the form $n^\alpha \phi^n$ and identify α .

2.6 Binary Divide-and-Conquer Recurrences and Binary Numbers.

Good algorithms for a broad variety of problems have been developed by applying the following fundamental algorithmic design paradigm: “Divide the problem into two subproblems of equal size, solve them recursively, then use the solutions to solve the original problem.” Mergesort is a prototype of such algorithms. For example (see the proof of Theorem 1.2 in §1.2), the number of comparisons used by mergesort is given by the solution to the recurrence

$$C_N = C_{\lfloor N/2 \rfloor} + C_{\lceil N/2 \rceil} + N \quad \text{for } N > 1 \text{ with } C_1 = 0. \quad (4)$$

This recurrence, and others similar to it, arise in the analysis of a variety of algorithms with the same basic structure as mergesort. It is normally possible to determine the asymptotic growth of functions satisfying such recurrences, but it is necessary to take special care in deriving exact results, primarily because of the simple reason that a problem of “size” N cannot be divided

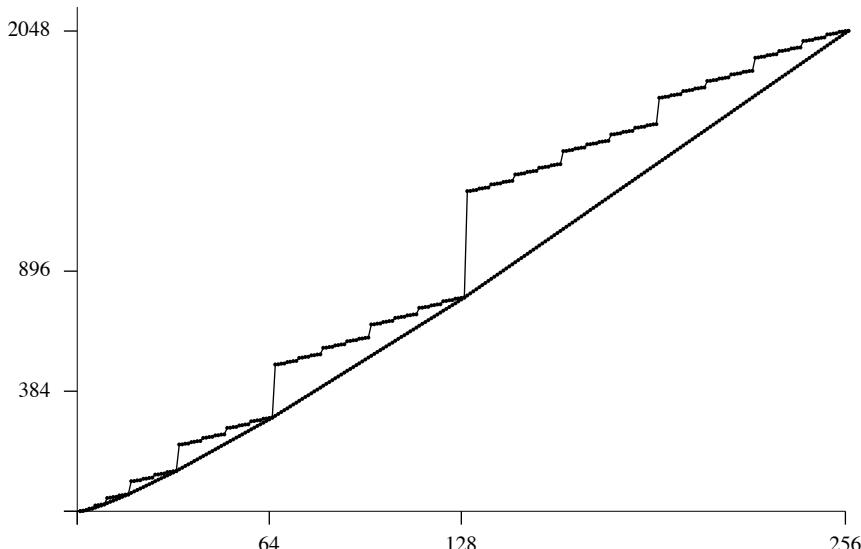


Figure 2.1 Solutions to binary divide-and-conquer recurrences

$$C_N = C_{\lfloor N/2 \rfloor} + C_{\lceil N/2 \rceil} + N \text{ (bottom)}$$

$$C_N = C_{\lceil N/2 \rceil} + C_{\lceil N/2 \rceil} + N \text{ (top)}$$

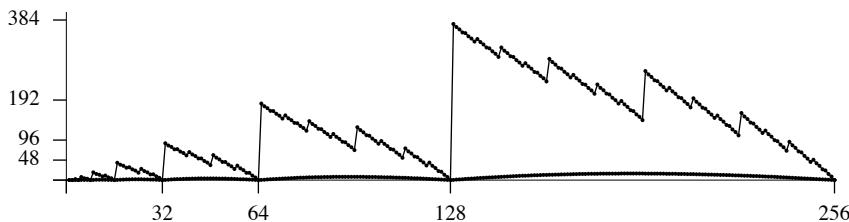


Figure 2.2 Periodic terms in binary divide-and-conquer recurrences

$$C_N = C_{\lfloor N/2 \rfloor} + C_{\lceil N/2 \rceil} + N \text{ (bottom)}$$

$$C_N = C_{\lceil N/2 \rceil} + C_{\lfloor N/2 \rfloor} + N \text{ (top)}$$

into equal-sized subproblems if N is odd: the best that can be done is to make the problem sizes differ by one. For large N , this is negligible, but for small N it is noticeable, and, as usual, the recursive structure ensures that many small subproblems will be involved.

As we shall soon see, this means that exact solutions tend to have periodicities, sometimes even severe discontinuities, and often cannot be described in terms of smooth functions. For example, Figure 2.1 shows the solution to the mergesort recurrence (4) and the similar recurrence

$$C_N = 2C_{\lceil N/2 \rceil} + N \quad \text{for } N > 1 \text{ with } C_1 = 0.$$

The former appears to be relatively smooth; the erratic fractal-based behavior that characterizes the solution to the latter is common in divide-and-conquer recurrences.

Both of the functions illustrated in Figure 2.1 are $\sim N \lg N$ and precisely equal to $N \lg N$ when N is a power of 2. Figure 2.2 is a plot of the same functions with $N \lg N$ subtracted out, to illustrate the periodic behavior of the linear term for both functions. The periodic function associated with mergesort is quite small in magnitude and continuous, with discontinuities in the derivative at powers of 2; the other function can be relatively large and is essentially discontinuous. Such behavior can be problematic when we are trying to make precise estimates for the purposes of comparing programs, even asymptotically. Fortunately, however, we typically can see the nature of the solutions quite easily when the recurrences are understood in terms of number representations. To illustrate this, we begin by looking at another important

algorithm that is a specific instance of a general problem-solving strategy that dates to antiquity.

Binary search. One of the simplest and best-known binary divide-and-conquer algorithms is called *binary search*. Given a fixed set of numbers, we wish to be able to determine quickly whether a given query number is in the set. To do so, we first sort the table. Then, for any query number, we can use the method shown in Program 2.2: Look in the middle and report success if the query number is there. Otherwise, (recursively) use the same method to look in the left half if the number is smaller than the middle number and in the right half if the query number is larger than the middle number.

Theorem 2.3 (Binary search). The number of comparisons used during an unsuccessful search with binary search in a table of size N in the worst case is equal to the number of bits in the binary representation of N . Both are described by the recurrence

$$B_N = B_{\lfloor N/2 \rfloor} + 1 \quad \text{for } N \geq 2 \text{ with } B_1 = 1,$$

which has the exact solution $B_N = \lfloor \lg N \rfloor + 1$.

Proof. After looking in “the middle,” one element is eliminated, and the two halves of the file are of size $\lfloor (N - 1)/2 \rfloor$ and $\lceil (N - 1)/2 \rceil$. The recurrence

```
public static int search(int key, int lo, int hi)
{
    if (lo > hi) return -1;
    int mid = lo + (hi - lo) / 2;
    if (key < a[mid])
        return search(key, lo, mid - 1);
    else if (key > a[mid])
        return search(key, mid + 1, hi);
    else return mid;
}
```

Program 2.2 Binary search

is established by checking separately for N odd and N even that the larger of these two is always $\lfloor N/2 \rfloor$. For example, in a table of size 83, both subfiles are of size 41 after the first comparison, but in a table of size 82, one is of size 40 and the other of size 41.

This is equal to the number of bits in the binary representation of N (ignoring leading 0s) because computing $\lfloor N/2 \rfloor$ is precisely equivalent to shifting the binary representation right by one bit position. Iterating the recurrence amounts to counting the bits, stopping when the leading 1 bit is encountered.

The number of bits in the binary representation of N is $n + 1$ for $2^n \leq N < 2^{n+1}$, or, taking logarithms, for $n \leq \lg N < n + 1$; that is to say, by definition, $n = \lfloor \lg N \rfloor$. ■

The functions $\lg N$ and $\lfloor \lg N \rfloor$ are plotted in Figure 2.3, along with the fractional part $\{\lg N\} \equiv \lg N - \lfloor \lg N \rfloor$.

Exercise 2.54 What is the number of comparisons used during an unsuccessful search with binary search in a table of size N in the *best* case?

Exercise 2.55 Consider a “ternary search” algorithm, where the file is divided into thirds, two comparisons are used to determine where the key could be, and the algorithm is applied recursively. Characterize the number of comparisons used by that algorithm in the worst case, and compare it to a binary search.

Exact solution of mergesort recurrence. The mergesort recurrence (2) is easily solved by differencing: if D_N is defined to be $C_{N+1} - C_N$, then D_N satisfies the recurrence

$$D_N = D_{\lfloor N/2 \rfloor} + 1 \quad \text{for } N \geq 2 \text{ with } D_1 = 2,$$

which iterates to

$$D_N = \lfloor \lg N \rfloor + 2,$$

and, therefore,

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

There are a number of ways to evaluate this sum to give an exact formula for C_N : as mentioned earlier, it is useful to adopt the approach of noting a relationship to the binary representations of integers. In particular, we just saw that $\lfloor \lg k \rfloor + 1$ is the number of bits in the binary representation of k .

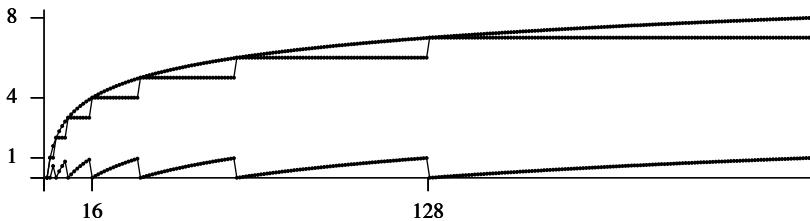


Figure 2.3 $\lg N$ (top); $\lfloor \lg N \rfloor$ (middle); $\{ \lg N \}$ (bottom)

(ignoring leading 0s), so C_N is precisely the number of bits in the binary representations of all the positive numbers less than N plus $N - 1$.

Theorem 2.4 (Mergesort). The number of comparisons used by mergesort is equal to $N - 1$ plus the number of bits in the binary representations of all the numbers less than N . Both quantities are described by the recurrence

$$C_N = C_{\lfloor N/2 \rfloor} + C_{\lceil N/2 \rceil} + N \quad \text{for } N \geq 2 \text{ with } C_1 = 0,$$

which has the exact solution $C_N = N \lfloor \lg N \rfloor + 2N - 2^{\lfloor \lg N \rfloor + 1}$.

Proof. The first part of the theorem is established by the earlier discussion. Now, all $N - 1$ of the numbers less than N have a rightmost bit; $N - 2$ of them (all except 1) have a second-to-rightmost bit; $N - 4$ of them (all except 1, 2, and 3) have a third-to-rightmost bit; $N - 8$ of them have a fourth-to-rightmost bit, and so on, so we must have

$$\begin{aligned} C_N &= (N - 1) + (N - 1) + (N - 2) + (N - 4) + \cdots + (N - 2^{\lfloor \lg N \rfloor}) \\ &= (N - 1) + N(\lfloor \lg N \rfloor + 1) - (1 + 2 + 4 + \dots + 2^{\lfloor \lg N \rfloor}) \\ &= N \lfloor \lg N \rfloor + 2N - 2^{\lfloor \lg N \rfloor + 1}. \end{aligned}$$

As noted earlier, $\lfloor \lg N \rfloor$ is a discontinuous function with periodic behavior. Also as mentioned, however, C_N itself is continuous, so the discontinuities (but not the periodicities) in the two functions involving $\lfloor \lg N \rfloor$ cancel out. This phenomenon is illustrated in Figure 2.4 and supported by the calculations in the corollary that follows. ■

Corollary $C_N = N\lg N + N\theta(1 - \{\lg N\})$, where $\theta(x) = 1 + x - 2^x$ is a positive function satisfying $\theta(0) = \theta(1) = 0$ and $0 < \theta(x) < .086$ for $0 < x < 1$.

Proof. Straightforward by substituting the decomposition $\lfloor \lg N \rfloor = \lg N - \{\lg N\}$. The value $.086 \approx 1 - \lg e + \lg \lg e$ is calculated by setting the derivative of $\theta(x)$ to zero. ■

Exercise 2.56 By considering the rightmost bits, give a direct proof that the number of bits in the binary representations of all the numbers less than N satisfies (4), but with an additive term of $N - 1$ instead of N .

Exercise 2.57 Prove that $N\lfloor \lg N \rfloor + 2N - 2^{\lfloor \lg N \rfloor + 1} = N\lceil \lg N \rceil + N - 2^{\lceil \lg N \rceil}$ for all positive N . (See Exercise 1.4.)

Other properties of binary numbers. We often study properties of binary integers because they naturally model the (binary) decision-making process in many basic algorithms. The quantities encountered earlier are likely to arise in the analysis of any algorithm that solves a problem by recursively dividing

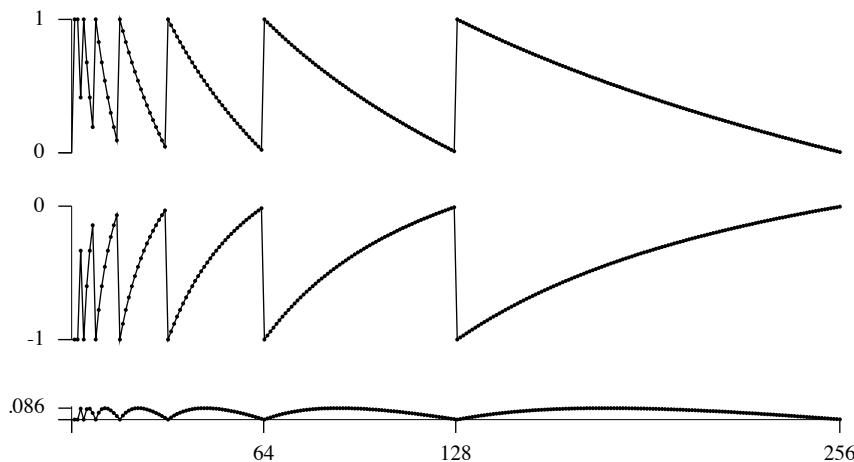


Figure 2.4 Composition of the periodic function $\theta(1 - \{\lg N\})$

$1 - \{\lg N\}$ (top)

$1 - 2^{1 - \{\lg N\}}$ (middle)

$2 - \{\lg N\} - 2^{1 - \{\lg N\}}$ (bottom)

it in two, the way that binary search and mergesort do, and similar quantities clearly will arise in other divide-and-conquer schemes. To complete our study of binary divide-and-conquer recurrences, we consider two more properties of the binary representation of numbers that frequently arise in the analysis of algorithms.

Definition Given an integer N , define the *population count* function ν_N to be the number of 1s in the binary representation of N and the *cumulated population count* function P_N to be the number of 1s in the binary representations of all the numbers less than N .

Definition Given an integer N , define the *ruler* function ψ_N to be the number of trailing 1s in the binary representation of N and the *cumulated ruler* function R_N to be the number of trailing 1s in the binary representations of all the numbers less than N .

Table 2.3 gives the values of the functions for $N < 16$. The reader may find it instructive to try to compute values of these functions for larger N . For example, the binary representation of 83 is 1010011, so $\nu_{83} = 4$ and $\psi_{83} = 2$. The cumulated values

$$P_N \equiv \sum_{0 \leq j < N} \nu_j \quad \text{and} \quad R_N \equiv \sum_{0 \leq j < N} \psi_j$$

are less easily computed. For example, $P_{84} = 215$ and $R_{84} = 78$.

It is not hard to see, for example, that

$$P_{2^n} = n2^{n-1} \quad \text{and} \quad R_{2^n} = 2^n - 1$$

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
ν_N	1	1	2	1	2	2	3	1	2	2	3	2	3	3	4
P_N	0	1	2	4	5	7	9	12	13	15	17	20	22	25	28
ψ_N	1	0	2	0	1	0	3	0	1	0	2	0	1	0	4
R_N	0	1	1	3	3	4	4	7	7	8	8	10	10	11	11

Table 2.3 Ruler and population count functions

and that

$$P_N = \frac{1}{2}N\lg N + O(N) \text{ and } R_N = N + O(\lg N),$$

but exact representations or more precise asymptotic estimates are difficult to derive, as indicated by the plot of $P_N - (N\lg N)/2$ in Figure 2.5.

As noted previously, functions of this type satisfy recurrences that are simply derived but can differ markedly, even when describing the same quantity. For example, considering the leftmost bits, we see that $2^{\lfloor \lg N \rfloor}$ of the numbers less than N start with 0 and the rest start with 1, so we have

$$P_N = P_{2^{\lfloor \lg N \rfloor}} + (N - 2^{\lfloor \lg N \rfloor}) + P_{N - 2^{\lfloor \lg N \rfloor}}.$$

But also, considering the rightmost bit, it is clear that

$$P_N = P_{\lceil N/2 \rceil} + P_{\lfloor N/2 \rfloor} + \lfloor N/2 \rfloor \quad \text{for } N > 1 \text{ with } P_1 = 0.$$

This is similar to the mergesort recurrence, which is associated with counting all the bits less than N (there should be about half as many 1 bits), but the function is remarkably different. (Compare Figure 2.5 with Figure 2.2.) The function counting the number of 0 bits is similar: both are fractal in behavior, but they cancel out to get the periodic but continuous function that we have seen before (see Figure 2.2). Delange [7] studied the function P_N in some detail, and expressed it in terms of a function that is nowhere differentiable.

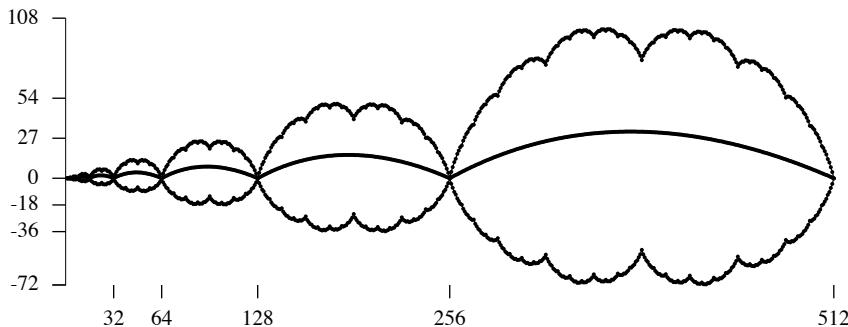


Figure 2.5 Periodic and fractal terms in bit counting

- {# 1 bits in numbers less than N } – $(N\lg N)/2$ (top)
- {# 0 bits in numbers less than N } – $(N\lg N)/2$ (bottom)
- {# bits in numbers less than N } – $N\lg N$ (middle)

Other recurrences. Binary search and mergesort are typical “binary divide-and-conquer” algorithms. The examples that we have given reveal relationships to properties of the binary representations of numbers that we can use to develop more precise solutions for other, similar, recurrences along the lines we have been discussing.

Table 2.4 shows a number of recurrences that commonly arise, along with approximate solutions derived from the general theorems given in the next section. In the table, $a_{N/2}$ means “ $a_{\lfloor N/2 \rfloor}$ or $a_{\lceil N/2 \rceil}$,” $2a_{N/2}$ means “ $a_{N/2} + a_{N/2}$,” and so forth—in the next section we discuss the fact that minor variations of this sort do not affect the asymptotic results given in Table 2.4 (though they do prevent us from giving more general estimates).

Normally, applications involving such recurrences involve worst-case results, as in Theorem 2.5 and Theorem 2.6, but if the subproblems are independent and still “random” after the divide step, then these results can also yield expected costs for some problems.

It is worth recognizing that it is only a small step from these properties of numbers to properties of bitstrings that are certainly more combinatorial

$a_N = a_{N/2} + 1$	$\lg N + O(1)$
$a_N = a_{N/2} + N$	$2N + O(\lg N)$
$a_N = a_{N/2} + N \lg N$	$\Theta(N \lg N)$
$a_N = 2a_{N/2} + 1$	$\Theta(N)$
$a_N = 2a_{N/2} + \lg N$	$\Theta(N)$
$a_N = 2a_{N/2} + N$	$N \log N + O(N)$
$a_N = 2a_{N/2} + N \lg N$	$\frac{1}{2}N \lg N^2 + O(N \log N)$
$a_N = 2a_{N/2} + N \lg^{\delta-1} N$	$\delta^{-1} N \lg^\delta N + O(N \lg^{\delta-1} N)$
$a_N = 2a_{N/2} + N^2$	$2N^2 + O(N)$
$a_N = 3a_{N/2} + N$	$\Theta(N^{\lg 3})$
$a_N = 4a_{N/2} + N$	$\Theta(N^2)$

Table 2.4 Binary divide-and-conquer recurrences and solutions

in nature. For example, suppose that we wanted to know the average length of the longest string of consecutive 0s in a random bitstring, say, to enable certain optimizations in designing an arithmetic unit. Is this a property of the number, or the bits that represent it? Such questions lead immediately to more generally applicable combinatorial studies, which we will consider in Chapter 8.

We encounter functions of this type frequently in the analysis of algorithms, and it is worthwhile to be cognizant of their relationship to simple properties of binary numbers. Beyond divide-and-conquer algorithms, these functions also arise, of course, in the direct analysis of arithmetic algorithms on numbers represented in binary. A famous example of this is the analysis of the expected length of the carry propagation chain in an adder, a problem dating back to von Neumann that was completely solved by Knuth [19]. These results relate directly to analyses of fundamental algorithms on strings, as we will see in Chapter 8. Binary divide-and-conquer recurrences and properties of their solutions are studied in more detail by Allouche and Shallit [2] and by Flajolet and Golin [9].

Exercise 2.58 Give recurrences for the functions plotted in Figure 2.5.

Exercise 2.59 Derive recurrences for R_N similar to those given above for P_N .

Exercise 2.60 Plot the solution to the recurrence

$$A_N = A_{\lfloor N/2 \rfloor} + A_{\lceil N/2 \rceil} + \lfloor \lg N \rfloor \quad \text{for } N \geq 2 \text{ with } A_1 = 0,$$

for $1 \leq N \leq 512$.

Exercise 2.61 Plot the solution to the recurrence

$$B_N = 3B_{\lceil N/2 \rceil} + N \quad \text{for } N \geq 2 \text{ with } B_1 = 0,$$

for $1 \leq N \leq 512$.

Exercise 2.62 Plot the solution to

$$D_N = D_{\lceil N/2 \rceil} + D_{\lfloor N/2 \rfloor} + C_N \quad \text{for } N > 1 \text{ with } D_1 = 0$$

where C_N is the solution to

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

Consider the variants of this problem derived by changing $\lceil N/2 \rceil$ to $\lfloor N/2 \rfloor$ in each of the terms.

Exercise 2.63 Take the binary representation of N , reverse it, and interpret the result as an integer, $\rho(N)$. Show that $\rho(N)$ satisfies a divide-and-conquer recurrence. Plot its values for $1 \leq N \leq 512$ and explain what you see.

Exercise 2.64 What is the average length of the initial string of 1s in the binary representation of a number less than N , assuming all such numbers are equally likely?

Exercise 2.65 What is the average length of the initial string of 1s in a random bitstring of length N , assuming all such strings are equally likely?

Exercise 2.66 What is the average and the variance of the length of the initial string of 1s in a (potentially infinite) sequence of random bits?

Exercise 2.67 What is the total number of carries made when a binary counter increments N times, from 0 to N ?

2.7 General Divide-and-Conquer Recurrences. More generally, efficient algorithms and upper bounds in complexity studies are very often derived by extending the divide-and-conquer algorithmic design paradigm along the following lines: “Divide the problem into smaller (perhaps overlapping) subproblems, solve them recursively, then use the solutions to solve the original problem.” A variety of “divide-and-conquer” recurrences arise that depend on the number and relative size of subproblems, the extent to which they overlap, and the cost of recombining them for the solution. It is normally possible to determine the asymptotic growth of functions satisfying such recurrences, but, as above, the periodic and fractal nature of functions that are involved make it necessary to specify details carefully.

In pursuit of a general solution, we start with the recursive formula

$$a(x) = \alpha a(x/\beta) + f(x) \quad \text{for } x > 1 \text{ with } a(x) = 0 \text{ for } x \leq 1$$

defining a *function* over the positive real numbers. In essence, this corresponds to a divide-and-conquer algorithm that divides a problem of size x into α subproblems of size x/β and recombines them at a cost of $f(x)$. Here $a(x)$ is a function defined for positive real x , so that $a(x/\beta)$ is well defined. In most applications, α and β will be integers, though we do not use that fact in developing the solution. We do insist that $\beta > 1$, of course.

For example, consider the case where $f(x) = x$ and we restrict ourselves to the integers $N = \beta^n$. In this case, we have

$$a_{\beta^n} = \alpha a_{\beta^{n-1}} + \beta^n \quad \text{for } n > 0 \text{ with } a_1 = 0.$$

Dividing both sides by α^n and iterating (that is, applying Theorem 2.1) we have the solution

$$a_{\beta^n} = \alpha^n \sum_{1 \leq j \leq n} \left(\frac{\beta}{\alpha}\right)^j.$$

Now, there are three cases: if $\alpha > \beta$, the sum converges to a constant; if $\alpha = \beta$, it evaluates to n ; and if $\alpha < \beta$, the sum is dominated by the latter terms and is $O(\beta/\alpha)^n$. Since $\alpha^n = (\beta^{\log_\beta \alpha})^n = (\beta^n)^{\log_\beta \alpha}$, this means that the solution to the recurrence is $O(N^{\log_\beta \alpha})$ when $\alpha > \beta$, $O(N \log N)$ when $\alpha = \beta$, and $O(N)$ when $\alpha < \beta$. Though this solution only holds for $N = \beta^n$, it illustrates the overall structure encountered in the general case.

Theorem 2.5 (Divide-and-conquer functions). If the function $a(x)$ satisfies the recurrence

$$a(x) = \alpha a(x/\beta) + x \quad \text{for } x > 1 \text{ with } a(x) = 0 \text{ for } x \leq 1$$

then

$$\begin{aligned} \text{if } \alpha < \beta \quad a(x) &\sim \frac{\beta}{\beta - \alpha} x \\ \text{if } \alpha = \beta \quad a(x) &\sim x \log_\beta x \\ \text{if } \alpha > \beta \quad a(x) &\sim \frac{\alpha}{\alpha - \beta} \left(\frac{\beta}{\alpha}\right)^{\{\log_\beta \alpha\}} x^{\log_\beta \alpha}. \end{aligned}$$

Proof. The basic idea, which applies to all divide-and-conquer recurrences, is to iterate the recurrence until the initial conditions are met for the subproblems. Here, we have

$$\begin{aligned} a(x) &= x + \alpha a(x/\beta) \\ &= x + \alpha \frac{x}{\beta} + \alpha a(x/\beta^2) \\ &= x + \alpha \frac{x}{\beta} + \alpha^2 \frac{x}{\beta^2} + \alpha a(x/\beta^3) \end{aligned}$$

and so on. After $t = \lfloor \log_\beta x \rfloor$ iterations, the term $a(x/\beta^t)$ that appears can be replaced by 0 and the iteration process terminates. This leaves an exact representation of the solution:

$$a(x) = x \left(1 + \frac{\alpha}{\beta} + \dots + \frac{\alpha^t}{\beta^t}\right).$$

Now, as mentioned earlier, three cases can be distinguished. First, if $\alpha < \beta$ then the sum converges and

$$a(x) \sim x \sum_{j \geq 0} \left(\frac{\alpha}{\beta}\right)^j = \frac{\beta}{\beta - \alpha} x.$$

Second, if $\alpha = \beta$ then each of the terms in the sum is 1 and the solution is simply

$$a(x) = x(\lfloor \log_\beta x \rfloor + 1) \sim x \log_\beta x.$$

Third, if $\alpha > \beta$ then the last term in the sum predominates, so that

$$\begin{aligned} a(x) &= x \left(\frac{\alpha}{\beta}\right)^t \left(1 + \frac{\beta}{\alpha} + \dots + \frac{\beta^t}{\alpha^t}\right) \\ &\sim x \frac{\alpha}{\alpha - \beta} \left(\frac{\alpha}{\beta}\right)^t. \end{aligned}$$

As mentioned previously, the periodic behavior of the expression in the third case can be isolated by separating the integer and fractional part of $\log_\beta x$ and writing $t \equiv \lfloor \log_\beta x \rfloor = \log_\beta x - \{\log_\beta x\}$. This gives

$$x \left(\frac{\alpha}{\beta}\right)^t = x \left(\frac{\alpha}{\beta}\right)^{\log_\beta x} \left(\frac{\alpha}{\beta}\right)^{-\{\log_\beta x\}} = x^{\log_\beta \alpha} \left(\frac{\beta}{\alpha}\right)^{\{\log_\beta x\}},$$

since $\alpha^{\log_\beta x} = x^{\log_\beta \alpha}$. This completes the proof.

For $\alpha \leq \beta$, the periodic behavior is not in the leading term, but for $\alpha > \beta$, the coefficient of $x^{\log_\beta \alpha}$ is a periodic function of $\log_\beta x$ that is bounded and oscillates between $\alpha/(\alpha - \beta)$ and $\beta/(\alpha - \beta)$. ■

Figure 2.6 illustrates how the relative values of α and β affect the asymptotic growth of the function. Boxes in the figures correspond to problem sizes for a divide-and-conquer algorithm. The top diagram, where a problem is split into two subproblems, each a third of the size of the original, shows how the performance is linear because the problem sizes go to 0 exponentially fast. The middle diagram, where a problem is split into three subproblems, each a third of the size of the original, shows how the total problem size is well balanced so a “log” multiplicative factor is needed. The last diagram, where a problem is split into four subproblems, each a third of the size of the original,

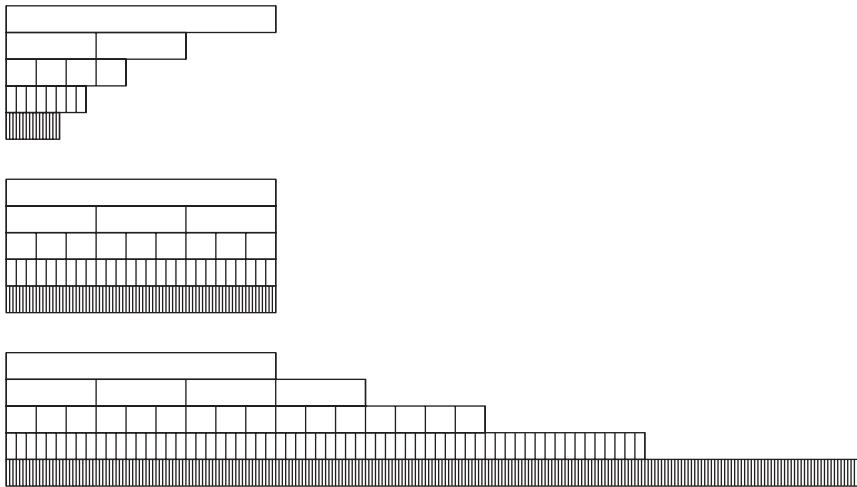


Figure 2.6 Divide-and-conquer for $\beta = 3$ and $\alpha = 2, 3, 4$

shows how the total problem size grows exponentially, so the total is dominated by the last term. This shows the asymptotic growth and is representative of what happens in general situations.

To generalize this to the point where it applies to practical situations, we need to consider other $f(x)$ and less restrictive subdivision strategies than precisely equal subproblem sizes (which will allow us to move back to recurrences on integers). For other $f(x)$, we proceed precisely as done earlier: at the top level we have one problem of cost $f(x)$, then we have α problems of cost $f(x/\beta)$, then α^2 problems of cost $f(x/\beta^2)$, and so on, so the total cost is

$$f(x) + \alpha f(x/\beta) + \alpha^2 f(x/\beta^2) + \dots .$$

As earlier, there are three cases: if $\alpha > \beta$, the later terms in the sum dominate; if $\alpha = \beta$, the terms are roughly equal; and if $\alpha < \beta$, the early terms dominate. Some “smoothness” restrictions on the function f are necessary to derive a precise answer. For example, if we restrict f to be of the form $x^\gamma(\log x)^\delta$ —

which actually represents a significant portion of the functions that arise in complexity studies—an argument similar to that given previously can be used to show that

$$\begin{aligned} \text{if } \gamma < \log_{\beta}\alpha & \quad a(x) \sim c_1 x^{\gamma} (\log x)^{\delta} \\ \text{if } \gamma = \log_{\beta}\alpha & \quad a(x) \sim c_2 x^{\gamma} (\log x)^{\delta+1} \\ \text{if } \gamma > \log_{\beta}\alpha & \quad a(x) = \Theta(x^{\log_{\beta}\alpha}) \end{aligned}$$

where c_1 and c_2 are appropriate constants that depend on α , β , and γ .

Exercise 2.68 Give explicit formulae for c_1 and c_2 . Start by doing the case $\delta = 0$.

Intuitively, we expect the same kind of result even when the subproblems are almost, but not necessarily exactly, the same size. Indeed, we are bound to consider this case because “problem sizes” must be integers: of course, dividing a file whose size is odd into two parts gives subproblems of almost, but not quite, the same size. Moreover, we expect that we need not have an exact value of $f(x)$ in order to estimate the growth of $a(x)$. Of course, we are also interested in functions that are defined only on the integers. Putting these together, we get a result that is useful for the analysis of a variety of algorithms.

Theorem 2.6 (Divide-and-conquer sequences). If a divide-and-conquer algorithm works by dividing a problem of size n into α parts, each of size $n/\beta + O(1)$, and solving the subproblems independently with additional cost $f(n)$ for dividing and combining, then if $f(n) = \Theta(n^{\gamma}(\log n)^{\delta})$, the total cost is given by

$$\begin{aligned} \text{if } \gamma < \log_{\beta}\alpha & \quad a_n = \Theta(n^{\gamma}(\log n)^{\delta}) \\ \text{if } \gamma = \log_{\beta}\alpha & \quad a_n = \Theta(n^{\gamma}(\log n)^{\delta+1}) \\ \text{if } \gamma > \log_{\beta}\alpha & \quad a_n = \Theta(n^{\log_{\beta}\alpha}). \end{aligned}$$

Proof. The general strategy is the same as used earlier: iterate the recurrence until the initial conditions are satisfied, then collect terms. The calculations involved are rather intricate and are omitted here. ■

In complexity studies, a more general formulation is often used, since less specific information about $f(n)$ may be available. Under suitable conditions on the smoothness of $f(n)$, it can be shown that

$$\begin{array}{ll} \text{if } f(n) = O(n^{\log_\beta \alpha - \epsilon}) & a_n = \Theta(n^{\log_\beta \alpha}) \\ \text{if } f(n) = \Theta(n^{\log_\beta \alpha}) & a_n = \Theta(n^{\log_\beta \alpha} \log n) \\ \text{if } f(n) = \Omega(n^{\log_\beta \alpha + \epsilon}) & a_n = \Theta(f(n)). \end{array}$$

This result is primarily due to Bentley, Haken, and Saxe [4]; a full proof of a similar result may also be found in [5]. This type of result is normally used to prove upper bounds and lower bounds on asymptotic behavior of algorithms, by choosing $f(n)$ to bound true costs appropriately. In this book, we are normally interested in deriving more accurate results for specific $f(n)$.

Exercise 2.69 Plot the periodic part of the solution to the recurrence

$$a_N = 3a_{\lfloor N/3 \rfloor} + N \quad \text{for } N > 3 \text{ with } a_1 = a_2 = a_3 = 1$$

for $1 \leq N \leq 972$.

Exercise 2.70 Answer the previous question for the other possible ways of dividing a problem of size N into three parts with the size of each part either $\lfloor N/3 \rfloor$ or $\lceil N/3 \rceil$.

Exercise 2.71 Give an asymptotic solution to the recurrence

$$a(x) = \alpha a_{x/\beta} + 2^x \quad \text{for } x > 1 \text{ with } a(x) = 0 \text{ for } x \leq 1.$$

Exercise 2.72 Give an asymptotic solution to the recurrence

$$a_N = a_{3N/4} + a_{N/4} + N \quad \text{for } N > 2 \text{ with } a_1 = a_2 = a_3 = 1.$$

Exercise 2.73 Give an asymptotic solution to the recurrence

$$a_N = a_{N/2} + a_{N/4} + N \quad \text{for } N > 2 \text{ with } a_1 = a_2 = a_3 = 1.$$

Exercise 2.74 Consider the recurrence

$$a_n = a_{f(n)} + a_{g(n)} + a_{h(n)} + 1 \quad \text{for } n > t \text{ with } a_n = 1 \text{ for } n < t$$

with the constraint that $f(n) + g(n) + h(n) = n$. Prove that $a_n = \Theta(n)$.

Exercise 2.75 Consider the recurrence

$$a_n = a_{f(n)} + a_{g(n)} + 1 \quad \text{for } N > t \text{ with } a_n = 1 \text{ for } n < t$$

with $f(n) + g(n) = n - h(n)$. Give the smallest value of $h(n)$ for which you can prove that $a_n/n \rightarrow 0$ as $n \rightarrow \infty$.

RECURRENCE relations correspond naturally to iterative and recursive programs, and they can serve us well in a variety of applications in the analysis of algorithms, so we have surveyed in this chapter the types of recurrence relations that can arise and some ways of coping with them. Understanding an algorithm sufficiently well to be able to develop a recurrence relation describing an important performance characteristic is often an important first step in analyzing it. Given a recurrence relation, we can often compute or estimate needed parameters for practical applications even if an analytic solution seems too difficult to obtain. On the other hand, as we will see, the existence of a recurrence often signals that our problem has sufficient structure that we can use general tools to develop analytic results.

There is a large literature on “difference equations” and recurrences, from which we have tried to select useful and relevant tools, techniques, and examples. There are general and essential mathematical tools for dealing with recurrences, but finding the appropriate path to solving a particular recurrence is often challenging. Nevertheless, a careful analysis can lead to understanding of the essential properties of a broad variety of the recurrences that arise in practice. We calculate values of the recurrence to get some idea of its rate of growth; try telescoping (iterating) it to get an idea of the asymptotic form of the solution; perhaps look for a summation factor, change of variable, or repertoire suite that can lead to an exact solution; or apply an approximation technique such as bootstrapping or perturbation to estimate the solution.

Our discussion has been exclusively devoted to recurrences on one index N . We defer discussion of multivariate and other types of recurrences until we have developed more advanced tools for solving them.

Studies in the theory of algorithms often depend on solving recurrences for estimating and bounding the performance characteristics of algorithms. Specifically, the “divide-and-conquer” recurrences that we considered at the end of the chapter arise particularly frequently in the theoretical computer science literature, as divide-and-conquer is a principal tool in algorithm design. Most such recurrences have a similar structure, which reflects the degree of balance in the algorithm design. They also are closely related to properties of number systems, and thus tend to exhibit fractal-like behavior. Approximate bounds such as those we have seen are appropriate (and widely used) for deriving upper bounds in complexity proofs, but not necessarily used for analyzing the performance of algorithms, because they do not always provide sufficiently accurate information to allow us to predict performance. We can

often get more precise estimates in situations where we have more precise information about $f(n)$ and the divide-and-conquer method.

Recurrences arise in a natural way in the study of performance characteristics of algorithms. As we develop detailed analyses of complicated algorithms, we encounter rather complex recurrences to be solved. In the next chapter, we introduce generating functions, which are fundamental to the analysis of algorithms. Not only can they help us solve recurrences, but also they have a direct connection to algorithms at a high level, allowing us to leave the detailed structure described by recurrences below the surface for many applications.

References

1. A. V. AHO AND N. J. A. SLOANE. "Some doubly exponential sequences," *Fibonacci Quarterly* **11**, 1973, 429–437.
2. J.-P. ALLOCHE AND J. SHALLIT. "The ring of k -regular sequences," *Theoretical Computer Science* **98**, 1992, 163–197.
3. C. M. BENDER AND S. A. ORSZAG. *Advanced Mathematical Methods for Scientists and Engineers*, McGraw-Hill, New York, 1978.
4. J. L. BENTLEY, D. HAKEN, AND J. B. SAXE. "A general method for solving divide-and-conquer recurrences," *SIGACT News*, Fall 1980, 36–44.
5. T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, AND C. STEIN. *Introduction to Algorithms*, MIT Press, New York, 3rd, edition, 2009.
6. N. G. DE BRUIJN. *Asymptotic Methods in Analysis*, Dover Publications, New York, 1981.
7. H. DELANGE. "Sur la fonction sommatoire de la fonction somme des chiffres," *L'enseignement Mathématique* **XXI**, 1975, 31–47.
8. P. FLAJOLET AND M. GOLIN. "Exact asymptotics of divide-and-conquer recurrences," in *Automata, Languages, and Programming*, A. Lingas, R. Karlsson, and S. Carlsson, eds., Lecture Notes in Computer Science #700, Springer Verlag, Berlin, 1993, 137–149.
9. P. FLAJOLET AND M. GOLIN. "Mellin transforms and asymptotics: the mergesort recurrence," *Acta Informatica* **31**, 1994, 673–696.
10. P. FLAJOLET, P. GRABNER, AND P. KIRSCHENHOFER. "Mellin transforms and asymptotics: digital sums," *Theoretical Computer Science* **123**, 1994, 291–314.
11. P. FLAJOLET, J.-C. RAOULT, AND J. VUILLEMIN. "The number of registers required to evaluate arithmetic expressions," *Theoretical Computer Science* **9**, 1979, 99–125.
12. P. FLAJOLET AND R. SEDGEWICK. *Analytic Combinatorics*, Cambridge University Press, 2009.
13. R. L. GRAHAM, D. E. KNUTH, AND O. PATASHNIK. *Concrete Mathematics*, 1st edition, Addison-Wesley, Reading, MA, 1989. Second edition, 1994.
14. D. H. GREENE AND D. E. KNUTH. *Mathematics for the Analysis of Algorithms*, Birkhäuser, Boston, 1981.

15. P. HENRICI. *Applied and Computational Complex Analysis*, 3 volumes, John Wiley, New York, 1974 (volume 1), 1977 (volume 2), 1986 (volume 3).
16. D. E. KNUTH. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*, 1st edition, Addison-Wesley, Reading, MA, 1969. Third edition, 1997.
17. D. E. KNUTH. *The Art of Computer Programming. Volume 3: Sorting and Searching*, 1st edition, Addison-Wesley, Reading, MA, 1973. Second edition, 1998.
18. D. E. KNUTH. “The average time for carry propagation,” *Indagationes Mathematicae* **40**, 1978, 238–242.
19. D. E. KNUTH AND A. SCHÖNHAGE. “The expected linearity of a simple equivalence algorithm,” *Theoretical Computer Science* **6**, 1978, 281–315.
20. G. LUEKER. “Some techniques for solving recurrences,” *Computing Surveys* **12**, 1980, 419–436.
21. Z. A. MELZAK. *Companion to Concrete Mathematics*, John Wiley, New York, 1968.
22. J. RIORDAN. *Combinatorial Identities*, John Wiley, New York, 1968.
23. R. SEDGEWICK. “The analysis of quicksort programs,” *Acta Informatica* **7**, 1977, 327–355.
24. A. YAO. “On random 2–3 trees,” *Acta Informatica* **9**, 1978, 159–170.

This page intentionally left blank

CHAPTER THREE

GENERATING FUNCTIONS

IN this chapter we introduce the central concept that we use in the analysis of algorithms and data structures: generating functions. This mathematical material is so fundamental to the rest of the book that we shall concentrate on presenting a synopsis somewhat apart from applications, though we do draw some examples from properties of algorithms.

After defining the basic notions of “ordinary” generating functions and “exponential” generating functions, we begin with a description of the use of generating functions to solve recurrence relations, including a discussion of necessary mathematical tools. For both ordinary and exponential generating functions, we survey many elementary functions that arise in practice, and consider their basic properties and ways of manipulating them. We discuss a number of examples, including a detailed look at solving the quicksort median-of-three recurrence from Chapter 1.

We normally are interested not just in counting combinatorial structures, but also in analyzing their properties. We look at how to use “bivariate” generating functions for this purpose, and how this relates to the use of “probability” generating functions.

The chapter concludes with a discussion of various special types of generating functions that can arise in applications in the analysis of algorithms.

Because they appear throughout the book, we describe basic properties and techniques for manipulating generating functions in some detail and provide a catalog of the most important ones in this chapter, for reference. We introduce a substantial amount of material, with examples from combinatorics and the analysis of algorithms, though our treatment of each particular topic is relatively concise. Fuller discussion of many of these topics may be found in our coverage of various applications in Chapters 6 through 9 and in the other references listed at the end of the chapter, primarily [1], [5], [4], and [19].

More important, we revisit generating functions in Chapter 5, where we characterize generating functions as our *central object of study* in the analysis of algorithms.

3.1 Ordinary Generating Functions. As we have seen, it is often our goal in the analysis of algorithms to derive specific expressions for the values of terms in a sequence of quantities a_0, a_1, a_2, \dots that measure some performance parameter. In this chapter we see the benefits of working with a single mathematical object that represents the whole sequence.

Definition Given a sequence $a_0, a_1, a_2, \dots, a_k, \dots$, the function

$$A(z) = \sum_{k \geq 0} a_k z^k$$

is called the *ordinary generating function (OGF)* of the sequence. We use the notation $[z^k]A(z)$ to refer to the coefficient a_k .

Some elementary ordinary generating functions and their corresponding sequences are given in Table 3.1. We discuss later how to derive these functions and various ways to manipulate them. The OGFs in Table 3.1 are fundamental and arise frequently in the analysis of algorithms. Each sequence can be described in many ways (with simple recurrence relations, for example), but we will see that there are significant advantages to representing them directly with generating functions.

The sum in the definition may or may not converge—for the moment we ignore questions of convergence, for two reasons. First, the manipulations that we perform on generating functions are typically well-defined formal manipulations on power series, even in the absence of convergence. Second, the sequences that arise in our analyses are normally such that convergence is assured, at least for some (small enough) z . In a great many applications in the analysis of algorithms, we are able to exploit formal relationships between power series and the algorithms under scrutiny to derive explicit formulae for generating functions in the first part of a typical analysis; and we are able to learn analytic properties of generating functions in detail (convergence plays an important role in this) to derive explicit formulae describing fundamental properties of algorithms in the second part of a typical analysis. We develop this theme in detail in Chapter 5.

Given generating functions $A(z) = \sum_{k \geq 0} a_k z^k$ and $B(z) = \sum_{k \geq 0} b_k z^k$ that represent the sequences $\{a_0, a_1, \dots, a_k, \dots\}$ and $\{b_0, b_1, \dots, b_k, \dots\}$, respectively, we can perform a number of simple transformations to get generating functions for other sequences. Several such operations are shown in Table 3.2. Examples of the application of these operations may be found in the relationships among the entries in Table 3.1.

1, 1, 1, 1, ..., 1, ...

$$\frac{1}{1-z} = \sum_{N \geq 0} z^N$$

0, 1, 2, 3, 4, ..., N , ...

$$\frac{z}{(1-z)^2} = \sum_{N \geq 1} Nz^N$$

0, 0, 1, 3, 6, 10, ..., $\binom{N}{2}$, ...

$$\frac{z^2}{(1-z)^3} = \sum_{N \geq 2} \binom{N}{2} z^N$$

0, ..., 0, 1, $M+1$, ..., $\binom{N}{M}$, ...

$$\frac{z^M}{(1-z)^{M+1}} = \sum_{N \geq M} \binom{N}{M} z^N$$

1, M , $\binom{M}{2}$..., $\binom{M}{N}$, ..., M , 1

$$(1+z)^M = \sum_{N \geq 0} \binom{M}{N} z^N$$

1, $M+1$, $\binom{M+2}{2}$, $\binom{M+3}{3}$, ...

$$\frac{1}{(1-z)^{M+1}} = \sum_{N \geq 0} \binom{N+M}{N} z^N$$

1, 0, 1, 0, ..., 1, 0, ...

$$\frac{1}{1-z^2} = \sum_{N \geq 0} z^{2N}$$

1, c , c^2 , c^3 , ..., c^N , ...

$$\frac{1}{1-cz} = \sum_{N \geq 0} c^N z^N$$

1, 1, $\frac{1}{2!}$, $\frac{1}{3!}$, $\frac{1}{4!}$, ..., $\frac{1}{N!}$, ...

$$e^z = \sum_{N \geq 0} \frac{z^N}{N!}$$

0, 1, $\frac{1}{2}$, $\frac{1}{3}$, $\frac{1}{4}$, ..., $\frac{1}{N}$, ...

$$\ln \frac{1}{1-z} = \sum_{N \geq 1} \frac{z^N}{N}$$

0, 1, $1 + \frac{1}{2}$, $1 + \frac{1}{2} + \frac{1}{3}$, ..., H_N , ...

$$\frac{1}{1-z} \ln \frac{1}{1-z} = \sum_{N \geq 1} H_N z^N$$

0, 0, 1, $3\left(\frac{1}{2} + \frac{1}{3}\right)$, $4\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4}\right)$, ...

$$\frac{z}{(1-z)^2} \ln \frac{1}{1-z} = \sum_{N \geq 0} N(H_N - 1) z^N$$

Table 3.1 Elementary ordinary generating functions

$$A(z) = \sum_{n \geq 0} a_n z^n \quad a_0, a_1, a_2, \dots, a_n, \dots$$

$$B(z) = \sum_{n \geq 0} b_n z^n \quad b_0, b_1, b_2, \dots, b_n, \dots$$

right shift

$$zA(z) = \sum_{n \geq 1} a_{n-1} z^n \quad 0, a_0, a_1, a_2, \dots, a_{n-1}, \dots$$

left shift

$$\frac{A(z) - a_0}{z} = \sum_{n \geq 0} a_{n+1} z^n \quad a_1, a_2, a_3, \dots, a_{n+1}, \dots$$

index multiply (differentiation)

$$A'(z) = \sum_{n \geq 0} (n+1) a_{n+1} z^n \quad a_1, 2a_2, \dots, (n+1)a_{n+1}, \dots$$

index divide (integration)

$$\int_0^z A(t) dt = \sum_{n \geq 1} \frac{a_{n-1}}{n} z^n \quad 0, a_0, \frac{a_1}{2}, \frac{a_2}{3}, \dots, \frac{a_{n-1}}{n}, \dots$$

scaling

$$A(\lambda z) = \sum_{n \geq 0} \lambda^n a_n z^n \quad a_0, \lambda a_1, \lambda^2 a_2, \dots, \lambda^n a_n, \dots$$

addition

$$A(z) + B(z) = \sum_{n \geq 0} (a_n + b_n) z^n \quad a_0 + b_0, \dots, a_n + b_n, \dots$$

difference

$$(1-z)A(z) = a_0 + \sum_{n \geq 1} (a_n - a_{n-1}) z^n \quad a_0, a_1 - a_0, \dots, a_n - a_{n-1}, \dots$$

convolution

$$A(z)B(z) = \sum_{n \geq 0} \left(\sum_{0 \leq k \leq n} a_k b_{n-k} \right) z^n \quad a_0 b_0, a_1 b_0 + a_0 b_1, \dots, \sum_{0 \leq k \leq n} a_k b_{n-k},$$

partial sum

$$\frac{A(z)}{1-z} = \sum_{n \geq 0} \left(\sum_{0 \leq k \leq n} a_k \right) z^n \quad a_1, a_1 + a_2, \dots, \sum_{0 \leq k \leq n} a_k, \dots$$

Table 3.2 Operations on ordinary generating functions

Theorem 3.1 (OGF operations). If two sequences $a_0, a_1, \dots, a_k, \dots$ and $b_0, b_1, \dots, b_k, \dots$ are represented by the OGFs $A(z) = \sum_{k \geq 0} a_k z^k$ and $B(z) = \sum_{k \geq 0} b_k z^k$, respectively, then the operations given in Table 3.2 produce OGFs that represent the indicated sequences. In particular:

$A(z) + B(z)$ is the OGF for $a_0 + b_0, a_1 + b_1, a_2 + b_2, \dots$

$zA(z)$ is the OGF for $0, a_0, a_1, a_2, \dots$

$A'(z)$ is the OGF for $a_1, 2a_2, 3a_3, \dots$

$A(z)B(z)$ is the OGF for $a_0 b_0, a_0 b_1 + a_1 b_0, a_0 b_2 + a_1 b_1 + a_2 b_0, \dots$

Proof. Most of these are elementary and can be verified by inspection. The convolution operation (and the *partial sum* special case) are easily proved by manipulating the order of summation:

$$\begin{aligned} A(z)B(z) &= \sum_{i \geq 0} a_i z^i \sum_{j \geq 0} b_j z^j \\ &= \sum_{i,j \geq 0} a_i b_j z^{i+j} \\ &= \sum_{n \geq 0} \left(\sum_{0 \leq k \leq n} a_k b_{n-k} \right) z^n. \end{aligned}$$

Taking $B(z) = 1/(1-z)$ in this formula gives the partial sum operation. The convolution operation plays a special role in generating function manipulations, as we shall see. ■

Corollary The OGF for the harmonic numbers is

$$\sum_{N \geq 1} H_N z^N = \frac{1}{1-z} \ln \frac{1}{1-z}.$$

Proof. Start with $1/(1-z)$ (the OGF for $1, 1, \dots, 1, \dots$), integrate (to get the OGF for $0, 1, 1/2, 1/3, \dots, 1/k, \dots$), and multiply by $1/(1-z)$. Similar examples may be found in the relationships among the entries in Table 3.1.

■

Readers unfamiliar with generating functions are encouraged to work through the following exercises to gain a basic facility for applying these transformations.

Exercise 3.1 Find the OGFs for each of the following sequences:

$$\{2^{k+1}\}_{k \geq 0}, \quad \{k2^{k+1}\}_{k \geq 0}, \quad \{kH_k\}_{k \geq 1}, \quad \{k^3\}_{k \geq 2}.$$

Exercise 3.2 Find $[z^N]$ for each of the following OGFs:

$$\frac{1}{(1-3z)^4}, \quad (1-z)^2 \ln \frac{1}{1-z}, \quad \frac{1}{(1-2z^2)^2}.$$

Exercise 3.3 Differentiate the OGF for harmonic numbers to verify the last line of Table 3.1.

Exercise 3.4 Prove that

$$\sum_{1 \leq k \leq N} H_k = (N+1)(H_{N+1} - 1).$$

Exercise 3.5 By factoring

$$\frac{z^M}{(1-z)^{M+1}} \ln \frac{1}{1-z}$$

in two different ways (and performing the associated convolutions), prove a general identity satisfied by the harmonic numbers and binomial coefficients.

Exercise 3.6 Find the OGF for

$$\left\{ \sum_{0 < k < n} \frac{1}{k(n-k)} \right\}_{n > 1}.$$

Generalize your answer.

Exercise 3.7 Find the OGF for $\{H_k/k\}_{k \geq 1}$.

Exercise 3.8 Find $[z^N]$ for each of the following OGFs:

$$\frac{1}{1-z} \left(\ln \frac{1}{1-z} \right)^2 \quad \text{and} \quad \left(\ln \frac{1}{1-z} \right)^3.$$

Use the notation

$$H_N^{(2)} \equiv 1 + \frac{1}{2^2} + \frac{1}{3^2} + \dots + \frac{1}{N^2}$$

for the “generalized harmonic numbers” that arise in these expansions.

Such elementary manipulations suffice to derive many of the sequences that we encounter in the analysis of algorithms, though more advanced tools are needed for many algorithms. From this point forward, it should become very clear that the analysis of algorithms revolves around the twin questions of determining an explicit formula for the generating function of a sequence and, conversely, determining an exact formula for members of the sequence from the generating-function representation. We will see many examples of this later in this chapter and in Chapters 6 through 9.

Formally, we could use any kernel family of functions $w_k(z)$ to define a “generating function”

$$A(z) = \sum_{k \geq 0} a_k w_k(z)$$

that encapsulates a sequence $a_0, a_1, \dots, a_k, \dots$. Though we focus almost exclusively on the kernels z^k and $z^k/k!$ (see the next section) in this book, other types do occasionally arise in the analysis of algorithms; we mention them briefly at the end of this chapter.

3.2 Exponential Generating Functions. Some sequences are more conveniently handled by a generating function that involves a normalizing factor:

Definition Given a sequence $a_0, a_1, a_2, \dots, a_k, \dots$, the function

$$A(z) = \sum_{k \geq 0} a_k \frac{z^k}{k!}$$

is called the *exponential generating function (EGF)* of the sequence. We use the notation $k![z^k]A(z)$ to refer to the coefficient a_k .

The EGF for $\{a_k\}$ is nothing more than the OGF for $\{a_k/k!\}$, but it arises in combinatorics and the analysis of algorithms for a specific and simple reason. Suppose that the coefficients a_k represent a count associated with a structure of k items. Suppose further that the k items are “labelled” so that each has a distinct identity. In some cases, the labelling is relevant (and EGFs are appropriate); in other cases it is not (and OGFs are appropriate). The factor of $k!$ accounts for all the arrangements of the labelled items that become indistinguishable if they are unlabelled. We will consider this situation in more detail in Chapter 5 when we look at the “symbolic method” for

associating generating functions and combinatorial objects; for the moment, we offer this explanation simply as justification for considering properties of EGFs in detail. They are well studied because labelled objects naturally arise in many applications.

Table 3.3 gives a number of elementary exponential generating functions that we will be encountering later in the book, and Table 3.4 gives some of the basic manipulations on EGFs. Note that the shift left/right operations for EGFs are the same as the index multiply/divide operations for OGFs (see

$$1, 1, 1, 1, \dots, 1, \dots$$

$$e^z = \sum_{N \geq 0} \frac{z^N}{N!}$$

$$0, 1, 2, 3, 4, \dots, N, \dots$$

$$ze^z = \sum_{N \geq 1} \frac{z^N}{(N-1)!}$$

$$0, 0, 1, 3, 6, 10, \dots, \binom{N}{2}, \dots$$

$$\frac{1}{2}z^2e^z = \frac{1}{2} \sum_{N \geq 2} \frac{z^N}{(N-2)!}$$

$$0, \dots, 0, 1, M+1, \dots, \binom{N}{M}, \dots$$

$$\frac{1}{M!}z^M e^z = \frac{1}{M!} \sum_{N \geq M} \frac{z^N}{(N-M)!}$$

$$1, 0, 1, 0, \dots, 1, 0, \dots$$

$$\frac{1}{2}(e^z + e^{-z}) = \sum_{N \geq 0} \frac{1 + (-1)^N}{2} \frac{z^N}{N!}$$

$$1, c, c^2, c^3, \dots, c^N, \dots$$

$$e^{cz} = \sum_{N \geq 0} \frac{c^N z^N}{N!}$$

$$1, \frac{1}{2}, \frac{1}{3}, \dots, \frac{1}{N+1}, \dots$$

$$\frac{e^z - 1}{z} = \sum_{N \geq 0} \frac{z^N}{(N+1)!}$$

$$1, 1, 2, 6, 24, \dots, N!, \dots$$

$$\frac{1}{1-z} = \sum_{N \geq 0} \frac{N! z^N}{N!}$$

Table 3.3 Elementary exponential generating functions

Table 3.2), and vice versa. As with OGFs, application of the basic operations from Table 3.4 on the basic functions in Table 3.3 yields a large fraction of the EGFs that arise in practice, and the reader is encouraged to work the exercises below to become familiar with these functions. Also as with OGFs, we can easily establish the validity of the basic operations.

Theorem 3.2 (EGF operations). If two sequences $a_0, a_1, \dots, a_k, \dots$ and $b_0, b_1, \dots, b_k, \dots$ are represented by the EGFs $A(z) = \sum_{k \geq 0} a_k z^k / k!$ and $B(z) = \sum_{k \geq 0} b_k z^k / k!$, respectively, then the operations given in Table 3.4 produce EGFs that represent the indicated sequences. In particular,

$A(z) + B(z)$ is the EGF for $a_0 + b_0, a_1 + b_1, a_2 + b_2 \dots$

$A'(z)$ is the EGF for $a_1, a_2, a_3 \dots$

$zA(z)$ is the EGF for $0, a_0, 2a_1, 3a_2, \dots$

$A(z)B(z)$ is the EGF for $a_0 b_0, a_0 b_1 + a_1 b_0, a_0 b_2 + 2a_1 b_1 + a_2 b_0, \dots$

Proof. As for Theorem 3.1, these are elementary and can be verified by inspection with the possible exception of binomial convolution, which is easily verified with OGF convolution:

$$\begin{aligned} A(z)B(z) &= \sum_{n \geq 0} \sum_{0 \leq k \leq n} \frac{a_k}{k!} \frac{b_{n-k}}{(n-k)!} z^n \\ &= \sum_{n \geq 0} \sum_{0 \leq k \leq n} \binom{n}{k} a_k b_{n-k} \frac{z^n}{n!}. \end{aligned}$$
■

Exercise 3.9 Find the EGFs for each of the following sequences:

$$\{2^{k+1}\}_{k \geq 0}, \quad \{k2^{k+1}\}_{k \geq 0}, \quad \{k^3\}_{k \geq 2}.$$

Exercise 3.10 Find the EGFs for $1, 3, 5, 7, \dots$ and $0, 2, 4, 6, \dots$

Exercise 3.11 Find $N![z^N]A(z)$ for each of the following EGFs:

$$A(z) = \frac{1}{1-z} \ln \frac{1}{1-z}, \quad A(z) = \left(\ln \frac{1}{1-z} \right)^2, \quad A(z) = e^{z+z^2}.$$

Exercise 3.12 Show that

$$N![z^N]e^z \int_0^z \frac{1-e^{-t}}{t} dt = H_N.$$

(Hint: Form a differential equation for the EGF $H(z) = \sum_{N \geq 0} H_N z^N / N!$.)

$$A(z) = \sum_{n \geq 0} a_n \frac{z^n}{n!} \quad a_0, a_1, a_2, \dots, a_n, \dots$$

$$B(z) = \sum_{n \geq 0} b_n \frac{z^n}{n!} \quad b_0, b_1, b_2, \dots, b_n, \dots$$

right shift (integration)

$$\int_0^z A(t) dt = \sum_{n \geq 1} a_{n-1} \frac{z^n}{n!} \quad 0, a_0, a_1, \dots, a_{n-1}, \dots$$

left shift (differentiation)

$$A'(z) = \sum_{n \geq 0} a_{n+1} \frac{z^n}{n!} \quad a_1, a_2, a_3, \dots, a_{n+1}, \dots$$

index multiply

$$zA(z) = \sum_{n \geq 0} n a_{n-1} \frac{z^n}{n!} \quad 0, a_0, 2a_1, 3a_2, \dots, n a_{n-1}, \dots$$

index divide

$$(A(z) - A(0))/z = \sum_{n \geq 1} \frac{a_{n+1}}{n+1} \frac{z^n}{n!} \quad a_1, \frac{a_2}{2}, \frac{a_3}{3}, \dots, \frac{a_{n+1}}{n+1}, \dots$$

addition

$$A(z) + B(z) = \sum_{n \geq 0} (a_n + b_n) \frac{z^n}{n!} \quad a_0 + b_0, \dots, a_n + b_n, \dots$$

difference

$$A'(z) - A(z) = \sum_{n \geq 0} (a_{n+1} - a_n) \frac{z^n}{n!} \quad a_1 - a_0, \dots, a_{n+1} - a_n, \dots$$

binomial convolution

$$A(z)B(z) = \sum_{n \geq 0} \left(\sum_{0 \leq k \leq n} \binom{n}{k} a_k b_{n-k} \right) \frac{z^n}{n!}$$

$$a_0 b_0, a_1 b_0 + a_0 b_1, \dots, \sum_{0 \leq k \leq n} \binom{n}{k} a_k b_{n-k}, \dots$$

binomial sum

$$e^z A(z) = \sum_{n \geq 0} \left(\sum_{0 \leq k \leq n} \binom{n}{k} a_k \right) \frac{z^n}{n!}$$

$$a_0, a_0 + a_1, \dots, \sum_{0 \leq k \leq n} \binom{n}{k} a_k, \dots$$

Table 3.4 Operations on exponential generating functions

It is not always clear whether an OGF or an EGF will lead to the most convenient solution to a problem: sometimes one will lead to a trivial solution and the other to difficult technical problems; other times either will work well. For many of the combinatorial and algorithmic problems that we encounter, the choice of whether to use OGFs or EGFs comes naturally from the structure of the problem. Moreover, interesting questions arise from an analytic standpoint: for example, can we automatically convert from the OGF for a sequence to the EGF for the same sequence, and vice versa? (Yes, by the Laplace transform; see Exercise 3.14.) In this book, we will consider many examples involving applications of both OGFs and EGFs.

Exercise 3.13 Given the EGF $A(z)$ for a sequence $\{a_k\}$, find the EGF for the sequence

$$\left\{ \sum_{0 \leq k \leq N} N! \frac{a_k}{k!} \right\}.$$

Exercise 3.14 Given the EGF $A(z)$ for a sequence $\{a_k\}$, show that the OGF for the sequence is given by

$$\int_0^\infty A(zt) e^{-t} dt,$$

if the integral exists. Check this for sequences that appear in Tables 3.1 and 3.3.

3.3 Generating Function Solution of Recurrences. Next, we examine the role that generating functions can play in the solution of recurrence relations, the second step in a classical approach to the analysis of algorithms: after a recurrence relationship describing some fundamental property of an algorithm is derived, generating functions can be used to solve the recurrence. Some readers may be familiar with this approach because of its widespread use and its basic and fundamental nature. We will see in Chapter 5 that it is often possible to avoid the recurrence and work with generating functions directly.

Generating functions provide a mechanical method for solving many recurrence relations. Given a recurrence describing some sequence $\{a_n\}_{n \geq 0}$, we can often develop a solution by carrying out the following steps:

- Multiply both sides of the recurrence by z^n and sum on n .
- Evaluate the sums to derive an equation satisfied by the OGF.
- Solve the equation to derive an explicit formula for the OGF.

- Express the OGF as a power series to get expressions for the coefficients (members of the original sequence).

The same method applies for EGFs, where we multiply by $z^n/n!$ and sum on n in the first step. Whether OGFs or EGFs are more convenient depends on the recurrence.

The most straightforward example of this method is its use in solving linear recurrences with constant coefficients (see Chapter 2).

Trivial linear recurrence. To solve the recurrence

$$a_n = a_{n-1} + 1 \quad \text{for } n \geq 1 \text{ with } a_0 = 0$$

we first multiply by z^n and sum to get

$$\sum_{n \geq 1} a_n z^n = \sum_{n \geq 1} a_{n-1} z^n + \frac{z}{1-z}.$$

In terms of the generating function $A(z) = \sum_{n \geq 0} a_n z^n$, this equation says

$$A(z) = zA(z) + \frac{z}{1-z}$$

or $A(z) = z/(1-z)^2$, and $a_n = n$, as expected.

Simple exponential recurrence. To solve the recurrence

$$a_n = 2a_{n-1} + 1 \quad \text{for } n \geq 1 \text{ with } a_0 = 1$$

we proceed as above to find that the generating function $A(z) = \sum_{n \geq 0} a_n z^n$ satisfies

$$A(z) - 1 = 2zA(z) + \frac{z}{1-z},$$

which simplifies to

$$A(z) = \frac{1}{(1-z)(1-2z)}.$$

From Table 3.1 we know that $1/(1-2z)$ is the generating function for the sequence $\{2^n\}$, and from Table 3.2 we know that multiplying by $1/(1-z)$ corresponds to taking partial sums:

$$a_n = \sum_{0 \leq k \leq n} 2^k = 2^{n+1} - 1.$$

Partial fractions. An alternative method for finding the coefficients in the preceding problem that is instructive preparation for more difficult problems is to use the *partial fractions* expansion for $A(z)$. By factoring the denominator, the generating function can be expressed as the sum of two fractions

$$\frac{1}{(1-z)(1-2z)} = \frac{c_0}{1-2z} + \frac{c_1}{1-z}$$

where c_0 and c_1 are constants to be determined. Cross-multiplying, we see that these constants must satisfy the simultaneous equations

$$\begin{aligned} c_0 + c_1 &= 1 \\ -c_0 - 2c_1 &= 0 \end{aligned}$$

so $c_0 = 2$ and $c_1 = -1$. Therefore,

$$[z^n] \frac{1}{(1-z)(1-2z)} = [z^n] \left(\frac{2}{1-2z} - \frac{1}{1-z} \right) = 2^{n+1} - 1.$$

This technique can be applied whenever we have a polynomial in the denominator, and leads to a general method for solving high-order linear recurrences that we discuss later in this section.

Fibonacci numbers. The generating function $F(z) = \sum_{k \geq 0} F_k z^k$ for the Fibonacci sequence

$$F_n = F_{n-1} + F_{n-2} \quad \text{for } n > 1 \text{ with } F_0 = 0 \text{ and } F_1 = 1$$

satisfies

$$F(z) = zF(z) + z^2F(z) + z.$$

This implies that

$$F(z) = \frac{z}{1-z-z^2} = \frac{1}{\sqrt{5}} \left(\frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi} z} \right)$$

by partial fractions, since $1-z-z^2$ factors as $(1-z\phi)(1-z\hat{\phi})$ where

$$\phi = \frac{1+\sqrt{5}}{2} \quad \text{and} \quad \hat{\phi} = \frac{1-\sqrt{5}}{2}$$

are the reciprocals of the roots of $1-z-z^2$. Now the series expansion is straightforward from Table 3.4:

$$F_n = \frac{1}{\sqrt{5}} (\phi^n - \hat{\phi}^n).$$

Of course, this strongly relates to the derivation given in Chapter 2. We examine this relationship in general terms next.

Exercise 3.15 Find the EGF for the Fibonacci numbers.

High-order linear recurrences. Generating functions make explicit the “factoring” process described in Chapter 2 to solve high-order recurrences with constant coefficients. Factoring the recurrence corresponds to factoring the polynomial that arises in the denominator of the generating function, which leads to a partial fraction expansion and an explicit solution. For example, the recurrence

$$a_n = 5a_{n-1} - 6a_{n-2} \quad \text{for } n > 1 \text{ with } a_0 = 0 \text{ and } a_1 = 1$$

implies that the generating function $a(z) = \sum_{n \geq 0} a_n z^n$ is

$$a(z) = \frac{z}{1 - 5z + 6z^2} = \frac{z}{(1 - 3z)(1 - 2z)} = \frac{1}{1 - 3z} - \frac{1}{1 - 2z}$$

so that we must have $a_n = 3^n - 2^n$.

Exercise 3.16 Use generating functions to solve the following recurrences:

$$\begin{aligned} a_n &= -a_{n-1} + 6a_{n-2} && \text{for } n > 1 \text{ with } a_0 = 0 \text{ and } a_1 = 1; \\ a_n &= 11a_{n-2} - 6a_{n-3} && \text{for } n > 2 \text{ with } a_0 = 0 \text{ and } a_1 = a_2 = 1; \\ a_n &= 3a_{n-1} - 4a_{n-2} && \text{for } n > 1 \text{ with } a_0 = 0 \text{ and } a_1 = 1; \\ a_n &= a_{n-1} - a_{n-2} && \text{for } n > 1 \text{ with } a_0 = 0 \text{ and } a_1 = 1. \end{aligned}$$

In general, the explicit expression for the generating function is the ratio of two polynomials; then partial fraction expansion involving roots of the denominator polynomial leads to an expression in terms of powers of roots. A precise derivation along these lines can be used to obtain a proof of Theorem 2.2.

Theorem 3.3 (OGFs for linear recurrences). If a_n satisfies the recurrence

$$a_n = x_1 a_{n-1} + x_2 a_{n-2} + \dots + x_t a_{n-t}$$

for $n \geq t$, then the generating function $a(z) = \sum_{n \geq 0} a_n z^n$ is a rational function $a(z) = f(z)/g(z)$, where the denominator polynomial is $g(z) = 1 - x_1 z - x_2 z^2 - \dots - x_t z^t$ and the numerator polynomial is determined by the initial values a_0, a_1, \dots, a_{t-1} .

Proof. The proof follows the general paradigm for solving recurrences described at the beginning of this section. Multiplying both sides of the recurrence by z^n and summing for $n \geq t$ yields

$$\sum_{n \geq t} a_n z^n = x_1 \sum_{n \geq t} a_{n-1} z^n + \cdots + x_t \sum_{n \geq t} a_{n-t} z^n.$$

The left-hand side evaluates to $a(z)$ minus the generating polynomial of the initial values; the first sum on the right evaluates to $za(z)$ minus a polynomial, and so forth. Thus $a(z)$ satisfies

$$a(z) - u_0(z) = (x_1 za(z) - u_1(z)) + \cdots + (x_t z^t a(z) - u_t(z)),$$

where the polynomials $u_0(z), u_1(z), \dots, u_t(z)$ are of degree at most $t - 1$ with coefficients depending only on the initial values a_0, a_1, \dots, a_{t-1} . This functional equation is linear.

Solving the equation for $a(z)$ gives the explicit form $a(z) = f(z)/g(z)$, where $g(z)$ has the form announced in the statement and

$$f(z) \equiv u_0(z) - u_1(z) - \cdots - u_t(z)$$

depends solely on the initial values of the recurrence and has degree less than t . ■

The general form immediately implies an alternate formulation for the dependence of $f(z)$ on the initial conditions, as follows. We have $f(z) = a(z)g(z)$ and we know that the degree of f is less than t . Therefore, we must have

$$f(z) = g(z) \sum_{0 \leq n < t} a_n z^n \pmod{z^t}.$$

This gives a shortcut to computing the coefficients of $f(z)$, which provides a quick exact solution to many recurrences.

Simple example. To solve the recurrence

$$a_n = 2a_{n-1} + a_{n-2} - 2a_{n-3} \quad \text{for } n > 2 \text{ with } a_0 = 0 \text{ and } a_1 = a_2 = 1$$

we first compute

$$g(z) = 1 - 2z - z^2 + 2z^3 = (1 - z)(1 + z)(1 - 2z)$$

then, using the initial conditions, we write

$$\begin{aligned} f(z) &= (z + z^2)(1 - 2z - z^2 + 2z^3) \pmod{z^3} \\ &= z - z^2 = z(1 - z). \end{aligned}$$

This gives

$$a(z) = \frac{f(z)}{g(z)} = \frac{z}{(1+z)(1-2z)} = \frac{1}{3} \left(\frac{1}{1-2z} - \frac{1}{1+z} \right),$$

so that $a_n = \frac{1}{3}(2^n - (-1)^n)$.

Cancellation. In the above recurrence, the $1 - z$ factor canceled, so there was no constant term in the solution. Consider the same recurrence with different initial conditions:

$$a_n = 2a_{n-1} + a_{n-2} - 2a_{n-3} \quad \text{for } n > 2 \text{ with } a_0 = a_1 = a_2 = 1.$$

The function $g(z)$ is the same as above, but now we have

$$\begin{aligned} f(z) &= (1 + z + z^2)(1 - 2z - z^2 + 2z^3) \pmod{z^3} \\ &= 1 - z - 2z^2 = (1 - 2z)(1 + z). \end{aligned}$$

In this case, we have cancellation to a trivial solution: $a(z) = f(z)/g(z) = 1/(1 - z)$ and $a_n = 1$ for all $n \geq 0$. The initial conditions can have drastic effects on the eventual growth rate of the solution by leading to cancellation of factors in this way.

We adopt the convention of factoring $g(z)$ in the form

$$g(z) = (1 - \beta_1 z) \cdot (1 - \beta_2 z) \cdots (1 - \beta_n z)$$

since it is slightly more natural in this context. Note that if a polynomial $g(z)$ satisfies $g(0) = 1$ (which is usual when $g(z)$ is derived from a recurrence as above), then the product of its roots is 1, and the $\beta_1, \beta_2, \dots, \beta_n$ in the equation above are simply the inverses of the roots. If $q(z)$ is the “characteristic polynomial” of Theorem 2.2, we have $g(z) = z^t q(1/z)$, so the β 's are the roots of the characteristic polynomial.

Complex roots. All the manipulations that we have been doing are valid for complex roots, as illustrated by the recurrence

$$a_n = 2a_{n-1} - a_{n-2} + 2a_{n-3} \quad \text{for } n > 2 \text{ with } a_0 = 1, a_1 = 0, \text{ and } a_2 = -1.$$

This gives

$$g(z) = 1 - 2z + z^2 - 2z^3 = (1 + z^2)(1 - 2z)$$

and

$$f(z) = (1 - z^4)(1 - 2z) \pmod{z^4} = 1 - 2z,$$

so

$$a(z) = \frac{f(z)}{g(z)} = \frac{1}{1 + z^2} = \frac{1}{2} \left(\frac{1}{1 - iz} - \frac{1}{1 + iz} \right),$$

and $a_n = \frac{1}{2}(i^n + (-i)^n)$. From this, it is easy to see that a_n is 0 for n odd, 1 when n is a multiple of 4, and -1 when n is even but not a multiple of 4 (this also follows directly from the form $a(z) = 1/(1 + z^2)$). For the initial conditions $a_0 = 1$, $a_1 = 2$, and $a_2 = 3$, we get $f(z) = 1$, so the solution grows like 2^n , but with periodic varying terms caused by the complex roots.

Multiple roots. When multiple roots are involved, we finish the derivation with the expansions given on the second and third lines of Table 3.1. For example, the recurrence

$$a_n = 5a_{n-1} - 8a_{n-2} + 4a_{n-3} \quad \text{for } n > 2 \text{ with } a_0 = 0, a_1 = 1, \text{ and } a_2 = 4$$

gives

$$g(z) = 1 - 5z + 8z^2 - 4z^3 = (1 - z)(1 - 2z)^2$$

and

$$f(z) = (z + 4z^2)(1 - 5z + 8z^2 - 4z^3) \pmod{z^3} = z(1 - z),$$

so $a(z) = z/(1 - 2z)^2$ and $a_n = n2^{n-1}$ from Table 3.1.

These examples illustrate a straightforward general method for developing exact solutions to linear recurrences:

- Derive $g(z)$ from the recurrence.
- Compute $f(z)$ from $g(z)$ and the initial conditions.

- Eliminate common factors in $f(z)/g(z)$.
- Use partial fractions to represent $f(z)/g(z)$ as a linear combination of terms of the form $(1 - \beta z)^{-j}$.
- Expand each term in the partial fractions expansion, using

$$[z^n](1 - \beta z)^{-j} = \binom{n+j-1}{j-1} \beta^n.$$

In essence, this process amounts to a constructive proof of Theorem 2.2.

Exercise 3.17 Solve the recurrence

$$a_n = 5a_{n-1} - 8a_{n-2} + 4a_{n-3} \quad \text{for } n > 2 \text{ with } a_0 = 1, a_1 = 2, \text{ and } a_2 = 4.$$

Exercise 3.18 Solve the recurrence

$$a_n = 2a_{n-2} - a_{n-4} \quad \text{for } n > 4 \text{ with } a_0 = a_1 = 0 \text{ and } a_2 = a_3 = 1.$$

Exercise 3.19 Solve the recurrence

$$a_n = 6a_{n-1} - 12a_{n-2} + 18a_{n-3} - 27a_{n-4} \quad \text{for } n > 4$$

with $a_0 = 0$ and $a_1 = a_2 = a_3 = 1$.

Exercise 3.20 Solve the recurrence

$$a_n = 3a_{n-1} - 3a_{n-2} + a_{n-3} \quad \text{for } n > 2 \text{ with } a_0 = a_1 = 0 \text{ and } a_2 = 1.$$

Solve the same recurrence with the initial condition on a_1 changed to $a_1 = 1$.

Exercise 3.21 Solve the recurrence

$$a_n = - \sum_{1 \leq k \leq t} \binom{t}{k} (-1)^k a_{n-k} \quad \text{for } n \geq t$$

with $a_0 = \dots = a_{t-2} = 0$ and $a_{t-1} = 1$.

Solving the quicksort recurrence with an OGF. When coefficients in a recurrence are polynomials in the index n , then the implied relationship constraining the generating function is a differential equation. As an example, let us revisit the basic recurrence from Chapter 1 describing the number of comparisons used by quicksort:

$$NC_N = N(N + 1) + 2 \sum_{1 \leq k \leq N} C_{k-1} \quad \text{for } N \geq 1 \text{ with } C_0 = 0. \quad (1)$$

We define the generating function

$$C(z) = \sum_{N \geq 0} C_N z^N \quad (2)$$

and proceed as described earlier to get a functional equation that $C(z)$ must satisfy. First, multiply both sides of (1) by z^N and sum on N to get

$$\sum_{N \geq 1} NC_N z^N = \sum_{N \geq 1} N(N + 1)z^N + 2 \sum_{N \geq 1} \sum_{1 \leq k \leq N} C_{k-1} z^N.$$

Now, we can evaluate each of these terms in a straightforward manner. The left-hand side is $zC'(z)$ (differentiate both sides of (2) and multiply by z) and the first term on the right is $2z/(1-z)^3$ (see Table 3.1). The remaining term, the double sum, is a partial sum convolution (see Table 3.2) that evaluates to $zC(z)/(1-z)$. Therefore, our recurrence relationship corresponds to a differential equation on the generating function

$$C'(z) = \frac{2}{(1-z)^3} + 2 \frac{C(z)}{1-z}. \quad (3)$$

We obtain the solution to this differential equation by solving the corresponding homogeneous equation $\rho'(z) = 2\rho(z)/(1-z)$ to get an “integration factor” $\rho(z) = 1/(1-z)^2$. This gives

$$\begin{aligned} ((1-z)^2 C(z))' &= (1-z)^2 C'(z) - 2(1-z)C(z) \\ &= (1-z)^2 \left(C'(z) - 2 \frac{C(z)}{1-z} \right) = \frac{2}{1-z}. \end{aligned}$$

Integrating, we get the result

$$C(z) = \frac{2}{(1-z)^2} \ln \frac{1}{1-z}. \quad (4)$$

Theorem 3.4 (Quicksort OGF). The average number of comparisons used by quicksort for a random permutation is given by

$$C_N = [z^N] \frac{2}{(1-z)^2} \ln \frac{1}{1-z} = 2(N+1)(H_{N+1} - 1).$$

Proof. The preceding discussion yields the explicit expression for the generating function, which completes the third step of the general procedure for solving recurrences with OGFs given at the beginning of this section. To extract coefficients, differentiate the generating function for the harmonic numbers. ■

The general approach for solving recurrences with OGFs that we have been discussing, while powerful, certainly cannot be relied upon to give solutions for *all* recurrence relations: various examples from the end of Chapter 2 can serve testimony to that. For some problems, it may not be possible to evaluate the sums to a simple form; for others, an explicit formula for the generating function can be difficult to derive; and for others, the expansion back to power series can present the main obstacle. In many cases, algebraic manipulations on the recurrence can simplify the process. In short, solving recurrences is not quite as “automatic” a process as we might like.

Exercise 3.22 Use generating functions to solve the recurrence

$$na_n = (n-2)a_{n-1} + 2 \quad \text{for } n > 1 \text{ with } a_1 = 1.$$

Exercise 3.23 [Greene and Knuth [6]] Solve the recurrence

$$na_n = (n+t-1)a_{n-1} \quad \text{for } n > 0 \text{ with } a_0 = 1.$$

Exercise 3.24 Solve the recurrence

$$a_n = n + 1 + \frac{t}{n} \sum_{1 \leq k \leq n} a_{k-1} \quad \text{for } n \geq 1 \text{ with } a_0 = 0$$

for $t = 2 - \epsilon$ and $t = 2 + \epsilon$, where ϵ is a small positive constant.

3.4 Expanding Generating Functions. Given an explicit functional form for a generating function, we would like a general mechanism for finding the associated sequence. This process is called “expanding” the generating function, as we take it from a compact functional form into an infinite series of terms. As we have seen in the preceding examples, we can handle many functions with algebraic manipulations involving the basic identities and transformations given in Tables 3.1–3.4. But where do the elementary expansions in Table 3.1 and Table 3.3 originate?

The *Taylor theorem* permits us to expand a function $f(z)$ given its derivatives at 0:

$$f(z) = f(0) + f'(0)z + \frac{f''(0)}{2!}z^2 + \frac{f'''(0)}{3!}z^3 + \frac{f''''(0)}{4!}z^4 + \dots$$

Thus, by calculating derivatives, we can, in principle, find the sequence associated with any given generating function.

Exponential sequence. Since all the derivatives of e^z are e^z , the easiest application of Taylor’s theorem is the fundamental expansion

$$e^z = 1 + z + \frac{z^2}{2!} + \frac{z^3}{3!} + \frac{z^4}{4!} + \dots$$

Geometric sequence. From Table 3.1, we know that the generating function for the sequence $\{1, c, c^2, c^3, \dots\}$ is $(1 - cz)^{-1}$. The k th derivative of $(1 - cz)^{-1}$ is $k!c^k(1 - cz)^{-k-1}$, which is simply $k!c^k$ when evaluated at $z = 0$, so Taylor’s theorem verifies that the expansion of this function is given by

$$\frac{1}{1 - cz} = \sum_{k \geq 0} c^k z^k,$$

as stated in Table 3.1.

Binomial theorem. The k th derivative of the function $(1 + z)^x$ is

$$x(x - 1)(x - 2) \cdots (x - k + 1)(1 + z)^{x-k},$$

so by Taylor’s theorem, we get a generalized version of the binomial theorem known as *Newton’s formula*:

$$(1 + z)^x = \sum_{k \geq 0} \binom{x}{k} z^k,$$

where the binomial coefficients are defined by

$$\binom{x}{k} \equiv x(x-1)(x-2) \cdots (x-k+1)/k!.$$

A particularly interesting case of this is

$$\frac{1}{\sqrt{1-4z}} = \sum_{k \geq 0} \binom{2k}{k} z^k,$$

which follows from the identity

$$\begin{aligned} \binom{-1/2}{k} &= \frac{-\frac{1}{2}(-\frac{1}{2}-1)(-\frac{1}{2}-2) \cdots (-\frac{1}{2}-k+1)}{k!} \\ &= \frac{(-1)^k}{2^k} \frac{1 \cdot 3 \cdot 5 \cdots (2k-1)}{k!} \frac{2 \cdot 4 \cdot 6 \cdots 2k}{2^k k!} \\ &= \frac{(-1)^k}{4^k} \binom{2k}{k}. \end{aligned}$$

An expansion closely related to this plays a central role in the analysis of algorithms, as we will see in several applications later in the book.

Exercise 3.25 Use Taylor's theorem to find the expansions of the following functions:

$$\sin(z), \quad 2^z, \quad ze^z.$$

Exercise 3.26 Use Taylor's theorem to verify that the coefficients of the series expansion of $(1 - az - bz^2)^{-1}$ satisfy a second-order linear recurrence with constant coefficients.

Exercise 3.27 Use Taylor's theorem to verify directly that

$$H(z) = \frac{1}{1-z} \ln \frac{1}{1-z}$$

is the generating function for the harmonic numbers.

Exercise 3.28 Find an expression for

$$[z^n] \frac{1}{\sqrt{1-z}} \ln \frac{1}{1-z}.$$

(Hint: Expand $(1-z)^{-\alpha}$ and differentiate with respect to α .)

Exercise 3.29 Find an expression for

$$[z^n] \left(\frac{1}{1-z} \right)^t \ln \frac{1}{1-z} \quad \text{for integer } t > 0.$$

In principle, we can always compute generating function coefficients by direct application of Taylor's theorem, but the process can become too complex to be helpful. Most often, we expand a generating function by decomposing it into simpler parts for which expansions are known, as we have done for several examples earlier, including the use of convolutions to expand the generating functions for binomial coefficients and the harmonic numbers and the use of partial fraction decomposition to expand the generating function for the Fibonacci numbers. Indeed, this is the method of choice, and we will be using it extensively throughout this book. For specific classes of problems, other tools are available to aid in this process—for example the *Lagrange inversion theorem*, which we will examine in §6.12.

Moreover, there exists something even more useful than a “general tool” for expanding generating functions to derive succinct representations for coefficients: a tool for directly deriving asymptotic estimates of coefficients, which allows us to ignore irrelevant detail, even for problems that may not seem amenable to expansion by decomposition. Though the general method involves complex analysis and is beyond the scope of this book, our use of partial fractions expansions for linear recurrences is based on the same intuition. For example, the partial fraction expansion of the Fibonacci numbers immediately implies that the generating function $F(z)$ does not converge when $z = 1/\phi$ or $z = 1/\hat{\phi}$. But it turns out that these “singularities” completely determine the asymptotic growth of the coefficients F_N . In this case, we are able to verify by direct expansion that the coefficients grow as ϕ^N (to within a constant factor). It is possible to state general conditions under which coefficients grow in this way and general mechanisms for determining other growth rates. By analyzing singularities of generating functions, we are very often able to reach our goal of deriving accurate estimates of the quantities of interest without having to resort to detailed expansions. This topic is discussed in §5.5, and in detail in [3].

But there are a large number of sequences for which the generating functions are known and for which simple algebraic manipulations of the generating function can yield simple expressions for the quantities of interest. Basic

generating functions for classic combinatorial sequences are discussed in further detail in this chapter, and Chapters 6 through 9 are largely devoted to building up a repertoire of familiar functions that arise in the analysis of combinatorial algorithms. We will proceed to discuss and consider detailed manipulations of these functions, secure in the knowledge that we have powerful tools available for getting the coefficients back, when necessary.

3.5 Transformations with Generating Functions. Generating functions succinctly represent infinite sequences. Often, their importance lies in the fact that simple manipulations on equations involving the generating function can lead to surprising relationships involving the underlying sequences that otherwise might be difficult to derive. Several basic examples of this follow.

Vandermonde's convolution. This identity relating binomial coefficients (see Chapter 2),

$$\sum_k \binom{r}{k} \binom{s}{N-k} = \binom{r+s}{N},$$

is trivial to derive, as it is the convolution of coefficients that express the functional relation

$$(1+z)^r(1+z)^s = (1+z)^{r+s}.$$

Similar identities can be derived in abundance from more complicated convolutions.

Quicksort recurrence. Multiplying OGFs by $(1-z)$ corresponds to differencing the coefficients, as stated in Table 3.2, and as we saw in Chapter 1 (without remarking on it) in the quicksort recurrence. Other transformations were involved to get the solution. Our point here is that these various manipulations are more easily done with the generating function representation than with the sequence representation. We will examine this in more detail later in this chapter.

Fibonacci numbers. The generating function for Fibonacci numbers can be written

$$F(z) = \frac{z}{1-y} \quad \text{with } y = z + z^2.$$

Expanding this in terms of y , we have

$$\begin{aligned}F(z) &= z \sum_{N \geq 0} y^N = z \sum_{N \geq 0} (z + z^2)^N \\&= \sum_{N \geq 0} \sum_k \binom{N}{k} z^{N+k+1}.\end{aligned}$$

But F_N is simply the coefficient of z^N in this, so we must have

$$F_N = \sum_k \binom{N-k-1}{k},$$

a well-known relationship between Fibonacci numbers and diagonals in Pascal's triangle.

Binomial transform. If $a^n = (1-b)^n$ for all n , then, obviously, $b^n = (1-a)^n$. Surprisingly, this generalizes to arbitrary sequences: given two sequences $\{a_n\}$ and $\{b_n\}$ related according to the equation

$$a_n = \sum_k \binom{n}{k} (-1)^k b_k,$$

we know that the associated generating functions satisfy $B(-z) = e^z A(z)$ (see Table 3.4). But then, of course, $A(-z) = e^z B(z)$, which implies that

$$b_n = \sum_k \binom{n}{k} (-1)^k a_k.$$

We will see more examples of such manipulations in ensuing chapters.

Exercise 3.30 Show that

$$\sum_k \binom{2k}{k} \binom{2N-2k}{N-k} = 4^N.$$

Exercise 3.31 What recurrence on $\{C_N\}$ corresponds to multiplying both sides of the differential equation (3) for the quicksort generating function by $(1-z)^2$?

Exercise 3.32 Suppose that an OGF satisfies the differential equation

$$A'(z) = -A(z) + \frac{A(z)}{1-z}.$$

What recurrence does this correspond to? Multiply both sides by $1-z$ and set coefficients equal to derive a different recurrence, then solve that recurrence. Compare this path to the solution with the method of directly finding the OGF and expanding.

Exercise 3.33 What identity on binomial coefficients is implied by the convolution

$$(1+z)^r(1-z)^s = (1-z^2)^s(1+z)^{r-s}$$

where $r > s$?

Exercise 3.34 Prove that

$$\sum_{0 \leq k \leq t} \binom{t-k}{r} \binom{k}{s} = \binom{t+1}{r+s+1}.$$

Exercise 3.35 Use generating functions to evaluate $\sum_{0 \leq k \leq N} F_k$.

Exercise 3.36 Use generating functions to find a sum expression for $[z^n] \frac{z}{1-e^z}$.

Exercise 3.37 Use generating functions to find a sum expression for $[z^n] \frac{1}{2-e^z}$.

Exercise 3.38 [Dobinski, cf. Comtet] Prove that

$$n![z^n]e^{e^z-1} = e^{-1} \sum_{k \geq 0} \frac{k^n}{n!}.$$

Exercise 3.39 Prove the binomial transform identity using OGFs. Let $A(z)$ and $B(z)$ be related by

$$B(z) = \frac{1}{1-z} A\left(\frac{z}{z-1}\right),$$

and then use the change of variable $z = y/(y-1)$.

Exercise 3.40 Prove the binomial transform identity directly, *without* using generating functions.

Exercise 3.41 [Faà di Bruno's formula, cf. Comtet] Let $f(z) = \sum_n f_n z^n$ and $g(z) = \sum_n g_n z^n$. Express $[z^n]f(g(z))$ using the multinomial theorem.

3.6 Functional Equations on Generating Functions. In the analysis of algorithms, recursion in an algorithm (or recurrence relationships in its analysis) very often leads to functional equations on the corresponding generating functions. We have seen some cases where we can find an explicit solution to the functional equation and then expand to find the coefficients. In other cases, we may be able to use the functional equation to determine the asymptotic behavior without ever finding an explicit form for the generating function, or to transform the problem to a similar form that can be more easily solved. We offer a few comments on the different types of functional equations in this section, along with some exercises and examples.

Linear. The generating function for the Fibonacci numbers is the prototypical example here:

$$f(z) = zf(z) + z^2f(z) + z.$$

The linear equation leads to an explicit formula for the generating function, which perhaps can be expanded. But *linear* here just refers to the function itself appearing only in linear combinations—the coefficients and consequent formulae could be arbitrarily complex.

Nonlinear. More generally, it is typical to have a situation where the generating function can be shown to be equal to an arbitrary function of itself, not necessarily a linear function. Famous examples of this include the GF for the Catalan numbers, which is defined by the functional equation

$$f(z) = zf(z)^2 + 1$$

and the GF for trees, which satisfies the functional equation

$$f(z) = ze^{f(z)}.$$

The former is discussed in some detail in §3.3 and the latter in §6.14. Depending on the nature of the nonlinear function, it may be possible to derive an explicit formula for the generating function algebraically.

Differential. The equation might involve derivatives of the generating function. We have already seen an example of this with quicksort,

$$f'(z) = \frac{2}{(1-z)^3} + 2\frac{f(z)}{1-z},$$

and will see a more detailed example below. Our ability to find an explicit formula for the generating function is, of course, directly related to our ability to solve the differential equation.

Compositional. In still other cases, the functional equation might involve linear or nonlinear functions on the *arguments* of the generating function of interest, as in the following examples from the analysis of algorithms:

$$\begin{aligned}f(z) &= e^{z/2} f(z/2) \\f(z) &= z + f(z^2 + z^3).\end{aligned}$$

The first is related to binary tries and radix-exchange sort (see Chapter 8), and the second counts 2–3 trees (see Chapter 6). Clearly, we could concoct arbitrarily complicated equations, with no assurance that solutions are readily available. Some general tools for attacking such equations are treated in [3].

These examples give some indication of what we can expect to encounter in the use of generating functions in the analysis of algorithms. We will be examining these and other functional equations on generating functions throughout the book. Often, such equations serve as a dividing line where detailed study of the algorithm leaves off and detailed application of analytic tools begins. However difficult the solution of the functional equation might appear, it is important to remember that we can use such equations to learn properties of the underlying sequence.

As with recurrences, the technique of *iteration*, simply applying the equation to itself successively, can often be useful in determining the nature of a generating function defined by a functional equation. For example, consider an EGF that satisfies the functional equation

$$f(z) = e^z f(z/2).$$

Then, provided that $f(0) = 1$, we must have

$$\begin{aligned}f(z) &= e^z e^{z/2} f(z/4) \\&= e^z e^{z/2} e^{z/4} f(z/8) \\\vdots \\&= e^{z+z/2+z/4+z/8+\dots} \\&= e^{2z}.\end{aligned}$$

This proves that 2^n is the solution to the recurrence

$$f_n = \sum_k \binom{n}{k} \frac{f_k}{2^k} \quad \text{for } n > 0 \text{ with } f_0 = 1.$$

Technically, we need to justify carrying out the iteration indefinitely, but the solution is easily verified from the original recurrence.

Exercise 3.42 Show that the coefficients f_n in the expansion

$$e^{z+z^2/2} = \sum_{n \geq 0} f_n \frac{z^n}{n!}$$

satisfy the second-order linear recurrence $f_n = f_{n-1} + (n-1)f_{n-2}$. (*Hint:* Find a differential equation satisfied by the function $f(z) = e^{z+z^2/2}$.)

Exercise 3.43 Solve

$$f(z) = e^{-z} f\left(\frac{z}{2}\right) + e^{2z} - 1$$

and, assuming that $f(z)$ is an EGF, derive the corresponding recurrence and solution.

Exercise 3.44 Find an explicit formula for the OGF of the sequence satisfying the divide-and-conquer recurrence

$$f_{2n} = f_{2n-1} + f_n \quad \text{for } n > 1 \text{ with } f_0 = 0;$$

$$f_{2n+1} = f_{2n} \quad \text{for } n > 0 \text{ with } f_1 = 1.$$

Exercise 3.45 Iterate the following equation to obtain an explicit formula for $f(z)$:

$$f(z) = 1 + zf\left(\frac{z}{1+z}\right).$$

Exercise 3.46 [Polya] Given $f(z)$ defined by the equation

$$f(z) = \frac{z}{1-f(z^2)},$$

find explicit expressions for $a(z)$ and $b(z)$ with $f(z) = a(z)/b(z)$.

Exercise 3.47 Prove that there is only one power series of the form $f(z) = \sum_{n \geq 1} f_n z^n$ that satisfies $f(z) = \sin(f(z))$.

Exercise 3.48 Derive an underlying recurrence from the functional equation for 2–3 trees and use the recurrence to determine the number of 2–3 trees of 100 nodes.

3.7 Solving the Quicksort Median-of-Three Recurrence with OGFs.

As a detailed example of manipulating functional equations on generating functions, we revisit the recurrence given in §1.5 that describes the average number of comparisons taken by the median-of-three quicksort. This recurrence would be difficult to handle without generating functions:

$$C_N = N + 1 + \sum_{1 \leq k \leq N} \frac{(N-k)(k-1)}{\binom{N}{3}} (C_{k-1} + C_{N-k}) \quad \text{for } N > 2$$

with $C_0 = C_1 = C_2 = 0$. We use $N + 1$ as the number of comparisons required to partition N elements for convenience in the analysis. The actual cost depends on how the median is computed and other properties of the implementation, but it will be within a small additive constant of $N + 1$. Also, the initial condition $C_2 = 0$ (and the implied $C_3 = 4$) is used for convenience in the analysis, though different costs are likely in actual implementations. As in §1.5, we can account for such details by taking linear combinations of the solution to this recurrence and other, similar, recurrences such as the one counting the number of partitioning stages (the same recurrence with cost 1 instead of $N + 1$).

We follow through the standard steps for solving recurrences with generating functions. Multiplying by $\binom{N}{3}$ and removing the symmetry in the sum, we have

$$\binom{N}{3} C_N = (N+1) \binom{N}{3} + 2 \sum_{1 \leq k \leq N} (N-k)(k-1) C_{k-1}.$$

Then, multiplying both sides by z^{N-3} and summing on N eventually leads to the differential equation:

$$C'''(z) = \frac{24}{(1-z)^5} + 12 \frac{C'(z)}{(1-z)^2}. \quad (5)$$

One cannot always hope to find explicit solutions for high-order differential equations, but this one is in fact of a type that can be solved explicitly. First, multiply both sides by $(1-z)^3$ to get

$$(1-z)^3 C'''(z) = 12(1-z)C'(z) + \frac{24}{(1-z)^2}. \quad (6)$$

Now, in this equation the degree equals the order of each term. Such a differential equation is known in the theory of ordinary differential equations as an *Euler equation*. We can decompose it by rewriting it in terms of an operator that both multiplies and differentiates. In this case, we define the operator

$$\Psi C(z) \equiv (1 - z) \frac{d}{dz} C(z),$$

which allows us to rewrite (6) as

$$\Psi(\Psi + 1)(\Psi + 2)C(z) = 12\Psi C(z) + \frac{24}{(1 - z)^2}.$$

Collecting all the terms involving Ψ into one polynomial and factoring, we have

$$\Psi(\Psi + 5)(\Psi - 2)C(z) = \frac{24}{(1 - z)^2}.$$

The implication of this equation is that we can solve for $C(z)$ by successively solving three first-order differential equations:

$$\begin{aligned} \Psi U(z) &= \frac{24}{(1 - z)^2} & \text{or} & & U'(z) &= \frac{24}{(1 - z)^3}, \\ (\Psi + 5)T(z) &= U(z) & \text{or} & & T'(z) &= -5 \frac{T(z)}{1 - z} + \frac{U(z)}{1 - z}, \\ (\Psi - 2)C(z) &= T(z) & \text{or} & & C'(z) &= 2 \frac{C(z)}{1 - z} + \frac{T(z)}{1 - z}. \end{aligned}$$

Solving these first-order differential equations exactly as for the simpler case that we solved to analyze regular quick sort, we arrive at the solution.

Theorem 3.5 (Median-of-three Quicksort). The average number of comparisons used by the median-of-three quicksort for a random permutation is given by

$$C_N = \frac{12}{7}(N + 1)\left(H_{N+1} - \frac{23}{14}\right) \quad \text{for } N \geq 6.$$

Proof. Continuing the earlier discussion, we solve the differential equations to get the result

$$\begin{aligned} U(z) &= \frac{12}{(1-z)^2} - 12; \\ T(z) &= \frac{12}{7} \frac{1}{(1-z)^2} - \frac{12}{5} + \frac{24}{35}(1-z)^5; \\ C(z) &= \frac{12}{7} \frac{1}{(1-z)^2} \ln \frac{1}{1-z} - \frac{54}{49} \frac{1}{(1-z)^2} + \frac{6}{5} - \frac{24}{245}(1-z)^5. \end{aligned}$$

Expanding this expression for $C(z)$ (and ignoring the last term) gives the result (see the exercises in §3.1). The leading term in the OGF differs from the OGF for standard quicksort only by a constant factor. ■

We can translate the decomposition into $U(z)$ and $T(z)$ into recurrences on the corresponding sequences. Consider the generating functions $U(z) = \sum U_N z^N$ and $T(z) = \sum T_N z^N$. In this case, manipulations on generating functions do correspond to manipulations on recurrences, but the tools used are more generally applicable and somewhat easier to discover and apply than would be a direct solution of the recurrence. Furthermore, the solution with generating functions can be used in the situation when a larger sample is used. Further details may be found in [9] or [14].

Besides serving as a practical example of the use of generating functions, this rather detailed example illustrates how precise mathematical statements about performance characteristics of interest can be used to help choose proper values for controlling parameters of algorithms (in this case, the size of the sample). For instance, the above analysis shows that we save about 14% of the cost of comparisons by using the median-of-three variant for quicksort, and a more detailed analysis, taking into account the extra costs (primarily, the extra exchanges required because the partitioning element is nearer the middle), shows that bigger samples lead to marginal further improvements.

Exercise 3.49 Show that $(1-z)^t C^{(t)}(z) = \Psi(\Psi+1) \dots (\Psi+t+1) C(z)$.

Exercise 3.50 Find the average number of exchanges used by median-of-three quicksort.

Exercise 3.51 Find the number of comparisons and exchanges used, on the average, by quicksort when modified to use the median of *five* elements for partitioning.

Exercise 3.52 [Euler] Discuss the solution of the differential equation

$$\sum_{0 \leq j \leq r} (1-z)^{r-j} \frac{d^j}{dz^j} f(z) = 0$$

and the inhomogeneous version where the right-hand side is of the form $(1-z)^\alpha$.

Exercise 3.53 [van Emden, cf. Knuth] Show that, when the median of a sample of $2t+1$ elements is used for partitioning, the number of comparisons used by quicksort is

$$\frac{1}{H_{2t+2} - H_{t+1}} N \ln N + O(N).$$

3.8 Counting with Generating Functions. So far, we have concentrated on describing generating functions as analytic tools for solving recurrence relationships. This is only part of their significance—they also provide a way to count combinatorial objects systematically. The “combinatorial objects” may be data structures being operated upon by algorithms, so this process plays a fundamental role in the analysis of algorithms as well.

Our first example is a classical combinatorial problem that also corresponds to a fundamental data structure that will be considered in Chapter 6 and in several other places in the book. A *binary tree* is a structure defined recursively to be either a single *external node* or an *internal node* that is connected to two binary trees, a *left subtree* and a *right subtree*. Figure 3.1 shows the binary trees with five or fewer nodes. Binary trees appear in many problems in combinatorics and the analysis of algorithms: for example, if internal nodes correspond to two-argument arithmetic operators and external nodes correspond to variables, then binary trees correspond to arithmetic expressions. The question at hand is, how many binary trees are there with N external nodes?

Counting binary trees. One way to proceed is to define a recurrence. Let T_N be the number of binary trees with $N+1$ external nodes. From Figure 3.1 we know that $T_0 = 1$, $T_1 = 1$, $T_2 = 2$, $T_3 = 5$, and $T_4 = 14$. Now, we can derive a recurrence from the recursive definition: if the left subtree in a binary tree with $N+1$ external nodes has k external nodes (there are T_{k-1} different such trees), then the right subtree must have $N-k+1$ external nodes (there

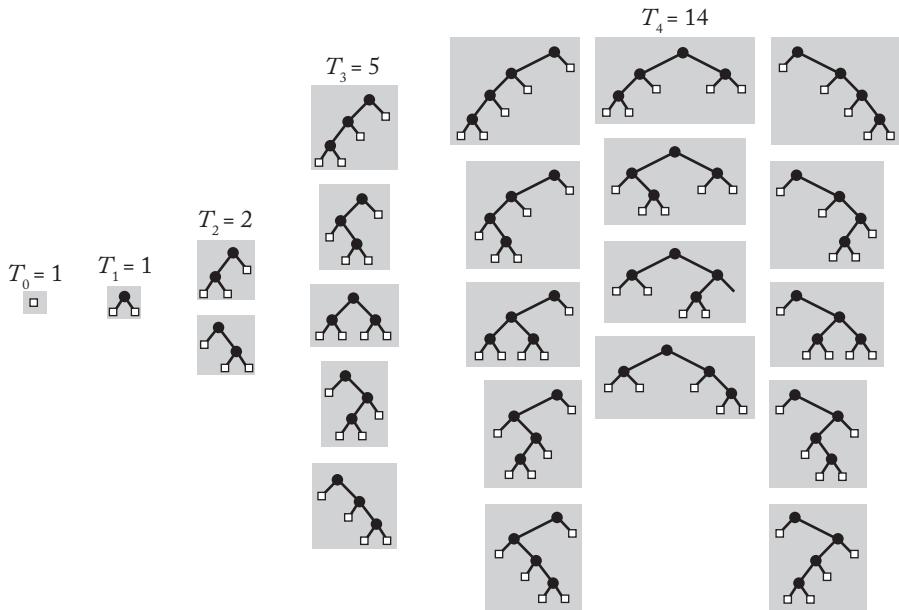


Figure 3.1 All binary trees with 1, 2, 3, 4, and 5 external nodes

are T_{N-k} possibilities), so T_N must satisfy

$$T_N = \sum_{1 \leq k \leq N} T_{k-1}T_{N-k} \quad \text{for } N > 0 \text{ with } T_0 = 1.$$

This is a simple convolution: multiplying by z^N and summing on N , we find that the corresponding OGF must satisfy the nonlinear functional equation

$$T(z) = zT(z)^2 + 1.$$

This formula for $T(z)$ is easily solved with the quadratic equation:

$$zT(z) = \frac{1}{2}(1 \pm \sqrt{1 - 4z}).$$

To get equality when $z = 0$, we take the solution with a minus sign.

Theorem 3.6 (OGF for binary trees). The number of binary trees with $N + 1$ external nodes is given by the Catalan numbers:

$$T_N = [z^{N+1}] \frac{1 - \sqrt{1 - 4z}}{2} = \frac{1}{N+1} \binom{2N}{N}.$$

Proof. The explicit representation of the OGF was derived earlier. To extract coefficients, use the binomial theorem with exponent 1/2 (Newton's formula):

$$zT(z) = -\frac{1}{2} \sum_{N \geq 1} \binom{1/2}{N} (-4z)^N.$$

Setting coefficients equal gives

$$\begin{aligned} T_N &= -\frac{1}{2} \binom{1/2}{N+1} (-4)^{N+1} \\ &= -\frac{1}{2} \frac{\frac{1}{2}(\frac{1}{2}-1)(\frac{1}{2}-2)\dots(\frac{1}{2}-N)(-4)^{N+1}}{(N+1)!} \\ &= \frac{1 \cdot 3 \cdot 5 \cdots (2N-1) \cdot 2^N}{(N+1)!} \\ &= \frac{1}{N+1} \frac{1 \cdot 3 \cdot 5 \cdots (2N-1)}{N!} \frac{2 \cdot 4 \cdot 6 \cdots 2N}{1 \cdot 2 \cdot 3 \cdots N} \\ &= \frac{1}{N+1} \binom{2N}{N}. \end{aligned}$$

■

As we will see in Chapter 6, every binary tree has exactly one more external node than internal node, so the Catalan numbers T_N also count the binary trees with N internal nodes. In the next chapter, we will see that the approximate value is $T_N \approx 4^N / N\sqrt{\pi N}$.

Counting binary trees (direct). There is a simpler way to determine the explicit expression for the generating function above, which gives more insight into the intrinsic utility of generating functions for counting. We define \mathcal{T} to be the set of all binary trees, and adopt the notation $|t|$ to represent, for

$t \in \mathcal{T}$, the number of internal nodes in t . Then we have the following derivation:

$$\begin{aligned} T(z) &= \sum_{t \in \mathcal{T}} z^{|t|} \\ &= 1 + \sum_{t_L \in \mathcal{T}} \sum_{t_R \in \mathcal{T}} z^{|t_L|+|t_R|+1}. \\ &= 1 + zT(z)^2 \end{aligned}$$

The first line is an alternative way to express $T(z)$ from its definition. Each tree with exactly k external nodes contributes exactly 1 to the coefficient of z^k , so the coefficient of z^k in the sum “counts” the number of trees with k internal nodes. The second line follows from the recursive definition of binary trees: either a binary tree has no internal nodes (which accounts for the 1), or it can be decomposed into two independent binary trees whose internal nodes comprise the internal nodes of the original tree, plus one for the root. The third line follows because the index variables t_L and t_R are independent. Readers are advised to study this fundamental example carefully—we will be seeing many other similar examples throughout the book.

Exercise 3.54 Modify the above derivation to derive directly the generating function for the number of binary trees with N *external* nodes.

Changing a dollar (Polya). A classical example of counting with generating functions, due to Polya, is to answer the following question: “How many ways are there to change a dollar, using pennies, nickels, dimes, quarters, and fifty-cent coins?” Arguing as in the direct counting method for binary trees, we find that the generating function is given by

$$D(z) = \sum_{p,n,d,q,f \geq 0} z^{p+5n+10d+25q+50f}$$

The indices of summation p , n , d , and so on, are the number of pennies, nickels, dimes, and other coins used. Each configuration of coins that adds up to k cents clearly contributes exactly 1 to the coefficient of z^k , so this is the desired generating function. But the indices of summation are all independent in this expression for $D(z)$, so we have

$$\begin{aligned} D(z) &= \sum_p z^p \sum_n z^{5n} \sum_d z^{10d} \sum_q z^{25q} \sum_f z^{50f} \\ &= \frac{1}{(1-z)(1-z^5)(1-z^{10})(1-z^{25})(1-z^{50})}. \end{aligned}$$

By setting up the corresponding recurrence, or by using a computer algebra system, we find that $[z^{100}]D(z) = 292$.

Exercise 3.55 Discuss the form of an expression for $[z^N]D(z)$.

Exercise 3.56 Write an efficient computer program that can compute $[z^N]D(z)$, given N .

Exercise 3.57 Show that the generating function for the number of ways to express N as a linear combination (with integer coefficients) of powers of 2 is

$$\prod_{k \geq 1} \frac{1}{1 - z^{2^k}}.$$

Exercise 3.58 [Euler] Show that

$$\frac{1}{1-z} = (1+z)(1+z^2)(1+z^4)(1+z^8)\dots$$

Give a closed form for the product of the first t factors. This identity is sometimes called the “computer scientist’s identity.” Why?

Exercise 3.59 Generalize the previous exercise to base 3.

Exercise 3.60 Express $[z^N](1-z)(1-z^2)(1-z^4)(1-z^8)\dots$ in terms of the binary representation of N .

Binomial distribution. How many binary sequences of length N have exactly k bits that are 1 (and $N-k$ bits that are 0)? Let \mathcal{B}_N denote the set of all binary sequences of length N and \mathcal{B}_{Nk} denote the set of all binary sequences of length N with the property that k of the bits are 1. Now we consider the generating function for the quantity sought:

$$B_N(z) = \sum_k |\mathcal{B}_{Nk}| z^k.$$

But we can note that each binary string b in \mathcal{B}_N with exactly k 1s contributes exactly 1 to the coefficient of z^k and rewrite the generating function so that it “counts” each string:

$$B_N(z) \equiv \sum_{b \in \mathcal{B}_N} z^{\{\# \text{ of 1 bits in } b\}} = \sum_{b \in \mathcal{B}_{Nk}} z^k \left(= \sum_k |\mathcal{B}_{Nk}| z^k \right).$$

Now the set of all strings of N bits with k 1s can be formed by taking the union of the set of all strings with $N - 1$ bits and k 1s (adding a 0 to the beginning of each string) and the set of all strings with $N - 1$ bits and $k - 1$ 1s (adding a 1 to the beginning of each string). Therefore,

$$\begin{aligned} B_N(z) &= \sum_{b \in \mathcal{B}_{(N-1)k}} z^k + \sum_{b \in \mathcal{B}_{(N-1)(k-1)}} z^k \\ &= B_{N-1}(z) + zB_{N-1}(z) \end{aligned}$$

so $B_N(z) = (1 + z)^N$. Expanding this function with the binomial theorem yields the expected answer $|\mathcal{B}_{Nk}| = \binom{N}{k}$.

To summarize informally, we can use the following method to “count” with generating functions:

- Write down a general expression for the GF involving a sum indexed over the combinatorial objects to be counted.
- Decompose the sum in a manner corresponding to the structure of the objects, to derive an explicit formula for the GF.
- Express the GF as a power series to get expressions for the coefficients.

As we saw when introducing generating functions for the problem of counting binary trees at the beginning of the previous section, an alternative approach is to use the objects’ structure to derive a recurrence, then use GFs to solve the recurrence. For simple examples, there is little reason to choose one method over the other, but for more complicated problems, the direct method just sketched can avoid the tedious calculations that sometimes arise with recurrences. In Chapter 5, we will consider a powerful general approach based on this idea, and we will see many applications later in the book.

3.9 Probability Generating Functions. An application of generating functions that is directly related to the analysis of algorithms is their use for manipulating probabilities, to simplify the calculation of averages and variances.

Definition Given a random variable X that takes on only nonnegative integer values, with $p_k \equiv \Pr\{X = k\}$, the function $P(u) = \sum_{k \geq 0} p_k u^k$ is called the probability generating function (PGF) for the random variable.

We have been assuming basic familiarity with computing averages and standard deviations for random variables in the discussion in §1.7 and in the examples of average-case analysis of algorithms that we have examined, but we review the definitions here because we will be doing related calculations in this and the next section.

Definition The *expected value* of X , or $E(X)$, also known as the *mean value* of X , is defined to be $\sum_{k \geq 0} kp_k$. In terms of $r_k \equiv \Pr\{X \leq k\}$, this is equivalent to $E(X) = \sum_{k \geq 0} (1 - r_k)$. The variance of X , or $\text{var}(X)$, is defined to be $\sum_{k \geq 0} (k - E(X))^2 p_k$. The standard deviation of X is defined to be $\sqrt{\text{var}(X)}$.

Probability generating functions are important because they can provide a way to find the average and the variance without tedious calculations involving discrete sums.

Theorem 3.7 (Mean and variance from PGFs). Given a PGF $P(z)$ for a random variable X , the expected value of X is given by $P'(1)$ with variance $P''(1) + P'(1) - P'(1)^2$.

Proof. If $p_k \equiv \Pr\{X = k\}$, then

$$P'(1) = \sum_{k \geq 0} kp_k u^{k-1} |_{u=1} = \sum_{k \geq 0} kp_k,$$

the expected value, by definition. Similarly, noting that $P(1) = 1$, the stated result for the variance follows directly from the definition:

$$\begin{aligned} \sum_{k \geq 0} (k - P'(1))^2 p_k &= \sum_{k \geq 0} k^2 p_k - 2 \sum_{k \geq 0} k P'(1) p_k + \sum_{k \geq 0} P'(1)^2 p_k \\ &= \sum_{k \geq 0} k^2 p_k - P'(1)^2 = P''(1) + P'(1) - P'(1)^2. \end{aligned} \quad \blacksquare$$

The quantity $E(X^r) = \sum_k k^r p_k$ is known as the *rth moment* of X . The expected value is the first moment and the variance is the difference between the second moment and the square of the first.

Composition rules such as the theorems that we will consider in §5.2 and §5.3 for enumeration through the symbolic method translate into statements about combining PGFs for independent random variables. For example, if $P(u), Q(u)$ are probability generating functions for independent random variables X and Y , then $P(u)Q(u)$ is the probability generating function for $X + Y$. Moreover, the average and variance of the distribution represented by the product of two probability generating functions is the sum of the individual averages and variances.

Exercise 3.61 Give a simple expression for $\text{var}(X)$ in terms of $r_k = \Pr\{X \leq k\}$.

Exercise 3.62 Define $\text{mean}(P) \equiv P'(1)$ and $\text{var}(P) \equiv P''(1) + P'(1) - P'(1)^2$. Prove that $\text{mean}(PQ) = \text{mean}(P) + \text{mean}(Q)$ and $\text{var}(PQ) = \text{var}(P) + \text{var}(Q)$ for any differentiable functions P and Q with $P(1) = Q(1) = 1$, not just PGFs.

Uniform discrete distribution. Given an integer $n > 0$, suppose that X_n is a random variable that is equally likely to take on each of the integer values $0, 1, 2, \dots, n - 1$. Then the probability generating function for X_n is

$$P_n(u) = \frac{1}{n} + \frac{1}{n}u + \frac{1}{n}u^2 + \cdots + \frac{1}{n}u^{n-1},$$

the expected value is

$$P'_n(1) = \frac{1}{n}(1 + 2 + \cdots + (n - 1)) = \frac{n - 1}{2},$$

and, since

$$P''_n(1) = \frac{1}{n}(1 \cdot 2 + 2 \cdot 3 + \cdots + (n - 2)(n - 1)) = \frac{1}{6}(n - 2)(n - 1),$$

the variance is

$$P''_n(1) + P'_n(1) - P'_n(1)^2 = \frac{n^2 - 1}{12}.$$

Exercise 3.63 Verify the above results from the closed form

$$P_n(u) = \frac{1 - u^n}{n(1 - u)},$$

using l'Hôpital's rule to compute the derivatives at 1.

Exercise 3.64 Find the PGF for the random variable that counts the number of leading 0s in a random binary string, and use the PGF to find the mean and standard deviation.

Binomial distribution. Consider a random string of N independent bits, where each bit is 0 with probability p and 1 with probability $q = 1 - p$. We can argue that the probability that exactly k of the N bits are 0 is

$$\binom{N}{k} p^k q^{N-k},$$

so the corresponding PGF is

$$P_N(u) = \sum_{0 \leq k \leq N} \binom{N}{k} p^k q^{N-k} u^k = (pu + q)^N.$$

Alternatively, we could observe that PGF for 0s in a single bit is $(pu + q)$ and the N bits are independent, so the PGF for the number of 0s in the N bits is $(pu + q)^N$. Now, the average number of 0s is $P'(1) = pN$ and the variance is $P''(1) + P'(1) - P'(1)^2 = pqN$, and so forth. We can make these calculations easily without ever explicitly determining individual probabilities.

One cannot expect to be so fortunate as to regularly encounter a full decomposition into independent PGFs in this way. In the binomial distribution, the count of the number of structures 2^N trivially factors into N simple factors, and, since this quantity appears as the denominator in calculating the average, it is not surprising that the numerator decomposes as well. Conversely, if the count does not factor in this way, as for example in the case of the Catalan numbers, then we might not expect to find easy independence arguments like these. For this reason, as described in the next section, we emphasize the use of cumulative and bivariate generating functions, not PGFs, in the analysis of algorithms.

Quicksort distribution. Let $Q_N(u)$ be the PGF for the number of comparisons used by quicksort. We can apply the composition rules for PGFs to show that function to satisfy the functional equation

$$Q_N(u) = \frac{1}{N} \sum_{1 \leq k \leq N} u^{N+1} Q_{k-1}(u) Q_{N-k}(u).$$

Though using this equation to find an explicit expression for $Q_N(u)$ appears to be quite difficult, it does provide a basis for calculation of the moments. For example, differentiating and evaluating at $u = 1$ leads directly to the standard quicksort recurrence that we addressed in §3.3. Note that the PGF corresponds to a sequence indexed by the number of comparisons; the OGF that we used to solve (1) in §3.3 is indexed by the number of elements in the file. In the next section we will see how to treat both with just one double generating function.

Though it would seem that probability generating functions are natural tools for the average-case analysis of algorithms (and they are), we generally give this point of view less emphasis than the approach of analyzing parameters of combinatorial structures, for reasons that will become more clear in the next section. When dealing with discrete structures, the two approaches are formally related if not equivalent, but counting is more natural and allows for more flexible manipulations.

3.10 Bivariate Generating Functions. In the analysis of algorithms, we are normally interested not just in counting structures of a given size, but also in knowing values of various parameters relating to the structures.

We use *bivariate* generating functions for this purpose. These are functions of two variables that represent doubly indexed sequences: one index for the problem size, and one index for the value of the parameter being analyzed. Bivariate generating functions allow us to capture both indices with just one generating function, of two variables.

Definition Given a doubly indexed sequence $\{a_{nk}\}$, the function

$$A(z, u) = \sum_{n \geq 0} \sum_{k \geq 0} a_{nk} z^n u^k$$

is called the bivariate generating function (BGF) of the sequence. We use the notation $[z^n u^k] A(z, u)$ to refer to a_{nk} ; $[z^n] A(z, u)$ to refer to $\sum_{k \geq 0} a_{nk} u^k$; and $[u^k] A(z, u)$ to refer to $\sum_{n \geq 0} a_{nk} z^n$.

As appropriate, a BGF may need to be made “exponential” by dividing by $n!$. Thus the exponential BGF of $\{a_{nk}\}$ is

$$A(z, u) = \sum_{n \geq 0} \sum_{k \geq 0} a_{nk} \frac{z^n}{n!} u^k.$$

Most often, we use BGFs to count parameter values in combinatorial structures as follows. For $p \in \mathcal{P}$, where \mathcal{P} is a class of combinatorial structures, let $\text{cost}(p)$ be a function that gives the value of some parameter defined for each structure. Then our interest is in the BGF

$$P(z, u) = \sum_{p \in \mathcal{P}} z^{|p|} u^{\{\text{cost}(p)\}} = \sum_{n \geq 0} \sum_{k \geq 0} p_{nk} z^n u^k,$$

where p_{nk} is the number of structures of size n and cost k . We also write

$$P(z, u) = \sum_{n \geq 0} p_n(u) z^n \quad \text{where} \quad p_n(u) = [z^n] A(z, u) = \sum_{k \geq 0} p_{nk} u^k$$

to separate out all the costs for the structures of size n , and

$$P(z, u) = \sum_{k \geq 0} q_k(z) u^k \quad \text{where} \quad q_k(z) = [u^k] P(z, u) = \sum_{n \geq 0} p_{nk} z^n$$

to separate out all the structures of cost k . Also, note that

$$P(z, 1) = \sum_{p \in \mathcal{P}} z^{|p|} = \sum_{n \geq 0} p_n(1) z^n = \sum_{k \geq 0} q_k(z)$$

is the ordinary generating function that enumerates \mathcal{P} .

Of primary interest is the fact that $p_n(u)/p_n(1)$ is the PGF for the random variable representing the cost, if all structures of size n are taken as equally likely. Thus, knowing $p_n(u)$ and $p_n(1)$ allows us to compute average cost and other moments, as described in the previous section. BGFs provide a convenient framework for such computations, based on counting and analysis of cost parameters for combinatorial structures.

Binomial distribution. Let \mathcal{B} be the set of all binary strings, and consider the “cost” function for a binary string to be the number of 1 bits. In this case, $\{a_{nk}\}$ is the number of n -bit binary strings with k 1s, so the associated BGF is

$$P(z, u) = \sum_{n \geq 0} \sum_{k \geq 0} \binom{n}{k} u^k z^n = \sum_{n \geq 0} (1+u)^n z^n = \frac{1}{1-(1+u)z}.$$

BGF expansions. Separating out the structures of size n as $[z^n]P(z, u) = p_n(u)$ is often called the “horizontal” expansion of the BGF. This comes from the natural representation of the full BGF expansion as a two-dimensional table, with powers of u increasing in the horizontal direction and powers of z increasing in the vertical direction. For example, the BGF for the binomial distribution may be written as follows:

$$\begin{aligned} & z^0(u^0) + \\ & z^1(u^0 + u^1) + \\ & z^2(u^0 + 2u^1 + u^2) + \\ & z^3(u^0 + 3u^1 + 3u^2 + u^3) + \\ & z^4(u^0 + 4u^1 + 6u^2 + 4u^3 + u^4) + \\ & z^5(u^0 + 5u^1 + 10u^2 + 10u^3 + 5u^4 + u^5) + \dots . \end{aligned}$$

Or, proceeding vertically through such a table, we can collect $[u^k]P(z, u) = q_k(z)$. For the binomial distribution, this gives

$$\begin{aligned} & u^0(z^0 + z^1 + z^2 + z^3 + z^4 + z^5 + \dots) + \\ & u^1(z^1 + 2z^2 + 3z^3 + 4z^4 + 5z^5 + \dots) + \\ & u^2(z^2 + 3z^3 + 6z^4 + 10z^5 \dots) + \\ & u^3(z^3 + 4z^4 + 10z^5 + \dots) + \\ & u^4(z^4 + 5z^5 + \dots) + \\ & u^5(z^5 + \dots) + \dots , \end{aligned}$$

the so-called vertical expansion of the BGF. As we will see, these alternate representations are important in the analysis of algorithms, especially when explicit expressions for the full BGF are not available.

Calculating moments “horizontally.” With these notations, calculations of probabilities and moments are straightforward. Differentiating with respect to u and evaluating at $u = 1$, we find that

$$p'_n(1) = \sum_{k \geq 0} kp_{nk}.$$

The partial derivative with respect to u of $P(z, u)$ evaluated at $u = 1$ is the generating function for this quantity. Now, $p_n(1)$ is the number of members of \mathcal{P} of size n . If we consider all members of \mathcal{P} of size n to be equally likely, then the probability that a structure of size n has cost k is $p_{nk}/p_n(1)$ and the average cost of a structure of size n is $p'_n(1)/p_n(1)$.

Definition Let \mathcal{P} be a class of combinatorial structures with BGF $P(z, u)$. Then the function

$$\frac{\partial P(z, u)}{\partial u} \Big|_{u=1} = \sum_{p \in \mathcal{P}} \text{cost}(p) z^{|p|}$$

is defined to be the *cumulative generating function* (CGF) for the class. Also, let \mathcal{P}_n denote the class of all the structures of size n in \mathcal{P} . Then the sum

$$\sum_{p \in \mathcal{P}_n} \text{cost}(p)$$

is defined to be the *cumulated cost* for the structures of size n .

This terminology is justified since the cumulated cost is precisely the coefficient of z^n in the CGF. The cumulated cost is sometimes referred to as the *unnormalized mean*, since the true mean is obtained by “normalizing,” or dividing by the number of structures of size n .

Theorem 3.8 (BGFs and average costs). Given a BGF $P(z, u)$ for a class of combinatorial structures, the average cost for all structures of a given size is given by the cumulated cost divided by the number of structures, or

$$\frac{[z^n] \frac{\partial P(z, u)}{\partial u} \Big|_{u=1}}{[z^n] P(1, z)}.$$

Proof. The calculations are straightforward, following directly from the observation that $p_n(u)/p_n(1)$ is the associated PGF, then applying Theorem 3.7. ■

The importance of the use of BGFs and Theorem 3.8 is that the average cost can be calculated by extracting coefficients *independently* from

$$\frac{\partial P(z, u)}{\partial u} \Big|_{u=1} \quad \text{and} \quad P(1, z)$$

and dividing. For more compact notation, we often write the partial derivative as $P_u(z, 1)$. The standard deviation can be calculated in a similar manner. These notations and calculations are summarized in Table 3.5.

For the example given earlier involving the binomial distribution, the number of binary strings of length n is

$$[z^n] \frac{1}{1 - (1+u)z} \Big|_{u=1} = [z^n] \frac{1}{(1-2z)} = 2^n,$$

and the cumulated cost (number of 1 bits in all n -bit binary strings) is

$$[z^n] \frac{\partial}{\partial u} \frac{1}{1 - (1+u)z} \Big|_{u=1} = [z^n] \frac{z}{(1-2z)^2} = n2^{n-1},$$

so the average number of 1 bits is thus $n/2$. Or, starting from $p_n(u) = (1+u)^n$, the number of structures is $p_n(1) = 2^n$ and the cumulated cost is $p'_n(1) = n2^{n-1}$. Or, we can compute the average by arguing directly that the number of binary strings of length n is 2^n and the number of 1 bits in all binary strings of length n is $n2^{n-1}$, since there are a total of $n2^n$ bits, half of which are 1 bits.

Exercise 3.65 Calculate the variance for the number of 1 bits in a random binary string of length n , using Table 3.5 and $p_n(u) = (1+u)^n$, as shown earlier.

Calculating moments “vertically.” Alternatively, the cumulated cost may be calculated using the vertical expansion:

$$[z^n] \sum_{k \geq 0} k q_k(z) = \sum_{k \geq 0} k p_{nk}.$$

Corollary The cumulated cost is also equal to

$$[z^n] \sum_{k \geq 0} (P(1, z) - r_k(z)) \quad \text{where} \quad r_k(z) \equiv \sum_{0 \leq j \leq k} q_j(z).$$

Proof. The function $r_k(z)$ is the generating function for all structures with cost no greater than k . Since $r_k(z) - r_{k-1}(z) = q_k(z)$, the cumulated cost is

$$[z^n] \sum_{k \geq 0} k(r_k(z) - r_{k-1}(z)),$$

which telescopes to give the stated result. ■

$$P(z, u) = \sum_{p \in \mathcal{P}} z^{|p|} u^{\{\text{cost}(p)\}} = \sum_{n \geq 0} \sum_{k \geq 0} p_{nk} u^k z^n = \sum_{n \geq 0} p_n(u) z^n = \sum_{k \geq 0} q_k(z) u^k$$

GF of costs for structures of size n $[z^n]P(z, u) \equiv p_n(u)$

GF enumerating structures with cost k $[u^k]P(z, u) \equiv q_k(z)$

cumulative generating function (CGF) $\frac{\partial P(z, u)}{\partial u} \Big|_{u=1} \equiv q(z)$
 $= \sum_{k \geq 0} k q_k(z)$

number of structures of size n $[z^n]P(1, z) = p_n(1)$

cumulated cost $[z^n] \frac{\partial P(z, u)}{\partial u} \Big|_{u=1} = \sum_{k \geq 0} k p_{nk}$
 $= p'_n(1)$
 $= [z^n]q(z)$

average cost $\frac{[z^n] \frac{\partial P(z, u)}{\partial u} \Big|_{u=1}}{[z^n]P(1, z)} = \frac{p'_n(1)}{p_n(1)}$
 $= \frac{[z^n]q(z)}{p_n(1)}$

variance $\frac{p''_n(1)}{p_n(1)} + \frac{p'_n(1)}{p_n(1)} - \left(\frac{p'_n(1)}{p_n(1)}\right)^2$

Table 3.5 Calculating moments from a bivariate generating function

As k increases in this sum, initial terms cancel (all small structures have cost no greater than k), so this representation lends itself to asymptotic approximation. We will return to this topic in detail in Chapter 6, where we first encounter problems for which the vertical formulation is appropriate.

Exercise 3.66 Verify from the vertical expansion that the mean for the binomial distribution is $n/2$ by first calculating $r_k(z)$ as described earlier.

Quicksort distribution. We have studied the average-case analysis of the running time of quicksort in some detail in §1.5 and §3.3, so it will be instructive to examine that analysis, including calculation of the variance, from the perspective of BGFs. We begin by considering the exponential BGF

$$Q(z, u) = \sum_{N \geq 0} \sum_{k \geq 0} q_{Nk} u^k \frac{z^N}{N!}$$

where q_{Nk} is the cumulative count of the number of comparisons taken by quicksort on all permutations of N elements. Now, because there are $N!$ permutations of N elements, this is actually a “probability” BGF: $[z^N]Q(z, u)$ is nothing other than the PGF $Q_N(u)$ introduced at the end of the previous section. As we will see in several examples in Chapter 7, this relationship between exponential BGFs and PGFs holds whenever we study properties of permutations. Therefore, by multiplying both sides of the recurrence from §3.9,

$$Q_N(u) = \frac{1}{N} \sum_{1 \leq k \leq N} u^{N+1} Q_{k-1}(u) Q_{N-k}(u),$$

by z^N and summing on N , we can derive the functional equation

$$\frac{\partial}{\partial z} Q(z, u) = u^2 Q^2(zu, u) \quad \text{with} \quad Q(u, 0) = 1$$

that must be satisfied by the BGF. This carries enough information to allow us to compute the moments of the distribution.

Theorem 3.9 (Quicksort variance). The variance of the number of comparisons used by quicksort is

$$7N^2 - 4(N + 1)^2 H_N^{(2)} - 2(N + 1)H_N + 13N \sim N^2 \left(7 - \frac{2\pi^2}{3} \right).$$

Proof. This calculation is sketched in the previous discussion and the following exercises, and is perhaps best done with the help of a computer algebra system. The asymptotic estimate follows from the approximations $H_N \sim \ln N$ (see the first corollary to Theorem 4.3) and $H_N^{(2)} \sim \pi^2/6$ (see Exercise 4.56). This result is due to Knuth [9]. ■

As discussed in §1.7, the standard deviation ($\approx .65N$) is asymptotically smaller than the average value ($\approx 2N\ln N - .846N$). This means that the observed number of comparisons when quicksort is used to sort a random permutation (or when partitioning elements are chosen randomly) should be close to the mean with high probability, and even more so as N increases.

Exercise 3.67 Confirm that

$$q^{[1]}(z) \equiv \frac{\partial}{\partial u} Q(z, u) \Big|_{u=1} = \frac{1}{(1-z)^2} \ln \frac{1}{1-z}$$

and show that

$$\begin{aligned} q^{[2]}(z) \equiv \frac{\partial^2}{\partial u^2} Q(z, u) \Big|_{u=1} &= \frac{6}{(1-z)^3} + \frac{8}{(1-z)^3} \ln \frac{1}{1-z} + \frac{8}{(1-z)^3} \ln^2 \frac{1}{1-z} \\ &\quad - \frac{6}{(1-z)^2} - \frac{12}{(1-z)^2} \ln \frac{1}{1-z} - \frac{4}{(1-z)^2} \ln^2 \frac{1}{1-z} \end{aligned}.$$

Exercise 3.68 Extract the coefficient of z^N in $q^{[2]}(z) + q^{[1]}(z)$ and verify the exact expression for the variance given in Theorem 3.9. (See Exercise 3.8.)

THE ANALYSIS OF LEAVES in binary trees and the analysis of the number of comparisons taken by quicksort are representative of numerous other examples, which we will see in Chapters 6 through 9, of the use of bivariate generating functions in the analysis of algorithms. As our examples here have illustrated, one reason for this is our ability to use symbolic arguments to encapsulate properties of algorithms and data structures in relationships among their generating functions. As also illustrated by our examples, another reason for this is the convenient framework provided by BGFs for computing moments, particularly the average.

3.11 Special Functions. We have already encountered a number of “special” sequences of numbers—such as the harmonic numbers, the Fibonacci numbers, binomial coefficients, and $N!$ —that are intrinsic to the problems under examination and that appear in so many different applications that they are worthy of study on their own merit. In this section, we briefly consider several more such sequences.

We define these sequences in Table 3.6 as the coefficients in the generating functions given. Alternatively, there are combinatorial interpretations that could serve to define these sequences, but we prefer to have the generating function serve as the definition to avoid biasing our discussion toward any particular application. We may view these generating functions as adding to our toolkit of “known” functions—these particular ones have appeared so frequently that their properties are quite well understood.

The primary heritage of these sequences is from combinatorics: each of them “counts” some basic combinatorial object, some of which are briefly described in this section. For example, $N!$ is the number of permutations of N objects, and H_N is the average number of times we encounter a value larger than all previously encountered when proceeding from left to right through a random permutation (see Chapter 7). We will avoid a full survey of the combinatorics of the special numbers, concentrating instead on those that play a role in fundamental algorithms and in the basic structures discussed in Chapters 6 through 9. Much more information about the special numbers may be found, for example, in the books by Comtet [1], by Graham, Knuth, and Patashnik [5], and by Goulden and Jackson [4]. The sequences also arise in analysis. For example, we can use them to translate from one way to represent a polynomial to another. We mention a few examples here but avoid considering full details.

The analysis of algorithms perhaps adds a new dimension to the study of special sequences: we resist the temptation to define the special sequences in terms of basic performance properties of fundamental algorithms, though it would be possible to do so for each of them, as discussed in Chapters 6 through 9. In the meantime, it is worthwhile to become familiar with these sequences because they arise so frequently—either directly, when we study algorithms that turn out to be processing fundamental combinatorial objects, or indirectly, when we are led to one of the generating functions discussed here. Whether or not we are aware of a specific combinatorial connection, well-understood properties of these generating functions are often exploited

binomial coefficients	$\frac{1}{1-z-uz} = \sum_{n,k \geq 0} \binom{n}{k} u^k z^n$
	$\frac{z^k}{(1-z)^{k+1}} = \sum_{n \geq k} \binom{n}{k} z^n$
	$(1+u)^n = \sum_{k \geq 0} \binom{n}{k} u^k$
Stirling numbers of the first kind	$\frac{1}{(1-z)^u} = \sum_{n,k \geq 0} \begin{Bmatrix} n \\ k \end{Bmatrix} u^k \frac{z^n}{n!}$
	$\frac{1}{k!} \left(\ln \frac{1}{1-z} \right)^k = \sum_{n \geq 0} \begin{Bmatrix} n \\ k \end{Bmatrix} \frac{z^n}{n!}$
	$u(u+1) \dots (u+n-1) = \sum_{k \geq 0} \begin{Bmatrix} n \\ k \end{Bmatrix} u^k$
Stirling numbers of the second kind	$e^{u(e^z-1)} = \sum_{n,k \geq 0} \begin{Bmatrix} n \\ k \end{Bmatrix} u^k \frac{z^n}{n!}$
	$\frac{1}{k!} (e^z - 1)^k = \sum_{n \geq 0} \begin{Bmatrix} n \\ k \end{Bmatrix} \frac{z^n}{n!}$
	$\frac{z^k}{(1-z)(1-2z) \dots (1-kz)} = \sum_{n \geq k} \begin{Bmatrix} n \\ k \end{Bmatrix} z^n$
Bernoulli numbers	$\frac{z}{(e^z-1)} = \sum_{n \geq 0} B_n \frac{z^n}{n!}$
Catalan numbers	$\frac{1-\sqrt{1-4z}}{2z} = \sum_{n \geq 0} \frac{1}{n+1} \binom{2n}{n} z^n$
harmonic numbers	$\frac{1}{1-z} \ln \frac{1}{1-z} = \sum_{n \geq 1} H_n z^n$
factorials	$\frac{1}{1-z} = \sum_{n \geq 0} n! \frac{z^n}{n!}$
Fibonacci numbers	$\frac{z}{1-z-z^2} = \sum_{n \geq 0} F_n z^n$

Table 3.6 Classic “special” generating functions

in the analysis of algorithms. Chapters 6 through 9 will cover many more details about these sequences with relevance to specific algorithms.

Binomial coefficients. We have already been assuming that the reader is familiar with properties of these special numbers: the number $\binom{n}{k}$ counts the number of ways to choose k objects out of n , without replacement; they are the coefficients that arise when the polynomial $(1+x)^n$ is expanded in powers of x . As we have seen, binomial coefficients appear often in the analysis of algorithms, ranging from elementary problems involving Bernoulli trials to Catalan numbers to sampling in quicksort to tries to countless other applications.

Stirling numbers. There are two kinds of Stirling numbers; they can be used to convert back and forth between the standard representation of a polynomial and a representation using so-called falling factorial powers $x^{\underline{k}} = x(x-1)(x-2)\dots(x-k+1)$:

$$x^n = \sum_k \begin{Bmatrix} n \\ k \end{Bmatrix} (-1)^{n-k} x^k \quad \text{and} \quad x^n = \sum_k \left\{ \begin{matrix} n \\ k \end{matrix} \right\} x^{\underline{k}}.$$

Stirling numbers have combinatorial interpretations similar to those for binomial coefficients: $\left\{ \begin{matrix} n \\ k \end{matrix} \right\}$ is the number of ways to divide a set of n objects into k nonempty subsets; and $\begin{Bmatrix} n \\ k \end{Bmatrix}$ is the number of ways to divide n objects into k nonempty cycles. We have touched on the $\begin{Bmatrix} n \\ k \end{Bmatrix}$ Stirling distribution already in §3.9 and will cover it in detail in Chapter 7. The $\left\{ \begin{matrix} n \\ k \end{matrix} \right\}$ Stirling distribution makes an appearance in Chapter 9, in our discussion of the coupon collector problem.

Bernoulli numbers. The sequence with EGF $z/(e^z - 1)$ arises in a number of combinatorial applications. For example, we need these numbers if we want to write down an explicit expression for the sum of the t th powers of the integers less than N , as a standard polynomial in N . We can deduce the first few terms in the sequence by setting coefficients of z equal in

$$z = \left(B_0 + B_1 z + \frac{B_2}{2} z^2 + \frac{B_3}{6} z^3 + \dots \right) \left(z + \frac{z^2}{2} + \frac{z^3}{6} + \dots \right).$$

This gives $B_0 = 1$, then $B_1 + B_0/2 = 0$ so $B_1 = -1/2$, then $B_2/2 + B_1/2 + B_0/6 = 0$ so $B_2 = 1/6$, and so on. If we let

$$S_{Nt} = \sum_{0 \leq k < N} k^t,$$

then the EGF is given by

$$S_N(z) = \sum_{t \geq 0} \sum_{0 \leq k \leq N} k^t \frac{z^t}{t!} = \sum_{0 \leq k \leq N} e^{kz} = \frac{e^{Nz} - 1}{e^z - 1}.$$

This is now a convolution of “known” generating functions, from which the explicit formula

$$S_{Nt} = \frac{1}{t+1} \sum_{0 \leq k \leq t} \binom{t+1}{k} B_k N^{t+1-k}$$

follows. We have

$$\sum_{1 \leq k \leq N} k = \frac{N^2}{2} + \frac{N}{2} = \frac{N(N+1)}{2},$$

$$\sum_{1 \leq k \leq N} k^2 = \frac{N^3}{3} + \frac{N^2}{2} + \frac{N}{6} = \frac{N(N+1)(2N+1)}{6},$$

$$\sum_{1 \leq k \leq N} k^3 = \frac{N^4}{4} + \frac{N^3}{2} + \frac{N^2}{4} = \frac{N^2(N+1)^2}{4},$$

and in general

$$\sum_{1 \leq k \leq N} k^t \sim \frac{N^{t+1}}{t+1}.$$

Beyond this basic use, Bernoulli numbers play an essential role in the Euler-Maclaurin summation formula (which is discussed in §4.5) and they arise naturally in other applications in the analysis of algorithms. For example, they appear in the generating functions for a family of algorithms related to the digital trees discussed in Chapter 8.

Bernoulli polynomials. The polynomials

$$B_m(x) = \sum_k \binom{m}{k} B_k x^{m-k}$$

have the EGF

$$\sum_{m \geq 0} B_m(x) \frac{z^m}{m!} = \frac{z}{e^z - 1} e^{xz}$$

from which a number of interesting properties can be proved. For example, differentiating this EGF gives the identity

$$B'_m(x) = m B_{m-1}(x) \quad \text{for } m > 1.$$

Our primary interest in the Bernoulli polynomials is an analytic application for approximating integrals—the Euler-Maclaurin summation formula—that we will examine in detail in the next chapter.

Exercise 3.69 Give closed-form expressions for the following:

$$\sum_{n,k \geq 0} \binom{n}{k} u^k \frac{z^n}{n!} \quad \sum_{n,k \geq 0} k! \begin{bmatrix} n \\ k \end{bmatrix} u^k \frac{z^n}{n!} \quad \sum_{n,k \geq 0} k! \left\{ \begin{matrix} n \\ k \end{matrix} \right\} u^k \frac{z^n}{n!}.$$

Exercise 3.70 Prove from the generating function that

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

Exercise 3.71 Prove from the generating function that

$$\begin{bmatrix} n \\ k \end{bmatrix} = (n-1) \begin{bmatrix} n-1 \\ k \end{bmatrix} + \begin{bmatrix} n-1 \\ k-1 \end{bmatrix}.$$

Exercise 3.72 Prove from the generating function that

$$\left\{ \begin{matrix} n \\ k \end{matrix} \right\} = k \left\{ \begin{matrix} n-1 \\ k \end{matrix} \right\} + \left\{ \begin{matrix} n-1 \\ k-1 \end{matrix} \right\}.$$

Exercise 3.73 Prove that $B_m(0) = B_m(1) = B_m$ for all $m > 1$.

Exercise 3.74 Prove from the generating function that B_k is zero for k odd, $k \geq 3$.

Other types of generating functions. As mentioned at the beginning of the chapter, kernel functions other than z^k and $z^k/k!$ lead to other types of generating functions. For example, using k^{-z} as the kernel function gives *Dirichlet generating functions* (DGFs), which play an important role in number

theory and in the analysis of several algorithms. These are best understood as functions of complex z , so their analytic properties are beyond the scope of this book. Nevertheless, we mention them to motivate the utility of other kernels and illustrate the kinds of formal manipulations that can arise. The Dirichlet generating function for $1, 1, 1, \dots$ is

$$\zeta(z) = \sum_{k \geq 1} \frac{1}{k^z},$$

the Riemann zeta function. This function plays a central role in analytic number theory.

In the analysis of algorithms, Dirichlet generating functions normally represent number-theoretic properties of sequences and are expressed in terms of the zeta function. For example,

$$\zeta(z)^2 = \sum_{k \geq 1} \frac{1}{k^z} \sum_{j \geq 1} \frac{1}{j^z} = \sum_{k \geq 1} \sum_{j \geq 1} \frac{1}{(kj)^z} = \sum_{N \geq 1} \sum_{j \text{ divides } N} \frac{1}{N^z} = \sum_{N \geq 1} \frac{d_N}{N^z}$$

where d_N is the number of divisors of N . In other words, $\zeta(z)^2$ is the Dirichlet generating function for $\{d_N\}$.

Dirichlet generating functions turn out to be especially useful when we need to work with the binary representation of numbers. For example, we can easily find the DGF for the characteristic sequence for the even numbers:

$$\sum_{\substack{N \geq 1 \\ N \text{ even}}} \frac{1}{N^z} = \sum_{N \geq 1} \frac{1}{(2N)^z} = \frac{1}{2^z} \zeta(z).$$

Again, while these formal manipulations are interesting, the analytic properties of these functions in the complex plane are an important facet of their use in the analysis of algorithms. Details may be found in [3] or [9].

By using other kernels—such as $z^k/(1 - z^k)$ (Lambert), $\binom{z}{k}$ (Newton), or $z^k/(1 - z)(1 - z^2) \dots (1 - z^k)$ (Euler)—we obtain other types of generating functions that have proved over the centuries to have useful properties in analysis. Such functions arise occasionally in the analysis of algorithms, and exploration of their properties is fascinating. We mention them in passing but we do not consider them in detail because they do not play the central role that OGFs and EGFs do. Much more information on these can be found in [5], [7], [11], and [17].

Exercise 3.75 Show that, for any $k \geq 0$, the DGF for the characteristic sequence of numbers whose binary representation ends in k 0s is $\zeta(z)/2^{kz}$.

Exercise 3.76 Find the DGF for the function ψ_N , the number of trailing 0s in the binary representation of N .

Exercise 3.77 Find the DGF for the characteristic function of $\{N^2\}$.

Exercise 3.78 Prove that

$$\sum_k \frac{z^k}{1-z^k} = \sum_N d_N z^N,$$

where d_N is the number of divisors of N .

GENERATING functions have long been used in combinatorics, probability theory, and analytic number theory; hence a rich array of mathematical tools have been developed that turn out to be germane to the analysis of algorithms. As described in detail in Chapter 5, the role of generating functions in the analysis of algorithms is central: we use generating functions both as *formal* objects to aid in the process of precisely accounting for quantities of interest and as *analytic* objects to yield solutions.

We have introduced generating functions as a tool to solve the recurrences that arise in the analysis of algorithms in order to emphasize their direct relationship to the quantity being studied (running time, or other characteristic parameter, as a function of problem size). This direct relationship provides a mathematical model that is amenable to classical techniques of all sorts to provide information about the algorithm at hand, as we have seen.

But we have also noted that a recurrence is simply one characterization of a sequence; the corresponding generating function itself is another. For many problems it is the case that direct arguments can yield explicit expressions for generating functions and recurrences can be avoided entirely. This theme is developed in the chapters that follow and formalized in [3].

Using the generating function representation, we are often able to transform a problem to see how the sequence of interest is expressed in terms of classical special number sequences. If an exact representation is not available, the generating function representation positions us to employ powerful mathematical techniques based on properties of functions of a complex variable in order to learn properties of our algorithms. For a great many of the kinds of

functions that arise in the analysis of algorithms, we can find precise estimates of asymptotic behavior of coefficients by expansion in terms of classical functions. If not, we can be secure that complex asymptotic methods are available for extracting estimates of the asymptotic values of coefficients, as discussed briefly in Chapter 5 and in detail in [3].

Ordinary, exponential, and bivariate generating functions provide a fundamental framework that enables us to develop a systematic approach to analyzing a large number of the fundamental structures that play a central role in the design of algorithms. With the additional help of the asymptotic methods to be developed in the next chapter, we are able to use the tools to develop results that we can use to predict the performance characteristics of a variety of important and useful algorithms. Again, this theme will be developed in detail in Chapters 6 through 9.

References

1. L. COMTET. *Advanced Combinatorics*, Reidel, Dordrecht, 1974.
2. P. FLAJOLET, B. SALVY, AND P. ZIMMERMAN. “Automatic average-case analysis of algorithms,” *Theoretical Computer Science* **79**, 1991, 37–109.
3. P. FLAJOLET AND R. SEDGEWICK. *Analytic Combinatorics*, Cambridge University Press, 2009.
4. I. GOULDEN AND D. JACKSON. *Combinatorial Enumeration*, John Wiley, New York, 1983.
5. R. L. GRAHAM, D. E. KNUTH, AND O. PATASHNIK. *Concrete Mathematics*, 1st edition, Addison-Wesley, Reading, MA, 1989. Second edition, 1994.
6. D. H. GREENE AND D. E. KNUTH. *Mathematics for the Analysis of Algorithms*, Birkhäuser, Boston, 1981.
7. G. H. HARDY. *Divergent Series*, Oxford University Press, 1947.
8. D. E. KNUTH. *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, 1st edition, Addison-Wesley, Reading, MA, 1968. Third edition, 1997.
9. D. E. KNUTH. *The Art of Computer Programming. Volume 3: Sorting and Searching*, 1st edition, Addison-Wesley, Reading, MA, 1973. Second edition, 1998.
10. D. E. KNUTH. *The Art of Computer Programming. Volume 4A: Combinatorial Algorithms, Part 1*, Addison-Wesley, Boston, 2011.
11. N. E. NORLUND. *Vorlesungen über Differenzenrechnung*, Chelsea Publishing Company, New York, 1954.
12. G. PÓLYA, R. E. TARJAN, AND D. R. WOODS. *Notes on Introductory Combinatorics*, Birkhauser, Boston, 1983.
13. J. RIORDAN. *Introduction to Combinatorial Analysis*, Princeton University Press, Princeton, NJ, 1980.
14. R. SEDGEWICK. “The analysis of quicksort programs,” *Acta Informatica* **7**, 1977, 327–355.
15. R. SEDGEWICK. *Quicksort*, Garland Publishing, New York, 1980.
16. R. P. STANLEY. *Enumerative Combinatorics*, Wadsworth & Brooks/Cole, 1986, 2nd edition, Cambridge, 2011.

17. R. P. STANLEY. "Generating functions," in *Studies in Combinatorics* (MAA Studies in Mathematics, 17, G. C. Rota, ed.), The Mathematical Association of America, 1978, 100–141.
18. 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.
19. H. WILF. *Generatingfunctionology*, Academic Press, San Diego, 1990, 2nd edition, A. K. Peters, 2006.

This page intentionally left blank

CHAPTER FOUR

ASYMPTOTIC APPROXIMATIONS

OUR initial general orientation in the analysis of algorithms is toward deriving *exact* mathematical results. However, such exact solutions may not be always available, or if available they may be too unwieldy to be of much use. In this chapter, we will examine some methods of deriving approximate solutions to problems or of approximating exact solutions; as a result, we may modify our primary orientation to be toward deriving *concise, accurate, and precise* estimates of quantities of interest.

In a manner similar to Chapter 3, our primary goal in this chapter is to provide an overview of the basic properties of asymptotic expansions, methods of manipulating them, and a catalog of those that we encounter most often in the analysis of algorithms. At times, this may seem to take us rather far from the analysis of algorithms, though we continue to draw examples and exercises directly from problems introduced in relation to specific algorithms in Chapter 1, and to lay the groundwork for studying a broad variety of algorithms in Chapters 6 through 9.

As we have been doing, we focus in this chapter on methods from real analysis. Asymptotic methods using complex analysis are sometimes required for the problems that we face in the analysis of algorithms. Such methods are a primary topic of [11]; the treatment in this chapter is a necessary background for learning them. We briefly introduce the principle upon which one of the most important techniques is based in §5.5.

We have seen that the analysis of computer algorithms involves tools from discrete mathematics, leading to answers most easily expressed in terms of discrete functions (such as harmonic numbers or binomial coefficients) rather than more familiar functions from analysis (such as logarithms or powers). However, it is generally true that these two types of functions are closely related—one reason to do asymptotic analysis is to “translate” between them.

Generally, a problem carries a notion of “size” and we are interested in approximations that become more accurate as the size becomes large. By the nature of the mathematics and the problems that we are solving, it is also often true that our answers, if they are to be expressed in terms of a single

parameter N , will be (primarily) expressed in terms of asymptotic series in N and $\log N$. These series are not necessarily convergent (indeed, they are often divergent), but the initial terms give very accurate estimates for many quantities that arise in the analysis of algorithms. Our approach will be to use such series to represent quantities of interest, manipulating them in well-defined ways to develop concise, accurate, and precise expressions.

One motivation for considering asymptotic methods is simply to find a convenient way to calculate good approximations to specific values for quantities of interest. Another motivation is to get all our quantities in a canonical form so that we can compare and combine them easily. For example, as we saw in Chapter 1, it is helpful to know that the number of comparisons taken by quicksort approaches $2NH_N$ as compared to the optimal $\lg N!$, but it is more useful to know that both are proportional to $N\lg N$ with coefficient $1.4421 \dots$ for the former and 1 for the latter, and that even more accurate estimates are available.

For another example, in §7.6 we encounter a sorting algorithm whose average running time is proportional to $N4^{N-1}/(2^N_N)$. This is a concise exact result, but how can we know the value of the quantity for, say, $N = 1000$? Using the formula to compute the value for large N is not a straightforward task, since it involves dividing two very large numbers or rearranging the computation to avoid doing so. In this chapter, we will see how to show that this quantity is very close to $N\sqrt{\pi N}/4$, which evaluates to 14,012 for $N = 1000$ and is only about a hundredth of a percent off the exact value, which is about 14,014. More important, the way in which the value of the quantity grows as N grows is clearly indicated by the approximate result (for example, we know by inspection that the value associated with $100N$ is about 1000 times the value associated with N), making it easier to compare the approximate result with similar results for other algorithms or other versions of the same algorithm.

Another important reason for working with approximate values is that they can substantially simplify symbolic calculations that might be involved in the analysis for many problems, allowing derivation of concise answers that might otherwise not be available. We touched upon this in Chapter 2 when discussing the solution of recurrences by solving similar, simpler, recurrences and estimating the error. Asymptotic analysis gives a systematic approach to aid in such arguments.

A primary topic of the chapter is our treatment of methods for computing approximate values of sums for which exact evaluation may be difficult or impossible. Specifically, we consider how to evaluate sums by approximating them with integrals using the *Euler-Maclaurin summation formula*. We also will look at the *Laplace method* for evaluating sums by adjusting the range of summation to make different approximations applicable in different parts of the range.

We consider several examples of the application of these concepts to find approximate values of some of the special number sequences introduced in Chapter 3 and other quantities that are likely to appear in the analysis of algorithms. In particular, we consider in some detail the *Ramanujan-Knuth Q-function* and related distributions, which arise frequently in the analysis. Then, we consider limits of the binomial distribution under various circumstances. The *normal approximation* and the *Poisson approximation* are classical results that are very useful in the analysis of algorithms and also provide excellent examples of the application of tools developed in this chapter.

The standard reference on these topics is the book by de Bruijn [6], which certainly should be read by anyone with a serious interest in asymptotic analysis. The survey by Odlyzko [18] also provides a great deal of information and a wealth of examples. Specific information about the normal and Poisson approximations may be found, for example, in Feller [8]. Detailed coverage of many of the topics that we consider may also be found in [2], [12], [13], [15], [19], and other references listed at the end of this chapter. Methods based on complex analysis are covered in detail in [11].

4.1 Notation for Asymptotic Approximations. The following notations, which date back at least to the beginning of the 20th century, are widely used for making precise statements about the approximate value of functions:

Definition Given a function $f(N)$, we write

$$g(N) = O(f(N))$$

if and only if $|g(N)/f(N)|$ is bounded from above as $N \rightarrow \infty$,

$$g(N) = o(f(N))$$

if and only if $g(N)/f(N) \rightarrow 0$ as $N \rightarrow \infty$,

$$g(N) \sim f(N)$$

if and only if $g(N)/f(N) \rightarrow 1$ as $N \rightarrow \infty$.

The O - and o -notations provide ways to express upper bounds (with o being the stronger assertion), and the \sim -notation provides a way to express asymptotic equivalence. The O -notation here coincides with the definition given in Chapter 1 for use in our discussion on computational complexity. A variety of similar notations and definitions have been proposed. A reader interested in pursuing implications may wish to read the discussion in [6] or [12].

Exercise 4.1 Show that

$$N/(N+1) = O(1), \quad 2^N = o(N!), \quad \text{and} \quad \sqrt[N]{e} \sim 1.$$

Exercise 4.2 Show that

$$\frac{N}{N+1} = 1 + O\left(\frac{1}{N}\right) \quad \text{and} \quad \frac{N}{N+1} \sim 1 - \frac{1}{N}.$$

Exercise 4.3 Show that $N^\alpha = o(N^\beta)$ if $\alpha < \beta$.

Exercise 4.4 Show that, for r fixed,

$$\binom{N}{r} = \frac{N^r}{r!} + O(N^{r-1}) \quad \text{and} \quad \binom{N+r}{r} = \frac{N^r}{r!} + O(N^{r-1}).$$

Exercise 4.5 Show that $\log N = o(N^\epsilon)$ for all $\epsilon > 0$.

Exercise 4.6 Show that

$$\frac{1}{2 + \ln N} = o(1) \quad \text{and} \quad \frac{1}{2 + \cos N} = O(1) \quad \text{but not } o(1).$$

As we will see later, it is not usually necessary to directly apply the definitions to determine asymptotic values of quantities of interest, because the O -notation makes it possible to develop approximations using a small set of basic algebraic manipulations.

The same notations are used when approximating functions of real or complex variables near any given point. For example, we say that

$$\frac{1}{1+x} = \frac{1}{x} - \frac{1}{x^2} + \frac{1}{x^3} + O\left(\frac{1}{x^4}\right) \quad \text{as } x \rightarrow \infty$$

and

$$\frac{1}{1+x} = 1 - x + x^2 - x^3 + O(x^4) \quad \text{as } x \rightarrow 0.$$

A more general definition of the O -notation that encompasses such uses is obtained simply by replacing $N \rightarrow \infty$ by $x \rightarrow x_0$ in the preceding definition, and specifying any restrictions on x (for example, whether it must be integer, real, or complex). The limiting value x_0 is usually 0 or ∞ , but it could be any value whatever. It is usually obvious from the context which set of numbers and which limiting value are of interest, so we normally drop the qualifying “ $x \rightarrow x_0$ ” or “ $N \rightarrow \infty$. ” Of course, the same remarks apply to the o - and \sim -notations.

In the analysis of algorithms, we avoid direct usages such as “the average value of this quantity is $O(f(N))$ ” because this gives scant information for the purpose of predicting performance. Instead, we strive to use the O -notation to bound “error” terms that have far smaller values than the main, or “leading,” term. Informally, we expect that the terms involved should be so small as to be negligible for large N .

O -approximations. We say that $g(N) = f(N) + O(h(N))$ to indicate that we can approximate $g(N)$ by calculating $f(N)$ and that the error will be bounded above by a constant factor of $h(N)$. As usual with the O -notation, the constant involved is unspecified, but the assumption that it is not large is often justified. As discussed later, we normally use this notation with $h(N) = o(f(N))$.

o -approximations. A stronger statement is to say that $g(N) = f(N) + o(h(N))$ to indicate that we can approximate $g(N)$ by calculating $f(N)$ and that the error will get smaller and smaller compared to $h(N)$ as N gets larger. An unspecified function is involved in the rate of decrease, but the assumption that it is never large numerically (even for small N) is often justified.

\sim -approximations. The notation $g(N) \sim f(N)$ is used to express the weakest nontrivial o -approximation $g(N) = f(N) + o(f(N))$.

These notations are useful because they can allow suppression of unimportant details without loss of mathematical rigor or precise results. If a more accurate answer is desired, one can be obtained, but most of the detailed calculations are suppressed otherwise. We will be most interested in methods that allow us to keep this “potential accuracy,” producing answers that could be calculated to arbitrarily fine precision if desired.

Exponentially small terms. When logarithms and exponentials are involved, it is worthwhile to be cognizant of “exponential differences” and avoid calculations that make truly negligible contributions to the ultimate answer of interest. For example, if we know that the value of a quantity is $2N + O(\log N)$, then we can be reasonably confident that $2N$ is within a few percent or a few thousandths of a percent of the true value when N is 1 thousand or 1 million, and that it may not be worthwhile to find the coefficient of $\log N$ or sharpen the expansion to within $O(1)$. Similarly, an asymptotic estimate of $2^N + O(N^2)$ is quite sharp. On the other hand, knowing that a quantity is $2N \ln N + O(N)$ might not be enough to estimate it within a factor of 2, even when N is 1 million. To highlight exponential differences, we often refer informally to a quantity as being *exponentially small* if it is smaller than any negative power of N —that is, $O(1/N^M)$ for any positive M . Typical examples of exponentially small quantities are e^{-N} , $e^{-\log^2 N}$, and $(\log N)^{-\log N}$.

Exercise 4.7 Prove that e^{-N^ϵ} is exponentially small for any positive constant ϵ . (That is, given ϵ , prove that $e^{-N^\epsilon} = O(N^{-M})$ for any fixed $M > 0$.)

Exercise 4.8 Prove that $e^{-\log^2 N}$ and $(\log N)^{-\log N}$ are exponentially small.

Exercise 4.9 If $\alpha < \beta$, show that α^N is exponentially small relative to β^N . For $\beta = 1.2$ and $\alpha = 1.1$, find the absolute and relative errors when $\alpha^N + \beta^N$ is approximated by β^N , for $N = 10$ and $N = 100$.

Exercise 4.10 Show that the product of an exponentially small quantity and any polynomial in N is an exponentially small quantity.

Exercise 4.11 Find the most accurate expression for a_n implied by each of the following recurrence relationships:

$$a_n = 2a_{n/2} + O(n)$$

$$a_n = 2a_{n/2} + o(n)$$

$$a_n \sim 2a_{n/2} + n.$$

In each case assume that $a_{n/2}$ is taken to be shorthand notation for $a_{\lfloor n/2 \rfloor} + O(1)$.

Exercise 4.12 Using the definitions from Chapter 1, find the most accurate expression for a_n implied by each of the following recurrence relationships:

$$a_n = 2a_{n/2} + O(n)$$

$$a_n = 2a_{n/2} + \Theta(n)$$

$$a_n = 2a_{n/2} + \Omega(n).$$

In each case assume that $a_{n/2}$ is taken to be shorthand notation for $a_{\lfloor n/2 \rfloor} + O(1)$.

Exercise 4.13 Let $\beta > 1$ and take $f(x) = x^\alpha$ with $\alpha > 0$. If $a(x)$ satisfies the recurrence

$$a(x) = a(x/\beta) + f(x) \quad \text{for } x \geq 1 \text{ with } a(x) = 0 \text{ for } x < 1$$

and $b(x)$ satisfies the recurrence

$$b(x) = b(x/\beta + c) + f(x) \quad \text{for } x \geq 1 \text{ with } b(x) = 0 \text{ for } x < 1$$

prove that $a(x) \sim b(x)$ as $x \rightarrow \infty$. Extend your proof to apply to a broader class of functions $f(x)$.

Asymptotics of linear recurrences. Linear recurrences provide an illustration of the way that asymptotic expressions can lead to substantial simplifications. We have seen in §2.4 and §3.3 that any linear recurrent sequence $\{a_n\}$ has a rational OGF and is a linear combination of terms of the form $\beta^n n^j$. Asymptotically speaking, it is clear that only a few terms need be considered, because those with larger β exponentially dominate those with smaller β (see Exercise 4.9). For example, we saw in §2.3 that the exact solution to

$$a_n = 5a_{n-1} - 6a_{n-2}, \quad n > 1; \quad a_0 = 0 \text{ and } a_1 = 1$$

is $3^n - 2^n$, but the approximate solution 3^n is accurate to within a thousandth of a percent for $n > 25$. In short, we need keep track only of terms associated with the largest absolute value or modulus.

Theorem 4.1 (Asymptotics of linear recurrences). Assume that a rational generating function $f(z)/g(z)$, with $f(z)$ and $g(z)$ relatively prime and $g(0) \neq 0$, has a unique pole $1/\beta$ of smallest modulus (that is, $g(1/\alpha) = 0$ and $\alpha \neq \beta$ implies that $|1/\alpha| > |1/\beta|$, or $|\alpha| < |\beta|$). Then, if the multiplicity of $1/\beta$ is ν , we have

$$[z^n] \frac{f(z)}{g(z)} \sim C \beta^n n^{\nu-1} \quad \text{where} \quad C = \nu \frac{(-\beta)^\nu f(1/\beta)}{g^{(\nu)}(1/\beta)}.$$

Proof. From the discussion in §3.3, $[z^n]f(z)/g(z)$ can be expressed as a sum of terms, one associated with each root $1/\alpha$ of $g(z)$, that is of the form $[z^n]c_0(1 -$

$\alpha z)^{-\nu_\alpha}$, where ν_α is the multiplicity of α . For all α with $|\alpha| < |\beta|$, such terms are exponentially small relative to the one associated with β because

$$[z^n] \frac{1}{(1 - \alpha z)^{\nu_\alpha}} = \binom{n + \nu_\alpha - 1}{\nu_\alpha - 1} \alpha^n$$

and $\alpha^n n^M = o(\beta^n)$ for any nonnegative M (see Exercise 4.10).

Therefore, we need only consider the term associated with β :

$$[z^n] \frac{f(z)}{g(z)} \sim [z^n] \frac{c_0}{(1 - \beta z)^\nu} \sim c_0 \binom{n + \nu - 1}{\nu - 1} \beta^n \sim \frac{c_0}{(\nu - 1)!} n^{\nu - 1} \beta^n$$

(see Exercise 4.4) and it remains to determine c_0 . Since $(1 - \beta z)$ is not a factor of $f(z)$, this computation is immediate from l'Hôpital's rule:

$$c_0 = \lim_{z \rightarrow 1/\beta} (1 - \beta z)^\nu \frac{f(z)}{g(z)} = f(1/\beta) \frac{\lim_{z \rightarrow 1/\beta} (1 - \beta z)^\nu}{\lim_{z \rightarrow 1/\beta} g(z)} = f(1/\beta) \frac{\nu!(-\beta)^\nu}{g^{(\nu)}(1/\beta)}. \blacksquare$$

For recurrences leading to $g(z)$ with a unique pole of smallest modulus, this gives a way to determine the asymptotic growth of the solution, including computation of the coefficient of the leading term. If $g(z)$ has more than one pole of smallest modulus, then, among the terms associated with such poles, the ones with highest multiplicity dominate (but not exponentially). This leads to a general method for determining the asymptotic growth of the solutions to linear recurrences, a modification of the method for exact solutions given at the end of §3.3.

- Derive $g(z)$ from the recurrence.
- Compute $f(z)$ from $g(z)$ and the initial conditions.
- Eliminate common factors in $f(z)/g(z)$. This could be done by factoring both $f(z)$ and $g(z)$ and cancelling, but full polynomial factorization of the functions is not required, just computation of the greatest common divisor.
- Identify terms associated with poles of highest multiplicity among those of smallest modulus.
- Determine the coefficients, using Theorem 4.1. As indicated above, this gives very accurate answers for large n because the terms neglected are exponentially small by comparison with the terms kept.

This process leads immediately to concise, accurate, and precise approximations to solutions for linear recurrences. For example, consider the recurrence

$$a_n = 2a_{n-1} + a_{n-2} - 2a_{n-3}, \quad n > 2; \quad a_0 = 0, a_1 = a_2 = 1.$$

We found in §3.3 that the generating function for the solution is

$$a(z) = \frac{f(z)}{g(z)} = \frac{z}{(1+z)(1-2z)}.$$

Here $\beta = 2$, $\nu = 1$, $g'(1/2) = -3$, and $f(1/2) = 1/2$, so Theorem 4.1 tells us that $a_n \sim 2^n/3$, as before.

Exercise 4.14 Use Theorem 4.1 to find an asymptotic solution to the recurrence

$$a_n = 5a_{n-1} - 8a_{n-2} + 4a_{n-3} \quad \text{for } n > 2 \text{ with } a_0 = 1, a_1 = 2, \text{ and } a_2 = 4.$$

Solve the same recurrence with the initial conditions on a_0 and a_1 changed to $a_0 = 1$ and $a_1 = 2$.

Exercise 4.15 Use Theorem 4.1 to find an asymptotic solution to the recurrence

$$a_n = 2a_{n-2} - a_{n-4} \quad \text{for } n > 4 \text{ with } a_0 = a_1 = 0 \text{ and } a_2 = a_3 = 1.$$

Exercise 4.16 Use Theorem 4.1 to find an asymptotic solution to the recurrence

$$a_n = 3a_{n-1} - 3a_{n-2} + a_{n-3} \quad \text{for } n > 2 \text{ with } a_0 = a_1 = 0 \text{ and } a_2 = 1.$$

Exercise 4.17 [Miles, cf. Knuth] Show that the polynomial $z^t - z^{t-1} - \dots - z - 1$ has t distinct roots and that exactly one of the roots has modulus greater than 1, for all $t > 1$.

Exercise 4.18 Give an approximate solution for the “ t th-order Fibonacci” recurrence

$$F_N^{[t]} = F_{N-1}^{[t]} + F_{N-2}^{[t]} + \dots + F_{N-t}^{[t]} \quad \text{for } N \geq t$$

with $F_0^{[t]} = F_1^{[t]} = \dots = F_{t-2}^{[t]} = 0$ and $F_{t-1}^{[t]} = 1$.

Exercise 4.19 [Schur] Show that the number of ways to change an N -denomination bill using coin denominations d_1, d_2, \dots, d_t with $d_1 = 1$ is asymptotic to

$$\frac{N^{t-1}}{d_1 d_2 \dots d_t (t-1)!}.$$

(See Exercise 3.55.)

4.2 Asymptotic Expansions. As mentioned earlier, we prefer the equation $f(N) = c_0g_0(N) + O(g_1(N))$ with $g_1(N) = o(g_0(N))$ to the equation $f(N) = O(g_0(N))$ because it provides the constant c_0 , and therefore allows us to provide specific estimates for $f(N)$ that improve in accuracy as N gets large. If $g_0(N)$ and $g_1(N)$ are relatively close, we might wish to find a constant associated with g_1 and thus derive a better approximation: if $g_2(N) = o(g_1(N))$, we write $f(N) = c_0g_0(N) + c_1g_1(N) + O(g_2(N))$.

The concept of an *asymptotic expansion*, developed by Poincaré (cf. [6]), generalizes this notion.

Definition Given a sequence of functions $\{g_k(N)\}_{k \geq 0}$ having the property that $g_{k+1}(N) = o(g_k(N))$ for $k \geq 0$, the formula

$$f(N) \sim c_0g_0(N) + c_1g_1(N) + c_2g_2(N) + \dots$$

is called an *asymptotic series* for f , or an *asymptotic expansion* of f . The asymptotic series represents the collection of equations

$$\begin{aligned} f(N) &= O(g_0(N)) \\ f(N) &= c_0g_0(N) + O(g_1(N)) \\ f(N) &= c_0g_0(N) + c_1g_1(N) + O(g_2(N)) \\ f(N) &= c_0g_0(N) + c_1g_1(N) + c_2g_2(N) + O(g_3(N)) \\ &\vdots \end{aligned}$$

and the $g_k(N)$ are referred to as an *asymptotic scale*.

Each additional term that we take from the asymptotic series gives a more accurate asymptotic estimate. Full asymptotic series are available for many functions commonly encountered in the analysis of algorithms, and we primarily consider methods that could be extended, in principle, to provide asymptotic expansions describing quantities of interest. We can use the \sim -notation to simply drop information on error terms or we can use the O -notation or the o -notation to provide more specific information.

For example, the expression $2N\ln N + (2\gamma - 2)N + O(\log N)$ allows us to make far more accurate estimates of the average number of comparison required for quicksort than the expression $2N\ln N + O(N)$ for practical values

of N , and adding the $O(\log N)$ and $O(1)$ terms provides even more accurate estimates, as shown in Table 4.1.

Asymptotic expansions extend the definition of the \sim — notation that we considered at the beginning of §4.1. The earlier use normally would involve just one term on the right-hand side, whereas the current definition calls for a series of (decreasing) terms.

Indeed, we primarily deal with *finite* expansions, not (infinite) asymptotic series, and use, for example, the notation

$$f(N) \sim c_0 g_0(N) + c_1 g_1(N) + c_2 g_2(N)$$

to refer to a finite expansion with the implicit error term $o(g_2(N))$. Most often, we use finite asymptotic expansions of the form

$$f(N) = c_0 g_0(N) + c_1 g_1(N) + c_2 g_2(N) + O(g_3(N)),$$

obtained by simply truncating the asymptotic series. In practice, we generally use only a few terms (perhaps three or four) for an approximation, since the usual situation is to have an asymptotic scale that makes later terms extremely small in comparison to early terms for large N . For the quicksort example shown in Table 4.1, the “more accurate” formula $2N\ln N + (2\gamma - 2)N + 2\ln N + 2\gamma + 1$ gives an absolute error less than .1 already for $N = 10$.

Exercise 4.20 Extend Table 4.1 to cover the cases $N = 10^5$ and 10^6 .

The full generality of the Poincaré approach allows asymptotic expansions to be expressed in terms of *any* infinite series of functions that decrease (in an o -notation sense). However, we are most often interested in a very

N	$2(N + 1)(H_{N+1} - 1)$	$2N\ln N$	$+(2\gamma - 2)N + 2(\ln N + \gamma) + 1$	
10	44.43	46.05	37.59	44.35
100	847.85	921.03	836.47	847.84
1000	12,985.91	13,815.51	12,969.94	12,985.91
10,000	175,771.70	184,206.81	175,751.12	175,771.70

Table 4.1 Asymptotic estimates for quicksort comparison counts

restricted set of functions: indeed, we are very often able to express approximations in terms of decreasing powers of N when approximating functions as N increases. Other functions occasionally are needed, but we normally will be content with an asymptotic scale consisting of terms of decreasing series of products of powers of N , $\log N$, iterated logarithms such as $\log\log N$, and exponentials.

When developing an asymptotic estimate, it is not necessarily clear *a priori* how many terms should be carried in the expansion to get the desired accuracy in the result. For example, frequently we need to subtract or divide quantities for which we only have asymptotic estimates, so cancellations might occur that necessitate carrying more terms. Typically, we carry three or four terms in an expansion, perhaps redoing the derivation to streamline it or to add more terms once the nature of the result is known.

Taylor expansions. Taylor series are the source of many asymptotic expansions: each (infinite) Taylor expansion gives rise to an asymptotic series as $x \rightarrow 0$. Table 4.2 gives asymptotic expansions for some of the basic functions, derived from truncating Taylor series. These expansions are classical,

exponential	$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + O(x^4)$
logarithmic	$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} + O(x^4)$
binomial	$(1+x)^k = 1 + kx + \binom{k}{2}x^2 + \binom{k}{3}x^3 + O(x^4)$
geometric	$\frac{1}{1-x} = 1 + x + x^2 + x^3 + O(x^4)$
trigonometric	$\sin(x) = x - \frac{x^3}{6} + \frac{x^5}{120} + O(x^7)$ $\cos(x) = 1 - \frac{x^2}{2} + \frac{x^4}{24} + O(x^6)$

Table 4.2 Asymptotic expansions derived from Taylor series ($x \rightarrow 0$)

and follow immediately from the Taylor theorem. In the sections that follow, we describe methods of manipulating asymptotic series using these expansions. Other similar expansions follow immediately from the generating functions given in the previous chapter. The first four expansions serve as the basis for many of the asymptotic calculations that we do (actually, the first three suffice, since the geometric expansion is a special case of the binomial expansion).

For a typical example of the use of Table 4.2, consider the problem of finding an asymptotic expansion for $\ln(N - 2)$ as $N \rightarrow \infty$. We do so by pulling out the leading term, writing

$$\ln(N - 2) = \ln N + \ln\left(1 - \frac{2}{N}\right) = \ln N - \frac{2}{N} + O\left(\frac{1}{N^2}\right).$$

That is, in order to use Table 4.2, we find a substitution ($x = -2/N$) with $x \rightarrow 0$.

Or, we can use more terms of the Taylor expansion to get a more general asymptotic result. For example, the expansion

$$\ln(N + \sqrt{N}) = \ln N + \frac{1}{\sqrt{N}} - \frac{1}{2N} + O\left(\frac{1}{N^{3/2}}\right)$$

follows from factoring out $\ln N$, then taking $x = 1/\sqrt{N}$ in the Taylor expansion for $\ln(1 + x)$. This kind of manipulation is typical, and we will see many examples of it later.

Exercise 4.21 Expand $\ln(1 - x + x^2)$ as $x \rightarrow 0$, to within $O(x^4)$.

Exercise 4.22 Give an asymptotic expansion for $\ln(N^\alpha + N^\beta)$, where α and β are positive constants with $\alpha > \beta$.

Exercise 4.23 Give an asymptotic expansion for $\frac{N}{N-1} \ln \frac{N}{N-1}$.

Exercise 4.24 Estimate the value of $e^{0.1} + \cos(.1) - \ln(.9)$ to within 10^{-4} , without using a calculator.

Exercise 4.25 Show that

$$\frac{1}{9801} = 0.000102030405060708091011\cdots 47484950\cdots$$

to within 10^{-100} . How many more digits can you predict? Generalize.

Nonconvergent asymptotic series. Any convergent series leads to a full asymptotic approximation, but it is very important to note that the converse is *not* true—an asymptotic series may well be divergent. For example, we might have a function

$$f(N) \sim \sum_{k \geq 0} \frac{k!}{N^k}$$

implying (for example) that

$$f(N) = 1 + \frac{1}{N} + \frac{2}{N^2} + \frac{6}{N^3} + O\left(\frac{1}{N^4}\right)$$

even though the infinite sum does not converge. Why is this allowed? If we take any fixed number of terms from the expansion, then the equality implied from the definition is meaningful, as $N \rightarrow \infty$. That is, we have an infinite collection of better and better approximations, but the point at which they start giving useful information gets larger and larger.

Stirling's formula. The most celebrated example of a divergent asymptotic series is *Stirling's formula*, which begins as follows:

$$N! = \sqrt{2\pi N} \left(\frac{N}{e} \right)^N \left(1 + \frac{1}{12N} + \frac{1}{288N^2} + O\left(\frac{1}{N^3}\right) \right).$$

N	$N!$	$\sqrt{2\pi N} \left(\frac{N}{e} \right)^N \left(1 + \frac{1}{12N} + \frac{1}{288N^2} \right)$	absolute error	relative error
1	1	1.002183625	.0022	10^{-2}
2	2	2.000628669	.0006	10^{-3}
3	6	6.000578155	.0006	10^{-4}
4	24	24.00098829	.001	10^{-4}
5	120	120.0025457	.002	10^{-4}
6	720	720.0088701	.009	10^{-4}
7	5040	5040.039185	.039	10^{-5}
8	40,320	40320.21031	.210	10^{-5}
9	362,880	362881.3307	1.33	10^{-5}
10	3,628,800	3628809.711	9.71	10^{-5}

Table 4.3 Accuracy of Stirling's formula for $N!$

In §4.6 we show how this formula is derived, using a method that gives a full (but divergent!) series in decreasing powers of N . The fact that the series is divergent is of little concern in practice because the first few terms give an extremely accurate estimate, as shown in Table 4.3 and discussed in further detail below. Now, the constant implicit in the O -notation means that, strictly speaking, such a formula does not give complete information about a specific value of N , since the constant is arbitrary (or unspecified). In principle, one can always go to the source of the asymptotic series and prove specific bounds on the constant to overcome this objection. For example, it is possible to show that

$$N! = \sqrt{2\pi N} \left(\frac{N}{e} \right)^N \left(1 + \frac{\theta_N}{12N} \right).$$

for all $N > 1$ where $0 < \theta_N < 1$ (see, for example, [1]). As in this example, it is normally safe to assume that the constants implicit in the O -notation are small and forgo the development of precise bounds on the error. Typically, if more accuracy is desired, the next term in the asymptotic series will eventually provide it, for large enough N .

Exercise 4.26 Use the nonasymptotic version of Stirling's formula to give a bound on the error made in estimating $N^{4^{N-1}/(2^N)}$ with $N\sqrt{\pi N}/4$.

Absolute error. As defined earlier, a finite asymptotic expansion has only one O -term, and we will discuss here how to perform various standard manipulations that preserve this property. If possible, we strive to express the final answer in the form $f(N) = g(N) + O(h(N))$, so that the unknown error

N	H_N	$\ln N$	$+\gamma$	$+\frac{1}{2N}$	$+\frac{1}{12N^2}$
10	2.9289683	2.3025851	2.8798008	2.9298008	2.9289674
100	5.1873775	4.6051702	5.1823859	5.1873859	5.1873775
1000	7.4854709	6.9077553	7.4849709	7.4854709	7.4854709
10,000	9.7876060	9.2103404	9.7875560	9.7876060	9.7876060
100,000	12.0901461	11.5129255	12.0901411	12.0901461	12.0901461
1,000,000	14.3927267	13.8155106	14.3927262	14.3927267	14.3927267

Table 4.4 Asymptotic estimates of the harmonic numbers

represented by the O -notation becomes negligible in an absolute sense as N increases (which means that $h(N) = o(1)$). In an asymptotic series, we get more accurate estimates by including more terms in $g(N)$ and taking smaller $h(N)$. For example, Table 4.4 shows how adding terms to the asymptotic series for the harmonic numbers gives more accurate estimates. We show how this series is derived later in this section. Like Stirling's formula, it is a divergent asymptotic series.

Relative error. We can always express estimates in the alternative form $f(N) = g(N)(1 + O(h(N)))$, where $h(N) = o(1)$. In some situations, we have to be content with an absolute error that may increase with N . The relative error decreases as N increases, but the absolute error is not necessarily "negligible" when trying to compute $f(N)$. We often encounter this type of estimate when $f(N)$ grows exponentially. For example, Table 4.3 shows the absolute and relative error in Stirling's formula. The logarithm of Stirling's expansion gives an asymptotic series for $\ln N!$ with very small absolute error, as shown in Table 4.5.

We normally use the "relative error" formulation only when working with quantities that are exponentially large in N , like $N!$ or the Catalan numbers. In the analysis of algorithms, such quantities typically appear at intermediate stages in the calculation; then operations such as dividing two such quantities or taking the logarithm take us back into the realm of absolute error for most quantities of interest in applications.

This situation is normal when we use the cumulative counting method for computing averages. For example, to find the number of leaves in binary trees in Chapter 3, we counted the total number of leaves in all trees, then di-

N	$\ln N!$	$(N + \frac{1}{2})\ln N - N + \ln\sqrt{2\pi} + \frac{1}{12N}$	error
10	15.104413	15.104415	10^{-6}
100	363.73937556	363.73937558	10^{-11}
1000	5912.128178488163	5912.128178488166	10^{-15}
10,000	82,108.9278368143533455	82,108.9278368143533458	10^{-19}

Table 4.5 Absolute error in Stirling's formula for $\ln N!$

vided by the Catalan numbers. In that case, we could compute an exact result, but for many other problems, it is typical to divide two asymptotic estimates. Indeed, this example illustrates a primary reason for using asymptotics. The average number of nodes satisfying some property in a tree of, say, 1000 nodes will certainly be less than 1000, and we may be able to use generating functions to derive an exact formula for the number in terms of Catalan numbers and binomial coefficients. But computing that number (which might involve multiplying and dividing numbers like 2^{1000} or $1000!$) might be a rather complicated chore without asymptotics. In the next section, we show basic techniques for manipulating asymptotic expansions that allow us to derive accurate asymptotic estimates in such cases.

Table 4.6 gives asymptotic series for special number sequences that are encountered frequently in combinatorics and the analysis of algorithms. Many of these approximations are derived in this chapter as examples of manipulating and deriving asymptotic series. We refer to these expansions frequently later in the book because the number sequences themselves arise naturally when studying properties of algorithms, and the asymptotic expansions therefore provide a convenient way to accurately quantify performance characteristics and appropriately compare algorithms.

Exercise 4.27 Assume that the constant C implied in the O -notation is less than 10 in absolute value. Give specific bounds for H_{1000} implied by the absolute formula $H_N = \ln N + \gamma + O(1/N)$ and by the relative formula $H_N = \ln N(1 + O(1/\log N))$.

Exercise 4.28 Assume that the constant C implied in the O -notation is less than 10 in absolute value. Give specific bounds for the 10th Catalan number implied by the relative formula

$$\frac{1}{N+1} \binom{2N}{N} = \frac{4^N}{\sqrt{\pi N^3}} \left(1 + O\left(\frac{1}{N}\right)\right).$$

Exercise 4.29 Suppose that $f(N)$ admits a convergent representation

$$f(N) = \sum_{k \geq 0} a_k N^{-k}$$

for $N \geq N_0$ where N_0 is a fixed constant. Prove that, for any $M > 0$,

$$f(N) = \sum_{0 \leq k < M} a_k N^{-k} + O(N^{-M}).$$

Exercise 4.30 Construct a function $f(N)$ such that $f(N) \sim \sum_{k \geq 0} \frac{k!}{N^k}$.

factorials (Stirling's formula)

$$\begin{aligned} N! &= \sqrt{2\pi N} \left(\frac{N}{e} \right)^N \left(1 + \frac{1}{12N} + \frac{1}{288N^2} + O\left(\frac{1}{N^3}\right) \right) \\ \ln N! &= \left(N + \frac{1}{2} \right) \ln N - N + \ln \sqrt{2\pi} + \frac{1}{12N} + O\left(\frac{1}{N^3}\right) \end{aligned}$$

harmonic numbers

$$H_N = \ln N + \gamma + \frac{1}{2N} - \frac{1}{12N^2} + O\left(\frac{1}{N^4}\right)$$

binomial coefficients

$$\begin{aligned} \binom{N}{k} &= \frac{N^k}{k!} \left(1 + O\left(\frac{1}{N}\right) \right) \quad \text{for } k = O(1) \\ &= \frac{2^{N/2}}{\sqrt{\pi N}} \left(1 + O\left(\frac{1}{N}\right) \right) \quad \text{for } k = \frac{N}{2} + O(1) \end{aligned}$$

normal approximation to the binomial distribution

$$\binom{2N}{N-k} \frac{1}{2^{2N}} = \frac{e^{-k^2/N}}{\sqrt{\pi N}} + O\left(\frac{1}{N^{3/2}}\right)$$

Poisson approximation to the binomial distribution

$$\binom{N}{k} p^k (1-p)^{N-k} = \frac{\lambda^k e^{-\lambda}}{k!} + o(1) \quad \text{for } p = \lambda/N$$

Stirling numbers of the first kind

$$\begin{bmatrix} N \\ k \end{bmatrix} \sim \frac{(N-1)!}{(k-1)!} (\ln N)^{k-1} \quad \text{for } k = O(1)$$

Stirling numbers of the second kind

$$\left\{ \begin{array}{c} N \\ k \end{array} \right\} \sim \frac{k^N}{k!} \quad \text{for } k = O(1)$$

Bernoulli numbers

$$B_{2N} = (-1)^N \frac{(2N)!}{(2\pi)^{2N}} (-2 + O(4^{-N}))$$

Catalan numbers

$$T_N \equiv \frac{1}{N+1} \binom{2N}{N} = \frac{4^N}{\sqrt{\pi N^3}} \left(1 + O\left(\frac{1}{N}\right) \right)$$

Fibonacci numbers

$$F_N = \frac{\phi^N}{\sqrt{5}} + O(\phi^{-N}) \quad \text{where } \phi = \frac{1+\sqrt{5}}{2}$$

Table 4.6 Asymptotic expansions for special numbers ($N \rightarrow \infty$)

4.3 Manipulating Asymptotic Expansions. We use asymptotic series, especially finite expansions, not only because they provide a succinct way to express approximate results with some control on accuracy, but also because they are relatively easy to manipulate, and allow us to perform complicated operations while still working with relatively simple expressions. The reason for this is that we rarely insist on maintaining the full asymptotic series for the quantity being studied, only the first few terms of the expansion, so that we are free to discard less significant terms each time we perform a calculation. In practice, the result is that we are able to get accurate expressions describing a wide variety of functions in a canonical form involving only a few terms.

Basic properties of the O -notation. A number of elementary identities, easily proven from the definition, facilitate manipulating expressions involving O -notation. These are intuitive rules, some of which we have been implicitly using already. We use an arrow to indicate that any expression containing the left-hand side of one of these identities can be simplified by using the corresponding right-hand side, on the right-hand side of an equation:

$$\begin{aligned} f(N) &\rightarrow O(f(N)) \\ cO(f(N)) &\rightarrow O(f(N)) \\ O(cf(N)) &\rightarrow O(f(N)) \\ f(N) - g(N) &= O(h(N)) \rightarrow f(N) = g(N) + O(h(N)) \\ O(f(N))O(g(N)) &\rightarrow O(f(N)g(N)) \\ O(f(N)) + O(g(N)) &\rightarrow O(g(N)) \quad \text{if } f(N) = O(g(N)). \end{aligned}$$

It is not strictly proper to use the O -notation on the left-hand side of an equation. We do often write expressions like $N^2 + N + O(1) = N^2 + O(N) = O(N^2)$ to avoid cumbersome formal manipulations with notations like the arrow used above, but we would not use the equations $N = O(N^2)$ and $N^2 = O(N^2)$ to reach the absurd conclusion $N = N^2$.

The O -notation actually makes possible a wide range of ways of describing any particular function, but it is common practice to apply these rules to write down simple canonical expressions without constants. We write $O(N^2)$, never $NO(N)$ or $2O(N^2)$ or $O(2N^2)$, even though these are all equivalent. It is conventional to write $O(1)$ for an unspecified constant, never something like $O(3)$. Also, we write $O(\log N)$ without specifying the base of the logarithm (when it is a constant), since specifying the base amounts to giving a constant, which is irrelevant because of the O -notation.

Manipulating asymptotic expansions generally reduces in a straightforward manner to the application of one of several basic operations, which we consider in turn. In the examples, we will normally consider series with one, two, or three terms (not counting the O -term). Of course, the methods apply to longer series as well.

Exercise 4.31 Prove or disprove the following, for $N \rightarrow \infty$:

$$e^N = O(N^2), \quad e^N = O(2^N), \quad 2^{-N} = O\left(\frac{1}{N^{10}}\right), \quad \text{and} \quad N^{\ln N} = O(e^{(\ln N)^2}).$$

Simplification. The main principle that we must be cognizant of when doing asymptotics is that an asymptotic series is only as good as its O -term, so anything smaller (in an asymptotic sense) may as well be discarded. For example, the expression $\ln N + O(1)$ is mathematically equivalent to the expression $\ln N + \gamma + O(1)$, but simpler.

Substitution. The simplest and most common asymptotic series derive from substituting appropriately chosen variable values into Taylor series expansions such as those in Table 4.2, or into other asymptotic series. For example, by taking $x = -1/N$ in the geometric series

$$\frac{1}{1-x} = 1 + x + x^2 + O(x^3) \quad \text{as } x \rightarrow 0$$

we find that

$$\frac{1}{N+1} = \frac{1}{N} - \frac{1}{N^2} + O\left(\frac{1}{N^3}\right) \quad \text{as } N \rightarrow \infty.$$

Similarly,

$$e^{1/N} = 1 + \frac{1}{N} + \frac{1}{2N^2} + \frac{1}{6N^3} + \cdots + \frac{1}{k!N^k} + O\left(\frac{1}{N^{k+1}}\right).$$

Exercise 4.32 Give an asymptotic expansion for $e^{1/(N+1)}$ to within $O(N^{-3})$.

Factoring. In many cases, the “approximate” value of a function is obvious upon inspection, and it is worthwhile to rewrite the function making this explicit in terms of relative or absolute error. For example, the function

$1/(N^2 + N)$ is obviously very close to $1/N^2$ for large N , which we can express explicitly by writing

$$\begin{aligned}\frac{1}{N^2 + N} &= \frac{1}{N^2} \frac{1}{1 + 1/N} \\ &= \frac{1}{N^2} \left(1 + \frac{1}{N} + O\left(\frac{1}{N^2}\right)\right) \\ &= \frac{1}{N^2} + \frac{1}{N^3} + O\left(\frac{1}{N^4}\right).\end{aligned}$$

If we are confronted with a complicated function for which the approximate value is not immediately obvious, then a short trial-and-error process might be necessary.

Multiplication. Multiplying two asymptotic series is simply a matter of doing the term-by-term multiplications, then collecting terms. For example,

$$\begin{aligned}(H_N)^2 &= \left(\ln N + \gamma + O\left(\frac{1}{N}\right)\right) \left(\ln N + \gamma + O\left(\frac{1}{N}\right)\right) \\ &= \left((\ln N)^2 + \gamma \ln N + O\left(\frac{\log N}{N}\right)\right) \\ &\quad + \left(\gamma \ln N + \gamma^2 + O\left(\frac{1}{N}\right)\right) \\ &\quad + \left(O\left(\frac{\log N}{N}\right) + O\left(\frac{1}{N}\right) + O\left(\frac{1}{N^2}\right)\right) \\ &= (\ln N)^2 + 2\gamma \ln N + \gamma^2 + O\left(\frac{\log N}{N}\right).\end{aligned}$$

In this case, the product has less absolute asymptotic “accuracy” than the factors—the result is only accurate to within $O(\log N/N)$. This is normal, and we typically need to begin a derivation with asymptotic expansions that have more terms than desired in the result. Often, we use a two-step process: do the calculation, and if the answer does not have the desired accuracy, express the original components more accurately and repeat the calculation.

Exercise 4.33 Calculate $(H_N)^2$ to within $O(1/N)$, then to within $o(1/N)$.

For another example, we estimate N factorial squared:

$$\begin{aligned}N!N! &= \left(\sqrt{2\pi N} \left(\frac{N}{e}\right)^N \left(1 + O\left(\frac{1}{N}\right)\right)\right)^2 \\ &= 2\pi N \left(\frac{N}{e}\right)^{2N} \left(1 + O\left(\frac{1}{N}\right)\right)\end{aligned}$$

since

$$\left(1 + O\left(\frac{1}{N}\right)\right)^2 = 1 + 2O\left(\frac{1}{N}\right) + O\left(\frac{1}{N^2}\right) = 1 + O\left(\frac{1}{N}\right).$$

Division. To compute the quotient of two asymptotic series, we typically factor and rewrite the denominator in the form $1/(1-x)$ for some symbolic expression x that tends to 0, then expand as a geometric series, and multiply. For example, to compute an asymptotic expansion of $\tan x$, we can divide the series for $\sin x$ by the series for $\cos x$, as follows:

$$\begin{aligned} \tan x &= \frac{\sin x}{\cos x} = \frac{x - x^3/6 + O(x^5)}{1 - x^2/2 + O(x^4)} \\ &= \left(x - x^3/6 + O(x^5)\right) \frac{1}{1 - x^2/2 + O(x^4)} \\ &= \left(x - x^3/6 + O(x^5)\right) (1 + x^2/2 + O(x^4)) \\ &= x + x^3/3 + O(x^5). \end{aligned}$$

Exercise 4.34 Derive an asymptotic expansion for $\cot x$ to $O(x^4)$.

Exercise 4.35 Derive an asymptotic expansion for $x/(e^x - 1)$ to $O(x^5)$.

For another example, consider approximating the middle binomial coefficients $\binom{2N}{N}$. We divide the series

$$(2N)! = 2\sqrt{\pi N} \left(\frac{2N}{e}\right)^{2N} \left(1 + O\left(\frac{1}{N}\right)\right)$$

by (from above)

$$N!N! = 2\pi N \left(\frac{N}{e}\right)^{2N} \left(1 + O\left(\frac{1}{N}\right)\right)$$

to get the result

$$\binom{2N}{N} = \frac{2^{2N}}{\sqrt{\pi N}} \left(1 + O\left(\frac{1}{N}\right)\right).$$

Multiplying this by $1/(N+1) = 1/N - 1/N^2 + O(1/N^3)$ gives the approximation for the Catalan numbers in Table 4.6.

Exponentiation/logarithm. Writing $f(x)$ as $\exp\{\ln(f(x))\}$ is often a convenient start for doing asymptotics involving powers or products. For example, an alternative way to derive the asymptotics of the Catalan numbers using the Stirling approximation is to write

$$\begin{aligned} \frac{1}{N+1} \binom{2N}{N} &= \exp\{\ln((2N)!) - 2\ln N! - \ln(N+1)\} \\ &= \exp\left\{\left(2N + \frac{1}{2}\right)\ln(2N) - 2N + \ln\sqrt{2\pi} + O\left(\frac{1}{N}\right)\right. \\ &\quad - 2\left(N + \frac{1}{2}\right)\ln N + 2N - 2\ln\sqrt{2\pi} + O\left(\frac{1}{N}\right) \\ &\quad \left.- \ln N + O\left(\frac{1}{N}\right)\right\} \\ &= \exp\left\{\left(2N + \frac{1}{2}\right)\ln 2 - \frac{3}{2}\ln N - \ln\sqrt{2\pi} + O\left(\frac{1}{N}\right)\right\} \end{aligned}$$

which is again equivalent to the approximation in Table 4.6 for the Catalan numbers.

Exercise 4.36 Carry out the expansion for the Catalan numbers to within $O(N^{-4})$ accuracy.

Exercise 4.37 Calculate an asymptotic expansion for $\binom{3N}{N}/(N+1)$.

Exercise 4.38 Calculate an asymptotic expansion for $(3N)!/(N!)^3$.

Another standard example of the exp/log manipulation is the following approximation for e :

$$\begin{aligned} \left(1 + \frac{1}{N}\right)^N &= \exp\left\{N\ln\left(1 + \frac{1}{N}\right)\right\} \\ &= \exp\left\{N\left(\frac{1}{N} + O\left(\frac{1}{N^2}\right)\right)\right\} \\ &= \exp\left\{1 + O\left(\frac{1}{N}\right)\right\} \\ &= e + O\left(\frac{1}{N}\right). \end{aligned}$$

The last step of this derivation is justified in the next subsection. Again, we can appreciate the utility of asymptotic analysis by considering how to compute the value of this expression when (say) N is 1 million or 1 billion.

Exercise 4.39 What is the approximate value of $\left(1 - \frac{\lambda}{N}\right)^N$?

Exercise 4.40 Give a three-term asymptotic expansion of $\left(1 - \frac{\ln N}{N}\right)^N$.

Exercise 4.41 Suppose that interest on a bank account is “compounded daily”—that is, $1/365$ of the interest is added to the account each day, for 365 days. How much more interest is paid in a year on an account with \$10,000, at a 10% interest rate compounded daily, as opposed to the \$1000 that would be paid if interest were paid once a year?

Composition. From substitution into the expansion of the exponential it is obvious that

$$e^{1/N} = 1 + \frac{1}{N} + O\left(\frac{1}{N^2}\right),$$

but this is slightly different from

$$e^{O(1/N)} = 1 + O\left(\frac{1}{N}\right),$$

which was assumed in the two derivations just given. In this case, substituting $O(1/N)$ into the expansion of the exponential is still valid:

$$\begin{aligned} e^{O(1/N)} &= 1 + O\left(\frac{1}{N}\right) + O\left(\left(O\left(\frac{1}{N}\right)\right)^2\right) \\ &= 1 + O\left(\frac{1}{N}\right). \end{aligned}$$

Since we usually deal with relatively short expansions, we can often develop simple asymptotic estimates for functions with a rather complicated appearance, just by power series substitution. Specific conditions governing such manipulations are given in [7].

Exercise 4.42 Simplify the asymptotic expression $\exp\{1+1/N+O(1/N^2)\}$ without losing asymptotic accuracy.

Exercise 4.43 Find an asymptotic estimate for $\ln(\sin((N!)^{-1}))$ to within $O(1/N^2)$.

Exercise 4.44 Show that $\sin(\tan(1/N)) \sim 1/N$ and $\tan(\sin(1/N)) \sim 1/N$. Then find the order of growth of $\sin(\tan(1/N)) - \tan(\sin(1/N))$.

Exercise 4.45 Find an asymptotic estimate for H_{T_N} , where T_N is the N th Catalan number, to within $O(1/N)$.

Reversion. Suppose that we have an asymptotic expansion

$$y = x + c_2x^2 + c_3x^3 + O(x^4).$$

We omit the constant term and the coefficient of the linear term to simplify calculations. This expansion can be transformed into an equation expressing x in terms of y through a bootstrapping process similar to that used to estimate approximate solutions to recurrences in Chapter 2. First, we clearly must have

$$x = O(y)$$

because $x/y = x/(x + c_2x^2 + O(x^3))$ is bounded as $x \rightarrow 0$. Substituting into the original expansion, this means that $y = x + O(y^2)$, or

$$x = y + O(y^2).$$

Substituting into the original expansion again, we have $y = x + c_2(y + O(y^2))^2 + O(y^3)$, or

$$x = y - c_2y^2 + O(y^3).$$

Each time we substitute back into the original expansion, we get another term. Continuing, we have $y = x + c_2(y - c_2y^2 + O(y^3))^2 + c_3(y - c_2y^2 + O(y^3))^3 + O(y^4)$, or

$$x = y - c_2y^2 + (2c_2^2 - c_3)y^3 + O(y^4).$$

Exercise 4.46 Let a_n be defined as the unique positive root of the equation

$$n = a_n e^{a_n}$$

for $n > 1$. Find an asymptotic estimate for a_n , to within $O(1/(\log n)^3)$.

Exercise 4.47 Give the reversion of the power series

$$y = c_0 + c_1x + c_2x^2 + c_3x^3 + O(x^4).$$

(Hint: Take $z = (y - c_0)/c_1$.)

4.4 Asymptotic Approximations of Finite Sums. Frequently, we are able to express a quantity as a finite sum, and therefore we need to be able to accurately estimate the value of the sum. As we saw in Chapter 2, some sums can be evaluated exactly, but in many more cases, exact values are not available. Also, it may be the case that we only have estimates for the quantities themselves being summed.

In [6], De Bruijn considers this topic in some detail. He outlines a number of different cases that frequently arise, oriented around the observation that it is frequently the case that the terms in the sum vary tremendously in value. We briefly consider some elementary examples in this section, but concentrate on the *Euler-Maclaurin formula*, a fundamental tool for estimating sums with integrals. We show how the Euler-Maclaurin formula gives asymptotic expansions for the harmonic numbers and factorials (Stirling's formula).

We consider a number of applications of Euler-Maclaurin summation throughout the rest of this chapter, particularly concentrating on summands involving classical “bivariate” functions exemplified by binomial coefficients. As we will see, these applications are predicated upon estimating summands differently in different parts of the range of summation, but they ultimately depend on estimating a sum with an integral by means of Euler-Maclaurin summation. Many more details on these and related topics may be found in [2], [3], [6], [12], and [19].

Bounding the tail. When the terms in a finite sum are rapidly decreasing, an asymptotic estimate can be developed by approximating the sum with an infinite sum and developing a bound on the size of the infinite tail. The following classical example, which counts the number of permutations that are “derangements” (see Chapter 6), illustrates this point:

$$N! \sum_{0 \leq k \leq N} \frac{(-1)^k}{k!} = N!e^{-1} - R_N \quad \text{where} \quad R_N = N! \sum_{k>N} \frac{(-1)^k}{k!}.$$

Now we can bound the tail R_N by bounding the individual terms:

$$|R_N| < \frac{1}{N+1} + \frac{1}{(N+1)^2} + \frac{1}{(N+1)^3} + \dots = \frac{1}{N}$$

so that the sum is $N!e^{-1} + O(1/N)$. In this case, the convergence is so rapid that it is possible to show that the value is always equal to $N!e^{-1}$ rounded to the nearest integer.

The infinite sum involved converges to a constant, but there may be no explicit expression for the constant. However, the rapid convergence normally means that it is easy to calculate the value of the constant with great accuracy. The following example is related to sums that arise in the study of tries (see Chapter 7):

$$\sum_{1 \leq k \leq N} \frac{1}{2^k - 1} = \sum_{k \geq 1} \frac{1}{2^k - 1} - R_N, \quad \text{where} \quad R_N = \sum_{k > N} \frac{1}{2^k - 1}.$$

In this case, we have

$$0 < R_N < \sum_{k > N} \frac{1}{2^{k-1}} = \frac{1}{2^{N-1}}$$

so that the constant $1 + 1/3 + 1/7 + 1/15 + \dots = 1.6066 \dots$ is an extremely good approximation to the finite sum. It is a trivial matter to calculate the value of this constant to any reasonable desired accuracy.

Using the tail. When the terms in a finite sum are rapidly increasing, the last term often suffices to give a good asymptotic estimate for the whole sum. For example,

$$\sum_{0 \leq k \leq N} k! = N! \left(1 + \frac{1}{N} + \sum_{0 \leq k \leq N-2} \frac{k!}{N!} \right) = N! \left(1 + O\left(\frac{1}{N}\right) \right).$$

The latter equality follows because there are $N - 1$ terms in the sum, each less than $1/(N(N - 1))$.

Exercise 4.48 Give an asymptotic estimate for $\sum_{1 \leq k \leq N} 1/(k^2 H_k)$.

Exercise 4.49 Give an asymptotic estimate for $\sum_{0 \leq k \leq N} 1/F_k$.

Exercise 4.50 Give an asymptotic estimate for $\sum_{0 \leq k \leq N} 2^k / (2^k + 1)$.

Exercise 4.51 Give an asymptotic estimate for $\sum_{0 \leq k \leq N} 2^{k^2}$.

Approximating sums with integrals. More generally, we expect that we should be able to estimate the value of a sum with an integral and to take advantage of the wide repertoire of known integrals.

What is the magnitude of the error made when we use

$$\int_a^b f(x)dx \quad \text{to estimate} \quad \sum_{a \leq k < b} f(k)?$$

The answer to this question depends on how “smooth” the function $f(x)$ is. Essentially, in each of the $b - a$ unit intervals between a and b , we are using $f(k)$ to estimate $f(x)$. Letting

$$\delta_k = \max_{k \leq x < k+1} |f(x) - f(k)|$$

denote the maximum error in each interval, we can get a rough approximation to the total error:

$$\sum_{a \leq k < b} f(k) = \int_a^b f(x)dx + \Delta, \quad \text{with } |\Delta| \leq \sum_{a \leq k < b} \delta_k.$$

If the function is monotone increasing or decreasing over the whole interval $[a, b]$, then the error term telescopes to simply $\Delta \leq |f(a) - f(b)|$. For example, for the harmonic numbers, this gives the estimate

$$H_N = \sum_{1 \leq k \leq N} \frac{1}{k} = \int_1^N \frac{1}{x} dx + \Delta = \ln N + \Delta$$

with $|\Delta| \leq 1 - 1/N$, an easy proof that $H_N \sim \ln N$; and for $N!$ this gives the estimate

$$\ln N! = \sum_{1 \leq k \leq N} \ln k = \int_1^N \ln x dx + \Delta = N \ln N - N + 1 + \Delta$$

with $|\Delta| \leq \ln N$ for $\ln N!$, an easy proof that $\ln N! \sim N \ln N - N$. The accuracy in these estimates depends on the care taken in approximating the error.

More precise estimates of the error terms depend on the derivatives of the function f . Taking these into account leads to an asymptotic series derived using the *Euler-Maclaurin summation formula*, one of the most powerful tools in asymptotic analysis.

4.5 Euler-Maclaurin Summation. In the analysis of algorithms, we approximate sums with integrals in two distinct ways. In the first case, we have a function defined on a fixed interval, and we evaluate a sum corresponding to sampling the function at an increasing number of points along the interval, with smaller and smaller step sizes, with the difference between the sum and the integral converging to zero. This is akin to classical Riemann integration. In the second case, we have a fixed function and a fixed discrete step size, so the interval of integration gets larger and larger, with the difference between the sum and the integral converging to a constant. We consider these two cases separately, though they both embody the same basic method, which dates back to the 18th century.

General form for Euler-Maclaurin summation formula. The method is based on integration by parts, and involves Bernoulli numbers (and Bernoulli polynomials), which are described in §3.11. We start from the formula

$$\int_0^1 g(x)dx = \left(x - \frac{1}{2}\right)g(x) \Big|_0^1 - \int_0^1 \left(x - \frac{1}{2}\right)g'(x)dx,$$

which is obtained by partial integration of $g(x)$ with the “clever” choice of the integration constant in $x - \frac{1}{2} = B_1(x)$. Using this formula with $g(x) = f(x + k)$, we get

$$\int_k^{k+1} f(x)dx = \frac{f(k+1) + f(k)}{2} - \int_k^{k+1} \left(\{x\} - \frac{1}{2}\right)f'(x)dx$$

where, as usual, $\{x\} \equiv x - [x]$ denotes the fractional part of x . Taking all values of k greater than or equal to a and less than b and summing these formulae gives

$$\int_a^b f(x)dx = \sum_{a \leq k \leq b} f(k) - \frac{f(a) + f(b)}{2} - \int_a^b \left(\{x\} - \frac{1}{2}\right)f'(x)dx$$

because $f(k)$ appears in two formulae for each value of k except a and b . Thus, rearranging terms, we have a precise relationship between a sum and the corresponding integral:

$$\sum_{a \leq k \leq b} f(k) = \int_a^b f(x)dx + \frac{f(a) + f(b)}{2} + \int_a^b \left(\{x\} - \frac{1}{2}\right)f'(x)dx.$$

To know how good an approximation this is, we need to be able to develop a bound for the integral at the end. We could do so by developing an absolute bound as at the end of the previous section, but it turns out that we can iterate this process, often leaving a very small error term because the derivatives of $f(x)$ tend to get smaller and smaller (as functions of N) and/or because the polynomial in $\{x\}$ that is also involved in the integral becomes smaller and smaller.

Theorem 4.2 (Euler-Maclaurin summation formula, first form). Let $f(x)$ be a function defined on an interval $[a, b]$ with a and b integers, and suppose that the derivatives $f^{(i)}(x)$ exist and are continuous for $1 \leq i \leq 2m$, where m is a fixed constant. Then

$$\sum_{a \leq k \leq b} f(k) = \int_a^b f(x)dx + \frac{f(a) + f(b)}{2} + \sum_{1 \leq i \leq m} \frac{B_{2i}}{(2i)!} f^{(2i-1)}(x) \Big|_a^b + R_m,$$

where B_{2i} are the Bernoulli numbers and R_m is a remainder term satisfying

$$|R_m| \leq \frac{|B_{2m}|}{(2m)!} \int_a^b |f^{(2m)}(x)|dx < \frac{4}{(2\pi)^{2m}} \int_a^b |f^{(2m)}(x)|dx.$$

Proof. We continue the argument above, using integration by parts and basic properties of the Bernoulli polynomials. For any function $g(x)$ that is differentiable in $[0, 1]$ and any $i > 0$, we can integrate $g(x)B'_{i+1}(x)$ by parts to get

$$\int_0^1 g(x)B'_{i+1}(x)dx = B_{i+1}(x)g(x) \Big|_0^1 - \int_0^1 g'(x)B_{i+1}(x)dx.$$

Now, from §3.11, we know that $B'_{i+1}(x) = (i+1)B_i(x)$ so, dividing by $(i+1)!$, we get a recurrence relation:

$$\int_0^1 g(x)\frac{B_i(x)}{i!}dx = \frac{B_{i+1}(x)}{(i+1)!}g(x) \Big|_0^1 - \int_0^1 g'(x)\frac{B_{i+1}(x)}{(i+1)!}dx.$$

Now, starting at $i = 0$ and iterating, this gives, formally,

$$\int_0^1 g(x)dx = \frac{B_1(x)}{1!}g(x) \Big|_0^1 - \frac{B_2(x)}{2!}g'(x) \Big|_0^1 + \frac{B_3(x)}{3!}g''(x) \Big|_0^1 - \dots$$

where the expansion can be pushed arbitrarily far for functions that are infinitely differentiable. More precisely, we stop the iteration after m steps, and

also note that $B_1(x) = x - \frac{1}{2}$ and $B_i(0) = B_i(1) = B_i$ for $i > 1$ with $B_i = 0$ for i odd and greater than 1 (see Exercises 3.86 and 3.87) to get the formula

$$\int_0^1 g(x)dx = \frac{g(0) + g(1)}{2} - \sum_{1 \leq i \leq m} \frac{B_{2i}}{(2i)!} g^{2i-1}(x) \Big|_0^1 - \int_0^1 g^{2m}(x) \frac{B_{2m}(x)}{(2m)!} dx.$$

Substituting $g(x) = f(x+k)$ and summing for $a \leq k < b$, this telescopes to the stated result with remainder term

$$|R_m| = \int_a^b \left| \frac{B_{2m}(\{x\})}{(2m)!} f^{(2m)}(x) \right| dx$$

in the same way as shown earlier. The stated bound on the remainder term follows from asymptotic properties of the Bernoulli numbers. (See De Bruijn [6] or Graham, Knuth, and Patashnik [12] for more details.) ■

For example, taking $f(x) = e^x$, the left-hand side is $(e^b - e^a)/(e-1)$, and all the derivatives are the same on the right-hand side, so we can divide through by $e^b - e^a$ and increase m to confirm that $1/(e-1) = \sum_k B_k/k!$.

Since the Bernoulli numbers grow to be quite large, this formula is often a *divergent* asymptotic series, typically used with small values of m . The first few Bernoulli numbers $B_0 = 1$, $B_1 = -1/2$, $B_2 = 1/6$, $B_3 = 0$, and $B_4 = -1/30$ suffice for typical applications of Theorem 4.2. We write the simpler form

$$\sum_{a \leq k \leq b} f(k) = \int_a^b f(x)dx + \frac{1}{2}f(a) + \frac{1}{2}f(b) + \frac{1}{12}f'(x) \Big|_a^b - \frac{1}{720}f'''(x) \Big|_a^b + \dots$$

with the understanding that the conditions of the theorem and the error bound have to be checked for the approximation to be valid.

Taking $f(x) = x^t$, the derivatives and remainder term vanish for large enough m , confirming the Bernoulli numbers as coefficients in expressing sums of powers of integers. We have

$$\sum_{1 \leq k \leq N} k = \frac{N^2}{2} + \frac{N}{2} = \frac{N(N+1)}{2},$$

$$\sum_{1 \leq k \leq N} k^2 = \frac{N^3}{3} + \frac{N^2}{2} + \frac{N}{6} = \frac{N(N+1)(2N+1)}{6},$$

and so forth, precisely as in §3.11.

Exercise 4.52 Use Euler-Maclaurin summation to determine the coefficients when $\sum_{1 \leq k \leq N} k^t$ is expressed as a sum of powers of N (see §3.11).

Corollary If h is an infinitely differentiable function, then

$$\sum_{0 \leq k \leq N} h(k/N) \sim N \int_0^1 h(x) dx + \frac{h(0) + h(1)}{2} + \sum_{i \geq 1} \frac{B_{2i}}{(2i)!} \frac{1}{N^{2i-1}} h^{(2i-1)}(x)|_0^1.$$

Proof. Apply Theorem 4.2 with $f(x) = h(x/N)$. ■

Dividing by N gives a Riemann sum relative to h . In other words, this corollary is a refinement of

$$\lim_{N \rightarrow \infty} \frac{1}{N} \sum_{0 \leq k \leq N} h(k/N) = \int_0^1 h(x) dx.$$

Euler-Maclaurin summation will be effective for obtaining asymptotic expansions for related sums of this type, such as

$$\sum_{0 \leq k \leq N} h(k^2/N).$$

We will be seeing applications of this shortly.

Exercise 4.53 Develop an asymptotic expansion for

$$\sum_{0 \leq k \leq N} \frac{1}{1+k/N}.$$

Exercise 4.54 Show that

$$\sum_{0 \leq k \leq N} \frac{1}{1+k^2/N^2} = \frac{\pi N}{4} + \frac{3}{4} - \frac{1}{24N} + O\left(\frac{1}{N^2}\right).$$

As stated, Theorem 4.2 will not provide sufficiently accurate estimates when the interval of summation/integration grows and the step size is fixed. For example, if we try to estimate

$$H_k = \sum_{1 \leq k \leq N} \frac{1}{k} \quad \text{with} \quad \int_a^b f(x) dx,$$

we encounter a difficulty because the difference between the sum and the integral tends to an unknown constant as $N \rightarrow \infty$. Next, we turn to a form of Euler-Maclaurin summation that addresses this problem.

Discrete form of Euler-Maclaurin summation. Taking $a = 1$ and $b = N$ in the discussion preceding Theorem 4.2 gives

$$\int_1^N f(x)dx = \sum_{1 \leq k \leq N} f(k) - \frac{1}{2}(f(1) + f(N)) - \int_1^N \left(\{x\} - \frac{1}{2}\right) f'(x)dx.$$

This formula relates the sum and the integral up to a constant factor if $f'(x)$ has a fast enough decay to 0 as $N \rightarrow \infty$. In particular, if the quantity

$$C_f = \frac{1}{2}f(1) + \int_1^\infty \left(\{x\} - \frac{1}{2}\right) f'(x)dx$$

exists, it defines the *Euler-Maclaurin constant* of f , and we have proved that

$$\lim_{N \rightarrow \infty} \left(\sum_{1 \leq k \leq N} f(k) - \int_1^N f(x)dx - \frac{1}{2}f(N) \right) = C_f.$$

Taking $f(x) = 1/x$, gives an approximation for the harmonic numbers. The Euler-Maclaurin constant for this case is known plainly as *Euler's constant*:

$$\gamma = \frac{1}{2} - \int_1^\infty \left(\{x\} - \frac{1}{2}\right) \frac{dx}{x^2}.$$

Thus

$$H_N = \ln N + \gamma + o(1).$$

The constant γ is approximately .57721 ··· and is not known to be a simple function of other fundamental constants.

Taking $f(x) = \ln x$ gives Stirling's approximation to $\ln N!$. In this case, the Euler-Maclaurin constant is

$$\int_1^\infty \left(\{x\} - \frac{1}{2}\right) \frac{dx}{x}.$$

This constant *does* turn out to be a simple function of other fundamental constants: it is equal to $\ln \sqrt{2\pi} - 1$. We will see one way to prove this in §4.7. The value $\sigma = \sqrt{2\pi}$ is known as *Stirling's constant*. It arises frequently in the analysis of algorithms and many other applications. Thus

$$\ln N! = N \ln N - N + \frac{1}{2} \ln N + \ln \sqrt{2\pi} + o(1).$$

When the Euler-Maclaurin constant is well defined, pursuing the analysis to obtain more terms in the asymptotic series is relatively easy. Summarizing the preceding discussion, we have shown that

$$\sum_{1 \leq k \leq N} f(k) = \int_1^N f(x)dx + \frac{1}{2}f(N) + C_f - \int_N^\infty \left(\{x\} - \frac{1}{2} \right) f'(x)dx.$$

Now, repeatedly integrating the remaining integral by parts, in the same fashion as earlier, we get an expansion involving Bernoulli numbers and higher-order derivatives. This often leads to a complete asymptotic expansion because it is a common fact that the high-order derivatives of functions of smooth behavior get smaller and smaller at ∞ .

Theorem 4.3 (Euler-Maclaurin summation formula, second form). Let $f(x)$ be a function defined on the interval $[1, \infty)$ and suppose that the derivatives $f^{(i)}(x)$ exist and are absolutely integrable for $1 \leq i \leq 2m$, where m is a fixed constant. Then

$$\sum_{1 \leq k \leq N} f(k) = \int_1^N f(x)dx + \frac{1}{2}f(N) + C_f + \sum_{1 \leq k \leq m} \frac{B_{2k}}{(2k)!} f^{(2k-1)}(N) + R_m,$$

where C_f is a constant associated with the function and R_{2m} is a remainder term satisfying

$$|R_{2m}| = O\left(\int_N^\infty |f^{(2m)}(x)|dx\right).$$

Proof. By induction, extending the argument in the earlier discussion. Details may be found in [6] or [12]. ■

Corollary The harmonic numbers admit a full asymptotic expansion in descending powers of N :

$$H_N \sim \ln N + \gamma + \frac{1}{2N} - \frac{1}{12N^2} + \frac{1}{120N^4} - \dots$$

Proof. Take $f(x) = 1/x$ in Theorem 4.3, use the constant γ discussed earlier and note that the remainder term is of the same order as the last term in the sum, to show that

$$H_N = \ln N + \gamma + \frac{1}{2N} - \sum_{1 \leq k < M} \frac{B_{2k}}{2kN^{2k}} + O\left(\frac{1}{N^{2m}}\right)$$

for any fixed m , which implies the stated result. ■

Corollary (Stirling's formula.) The functions $\ln N!$ and $N!$ admit a full asymptotic expansion in descending powers of N :

$$\ln N! \sim \left(N + \frac{1}{2}\right) \ln N - N + \ln \sqrt{2\pi} + \frac{1}{12N} - \frac{1}{360N^3} + \dots$$

and

$$N! \sim \sqrt{2\pi N} \left(\frac{N}{e}\right)^N \left(1 + \frac{1}{12N} + \frac{1}{288N^2} - \frac{139}{5140N^3} + \dots\right).$$

Proof. Take $f(x) = \ln x$ in Theorem 4.3 and argue as above to develop the expansion for $\ln N!$. As mentioned earlier, the first derivative is not absolutely integrable, but the Euler-Maclaurin constant exists, so Theorem 4.3 clearly holds. The expansion for $N!$ follows by exponentiation and basic manipulations discussed in §4.3. ■

Corollary The Catalan numbers admit a full asymptotic expansion in descending powers of N :

$$\frac{1}{N+1} \binom{2N}{N} \sim \frac{4^N}{N\sqrt{\pi N}} \left(1 - \frac{9}{8N} + \frac{145}{128N^2} - \dots\right).$$

Proof. This follows from elementary manipulations with the asymptotic series for $(2N)!$ and $N!$. Many of the details are given in the examples in §4.3. ■

Euler-Maclaurin summation is a general tool, which is useful only subject to the caveats that the function must be “smooth” (as many derivatives must exist as terms in the asymptotic series desired), and we must be able to calculate the integral involved. The asymptotic expansions just given for factorials, harmonic numbers, and Catalan numbers play a central role in the analysis of many fundamental algorithms, and the method arises in many other applications.

Exercise 4.55 Evaluate γ to 10 decimal places.

Exercise 4.56 Show that the generalized (second-order) harmonic numbers admit the asymptotic expansion

$$H_N^{(2)} \equiv \sum_{1 \leq k \leq N} \frac{1}{k^2} \sim \frac{\pi^2}{6} - \frac{1}{N} + \frac{1}{2N^2} - \frac{1}{6N^3} + \dots$$

Exercise 4.57 Derive an asymptotic expansion for

$$H_N^{(3)} \equiv \sum_{1 \leq k \leq N} \frac{1}{k^3}$$

to within $O(N^3)$.

Exercise 4.58 Use Euler-Maclaurin summation to estimate

$$\sum_{1 \leq k \leq N} \sqrt{k}, \quad \sum_{1 \leq k \leq N} \frac{1}{\sqrt{k}}, \quad \text{and} \quad \sum_{1 \leq k \leq N} \frac{1}{\sqrt[3]{k}}$$

to within $O(1/N^2)$.

Exercise 4.59 Derive full asymptotic expansions for

$$\sum_{1 \leq k \leq N} \frac{(-1)^k}{k} \quad \text{and} \quad \sum_{1 \leq k \leq N} \frac{(-1)^k}{\sqrt{k}}.$$

Exercise 4.60 The *Gamma function* is defined for positive real x as follows:

$$\Gamma(x) = \int_0^\infty e^{-t} t^{x-1} dt.$$

Since $\Gamma(1) = 1$ and $\Gamma(x+1) = x\Gamma(x)$ (integration by parts), $\Gamma(n) = (n-1)!$ for positive integers n . In other words, the Gamma function extends the factorial to the reals (and eventually to the complex plane, except for negative integers). Use Euler-Maclaurin summation to approximate the generalized binomial coefficient that follows immediately. Specifically, show that

$$\binom{n+\alpha}{n} \equiv \frac{\Gamma(n+\alpha+1)}{\Gamma(n+1)\Gamma(\alpha+1)} \sim \frac{n^\alpha}{\Gamma(\alpha+1)}.$$

4.6 Bivariate Asymptotics. Many of the most challenging problems that we face in approximating sums have to do with so-called bivariate asymptotics, where the summands depend both on the index of summation and on the “size” parameter that describes asymptotic growth. Suppose that the two parameters are named k and N , respectively, as we have done many times. Now, the relative values of k and N and the rate at which they grow certainly dictate the significance of asymptotic estimates. For a simple example of this, note that the function $k^N/k!$ grows exponentially as $N \rightarrow \infty$ for fixed k , but is exponentially small as $k \rightarrow \infty$ for fixed N .

In the context of evaluating sums, we generally need to consider all values of k less than N (or some function of N), and therefore we are interested in developing accurate asymptotic estimates for as large a range of k as possible. Evaluating the sum eliminates the k and leaves us back in the univariate case. In this section we will conduct a detailed examination of some bivariate functions of central importance in the analysis of algorithms: Ramanujan functions and binomial coefficients. In subsequent sections, we will see how the estimates developed here are used to obtain asymptotic approximations to sums involving these functions, and how these relate to applications in the analysis of algorithms.

Ramanujan distributions. Our first example concerns a distribution that was first studied by Ramanujan (see [4]) and later, because it arises in so many applications in the analysis of algorithms, by Knuth [16]. As we will see in Chapter 9, the performance characteristics of a variety of algorithms depend on the function

$$Q(N) \equiv \sum_{1 \leq k \leq N} \frac{N!}{(N-k)!N^k}.$$

This function is also well known in probability theory as the *birthday function*: $Q(N) + 1$ is the expected number of trials needed to find two people with the same birthday (when the year has N days). The summand is tabulated for small values of N and k in Table 4.7 and plotted in Figure 4.1. In the figure, separate curves are given for each value of N , with successive values of k connected by straight lines. The k -axis for each curve is scaled so that the curve fills the whole figure. In order to be able to estimate the value of the sum, we first need to be able to estimate the value of the summand accurately for *all* values of k , as N grows.

Theorem 4.4 (Ramanujan Q -distribution). As $N \rightarrow \infty$, the following (relative) approximation holds for $k = o(N^{2/3})$:

$$\frac{N!}{(N-k)!N^k} = e^{-k^2/(2N)} \left(1 + O\left(\frac{k}{N}\right) + O\left(\frac{k^3}{N^2}\right)\right).$$

In addition, the following (absolute) approximation holds for all k :

$$\frac{N!}{(N-k)!N^k} = e^{-k^2/(2N)} + O\left(\frac{1}{\sqrt{N}}\right).$$

Proof. The relative error bound is proved with the “exp/log” technique given in §4.3. We write

$$\begin{aligned} \frac{N!}{(N-k)!N^k} &= \frac{N(N-1)(N-2)\dots(N-k+1)}{N^k} \\ &= 1 \cdot \left(1 - \frac{1}{N}\right) \left(1 - \frac{2}{N}\right) \cdots \left(1 - \frac{1}{N}\right) \\ &= \exp\left\{\ln\left(1 - \frac{1}{N}\right) \left(1 - \frac{2}{N}\right) \cdots \left(1 - \frac{1}{N}\right)\right\} \\ &= \exp\left\{\sum_{1 \leq j < k} \ln\left(1 - \frac{j}{N}\right)\right\}. \end{aligned}$$

\downarrow	N	$k \rightarrow 1$	2	3	4	5	6	7	8	9	10
2		1	.5000								
3		1	.6667	.2222							
4		1	.7500	.3750	.0938						
5		1	.8000	.4800	.1920	.0384					
6		1	.8333	.5556	.2778	.0926	.0154				
7		1	.8571	.6122	.3499	.1499	.0428	.0061			
8		1	.8750	.6563	.4102	.2051	.0769	.0192	.0024		
9		1	.8889	.6914	.4609	.2561	.1138	.0379	.0084	.0009	
10		1	.9000	.7200	.5040	.3024	.1512	.0605	.0181	.0036	.0004

Table 4.7 Ramanujan Q -distribution $\frac{N!}{(N-k)!N^k}$

Now, for $k = o(N)$, we can apply the approximation

$$\ln(1 + x) = x + O(x^2) \quad \text{with } x = -j/N$$

from Table 4.2 and evaluate the sum:

$$\begin{aligned} \frac{N!}{(N-k)!N^k} &= \exp\left\{\sum_{1 \leq j < k}\left(-\frac{j}{N} + O\left(\frac{j^2}{N^2}\right)\right)\right\} \\ &= \exp\left\{-\frac{k(k-1)}{2N} + O\left(\frac{k^3}{N^2}\right)\right\}. \end{aligned}$$

Finally, for $k = o(N^{2/3})$ we can use the approximation $e^x = 1 + O(x)$ from Table 4.2 to get the stated relative approximation.

We need to carry both O -terms to cover the range in values of k . The $O(k^3/N^2)$ term is not sufficient by itself because, for example, it is $O(1/N^2)$

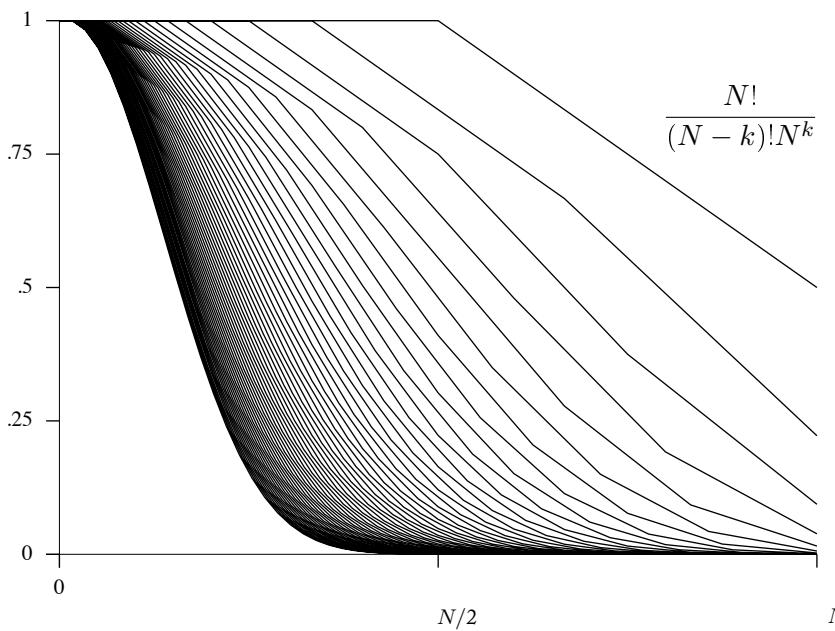


Figure 4.1 Ramanujan Q -distribution, $2 \leq N \leq 60$ (k -axes scaled to N)

for $k = O(1)$, when $O(1/N)$ is called for. The $O(k/N)$ term is not sufficient by itself because, for example, it is $O(1/N^{2/5})$ for $k = O(N^{3/5})$, when $O(1/N^{1/5})$ is called for. Now, $N^{1/5}$ is quite small for any practical value of N , and the precise value $1/5$ is not particularly critical, but we need to choose some cutoff to be able to discard a large number of terms that are even smaller. This situation illustrates the care that is necessary in bivariate asymptotics.

To develop the absolute error bound, we first consider the case where k is “small,” say $k \leq k_0$ where k_0 is the nearest integer to $N^{3/5}$. The relative approximation certainly holds, and we have

$$\frac{N!}{(N-k)!N^k} = e^{-k^2/(2N)} + e^{-k^2/(2N)}O\left(\frac{k}{N}\right) + e^{-k^2/(2N)}O\left(\frac{k^3}{N^2}\right).$$

Now, the second term is $O(1/\sqrt{N})$ because we can rewrite it in the form $xe^{-x^2}O(1/\sqrt{N})$ and $xe^{-x^2} = O(1)$ for all $x \geq 0$. Similarly, the third term is of the form $x^3e^{-x^2}O(1/\sqrt{N})$ and is $O(1/\sqrt{N})$ because $x^3e^{-x^2} = O(1)$ for all $x \geq 0$.

Next, consider the case where k is “large,” or $k \geq k_0$. The argument just given shows that

$$\frac{N!}{(N-k_0)!N^{k_0}} = e^{-\sqrt[5]{N}/2} + O\left(\frac{1}{\sqrt{N}}\right).$$

The first term is exponentially small, and the coefficients decrease as k increases, so this implies that

$$\frac{N!}{(N-k)!N^k} = O\left(\frac{1}{\sqrt{N}}\right)$$

for $k \geq k_0$. But $\exp\{-k^2/(2N)\}$ is also exponentially small for $k \geq k_0$, so the absolute error bound in the statement holds for $k \geq k_0$.

The above two paragraphs establish the absolute error bound for all $k \geq 0$. As mentioned above, the cutoff point $N^{3/5}$ is not particularly critical in this case: it need only be small enough that the relative error bound holds for smaller k (slightly smaller than $N^{2/3}$) and large enough that the terms are exponentially small for larger k (slightly larger than \sqrt{N}). ■

Corollary For all k and N , $\frac{N!}{(N-k)!N^k} \leq e^{-k(k-1)/(2N)}$.

Proof. Use the inequality $\ln(1-x) \leq -x$ instead of the asymptotic estimate in the preceding derivation. ■

A virtually identical set of arguments apply to another function studied by Ramanujan, the so-called *R-distribution*. This function is tabulated for small values of N and k in Table 4.8 and plotted in Figure 4.2. We include a detailed statement of the asymptotic results for this function as well, for reasons that will become clear later.

Theorem 4.5 (Ramanujan R-distribution). As $N \rightarrow \infty$, the following (relative) approximation holds for $k = o(N^{2/3})$:

$$\frac{N!N^k}{(N+k)!} = e^{-k^2/(2N)} \left(1 + O\left(\frac{k}{N}\right) + O\left(\frac{k^3}{N^2}\right) \right).$$

In addition, the following (absolute) approximation holds for all k :

$$\frac{N!N^k}{(N+k)!} = e^{-k^2/(2N)} + O\left(\frac{1}{\sqrt{N}}\right).$$

N									
$\downarrow k \rightarrow$	1	2	3	4	5	6	7	8	9
2	.6667	.3333							
3	.7500	.4500	.2250						
4	.8000	.5333	.3048	.1524					
5	.8333	.5952	.3720	.2067	.1033				
6	.8571	.6429	.4286	.2571	.1403	.0701			
7	.8750	.6806	.4764	.3032	.1768	.0952	.0476		
8	.8889	.7111	.5172	.3448	.2122	.1212	.0647	.0323	
9	.9000	.7364	.5523	.3823	.2458	.1475	.0830	.0439	.0220

Table 4.8 Ramanujan *R*-distribution $\frac{N!N^k}{(N+k)!}$

Proof. After the first step, the proof is virtually identical to the proof for the Q -distribution:

$$\begin{aligned} \frac{N!N^k}{(N-k)!} &= \frac{N^k}{(N+k)(N+k-1)\dots(N+1)} \\ &= \frac{1}{\left(1+\frac{k}{N}\right)\left(1+\frac{k-1}{N}\right)\cdots\left(1+\frac{1}{N}\right)} \\ &= \exp\left\{-\sum_{1\leq j\leq k}\ln\left(1+\frac{j}{N}\right)\right\} \\ &= \exp\left\{\sum_{1\leq j\leq k}\left(-\frac{j}{N}+O\left(\frac{j^2}{N^2}\right)\right)\right\} \\ &= \exp\left\{-\frac{k(k+1)}{2N}+O\left(\frac{k^3}{N^2}\right)\right\} \\ &= e^{-k^2/(2N)}\left(1+O\left(\frac{k}{N}\right)+O\left(\frac{k^3}{N^2}\right)\right). \end{aligned}$$

The absolute error bound follows as for the Q -distribution. ■

Corollary For all k and N with $k \leq N$, $\frac{N!N^k}{(N+k)!} \leq e^{-k(k+1)/(4N)}$.

Proof. Use the inequality $-\ln(1+x) \leq -x/2$ (valid for $0 \leq x \leq 1$) instead of the asymptotic estimate in the derivation above. ■

Exercise 4.61 Prove that $\frac{N!N^k}{(N+k)!} \geq e^{-k(k+1)/(2N)}$ for all k and N .

We will return to the Ramanujan distributions for several applications later in this chapter. Before doing so, however, we turn our attention to a bivariate distribution that plays an even more central role in the analysis of algorithms, the familiar *binomial distribution*. It turns out that the development of asymptotic approximations for the Ramanujan distributions given earlier encapsulates the essential aspects of approximating the binomial distribution.

Exercise 4.62 Use Stirling's formula for $\ln N!$ to prove the relative bounds for the Ramanujan Q - and R -distributions given in Theorem 4.4 and Theorem 4.5, respectively.

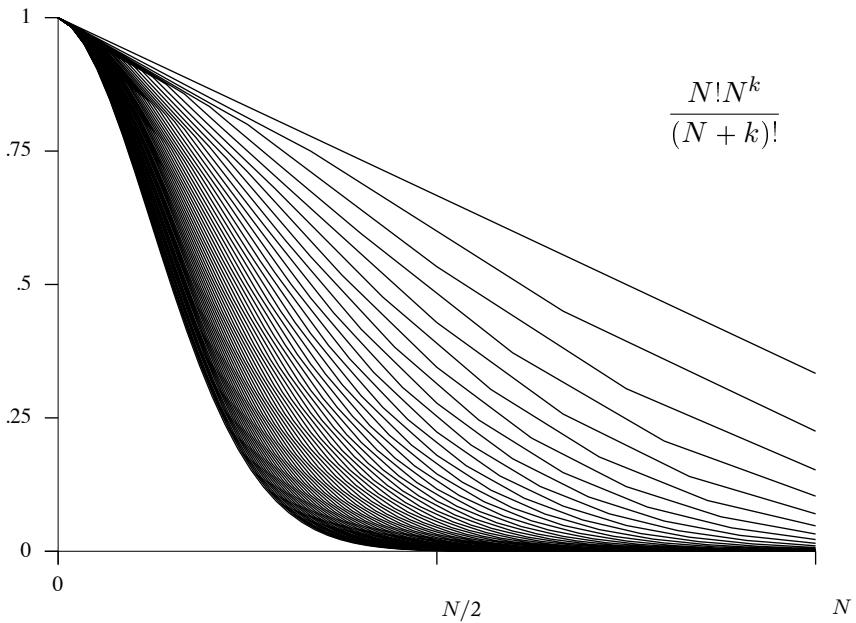


Figure 4.2 Ramanujan R -distribution, $2 \leq N \leq 60$ (k -axes scaled to N)

Binomial distribution. Given N random bits, the probability that exactly k of them are 0 is the familiar *binomial distribution*, also known as the *Bernoulli distribution*:

$$\frac{1}{2^N} \binom{N}{k} = \frac{1}{2^N} \frac{N!}{k!(N-k)!}.$$

The interested reader may consult Feller's classic text [8] or any of a number of standard references for basic information on properties of this distribution and applications in probability theory. Since it appears frequently in the analysis of algorithms, we summarize many of its important properties here.

Table 4.9 gives exact values of the binomial distribution for small N and approximate values for larger N . As usual, one motivation for doing asymptotic analysis of this function is the desire to compute such values. The value of $\binom{10000}{5000}/2^{5000}$ is about .007979, but one would not compute that by first computing $10000!$, then dividing by $5000!$, and so on. Indeed, this distribution has been studied for three centuries, and the motivation for finding easily

computed approximations was present well before computers arrived on the scene.

Exercise 4.63 Write a program to compute exact values of the binomial distribution to single-precision floating point accuracy.

We have already computed the approximate value of the middle binomial coefficients:

$$\binom{2N}{N} = \frac{2^{2N}}{\sqrt{\pi N}} \left(1 + O\left(\frac{1}{N}\right)\right).$$

That is, the middle entries in Table 4.9 decrease like $1/\sqrt{N}$. How does the distribution behave for other values of k ? Figure 4.3 shows a scaled version of the distribution that gives some indication, and a precise asymptotic analysis is given here.

The limiting curve is the familiar “bell curve” described by the normal probability density function $e^{-x^2/2}/\sqrt{2\pi}$. The top of the curve is at $(\lfloor N/2 \rfloor)/2^N \sim 1/\sqrt{\pi N/2}$, which is about .103 for $N = 60$.

Our purpose here is to analyze properties of the bell curve, using the asymptotic tools that we have been developing. The results that we present are classical, and play a central role in probability and statistics. Our interest

N	↓	$k \rightarrow$	0	1	2	3	4	5	6	7	8	9
1			.5000	.5000								
2			.2500	.5000	.2500							
3			.1250	.3750	.3750	.1250						
4			.0625	.2500	.3750	.2500	.0625					
5			.0312	.1562	.3125	.3125	.1562	.0312				
6			.0156	.0938	.2344	.3125	.2344	.0938	.0156			
7			.0078	.0547	.1641	.2734	.2734	.1641	.0547	.0078		
8			.0039	.0312	.1094	.2188	.2734	.2188	.1094	.0312	.0039	
9			.0020	.0176	.0703	.1641	.2461	.2461	.1641	.0703	.0176	.0020

Table 4.9 Binomial distribution $\binom{N}{k}/2^N$

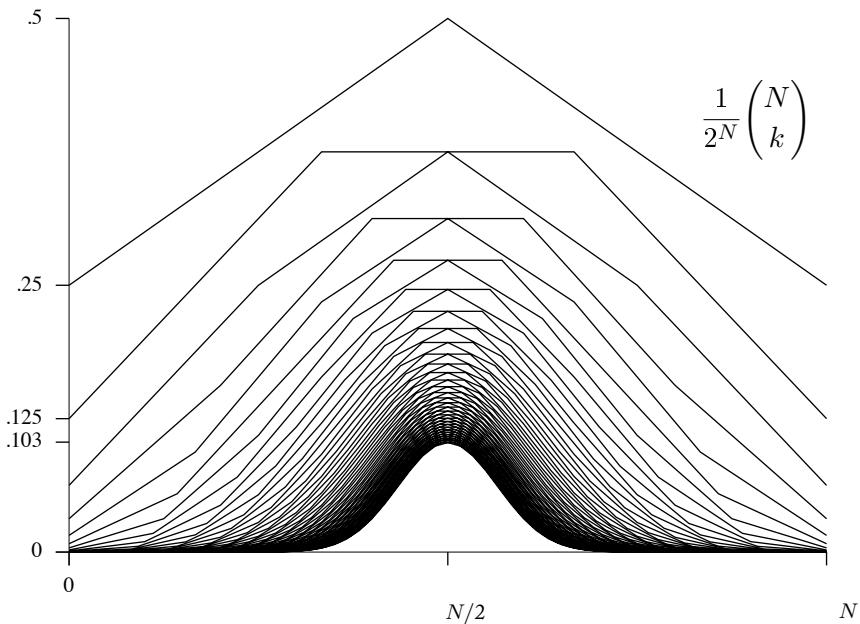


Figure 4.3 Binomial distribution, $2 \leq N \leq 60$ (k -axes scaled to N)

in these results stems not only from the fact that they are directly useful in the analysis of many algorithms, but also from the fact that the techniques used to develop asymptotic estimates for the binomial distribution are of direct use in a plethora of similar problems that arise in the analysis of algorithms. For a treatment of the normal approximation in the context of probability and statistics, see, for example, Feller [8].

Figure 4.3 makes it plain that the most significant part of the curve is near the center—as N increases, the values near the edge become negligible. This is intuitive from the probabilistic model that we started with: we expect the number of 0s and 1s in a random bitstream to be about equal, and the probability that the bits are nearly all 0 or nearly all 1 to become vanishingly small as the size of the bitstream increases. We now turn to quantifying such statements more precisely.

Normal approximation. Since the significant values of the binomial distribution are near the center, it is convenient to rewrite it and consider es-

imating $\binom{2N}{N-k}/2^{2N}$. This is symmetric about $k = 0$ and decreases as $|k|$ increases from 0 to N . This is an important step in working with any distribution: putting the largest terms at the beginning and the small terms in the tails makes it more convenient to bound the tails and concentrate on the main terms, particularly when using the approximation in evaluating sums, as we will see here.

As it turns out, we have already seen the basic methods required to prove the classic *normal approximation* to the binomial distribution.

Theorem 4.6 (Normal approximation). As $N \rightarrow \infty$, the following (relative) approximation holds for $k = o(N^{3/4})$:

$$\frac{1}{2^{2N}} \binom{2N}{N-k} = \frac{e^{-k^2/N}}{\sqrt{\pi N}} \left(1 + O\left(\frac{1}{N}\right) + O\left(\frac{k^4}{N^3}\right) \right).$$

In addition, the following (absolute) approximation holds for all k :

$$\frac{1}{2^{2N}} \binom{2N}{N-k} = \frac{e^{-k^2/N}}{\sqrt{\pi N}} + O\left(\frac{1}{N^{3/2}}\right).$$

Proof. If we write

$$\frac{1}{2^{2N}} \binom{2N}{N-k} = \frac{1}{2^{2N}} \frac{(2N)!}{N!N!} \frac{N!}{(N-k)!N^k} \frac{N!N^k}{(N+k)!}$$

we see that the binomial distribution is precisely the product of

$$\frac{1}{2^{2N}} \binom{2N}{N} = \frac{1}{\sqrt{\pi N}} \left(1 + O\left(\frac{1}{N}\right) \right),$$

the Ramanujan Q -distribution, and the Ramanujan R -distribution (!). Accordingly, we can obtain a relative approximation accurate to $O(k^3/N^2)$ by simply multiplying the asymptotic estimates for these quantities (given in Theorem 4.4, Theorem 4.5, and the third corollary to Theorem 4.3) to get the result. However, taking an extra term in the derivation leads to cancellation that gives

more accuracy. As in the proofs of Theorem 4.4 and Theorem 4.5, we have

$$\begin{aligned}
 \frac{N!}{(N-k)!} \frac{N!}{(N+k)!} &= \exp \left\{ \sum_{1 \leq j < k} \ln \left(1 - \frac{j}{N} \right) - \sum_{1 \leq j \leq k} \ln \left(1 + \frac{j}{N} \right) \right\} \\
 &= \exp \left\{ \sum_{1 \leq j < k} \left(-\frac{j}{N} - \frac{j^2}{2N^2} + O\left(\frac{j^3}{N^3}\right) \right) \right. \\
 &\quad \left. - \sum_{1 \leq j \leq k} \left(\frac{j}{N} - \frac{j^2}{2N^2} + O\left(\frac{j^3}{N^3}\right) \right) \right\} \\
 &= \exp \left\{ -\frac{k(k-1)}{2N} - \frac{k(k+1)}{2N} + O\left(\frac{k^2}{N^2}\right) + O\left(\frac{k^4}{N^3}\right) \right\} \\
 &= \exp \left\{ -\frac{k^2}{N} + O\left(\frac{k^2}{N^2}\right) + O\left(\frac{k^4}{N^3}\right) \right\} \\
 &= e^{-k^2/N} \left(1 + O\left(\frac{k^2}{N^2}\right) + O\left(\frac{k^4}{N^3}\right) \right).
 \end{aligned}$$

The improved accuracy results from cancellation of the j^2/N^2 terms in the sums. The $O(k^2/N^2)$ term can be replaced by $O(1/N)$ because k^2/N^2 is $O(1/N)$ if $k \leq \sqrt{N}$ and $O(k^4/N^3)$ if $k \geq \sqrt{N}$.

The same procedure as in the proof of Theorem 4.4 can be used to establish the absolute error bound for all $k \geq 0$, and by symmetry in the binomial coefficients, it holds for all k . ■

This is the normal approximation to the binomial distribution: the function $e^{-k^2/N}/\sqrt{\pi N}$ is the well-known “bell curve” at the bottom of Figure 4.3. Most of the curve is within plus or minus a small constant times \sqrt{N} of the mean, and the tails decay exponentially outside that range. Typically, when working with the normal approximation, we need to make use of both of these facts.

Corollary For all k and N , $\frac{1}{2^{2N}} \binom{2N}{N-k} \leq e^{-k^2/(2N)}$.

Proof. Note that $\binom{2N}{N}/2^{2N} < 1$, and multiply the bounds in the corollaries to Theorem 4.4 and Theorem 4.5. ■

This kind of result is often used to bound the tail of the distribution in the following manner. Given $\epsilon > 0$, we have, for $k > \sqrt{2N^{1+\epsilon}}$,

$$\frac{1}{2^{2N}} \binom{2N}{N-k} \leq e^{-(2N)^\epsilon}.$$

That is, the tails of the distribution are exponentially small when k grows slightly faster than \sqrt{N} .

Exercise 4.64 Carry out the normal approximation to $O(1/N^2)$ for the case $k = \sqrt{N} + O(1)$.

Exercise 4.65 Plot the smallest k for which the binomial probabilities are greater than .001 as a function of N .

Poisson approximation. In slightly more general form, the binomial distribution gives the probability of k successes in N independent trials, each having a probability p of success:

$$\binom{N}{k} p^k (1-p)^{N-k}.$$

As we discuss in detail in Chapter 9, this situation is often studied in terms of the “occupancy distribution” when N balls are distributed into M urns. Taking $p = 1/M$, the probability that exactly k balls fall into a particular urn (for instance, the first urn) is

$$\binom{N}{k} \left(\frac{1}{M}\right)^k \left(1 - \frac{1}{M}\right)^{N-k}.$$

The “balls-and-urns” model is classical (see, for example, [8]), and it also turns out to be directly applicable to a variety of fundamental algorithms that are in widespread use, as we will see in Chapters 8 and 9.

Whenever M is a constant, the distribution can still be approximated by a normal distribution centered at Np , as verified in Exercise 4.67 and illustrated in Figure 4.4, for $p = 1/5$.

The case where M varies with N is of special interest. In other words, we take $p = 1/M = \lambda/N$, where λ is a constant. This corresponds to performing N trials, each of which has a small (λ/N) probability of success. We

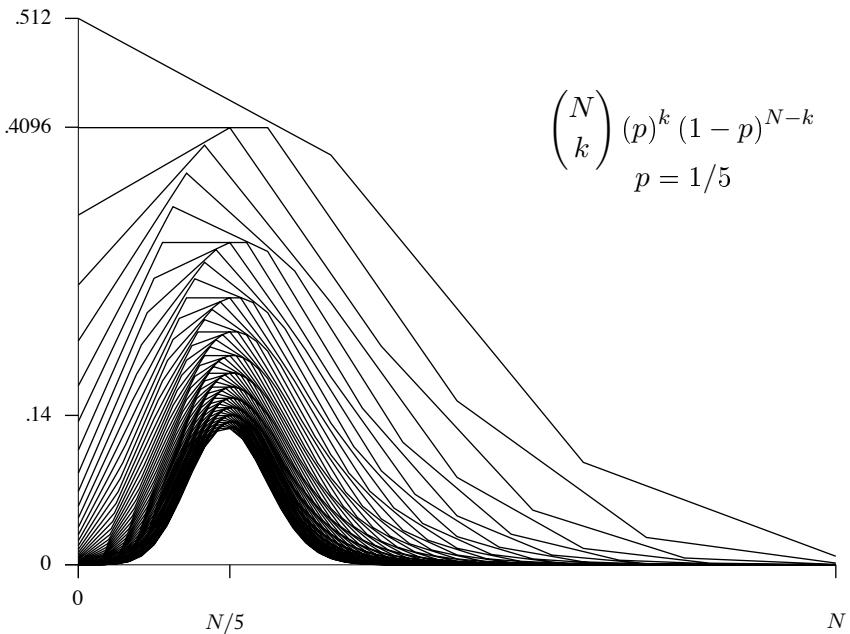


Figure 4.4 Binomial distribution, $3 \leq N \leq 60$ (k -axes scaled to N)

thus expect an average of λ trials to succeed. The probability law corresponding to this (in the asymptotic limit) is called the *Poisson law*. It applies when we want to describe the collective behavior of a large number of “agents,” each with a small probability of being “active,” “successful,” or somehow otherwise distinguished. One of the very first applications of the Poisson law (due to Von Bortkiewicz in the 19th century) was to characterize the number of cavalrymen killed by horse kicks in the Prussian Army. Since then, it has been used to describe a large variety of situations, from radioactive decay to bomb hits on London (cf. [8]). In the present context, we are interested in the Poisson law because it is an appropriate model for many fundamental algorithms, most notably hashing algorithms (see Chapter 9).

The Poisson law describes the situation of any given urn (e.g., the first one) in the balls-and-urns model when the number of urns is within a constant factor of the number of balls. Ultimately, this behavior is described by a simple asymptotic expression

Theorem 4.7 (Poisson approximation). For fixed λ , $N \rightarrow \infty$ and all k

$$\binom{N}{k} \left(\frac{\lambda}{N}\right)^k \left(1 - \frac{\lambda}{N}\right)^{N-k} = \frac{\lambda^k e^{-\lambda}}{k!} + o(1).$$

In particular, for $k = O(N^{1/2})$

$$\binom{N}{k} \left(\frac{\lambda}{N}\right)^k \left(1 - \frac{\lambda}{N}\right)^{N-k} = \frac{\lambda^k e^{-\lambda}}{k!} \left(1 + O\left(\frac{1}{N}\right) + O\left(\frac{k}{N}\right)\right).$$

Proof. Rewriting the binomial coefficient in yet another form,

$$\binom{N}{k} \left(\frac{\lambda}{N}\right)^k \left(1 - \frac{\lambda}{N}\right)^{N-k} = \frac{\lambda^k}{k!} \frac{N!}{(N-k)! N^k} \left(1 - \frac{\lambda}{N}\right)^{N-k},$$

we see that the Ramanujan Q -distribution again appears. Combining the result of Theorem 4.4 with

$$\begin{aligned} \left(1 - \frac{\lambda}{N}\right)^{N-k} &= \exp\left\{(N-k)\ln\left(1 - \frac{\lambda}{N}\right)\right\} \\ &= \exp\left\{(N-k)\left(-\frac{\lambda}{N} + O\left(\frac{1}{N^2}\right)\right)\right\} \\ &= e^{-\lambda} \left(1 + O\left(\frac{1}{N}\right) + O\left(\frac{k}{N}\right)\right) \end{aligned}$$

we get the stated relative error bound.

To prove the first part of the theorem (the absolute error bound), observe that for $k > \sqrt{N}$ (and λ fixed) both the Poisson and the binomial terms are exponentially small, and in particular $o(1)$. ■

As k grows, the $\lambda^k/k!$ term grows until $k = \lfloor \lambda \rfloor$, then becomes very small very quickly. As with the normal approximation, a bound on the tail can easily be derived by working through the above derivation, using inequalities rather than the O -notation. Figure 4.5 shows how the distribution evolves for small fixed λ as N grows. Table 4.10 gives the binomial distribution and Poisson approximation for $\lambda = 3$. The approximation is a fairly accurate estimate for the binomial distribution whenever probabilities are relatively small, even for small N .

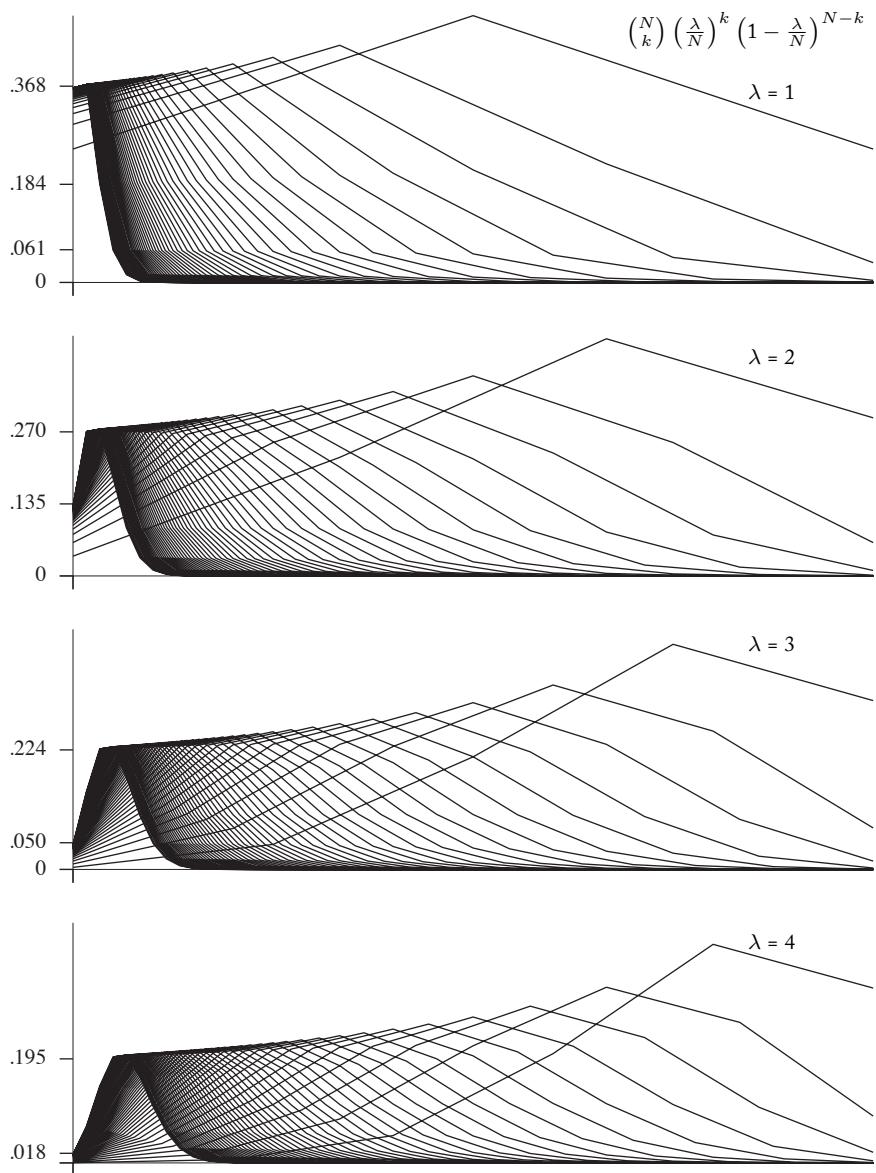


Figure 4.5 Binomial distributions, $3 \leq N \leq 60$ with $p = \lambda/N$, tending to Poisson distributions $\lambda^k e^{-\lambda}/k!$ (k -axes scaled to N)

The function $\lambda^k e^{-\lambda} / k!$ is the *Poisson distribution*, which, as mentioned earlier, models many stochastic processes. The PGF of this distribution is $e^{\lambda(z-1)}$, whence both the mean and the variance are λ . See Feller [8] for more details.

Exercise 4.66 Give an asymptotic approximation to $\binom{N}{pN} p^{pN} (1-p)^{N-pN}$.

Exercise 4.67 Give an asymptotic approximation to $\binom{N}{k} p^k (1-p)^{N-k}$ for p fixed.
(Hint: Shift so that the largest terms in the distribution are at $k = 0$.)

Exercise 4.68 Give an asymptotic approximation of the binomial distribution for the case where $p = \lambda/\sqrt{N}$.

Exercise 4.69 Give an asymptotic approximation of the binomial distribution for the case where $p = \lambda/\ln N$.

N	\downarrow	$k \rightarrow$	0	1	2	3	4	5	6	7	8	9
4			.0039	.0469	.2109	.4219	.3164					
5			.0102	.0768	.2304	.3456	.2592	.0778				
6			.0156	.0938	.2344	.3125	.2344	.0938	.0156			
7			.0199	.1044	.2350	.2938	.2203	.0991	.0248	.0027		
8			.0233	.1118	.2347	.2816	.2112	.1014	.0304	.0052	.0004	
9			.0260	.1171	.2341	.2731	.2048	.1024	.0341	.0073	.0009	
10			.0282	.1211	.2335	.2668	.2001	.1029	.0368	.0090	.0014	.0001
11			.0301	.1242	.2329	.2620	.1965	.1031	.0387	.0104	.0019	.0002
12			.0317	.1267	.2323	.2581	.1936	.1032	.0401	.0115	.0024	.0004
13			.0330	.1288	.2318	.2550	.1912	.1033	.0413	.0124	.0028	.0005
14			.0342	.1305	.2313	.2523	.1893	.1032	.0422	.0132	.0031	.0006
100			.0476	.1471	.2252	.2275	.1706	.1013	.0496	.0206	.0074	.0023
∞			.0498	.1494	.2240	.2240	.1680	.1008	.0504	.0216	.0027	.0009

Table 4.10 Binomial distribution $\binom{N}{k} \left(\frac{3}{N}\right)^k \left(1 - \frac{3}{N}\right)^{N-k}$, tending to Poisson distribution $3^k e^{-3}/k!$

4.7 Laplace Method. In the theorems of the previous section, we saw that different bounds are appropriate for different parts of the range for bivariate distributions. When estimating sums across the whole range, we want to take advantage of our ability to get accurate estimates of the summand in various different parts of the range. On the other hand, it is certainly more convenient if we can stick to a single function across the entire range of interest.

In this section, we discuss a general method that allows us to do both—the *Laplace method* for estimating the value of sums and integrals. We frequently encounter sums in the analysis of algorithms that can be estimated with this approach. Generally, we are also taking advantage of our ability to approximate sums with integrals in such cases. Full discussion and many ex-

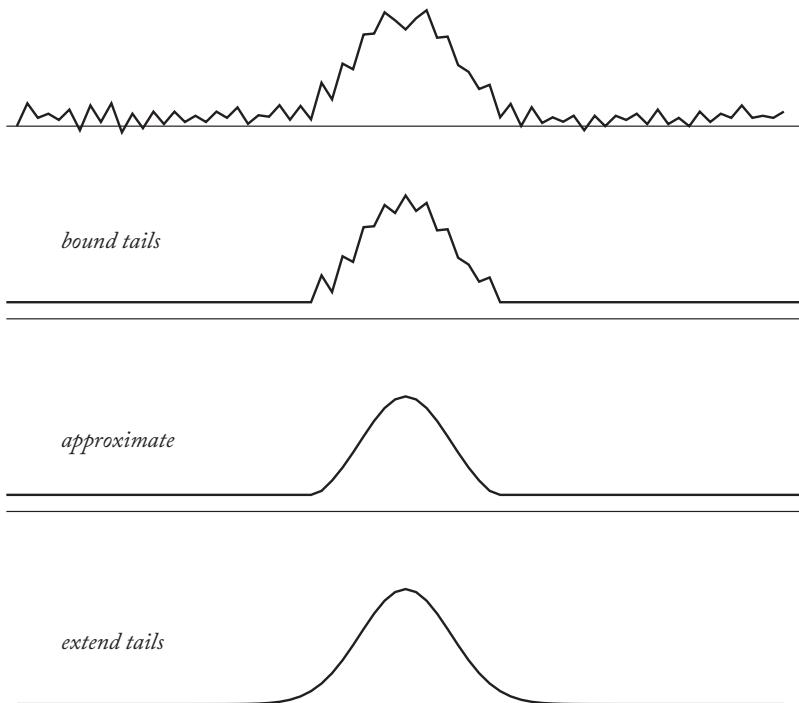


Figure 4.6 Laplace method

amples may be found in Bender and Orszag [2] or De Bruijn [6]. The method is centered on the following three steps for evaluating sums:

- Restrict the range to an area that contains the largest summands.
- Approximate the summand and bound the tails.
- Extend the range and bound the new tails, to get a simpler sum.

Figure 4.6 illustrates the method in a schematic manner. Actually, when approximating sums, the functions involved are all step functions; usually a “smooth” function makes an appearance at the end, in an application of the Euler-Maclaurin formula.

A typical case of the application of the Laplace method is the evaluation of the Ramanujan Q -function that we introduced at the beginning of the previous section. As mentioned at that time, it is of interest in the analysis of algorithms because it arises in many applications, including hashing algorithms, random maps, equivalence algorithms, and analysis of memory cache performance (see Chapter 9).

Theorem 4.8 (Ramanujan Q -function). As $N \rightarrow \infty$,

$$Q(N) \equiv \sum_{1 \leq k \leq N} \frac{N!}{(N-k)!N^k} = \sqrt{\pi N/2} + O(1).$$

Proof. Neither of the estimates given in Theorem 4.4 is useful across the entire range of summation, so we have to restrict each to the portion of the range to which it applies. More precisely, we define k_0 to be an integer that is $o(N^{2/3})$ and divide the sum into two parts:

$$\sum_{1 \leq k \leq N} \frac{N!}{(N-k)!N^k} = \sum_{1 \leq k \leq k_0} \frac{N!}{(N-k)!N^k} + \sum_{k_0 < k \leq N} \frac{N!}{(N-k)!N^k}.$$

We approximate the two parts separately, using the different restrictions on k in the two parts to advantage. For the first (main) term, we use the relative approximation in Theorem 4.4. For the second term (the tail), the restriction $k > k_0$ and the fact that the terms are decreasing imply that they are all exponentially small, as discussed in the proof of Theorem 4.4. Putting these two observations together, we have shown that

$$Q(N) = \sum_{1 \leq k \leq k_0} e^{-k^2/(2N)} \left(1 + O\left(\frac{k}{N}\right) + O\left(\frac{k^3}{N^2}\right) \right) + \Delta.$$

Here we use Δ as a notation to represent a term that is exponentially small, but otherwise unspecified. Moreover, $\exp(-k^2/(2N))$ is *also* exponentially small for $k > k_0$ and we can add the terms for $k > k_0$ back in, so we have

$$Q(N) = \sum_{k \geq 1} e^{-k^2/(2N)} + O(1).$$

Essentially, we have replaced the tail of the original sum with the tail of the approximation, which is justified because both are exponentially small. We leave the proof that the error terms contribute an absolute error that is $O(1)$ as an exercise below, because it is a slight modification of the proof for the main term, which is discussed in the next paragraph. Of course, the $O(1)$ also absorbs the exponentially small terms.

The remaining sum is the sum of values of the function $e^{x^2/2}$ at regularly spaced points with step $1/\sqrt{N}$. Thus, the Euler-Maclaurin theorem provides the approximation

$$\sum_{k \geq 1} e^{-k^2/(2N)} = \sqrt{N} \int_0^\infty e^{-x^2/2} dx + O(1).$$

The value of this integral is well known to be $\sqrt{\pi/2}$. Substituting into the above expression for $Q(N)$ gives the stated result. ■

Note that, in this case, the large terms occur for small values of k , so that we had only one tail to take care of. In general, as depicted in Figure 4.6, the dominant terms occur somewhere in the middle of the range, so that both left and right tails have to be treated.

Exercise 4.70 By applying Euler-Maclaurin summation to the functions $xe^{-x^2/2}$ and $x^3e^{-x^3/2}$, show that

$$\sum_{1 \leq k \leq k_0} e^{-k^2/(2N)} O\left(\frac{k}{N}\right) \quad \text{and} \quad \sum_{1 \leq k \leq k_0} e^{-k^2/(2N)} O\left(\frac{k^3}{N^2}\right)$$

are both $O(1)$.

The Q -function takes on several forms; also Knuth [15] defines two related functions, the P -function and the R -function (which is a sum of the

$$\begin{aligned}
 Q(N) &= \sum_{1 \leq k \leq N} \frac{N!}{(N-k)!N^k} \\
 &= \sum_{1 \leq k \leq N} \prod_{1 \leq j < k} \left(1 - \frac{j}{N}\right) = \sum_{1 \leq k \leq N} \left(1 - \frac{1}{N}\right) \left(1 - \frac{2}{N}\right) \cdots \left(1 - \frac{k-1}{N}\right) \\
 &= \sum_{0 \leq k < N} \frac{N!}{k!} \frac{N^k}{N^N} \\
 &= \sum_k \binom{N}{k} \frac{k!}{N^k} - 1 \\
 &= \sqrt{\frac{\pi N}{2}} - \frac{1}{3} + \frac{1}{12} \sqrt{\frac{\pi}{2N}} + O\left(\frac{1}{N}\right)
 \end{aligned}$$

$$\begin{aligned}
 P(N) &= \sum_{0 \leq k \leq N} \frac{(N-k)^k (N-k)!}{N!} \\
 &= \sum_{1 \leq k \leq N} \prod_{1 \leq j < k} \left(\frac{N-k}{N-j}\right) \\
 &= \sum_{0 \leq k < N} \frac{k!}{N!} \frac{k^N}{k^k} \\
 &= \sum_k \frac{(N-k)^k}{k! \binom{N}{k}} \\
 &= \sqrt{\frac{\pi N}{2}} - \frac{2}{3} + \frac{11}{24} \sqrt{\frac{\pi}{2N}} + O\left(\frac{1}{N}\right)
 \end{aligned}$$

$$\begin{aligned}
 R(N) &= \sum_{k \geq 0} \frac{N! N^k}{(N+k)!} \\
 &= \sum_{1 \leq k \leq N} \prod_{1 \leq j < k} \left(\frac{N}{N+j}\right) \\
 &= \sum_{k \geq N} \frac{N!}{k!} \frac{N^k}{N^N} \\
 &= \sqrt{\frac{\pi N}{2}} + \frac{1}{3} + \frac{1}{12} \sqrt{\frac{\pi}{2N}} + O\left(\frac{1}{N}\right)
 \end{aligned}$$

Table 4.11 Ramanujan P -, Q -, and R -functions

R -distribution that we considered earlier). These, along with the asymptotic estimates given by Knuth [15], are shown in Table 4.11. Note that

$$Q(N) + R(N) = \sum_k \frac{N!}{k!} \frac{N^k}{N^N} = \frac{N!}{N^N} e^N = \sqrt{2\pi N} + \frac{1}{6} \sqrt{\frac{\pi}{2N}} + O\left(\frac{1}{N}\right)$$

by Stirling's approximation.

Exercise 4.71 Show that

$$P(N) = \sum_{k \geq 0} \frac{(N-k)^k (N-k)!}{N!} = \sqrt{\pi N/2} + O(1)$$

Exercise 4.72 Find a direct argument showing that $R(N) - Q(N) \sim 2/3$.

4.8 “Normal” Examples from the Analysis of Algorithms. The analysis of several algorithms depends on the evaluation of a sum that is similar to the binomial distribution:

$$\sum_k F(k) \binom{2N}{N-k} / \binom{2N}{N}.$$

When $F(k)$ is reasonably well behaved, we can use the Laplace method and the Euler-Maclaurin formula (Theorem 4.2) to accurately estimate such a sum. We consider this sum in some detail because it is representative of many similar problems that arise in the analysis of algorithms and because it provides a good vehicle for further illustrating the Laplace method.

The nature of the applications involving various $F(k)$ is only briefly sketched here, since the applications are described fully in the cited sources, and we will cover related basic concepts in the chapters that follow. Our purpose here is to provide a concrete example of how asymptotic methods can give accurate estimates for complex expressions that arise in practice. Such sums are sometimes called *Catalan sums* as they arise in connection with tree enumerations and Catalan numbers, as we will see in Chapter 5. They also occur in connection with path enumerations and merging algorithms, as we will see in Chapter 6.

2-ordered sequences. A sequence of numbers is said to be *2-ordered* if the numbers in the odd positions are in ascending order and the numbers in the even positions are in ascending order. Taking a pair of ordered sequences and “shuffling” them (alternate taking one number from each sequence) gives a 2-ordered sequence. In Chapters 6 and 7, we will examine the combinatorial properties of such sequences. Analysis of a number of merging and sorting methods leads to the study of properties of 2-ordered sequences. Taking $F(k) = k^2$ in the Catalan sum gives the average number of *inversions* in a 2-ordered sequence, which is proportional to the average running time of a simple sorting method for the sequence.

Batcher’s odd-even merge. Another example involves a sorting method due to Batcher (see [16] and [20]) that is suitable for hardware implementation. Taking $F(k) = k \log k$ gives the leading term in the running time for this method, and a more accurate analysis is possible, involving a much more complicated $F(k)$.

For $F(k) = 1$, we immediately have the result $4^N / \binom{2N}{N}$, which we have already shown to be asymptotic to $\sqrt{\pi N}$. Similarly, for $F(k) = k$ and $F(k) = k^2$, it is also possible to derive an exact value for the sum as a linear combination of a few binomial coefficients, then develop asymptotic estimates using Stirling’s approximation. The methods of this section come into play when $F(k)$ is more complicated, as in the analysis of Batcher’s method. The primary assumption that we make about $F(k)$ is that it is bounded by a polynomial.

As discussed in the proof of Theorem 4.6, we are working with the product of the Ramanujan Q -distribution and R -distribution:

$$\begin{aligned} \frac{\binom{2N}{N-k}}{\binom{2N}{N}} &= \frac{\frac{(2N)!}{(N-k)!(N+k)!}}{\frac{(2N)!}{N!N!}} \\ &= \frac{N!N!}{(N-k)!(N+k)!} = \frac{N!}{(N-k)!N^k} \frac{N!N^k}{(N+k)!}. \end{aligned}$$

Thus we can use the results derived during the proof of Theorem 4.6 and its corollary in an application of the Laplace method for this problem. Choosing the cutoff point $k_0 = \sqrt{2N^{1+\epsilon}}$ for a small constant $\epsilon > 0$, we have the

approximations

$$\frac{N!N!}{(N-k)!(N+k)!} = \begin{cases} e^{-k^2/N} \left(1 + O\left(\frac{1}{N^{1-2\epsilon}}\right)\right) & \text{for } k < k_0 \\ O\left(e^{-(2N)^\epsilon}\right) & \text{for } k \geq k_0. \end{cases}$$

This is the basic information that we need for approximating the summand and then using the Laplace method. As earlier, we use the first part of this to approximate the main contribution to the sum and the second part to bound the tails:

$$\sum_k \frac{\binom{2N}{N-k}}{\binom{2N}{N}} F(k) = \sum_{|k| \leq k_0} F(k) e^{-k^2/N} \left(1 + O\left(\frac{1}{N^{1-2\epsilon}}\right)\right) + \Delta$$

where, again, Δ represents an exponentially small term. As before, we can add the tails back in because $\exp(-k^2/N)$ is also exponentially small for $k > k_0$, which leads to the following theorem.

Theorem 4.9 (Catalan sums). If $F(k)$ is bounded by a polynomial, then

$$\sum_k F(k) \binom{2N}{N-k} / \binom{2N}{N} = \sum_k e^{-k^2/N} F(k) \left(1 + O\left(\frac{1}{N^{1-2\epsilon}}\right)\right) + \Delta,$$

where Δ denotes an exponentially small error term.

Proof. See the discussion above. The condition on F is required to keep the error term exponentially small. ■

If the sequence $\{F(k)\}$ is the specialization of a real function $F(x)$ that is sufficiently smooth, then the sum is easily approximated with the Euler-Maclaurin formula. In fact, the exponential and its derivatives vanish very quickly at ∞ , so all the error terms vanish there, and, under suitable conditions on the behavior of F , we expect to have

$$\sum_{-\infty < k < \infty} F(k) \binom{2N}{N-k} / \binom{2N}{N} = \int_{-\infty}^{\infty} e^{-x^2/N} F(x) dx \left(1 + O\left(\frac{1}{N^{1-2\epsilon}}\right)\right).$$

A similar process works for one-sided sums, where k is restricted to be, say, nonnegative, leading to integrals that are similarly restricted.

Stirling's constant. Taking $F(x) = 1$ gives a familiar integral that leads to the expected solution $\sqrt{\pi N}$. Indeed, this constitutes a derivation of the value of Stirling's constant, as promised in §4.5: we know that the sum is equal to $4^N / \binom{2N}{N}$, which is asymptotic to $\sigma \sqrt{N/2}$ by the same elementary manipulations we did in §4.3, but leaving σ as an unknown in Stirling's formula. Taking $N \rightarrow \infty$, we get the result $\sigma = \sqrt{2\pi}$.

Other examples. The average number of inversions in a 2-ordered file is given by the one-sided version (that is, $k \geq 0$) of the Catalan sum with $F(x) = x^2$. This integral is easily evaluated (integration by parts) to give the asymptotic result $N\sqrt{\pi N}/4$. For Batcher's merging method, we use the one-sided sum with $F(x) = x \lg x + O(x)$ to get the estimate

$$\int_0^\infty e^{-x^2/N} x \lg x \, dx + O(N).$$

The substitution $t = x^2/N$ transforms the integral to another well-known integral, the “exponential integral function,” with the asymptotic result

$$\frac{1}{4} N \lg N + O(N).$$

It is not easy to get a better approximation for $F(x)$, and it turns out that complex analysis can be used to get a more accurate answer, as described in [20]. These results are summarized in Table 4.12.

$F(k)$	$\sum_{k \geq 0} F(k) \binom{2N}{N-k} / \binom{2N}{N}$
1	$\sim \frac{\sqrt{\pi N}}{2}$
k	$\frac{N}{2}$
$k \lg k$	$\sim \frac{N \lg N}{4}$
k^2	$N 4^{N-1} / \binom{2N}{N} \sim \frac{\sqrt{\pi N^3}}{4}$

Table 4.12 Catalan sums

Exercise 4.73 Find an asymptotic estimate for $\sum_{k \geq 0} \binom{N}{k}^3$.

Exercise 4.74 Find an asymptotic estimate for $\sum_{k \geq 0} \frac{N!}{(N-2k)!k!2^k}$.

Exercise 4.75 Find an asymptotic estimate for $\sum_{k \geq 0} \binom{N-k}{k}^2$.

4.9 “Poisson” Examples from the Analysis of Algorithms. Several other basic algorithms lead to the evaluation of sums where the largest terms are at the beginning, with an eventual exponential decrease. This kind of sum is more like the Poisson distribution with $\lambda < 1$. An example of such a sum is

$$\sum_k \binom{N}{k} \frac{f(k)}{2^k}$$

where, for example, $f(k)$ is a fast-decreasing function.

For an example, we consider the radix-exchange sorting method mentioned in Chapter 1, which is closely related to the “trie” data structure described in detail in Chapter 8. Below we show that the solution to the recurrence describing the number of bit inspections in radix-exchange sorting involves the “trie sum”

$$S_N = \sum_{j \geq 0} \left(1 - \left(1 - \frac{1}{2^j} \right)^N \right).$$

Expanding the binomial gives sums of terms of the form shown above with $f(k) = (-1)^k, (-1/2)^k, (-1/4)^k, (-1/8)^k$, and so forth. Precise evaluation of this sum is best done with complex analysis, but a very good estimate is easy to obtain from the approximation

$$1 - \left(1 - \frac{1}{2^j} \right)^N \sim 1 - e^{-N/2^j}.$$

We can show that the approximation is good to within $O(1/N)$ for $j < \lg N$ and that both sides are very small for large $j \gg \lg N$, so it is elementary to show that

$$S_N = \sum_{j \geq 0} (1 - e^{-N/2^j}) + o(1).$$

By splitting the range of summation into three parts, it is not difficult to get a good estimate for this sum. As shown in Figure 4.7, the summand is near 1 for small j , near 0 for large j , and in transition from 1 to 0 for j near $\lg N$. More precisely, for $j < (1 - \epsilon)\lg N$, the summand is very close to 1; for $j > (1 + \epsilon)\lg N$, it is very close to 0; and for j in between the bounds, the sum is certainly between 0 and 1. This argument proves that, up to smaller-order terms, the sum is between $(1 - \epsilon)\lg N$ and $(1 + \epsilon)\lg N$ for any ϵ ; a more careful choice of the bounds will show that the sum is $\sim \lg N$.

Theorem 4.10 (Trie sum). As $N \rightarrow \infty$,

$$S_N = \sum_{j \geq 0} \left(1 - \left(1 - \frac{1}{2^j} \right)^N \right) = \lg N + O(\log \log N).$$

Proof. Use bounds of $\lg N \pm \ln \ln N$ in the argument above. A more detailed argument can be used to show that $S_N = \lg N + O(1)$. We will analyze the function $S_N - \lg N$ in some detail in Chapter 8. ■

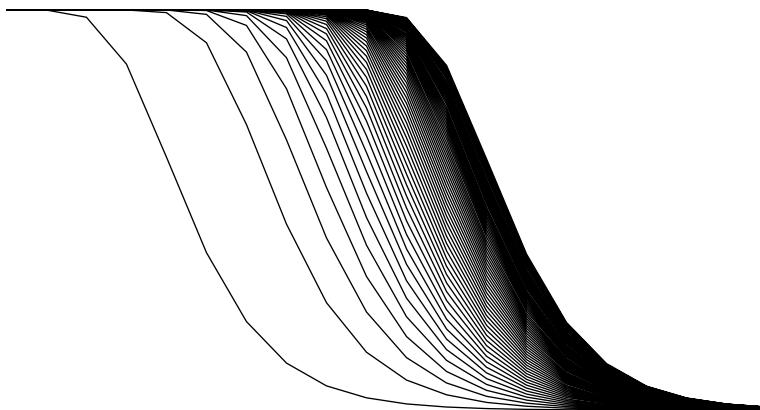


Figure 4.7 Asymptotic behavior of the terms in $\sum_{j \geq 0} (1 - e^{-N/2^j})$

Corollary The average number of bit inspections used by radix-exchange sort is $N \lg N + O(N)$.

Proof. As discussed in §1.5 this quantity satisfies the recurrence

$$C_N = N + \frac{2}{2^N} \sum_k \binom{N}{k} C_k \quad \text{for } N > 1 \text{ with } C_0 = 0 \text{ and } C_1 = 0.$$

Multiplying by z^N and summing on N leaves a straightforward convolution that simplifies to demonstrate that the EGF $\sum_{N \geq 0} C_N z^N / N!$ must satisfy the functional equation

$$C(z) = ze^z - z + 2e^{z/2}C(z/2).$$

As we might expect, this equation is also available via the symbolic method [10]. Iterating the equation, we find that

$$\begin{aligned} C(z) &= ze^z - z + 2e^{z/2}C(z/2) \\ &= ze^z - z + 2e^{z/2}\left(\frac{z}{2}e^{z/2} - \frac{z}{2} + 2e^{z/4}C(z/4)\right) \\ &= z(e^z - 1) + z(e^z - e^{z/2}) + 4e^{3z/4}C(z/4) \\ &= z(e^z - 1) + z(e^z - e^{z/2}) + z(e^z - e^{3z/4}) + 8e^{7z/8}C(z/8) \\ &\vdots \\ &= z \sum_{j \geq 0} \left(e^z - e^{(1-2^{-j})z} \right) \end{aligned}$$

and therefore

$$C_N = N![z^N]C(z) = N \sum_{j \geq 0} \left(1 - \left(1 - \frac{1}{2^j} \right)^{N-1} \right).$$

Thus we have $C_N = NS_{N-1}$, and the claimed result is established. ■

As discussed further in Chapter 8, the linear term oscillates in value. This is perhaps not surprising because the algorithm deals with bits and has the “binary divide-and-conquer” flavor of some of the algorithms discussed in §2.6. It presents significant analytic challenges nonetheless, which are best met by complex analysis methods.

Exercise 4.76 Find an asymptotic estimate for $\sum_{0 \leq j \leq N} \left(1 - \frac{1}{2^j}\right)^N$.

Exercise 4.77 Find an asymptotic estimate for $\sum_{j \geq 0} (1 - e^{-N/j^t})$ for $t > 1$.

ASYMPTOTIC methods play an essential role in the analysis of algorithms. Without asymptotics we might be left with hopelessly complex exact answers or hopelessly difficult closed-form solutions to evaluate. With asymptotics, we can focus on those parts of the solution that contribute most to the answer. This extra insight from the analysis also plays a role in the algorithm design process: when seeking to improve the performance of an algorithm, we focus on precisely those parts of the algorithm that lead to the terms we focus on in the asymptotic analysis.

Understanding distinctions among the various notations that are commonly used in asymptotic analysis is critically important. We have included many exercises and examples in this chapter to help illustrate such distinctions, and the reader is urged to study them carefully. Proper use of elementary definitions and manipulations can greatly simplify asymptotic formulas.

As with GFs, a good deal of the basic material for asymptotic analysis follows from combining well-known classical expansions, for generating functions associated with well-known special numbers such as Stirling numbers, harmonic numbers, geometric series, and binomial coefficients. Algebraic manipulations and simplification also play an important role. Indeed, the ability to suppress detail in such calculations makes the asymptotic representation very attractive.

We considered in detail the use of these methods to derive two of the most important approximations to the binomial distribution: normal and Poisson. We also considered the related functions that were studied by Ramanujan and Knuth that appear in the analysis of many algorithms. Not only are these approximations extremely useful in the analysis of algorithms, but they also are prototypical of the kind of manipulations that arise often.

We have concentrated on so-called elementary methods, from real analysis. Proficiency in elementary methods is important, especially simplifying complicated asymptotic expressions, refining estimates, approximating sums with integrals, and bounding tails, including the Laplace method. Better understanding of asymptotic analysis relies on understanding of properties of functions in the complex plane. Surprisingly, powerful methods derive

from only a few basic properties, especially singularities of generating functions. We have given a general idea of the basic method, which is considered in detail in [11]. Advanced techniques often cannot be avoided in detailed asymptotics, since complex values appear in answers. In particular, we see many examples where an oscillating phenomenon appears as the function under study grows. This seems surprising, until we recall that it appears in our most fundamental algorithms, including divide-and-conquer methods such as mergesort or methods involving binary representation of integers. Complex analysis provides a simple way to explain such phenomena.

We can conclude this chapter in a similar fashion to the previous chapters: the techniques given here can take us reasonably far in studying important properties of fundamental computer algorithms. They play a crucial role in our treatment of algorithms associated with trees, permutations, strings, and maps in Chapters 6 through 9.

References

1. M. ABRAMOWITZ AND I. STEGUN. *Handbook of Mathematical Functions*, Dover, New York, 1972.
2. C. M. BENDER AND S. A. ORSZAG. *Advanced Mathematical Methods for Scientists and Engineers*, McGraw-Hill, New York, 1978.
3. E. A. BENDER. "Asymptotic methods in enumeration," *SIAM Review* **16**, 1974, 485–515.
4. B. C. BERNDT. *Ramanujan's Notebooks, Parts I and II*, Springer-Verlag, Berlin, 1985 and 1989.
5. L. COMTET. *Advanced Combinatorics*, Reidel, Dordrecht, 1974.
6. N. G. DE BRUIJN. *Asymptotic Methods in Analysis*, Dover, New York, 1981.
7. A. ERDÉLYI. *Asymptotic Expansions*, Dover, New York, 1956.
8. W. FELLER. *An Introduction to Probability Theory and Its Applications*, volume 1, John Wiley, New York, 1957, 2nd edition, 1971.
9. P. FLAJOLET AND A. ODLYZKO. "Singularity analysis of generating functions," *SIAM Journal on Discrete Mathematics* **3**, 1990, 216–240.
10. P. FLAJOLET, M. REGNIER, AND D. SOTTEAU. "Algebraic methods for tries statistics," *Annals of Discrete Mathematics* **25**, 1985, 145–188.
11. P. FLAJOLET AND R. SEDGEWICK. *Analytic Combinatorics*, Cambridge University Press, 2009.
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, 1981.
14. PETER HENRICI. *Applied and Computational Complex Analysis*, 3 volumes, John Wiley, New York, 1977.
15. D. E. KNUTH. *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, 1st edition, Addison-Wesley, Reading, MA, 1968. Third edition, 1997.
16. D. E. KNUTH. *The Art of Computer Programming. Volume 3: Sorting and Searching*, 1st edition, Addison-Wesley, Reading, MA, 1973. Second edition, 1998.

17. D. E. KNUTH. "Big omicron and big omega and big theta," *SIGACT News*, April-June 1976, 18–24.
18. A. ODLYZKO. "Asymptotic enumeration methods," in *Handbook of Combinatorics*, volume 2, R. Graham, M. Grötschel, and L. Lovász, eds., North Holland, 1995.
19. F. W. J. OLVER. *Asymptotics and Special Functions*, Academic Press, New York, 1974, reprinted by A. K. Peters, 1997.
20. R. SEDGEWICK. "Data movement in odd-even merging," *SIAM Journal on Computing* 7, 1978, 239–272.
21. E. T. WHITTAKER AND G. N. WATSON. *A Course of Modern Analysis*, Cambridge University Press, 4th edition, 1927.
22. H. WILF. *Generatingfunctionology*, Academic Press, San Diego, 1990, 2nd edition, A. K. Peters, 2006.

This page intentionally left blank

CHAPTER FIVE

ANALYTIC COMBINATORICS

THIS chapter introduces *analytic combinatorics*, a modern approach to the study of combinatorial structures of the sort that we encounter frequently in the analysis of algorithms. The approach is predicated on the idea that combinatorial structures are typically defined by simple formal rules that are the key to learning their properties. One eventual outgrowth of this observation is that a relatively small set of *transfer theorems* ultimately yields accurate approximations of the quantities that we seek. Figure 5.1 gives an general overview of the process.

Generating functions are the central objects of study in analytic combinatorics. In the first place, we directly translate formal definitions of combinatorial objects into definitions of generating functions that enumerate objects or describe their properties. In the second place, we use classical mathematical analysis to extract estimates of generating function coefficients.

First, we treat generating functions as formal objects, which provides a convenient, compact, and elegant approach to understanding relationships among classes of combinatorial objects. The key idea is to develop a set of intuitive *combinatorial constructions* that immediately translate to equations that the associated generating functions must satisfy. We refer to this approach as the *symbolic method*, as it formalizes the idea that an object's description can be captured via symbolic mathematics.

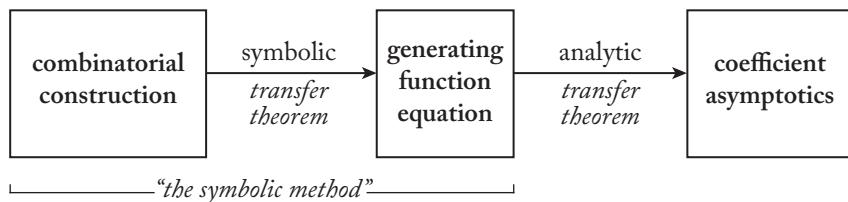


Figure 5.1 An overview of analytic combinatorics

The symbolic method yields explicit or implicit descriptions of generating functions. Treating those same generating functions as analytic objects is fruitful, as we have already seen. We can extract the coefficients that carry the information that we seek (as described in Chapter 3) and then derive asymptotic estimates of their growth (as described in Chapter 4). This process can often present significant technical challenges, but it is often the case that general transfer theorems can *immediately* provide exact or asymptotic information. In general, complex analysis is really required, so we only provide an introduction here, for examples where real analysis suffices.

Classical research in combinatorics provides the underlying framework for analytic combinatorics, but the motivation provided by the analysis of algorithms is immediate. Modern programmers define *data structures* to systematically organize the data that needs to be processed in applications. A data structure is nothing more than a combinatorial object that is formally defined (in a programming language) and so is immediately amenable to study with analytic combinatorics. Our current state of knowledge stops far short of being able to analyze any program that can be defined in any programming language, but the approach has proved to be quite successful in studying many of the most important algorithms and data structures that play a critical role in the computational infrastructure that surrounds us.

5.1 Formal Basis. To understand the motivation for the symbolic method, consider the problem of enumerating binary trees. The functional equation for binary trees that we derived in §3.8 is quite simple, so the question naturally arises whether an even more direct derivation is available. Indeed, the similarity between the recursive definition of trees and the quadratic equation of the OGF is striking. With the symbolic method, we can demonstrate that this similarity is not coincidental, but essential; we can interpret

$$T(z) = 1 + zT(z)^2$$

as a direct consequence of the definition “a binary tree is either an external node or an internal node connected to two binary trees.”

The approach that we use to do this has two primary features. First, it is symbolic, using only a few algebraic rules to manipulate symbolic information. Some authors emphasize this by actually using symbolic diagrams rather than variables like z in the generating formulae. Second, it directly mirrors

the way in which we define the structure. We generate structures, as opposed to dissecting them for analysis.

The formal basis for the symbolic method is a precise definition that captures the essential properties of any combinatorial enumeration problem.

Definition A *combinatorial class* is a set of discrete objects and an associated *size* function, where the size of each object is a nonnegative integer and the number of objects of any given size is finite.

Combinatorial objects are built from *atoms*, which are defined to be objects of size 1. Examples of atoms include 0 or 1 bits in bitstrings and internal or external nodes in trees. For a class \mathcal{A} , we denote the number of members of the class of size n by a_n and refer to this sequence as the *counting sequence* associated with \mathcal{A} . Typically the size function enumerates certain atoms, so that a combinatorial object of size n consists of n such atoms.

To specify classes, we also make use of *neutral* objects ϵ of size 0 and the *neutral class* \mathcal{E} that contains a single neutral object. We use \mathcal{Z} to represent the class containing a single atom and we use subscripts to distinguish different types of atoms. Also, we denote the empty class by ϕ .

Objects may be unlabelled (where atoms are indistinguishable) or labelled (where the atoms are all different and we consider objects with atoms appearing in different order to be different). We consider the simpler of the two cases (unlabelled objects) first.

5.2 Symbolic Method for Unlabelled Classes. For reference, Figure 5.2 gives examples of three basic unlabelled combinatorial classes. The first is just a sequence of atoms—there is only one object of each size, so this is tantamount to encoding natural numbers in unary. The second is a sequence built from one of two atoms—the objects of size N are the N -bit binary numbers, or bitstrings. The third is the class of binary trees that we considered in Chapter 3 (just built from internal nodes in this example).

Given an unlabelled class \mathcal{A} with counting sequence $\{a_n\}$, we are interested in the OGF

$$A(z) = \sum_{n \geq 0} a_n z^n.$$

Note that \mathcal{E} has OGF 1, and ϕ has OGF 0. The OGF of \mathcal{Z} is z when the associated atom is counted in the size function and 1 when it is not (this distinction will be made clear in later examples).

Natural numbers (sequences or sets of atoms)

$$I_1 = 1$$



$$I_2 = 1$$



$$I_3 = 1$$



$$I_4 = 1$$



Bitstrings (sequences of bits)

$$B_3 = 8$$

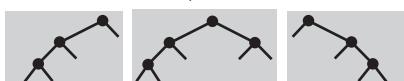
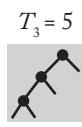
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

$$B_4 = 16$$

0 0 0 0	1 0 0 0
0 0 0 1	1 0 0 1
0 0 1 0	1 0 1 0
0 0 1 1	1 0 1 1
0 1 0 0	1 1 0 0
0 1 0 1	1 1 0 1
0 1 1 0	1 1 1 0
0 1 1 1	1 1 1 1

Binary trees (sized by number of internal nodes)

$$T_4 = 14$$



$$T_1 = 1$$



$$T_2 = 2$$

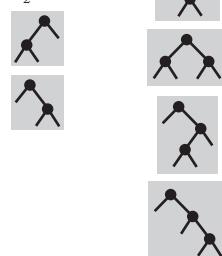


Figure 5.2 Basic unlabelled classes, $1 \leq N \leq 4$

As we have seen, the fundamental identity

$$A(z) = \sum_{n \geq 0} a_n z^n = \sum_{a \in \mathcal{A}} z^{|a|}.$$

allows us to view the OGF as an analytic form representing the counting sequence (the left sum) or as a combinatorial form representing all the individual objects (the right sum). Table 5.1 gives the counting sequences and OGFs for the classes in Figure 5.2. Generally, we use the same letter in different fonts to refer to a combinatorial class, its counting sequence, and its generating function. Often, we use lowercase when discussing generic classes and uppercase when discussing specific classes.

The essence of the symbolic method is a mechanism for deriving such OGFs *while at the same time specifying the corresponding class*. Instead of the informal English-language descriptions that we have been using, we turn to simple formal operations. With just a little experience, you will see that such operations actually *simplify* the process of specifying the combinatorial objects and classes that we are studying. To start, we define three simple and intuitive operations. Given two classes \mathcal{A} and \mathcal{B} of combinatorial objects, we can build new classes as follows:

- $\mathcal{A} + \mathcal{B}$ is the class consisting of disjoint copies of the members of \mathcal{A} and \mathcal{B} ,
- $\mathcal{A} \times \mathcal{B}$ is the class of ordered pairs of objects, one from \mathcal{A} and one from \mathcal{B} ,
- and $SEQ(\mathcal{A})$ is the class $\epsilon + \mathcal{A} + \mathcal{A} \times \mathcal{A} + \mathcal{A} \times \mathcal{A} \times \mathcal{A} + \dots$.

	atoms	size function	counting sequence	OGF
natural numbers	•	# of •s	1	$\frac{1}{1-z}$
bitstrings	0 bits 1 bits	# of bits	2^N	$\frac{1}{1-2z}$
binary trees	•	# of •s	$\frac{1}{N+1} \binom{2N}{N}$	$\frac{1 - \sqrt{1 - 4z}}{2}$

Table 5.1 Counting sequences and OGFs for classes in Figure 5.2

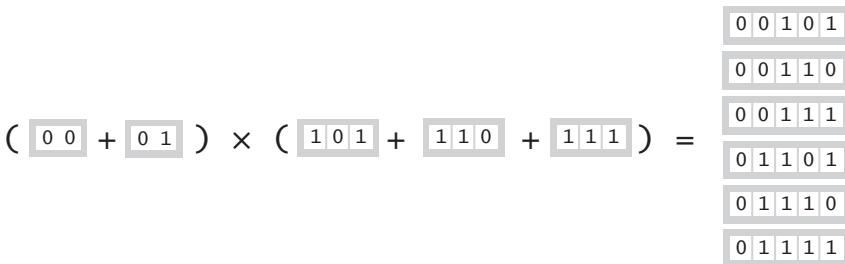


Figure 5.3 A simple combinatorial construction (unlabelled)

We refer to these operations as the *disjoint union*, *Cartesian product*, and *sequence* operations, respectively. The disjoint union operation is sometimes called the *combinatorial sum*. It is the same as the standard set union if the two arguments are disjoint but builds multisets when they are not. The sequence operation is like the concatenation operation that is used to define formal languages (sets of strings).

A *combinatorial construction* is an algebraic expression involving these operations, where each operand may be a symbol representing an atom, a symbol representing a combinatorial class, or a parenthesized combinatorial construction. Figure 5.3 gives an example of a combinatorial construction.

At this introductory level, combinatorial constructions are formally quite similar to arithmetic expressions in elementary algebra or regular expressions and other formal languages in theoretical computer science. We shall examine this latter connection in more detail in Chapter 8. Of course, the difference is in the interpretation: with combinatorial constructions, our purpose is to specify combinatorial classes in such a manner that we can solve enumeration problems. The disjoint union, Cartesian product, and sequence operations are only a beginning: several other operations that greatly expand the combinatorial classes that we can study are described in [8] (see also Exercise 5.3 and Exercise 5.4).

The importance of combinatorial constructions in analytic combinatorics stems from not just the fact that they provide a way to specify combinatorial classes of all sorts but also the fact that they imply functional relationships on generating functions. The following theorem provides a simple

correspondence between the three operations in combinatorial constructions that we have defined and their associated generating functions.

Theorem 5.1 (Symbolic method for unlabelled class OGFs). Let \mathcal{A} and \mathcal{B} be unlabelled classes of combinatorial objects. If $A(z)$ is the OGF that enumerates \mathcal{A} and $B(z)$ is the OGF that enumerates \mathcal{B} , then

$A(z) + B(z)$ is the OGF that enumerates $\mathcal{A} + \mathcal{B}$

$A(z)B(z)$ is the OGF that enumerates $\mathcal{A} \times \mathcal{B}$

$\frac{1}{1 - A(z)}$ is the OGF that enumerates $SEQ(\mathcal{A})$.

Proof. The first part of the proof is trivial. If a_n is the number of objects of size n in \mathcal{A} and b_n is the number of objects of size n in \mathcal{B} , then $a_n + b_n$ is the number of objects of size n in $\mathcal{A} + \mathcal{B}$.

To prove the second part, note that, for every k from 0 to n , we can pair any of the a_k objects of size k from \mathcal{A} with any of the b_{n-k} objects of size $n - k$ from \mathcal{B} to get an object of size n in $\mathcal{A} \times \mathcal{B}$. Thus, the number of objects of size n in $\mathcal{A} \times \mathcal{B}$ is

$$\sum_{0 \leq k \leq n} a_k b_{n-k},$$

a simple convolution that implies the stated result. Alternatively, using the combinatorial form of the OGFs, we have

$$\sum_{\gamma \in \mathcal{A} \times \mathcal{B}} z^{|\gamma|} = \sum_{\alpha \in \mathcal{A}} \sum_{\beta \in \mathcal{B}} z^{|\alpha|+|\beta|} = \sum_{\alpha \in \mathcal{A}} z^{|\alpha|} \sum_{\beta \in \mathcal{B}} z^{|\beta|} = A(z)B(z).$$

This equation is deceptively simple: be sure that you understand it before reading further.

The result for sequences follows from the definition

$$SEQ(\mathcal{A}) = \epsilon + \mathcal{A} + \mathcal{A} \times \mathcal{A} + \mathcal{A} \times \mathcal{A} \times \mathcal{A} + \dots$$

From the first two parts of the theorem, the generating function that enumerates this class is

$$1 + A(z) + A(z)^2 + A(z)^3 + A(z)^4 + \dots = \frac{1}{1 - A(z)}. \quad \blacksquare$$

Applying Theorem 5.1 to find OGFs associated with simple classes is a good way to clarify the definitions of disjoint union, Cartesian product, and sequence. For example, $\mathcal{Z} + \mathcal{Z} + \mathcal{Z}$ is a multiset containing three objects of size 1, with OGF $3z$, while $\mathcal{Z} \times \mathcal{Z} \times \mathcal{Z}$ contains one object of size 3 (a sequence of length 3), with OGF z^3 . The class $SEQ(\mathcal{Z})$ represents the natural numbers (one object corresponding to each number, by size) and has OGF $1/(1 - z)$, as you might expect.

Bitstrings. Let \mathcal{B} be the set of all binary strings (bitstrings), where the size of a bitstring is its length. Enumeration is elementary: the number of bitstrings of length N is 2^N . One way to define \mathcal{B} is to use recursion, as follows: A bitstring is either empty or corresponds precisely to an ordered pair consisting of a 0 or a 1 followed by a bitstring. Symbolically, this argument gives the combinatorial construction

$$\mathcal{B} = \epsilon + (\mathcal{Z}_0 + \mathcal{Z}_1) \times \mathcal{B}.$$

Theorem 5.1 allows us to translate directly from this symbolic form to a functional equation satisfied by the generating function. We have

$$B(z) = 1 + 2zB(z),$$

so $B(z) = 1/(1 - 2z)$ and $B_N = 2^N$, as expected. Alternatively, we may view \mathcal{B} as being formed of sequences of bits, so that we can use the combinatorial construction

$$\mathcal{B} = SEQ(\mathcal{Z}_0 + \mathcal{Z}_1)$$

and then $B(z) = 1/(1 - 2z)$ by the sequence rule of Theorem 5.1, since the OGF of $\mathcal{Z}_0 + \mathcal{Z}_1$ is just $2z$. This example is fundamental, and just a starting point.

A variation. The importance of the symbolic method is that it vastly simplifies the analysis of *variations* of such fundamental constructs. As an example, consider the class \mathcal{G} of binary strings having no two consecutive 0 bits. Such strings are either ϵ , a single 0, or 1 or 01 followed by a string with no two consecutive 0 bits. Symbolically,

$$\mathcal{G} = \epsilon + \mathcal{Z}_0 + (\mathcal{Z}_1 + \mathcal{Z}_0 \times \mathcal{Z}_1) \times \mathcal{G}.$$

$G_1 = 2$	$G_2 = 3$	$G_3 = 5$	$G_4 = 8$	$G_5 = 13$
0	1 0 1 1 0 1	1 1 0 1 1 1 1 0 1 0 1 0 0 1 1	1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 0 1 0 1 1 0 1 1 1 0 1 1 0 1 0 1 1 1 1 0 1 0 1 0 1 1 1 0 0 1 1 1 1 0 1 1 0 1 0 1 0 1 0 0 1 0 1 1	1 1 1 1 0 1 1 1 1 1 1 1 1 0 1 1 1 0 1 0 1 1 0 1 1 1 0 1 1 0 1 0 1 1 1 1 0 1 0 1 0 1 1 1 0 0 1 1 1 1 0 1 1 0 1 0 1 0 1 0 0 1 0 1 1

Figure 5.4 Bitstrings with no 00, $1 \leq N \leq 4$

Again, Theorem 5.1 allows us to translate this immediately into a formula for the generating function $G(z)$ that enumerates such strings:

$$G(z) = 1 + z + (z + z^2)G(z).$$

Thus we have $G(z) = (1 + z)/(1 - z - z^2)$, which leads directly to the result that the number of strings of length N with no two consecutive 0 bits is $F_N + F_{N+1} = F_{N+2}$, a Fibonacci number. Chapter 8 covers many variations of this kind.

Exercise 5.1 How many bitstrings of length N have no 000?

Exercise 5.2 How many bitstrings of length N have no 01?

Binary trees. For the class of binary trees, we have the construction

$$\mathcal{T} = \mathcal{Z}_{\square} + \mathcal{Z}_{\bullet} \times \mathcal{T} \times \mathcal{T}.$$

If the size of a tree is its number of internal nodes, we translate \mathcal{Z}_{\square} to 1 and \mathcal{Z}_{\bullet} to z to get

$$T^{\bullet}(z) = 1 + zT^{\bullet}(z)^2,$$

the functional equation defining the Catalan numbers that we have derived. If the size of a tree is its number of external nodes, we translate \mathcal{Z}_{\bullet} to 1 and \mathcal{Z}_{\square} to z to get

$$T^{\square}(z) = z + T^{\square}(z)^2,$$

which we can solve in a similar manner. These equations also imply, for example, that $T^{\square}(z) = zT^{\bullet}(z)$, a consequence of the fact that the number of external nodes in a binary tree is one greater than the number of internal nodes (see Chapter 6). The simplicity of the symbolic approach in studying properties of various types of trees is compelling, especially by comparison with analyses using recurrences. We shall cover this topic in detail in Chapter 6.

Exercise 5.3 Let \mathcal{U} be the set of binary trees with the size of a tree defined to be the total number of nodes (internal plus external), so that the generating function for its counting sequence is $U(z) = z + z^3 + 2z^5 + 5z^7 + 14z^9 + \dots$. Derive an explicit expression for $U(z)$.

Exercise 5.4 Define a "superleaf" in a binary tree to be an internal node whose four grandchildren are all external nodes. What fraction of binary trees with N nodes have no superleaves? From Figure 5.2, the answer for $N = 1, 2, 3$, and 4 is 0, 0, 4/5, and 6/7, respectively.

THEOREM 5.1 IS CALLED a "transfer theorem" because it directly transfers one mathematical formulation to another—it transfers a symbolic formula that defines structures to an equation involving a generating function that enumerates them. The ability to do so is quite powerful. As long as we use operations to define a structure for which we have such a theorem, we know that we will be able to learn something about the corresponding generating function.

There are a number of other operations that we use to build combinatorial structures besides the *union*, *Cartesian product*, and *sequence* operations corresponding to Theorem 5.1. Examples include *set of* or *multiset of*, as explored in the following two exercises. These and other operations are covered

thoroughly in [8]. With these sorts of operations, we can define and study an unlimited range of combinatorial structures, including a great many of those that have been studied in classical combinatorics and a great many of those that are important in the analysis of algorithms, as we will see in Chapters 6 through 9.

Exercise 5.5 Let \mathcal{B} be defined as the collection of all finite subsets of \mathcal{A} . If $A(z)$ and $B(z)$ are the OGFs of \mathcal{A} and \mathcal{B} , show that

$$B(z) = \prod_{n \geq 1} (1 + z^n)^{A_n} = \exp\left(A(z) - \frac{1}{2}A(z^2) + \frac{1}{3}A(z^3) - \dots\right).$$

Exercise 5.6 Let \mathcal{B} be defined as the collection of all finite multisets of \mathcal{A} (subsets with repetitions allowed). If $A(z)$ and $B(z)$ are the OGFs of \mathcal{A} and \mathcal{B} , show that

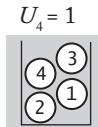
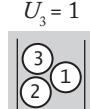
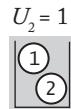
$$B(z) = \prod_{n \geq 1} \frac{1}{(1 - z^n)^{A_n}} = \exp\left(A(z) + \frac{1}{2}A(z^2) + \frac{1}{3}A(z^3) + \dots\right).$$

5.3 Symbolic Method for Labelled Classes. A primary feature of the preceding section is that the individual items from which the combinatorial objects are assembled are indistinguishable. An alternative paradigm is to assume that the individual items are *labelled* and that, therefore, the order in which items appear when assembled to make combinatorial objects is significant. Labelled objects are normally enumerated by exponential generating functions. We will illustrate fundamental principles with basic examples here, then we will study various families of labelled objects in detail in later chapters.

To be specific, if a combinatorial structure consists of N atoms, we consider their labels to be the integers 1 to N , and we consider objects to be different when the labels are in different order in the structure.

Figure 5.5 gives examples of three basic labelled combinatorial classes. The first is just a set of atoms—again, there is only one object of each size and this is another way to encode natural numbers in unary (for reasons that will become clear in Chapter 9, we refer to these as *urns*). The second consists of sequences of labelled atoms—the objects of size N are the *permutations* of length N , all the possible ways to order the integers 1 through N , so there are $N!$ permutations of size N . The third consists of *cyclic* sequences of labelled

Urns (sets of labelled atoms)



Permutations (sequences of labelled atoms)

$$P_1 = 1$$

$$P_2 = 2$$

(1)(2)	(1)(3)
(2)(1)	(2)(3)

$$P_3 = 6$$

(1)(2)(3)	(1)(3)(2)
(2)(1)(3)	(2)(3)(1)
(3)(1)(2)	(3)(1)(2)
(3)(2)(1)	

$$P_4 = 24$$

(1)(2)(3)(4)	(2)(1)(3)(4)
(1)(2)(4)(3)	(2)(1)(4)(3)
(1)(3)(2)(4)	(2)(3)(1)(4)
(1)(3)(4)(2)	(2)(3)(4)(1)
(1)(4)(2)(3)	(2)(4)(1)(3)
(1)(4)(3)(2)	(2)(4)(3)(1)
(3)(1)(2)(4)	(4)(1)(2)(3)
(3)(1)(4)(2)	(4)(1)(3)(2)
(3)(2)(1)(4)	(4)(2)(1)(3)
(3)(2)(4)(1)	(4)(2)(3)(1)
(3)(4)(1)(2)	(4)(3)(1)(2)
(3)(4)(2)(1)	(4)(3)(2)(1)

Cycles (cyclic sequences of labelled atoms)

$$C_1 = 1$$

$$C_2 = 1$$

$$C_3 = 2$$

$$C_4 = 6$$

Figure 5.5 Basic labelled classes, $1 \leq N \leq 4$

atoms—the objects of size N are known as the *cycles* of length N . There are $(N-1)!$ cycles of size N since any of the $(N-1)!$ permutations of size $N-1$ can appear after the 1 in order on the cycle.

Given a labelled class \mathcal{A} with counting sequence $\{a_n\}$, we are interested in the EGF

$$A(z) = \sum_{n \geq 0} a_n \frac{z^n}{n!} = \sum_{a \in \mathcal{A}} \frac{z^{|a|}}{|a|!}.$$

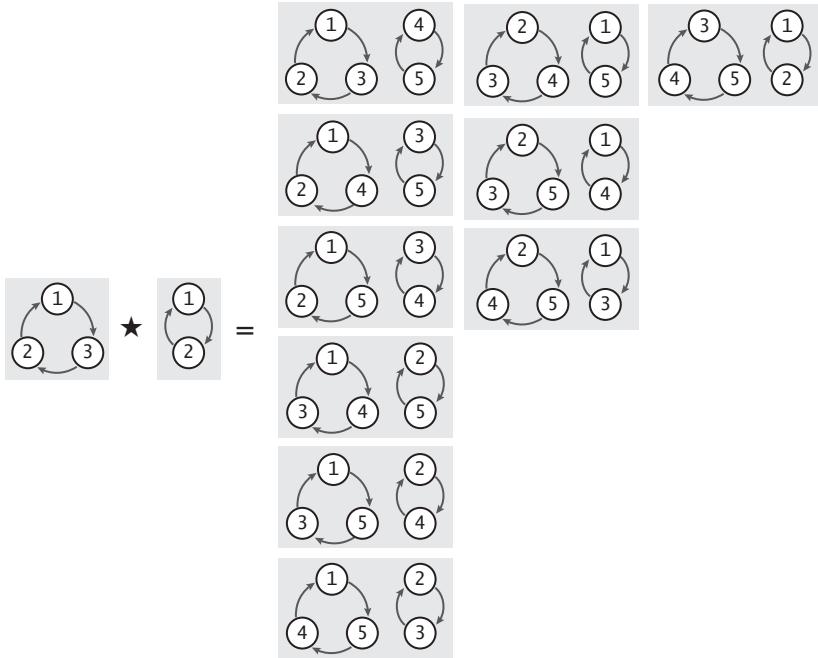
Again, this fundamental identity allows us to view the EGF as an analytic form representing the counting sequence (the left sum) or as a combinatorial form representing all the individual objects (the right sum). Table 5.2 gives the counting sequences and EGFs for the classes in Figure 5.5.

For unlabelled objects, we described ways to assemble combinatorial structures by means of the disjoint union, Cartesian product, and sequence operations. Here, we are going to define analogous constructions for labelled objects. The primary difference is that, in the analog to the Cartesian product operation, it is necessary to relabel in a consistent manner so that only the labels 1 to N appear if the resulting object has size N .

Specifically, we define the *star product* operation for two classes \mathcal{A} and \mathcal{B} of labelled objects $\mathcal{A} \star \mathcal{B}$ that gives the class of ordered pairs of objects, one from \mathcal{A} and one from \mathcal{B} *relabelled in all consistent ways*. Figure 5.5 depicts an example, the star product of a 3-cycle and a 2-cycle. The result is a pair of cycles that must be labelled with the labels 1 to 5. To do so, pick the $\binom{5}{3} = 10$ different possibilities for the 3-cycle labels, and assign them to the atoms of the 3-cycle, replacing the 1 with the smallest of the three, the 2 with

	<i>atoms</i>	<i>size function</i>	<i>counting sequence</i>	<i>EGF</i>
<i>urns</i>	(i)	# of (i)s	1	e^z
<i>permutations</i>	(i)	# of (i)s	$N!$	$\frac{1}{1-z}$
<i>cycles</i>	(i)	# of (i)s	$(N-1)!$	$\ln \frac{1}{1-z}$

Table 5.2 Familiar labelled combinatorial classes

**Figure 5.6** Star product example

the middle value, and the 3 with the largest of the three. Then assign the remaining 2 labels to the 2-cycle, in the same way. This relabelling algorithm is effective for any pair of labelled objects.

Our basic toolbox for assembling new classes of structured labelled objects consists of the sum, star product, and these three additional operations:

$SEQ(\mathcal{A})$ is the class of sequences of elements of \mathcal{A} ,

$SET(\mathcal{A})$ is the class of sets of elements of \mathcal{A} , and

$CYC(\mathcal{A})$ is the class of cyclic sequences of elements of \mathcal{A} .

As for unlabelled objects, $SEQ(\mathcal{A})$ is the class $\epsilon + \mathcal{A} + \mathcal{A} * \mathcal{A} + \dots$. With these definitions, we have the following transfer theorem for labelled classes.

Theorem 5.2 (Symbolic method for labelled class EGFs). Let \mathcal{A} and \mathcal{B} be classes of labelled combinatorial objects. If $A(z)$ is the EGF that enumerates \mathcal{A} and $B(z)$ is the EGF that enumerates \mathcal{B} , then

$A(z) + B(z)$ is the EGF that enumerates $\mathcal{A} + \mathcal{B}$,

$A(z)B(z)$ is the EGF that enumerates $\mathcal{A} \star \mathcal{B}$,

$\frac{1}{1 - A(z)}$ is the EGF that enumerates $SEQ(\mathcal{A})$,

$e^{A(z)}$ is the EGF that enumerates $SET(\mathcal{A})$, and

$\ln \frac{1}{1 - A(z)}$ is the EGF that enumerates $CYC(\mathcal{A})$.

Proof. The first part of the proof is the same as for Theorem 5.1. To prove the second part, note that, for every k from 0 to n , we can pair any of the a_k objects of size k from \mathcal{A} with any of the b_{n-k} objects of size $n - k$ from \mathcal{B} to get an object of size n in $\mathcal{A} \star \mathcal{B}$. The relabellings can be done in $\binom{n}{k}$ ways (simply choose the labels; their assignment is determined). Thus, the number of objects of size n in $\mathcal{A} \star \mathcal{B}$ is

$$\sum_{0 \leq k \leq n} \binom{n}{k} a_k b_{n-k}.$$

Again, a simple convolution leads to the desired result.

The result for sequences is the same as for Theorem 5.1. Now, if there are k identical components in a sequential arrangement and we relabel them in all possible ways, each set of components appears $k!$ times, because we “forget” about the order between components. Thus, we can compute the number of sets by dividing the number of sequential arrangements by $k!$. Similarly, each cyclic sequence appears k times, so the number of sequences is the number of sequential arrangements divided by k . From these observations, if we define the more specific operations

$SEQ_k(\mathcal{A})$, the class of k -sequences of elements of \mathcal{A} ;

$SET_k(\mathcal{A})$, the class of k -sets of elements of \mathcal{A} ;

$CYC_k(\mathcal{A})$, the class of k -cycles of elements of \mathcal{A} ;

then it is immediate that

$A(z)^k$ is the EGF that enumerates $SEQ_k(\mathcal{A})$,

$A(z)^k/k!$ is the EGF that enumerates $SET_k(\mathcal{A})$,

$A(z)^k/k$ is the EGF that enumerates $CYC_k(\mathcal{A})$,

and the results stated in the theorem follow by summing on k . ■

Again, thinking about tiny examples is a good way to clarify these definitions and associated EGF operations. For example, $\mathcal{Z} + \mathcal{Z} + \mathcal{Z}$ is a class containing three objects of size 1 (a multiset), with EGF $3z$; $\mathcal{Z} \star \mathcal{Z} \star \mathcal{Z}$ is an object of size 3 (a sequence), with EGF $z^3/6$; $SEQ_3(\mathcal{Z}) = \mathcal{Z} \star \mathcal{Z} \star \mathcal{Z}$ is a class that contains six objects of size 3, with EGF z^3 ; $SET_3(\mathcal{Z})$ is a class that contains one object of size 3, with EGF $z^3/6$; and $CYC_3(\mathcal{Z})$ is a class that contains two objects of size 3, with EGF $z^3/3$.

Permutations. Let \mathcal{P} be the set of all permutations, where the size of a permutation is its length. Enumeration is elementary: the number of permutations of length N is $N!$. One way to define \mathcal{P} is to use recursion, as follows:

$$\mathcal{P} = \epsilon + \mathcal{Z} \star \mathcal{P}.$$

A permutation is either empty or corresponds precisely to the star product of an atom and a permutation. Or, more specifically, if \mathcal{P}_N is the class of permutations of size N , then $\mathcal{P}_0 = \phi$ and, for $N > 0$,

$$\mathcal{P}_N = \mathcal{Z} \star \mathcal{P}_{N-1},$$

as illustrated in Figure 5.6. Theorem 5.2 allows us to translate directly from these symbolic forms to functional equation satisfied by the corresponding generating functions. We have

$$P(z) = 1 + zP(z),$$

so $P(z) = 1/(1 - z)$ and $P_N = N![z^N]P(z) = N!$, as expected. Or, we can translate the construction for \mathcal{P}_N to get

$$P_N(z) = zP_{N-1}(z),$$

which telescopes to give $P_N(z) = z^N$ and $P_N = N![z^N]P_N(z) = N!$ or $P(z) = 1/(1 - z)$ by the sum rule of Theorem 5.2, since $\mathcal{P} = \sum_{N \geq 0} \mathcal{P}_N$. A

$$\begin{array}{l}
 \text{①} \star \text{①②③} = \begin{array}{c} (1\ 2\ 3\ 4) \\ (2\ 1\ 3\ 4) \\ (3\ 1\ 2\ 4) \\ (4\ 1\ 2\ 3) \end{array} \\
 \text{①} \star \text{①③②} = \begin{array}{c} (1\ 2\ 4\ 3) \\ (2\ 1\ 4\ 3) \\ (3\ 1\ 4\ 2) \\ (4\ 1\ 3\ 2) \end{array} \\
 \text{①} \star \text{②①③} = \begin{array}{c} (1\ 3\ 2\ 4) \\ (2\ 3\ 1\ 4) \\ (3\ 2\ 1\ 4) \\ (4\ 2\ 1\ 3) \end{array} \\
 \text{①} \star \text{②③①} = \begin{array}{c} (1\ 3\ 4\ 2) \\ (2\ 3\ 4\ 1) \\ (3\ 2\ 4\ 1) \\ (4\ 2\ 3\ 1) \end{array} \\
 \text{①} \star \text{③①②} = \begin{array}{c} (1\ 4\ 2\ 3) \\ (2\ 4\ 1\ 3) \\ (3\ 4\ 1\ 2) \\ (4\ 3\ 1\ 2) \end{array} \\
 \text{①} \star \text{③②①} = \begin{array}{c} (1\ 4\ 3\ 2) \\ (2\ 4\ 3\ 1) \\ (3\ 4\ 2\ 1) \\ (4\ 3\ 2\ 1) \end{array}
 \end{array}$$

Figure 5.7 $\mathcal{Z} \star \mathcal{P}_3 = \mathcal{P}_4$

third alternative is to view \mathcal{P} as being formed of sequences of labelled atoms, defined by the combinatorial construction $\mathcal{P} = SEQ(\mathcal{Z})$, which again gives $P(z) = 1/(1 - z)$ by the sequence rule of Theorem 5.2. Again, this example is fundamental, and just a starting point.

Sets and cycles. Similarly, we may view the class of urns \mathcal{U} as being formed of sets of labelled atoms and the class of cycles \mathcal{C} as being formed of cycles of labelled atoms, so that $U(z) = e^z$ by the set rule of Theorem 5.2 and $C(z) = \ln(1/(1 - z))$ by the cycle rule of Theorem 5.2, which gives $U_N = 1$ and $C_{N-1} = (N - 1)!$ as expected.

Sets of cycles. Next, we consider a fundamental example where we compound two combinatorial constructions. Specifically, consider the class \mathcal{P}^* formed by taking sets of cycles, as illustrated for small sizes in Figure 5.7.

Symbolically, we have the construction

$$\mathcal{P}^* = \text{SET}(\text{CYC}(\mathcal{Z})),$$

which, by the cycle and sequence rules in Theorem 5.2, gives the EGF

$$P^*(z) = \exp\left(\ln\frac{1}{1-z}\right) = \frac{1}{1-z},$$

so that $P_N^* = N![z^N]P^*(z) = N!$. That is, the number of labelled sets of cycles of N items is precisely $N!$, the number of permutations. This is a fundamental result, which we now briefly examine in more detail (foreshadowing Chapter 7).

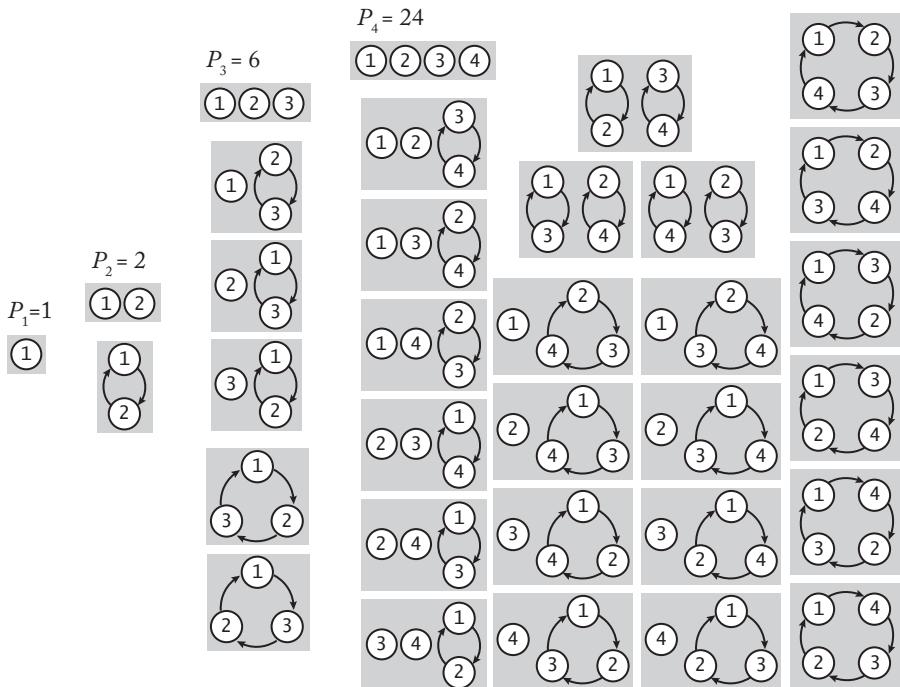


Figure 5.8 Sets of cycles, $1 \leq N \leq 4$

A bijection. Figure 5.9 illustrates a 1:1 correspondence between permutations and sets of cycles. To more easily visualize the problem, suppose that N students have the names 1 through N and each owns a hat labelled with their name. On graduation each throws their hat in the air and then grabs a random hat. This process defines a random permutation, drawn from the labelled class that we have just considered. Permutations are so well studied in combinatorics that many other colorful images have been proposed, ranging from hats at the opera to beds occupied by drunken sailors—use your imagination!

One way to write down a permutation, called the *two-line representation*, is to write the student names 1 through N in order on the first line and the number of the hat that each student grabbed below her or his name on the second line. In the permutation depicted in Figure 5.9, 1 grabbed 9's hat, 2 grabbed 3's hat, and so forth. Now, imagine that student 1, having hat 9, asks student 9 whether he or she has hat 1, then since student 9 has hat 4, goes on to ask student 4, and so forth, continuing until finding hat 1. (Take a moment to convince yourself that you could always find your hat in this way.) This leads to another way to write down a permutation, called the *set-of-cycles* representation. In the permutation depicted in Figure 5.9, 1 has 9's hat, 9 has 4's hat, and 4 has 1's hat, and so forth. Everyone is on a cycle and the cycles are disjoint since everyone has one hat, and their order is irrelevant, so every permutation corresponds to a set of cycles. Conversely, given a set of cycles, one can easily write down the two-line representation, so we have defined a bijection. This bijection is an alternate proof that the number of labelled sets of cycles of N items is precisely $N!$.

In the present context, our interest is in the fact that the simple structure of the set-of-cycles combinatorial construction leads to easy solutions to problems that are considerably more difficult to solve otherwise. We consider one example next and several similar examples in Chapters 7 and 9.

two-line representation	1	2	3	4	5	6	7	8	9
	9	3	2	1	8	5	7	6	4

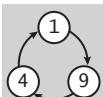
set-of-cycles representation				
------------------------------	---	---	---	---

Figure 5.9 A permutation is a set of cycles.

Derangements. Again, the true power of analytic combinatorics is apparent when we consider variations of fundamental constructions. As an example, we consider the famous *derangements* problem: When our N graduating students throw their hats in the air as described previously, what is the probability that everyone gets someone else's hat? As you can see from Figure 5.10, the answer for $N = 2, 3$, and 4 is $1/2, 1/3$, and $3/8$, respectively. This is precisely the number of sets of cycles of N items with no singleton cycles, divided by $N!$. For the present our interest is to note that the desired quantity is $[z^N]D(z)$, where $D(z)$ is the EGF associated with the combinatorial class \mathcal{D} , the sets of cycles of N items where all cycles are of size greater than 1. (In this case, the EGF is a PGF.) Then the simple combinatorial construction

$$\mathcal{D} = \text{SET}(\text{CYC}_2(\mathcal{Z}) + \text{CYC}_3(\mathcal{Z}) + \text{CYC}_4(\mathcal{Z}) + \dots)$$

immediately translates to the EGF

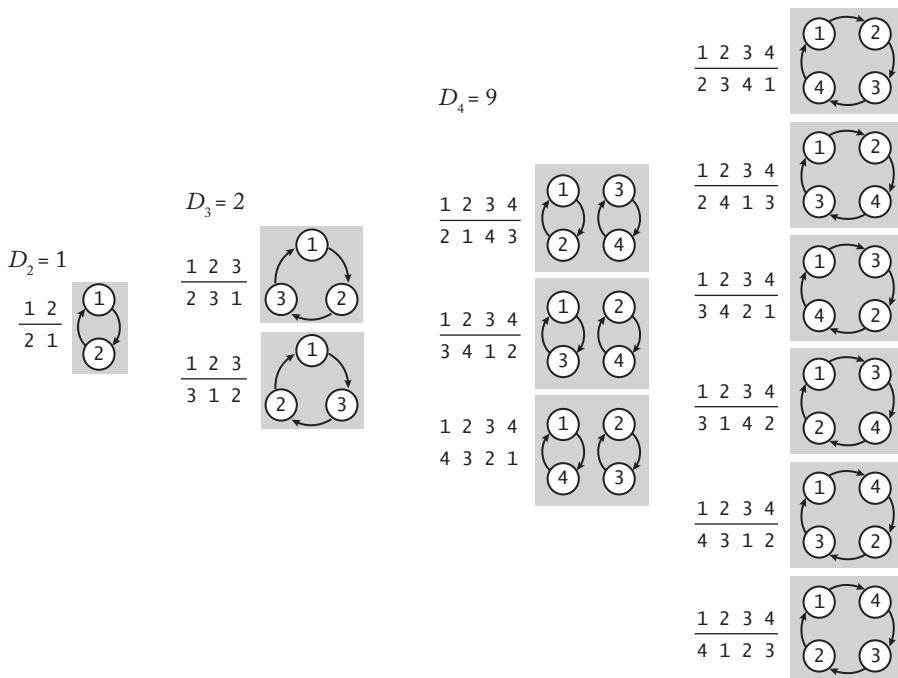


Figure 5.10 Derangements, $2 \leq N \leq 4$

$$\begin{aligned} D(z) &= \exp\left(\frac{z^2}{2} + \frac{z^3}{3} + \frac{z^4}{4} + \dots\right) \\ &= \exp\left(\ln\frac{1}{1-z} - z\right) \\ &= \frac{e^{-z}}{1-z}. \end{aligned}$$

An even simpler derivation is to note that a permutation is the star product of a set of singleton cycles (an urn) and a derangement, so we have the construction

$$SET(\mathcal{Z}) \star \mathcal{D} = \mathcal{P},$$

which translates to the EGF equation

$$e^z D(z) = \frac{1}{1-z}$$

and leads to the same result. Thus $D(z)$ is a simple convolution, and

$$D_N = \sum_{0 \leq k \leq N} \frac{(-1)^k}{k!} \sim 1/e.$$

Note that the exact formula agrees with the observed values for $N = 2, 3$, and 4 ; the asymptotic estimate is one that we discussed in §4.4.

Generalized derangements. Again, the true power of analytic combinatorics is apparent when we consider variations of fundamental constructions. For example, when our students throw their hats in the air, what is the probability that every group of M students gets a hat from someone outside the group? This is precisely the number of sets of cycles of N items where all cycles are of size greater than M , divided by $N!$. As earlier, the desired quantity is $[z^N]P_{>M}^*(z)$, where $P_{>M}^*(z)$ is the EGF associated with the combinatorial class $\mathcal{P}_{>M}$, the combinatorial class of sets of cycles of N items where all cycles are of size greater than M . Then we have the construction

$$\mathcal{P}_{>M}^* = SET(CYC_{>M}(\mathcal{Z}))$$

(where, for brevity, we use the abbreviation $CYC_{>M}(\mathcal{Z})$ for $CYC_{M+1}(\mathcal{Z}) + CYC_{M+2}(\mathcal{Z}) + CYC_{M+3}(\mathcal{Z}) + \dots$), which immediately translates to the EGF

$$\begin{aligned} P_{>M}^*(z) &= \exp\left(\frac{z^{M+1}}{M+1} + \frac{z^{M+2}}{M+2} + \frac{z^{M+3}}{M+3} + \dots\right) \\ &= \exp\left(\ln\frac{1}{1-z} - z - \frac{z^2}{2} - \dots - \frac{z^M}{M}\right) \\ &= \frac{e^{-z-z^2/2-\dots-z^M/M}}{1-z}. \end{aligned}$$

In summary, the symbolic method immediately leads to a simple expression for the EGF that might otherwise be complicated to derive. Extracting the coefficients from this generating function appears to be not so easy, but we shall soon consider a transfer theorem that gives an asymptotic estimate directly.

Exercise 5.7 Derive an EGF for the number of permutations whose cycles are all of odd length.

Exercise 5.8 Derive an EGF for *sequences* of cycles.

Exercise 5.9 Derive an EGF for cycles of sequences.

THEOREM 5.2 IS A transfer theorem for labelled objects that immediately gives EGF equations for the broad variety of combinatorial classes that can be described by the operations we have considered. We will consider a number of examples in Chapters 6 through 9. As for unlabelled classes, several additional operations for labelled classes have been invented (and are covered thoroughly in [8]), but these basic operations serve us well for studying a broad variety of classes in Chapters 6 through 9.

5.4 Symbolic Method for Parameters. The symbolic method is also effective for developing equations satisfied by BGFs associated with combinatorial parameters, as introduced in Chapter 3. Indeed, transfer theorems to BGF equations associated with natural parameters from the *very same* combinatorial constructions that we have already considered are readily available.

In this section, we state the theorems for unlabelled and labelled classes and then give a basic example of the application of each, reserving more applications for our studies of various kinds of combinatorial classes in Chapters 6 through 9. Once one has understood the basic theorems for enumeration, the corresponding theorems for analysis of parameters are straightforward. For brevity, we state the theorems only for the basic constructions and leave the proofs for exercises.

Given an unlabelled class \mathcal{A} with a parameter defined as a cost function that is defined for every object in the class, we are interested in the ordinary *bivariate* generating function (OBGF)

$$A(z, u) = \sum_{n \geq 0} \sum_{k \geq 0} a_{nk} z^n u^k,$$

where a_{nk} is the number of objects of size n and cost k . As we have seen, the fundamental identity

$$A(z, u) = \sum_{n \geq 0} \sum_{k \geq 0} a_{nk} z^n u^k = \sum_{a \in \mathcal{A}} z^{|a|} u^{\text{cost}(a)}$$

allows us to view the OGF as an analytic form representing the double counting sequence for size and cost (the left sum) or as a combinatorial form representing all the individual objects (the right sum).

Theorem 5.3 (Symbolic method for unlabelled class OBGFs). Let \mathcal{A} and \mathcal{B} be unlabelled classes of combinatorial objects. If $A(z, u)$ and $B(z, u)$ are the OBGFs associated with \mathcal{A} and \mathcal{B} , respectively, where z marks size and u marks a parameter, then

$A(z, u) + B(z, u)$ is the OBGF associated with $\mathcal{A} + \mathcal{B}$,

$A(z, u)B(z, u)$ is the OBGF associated with $\mathcal{A} \times \mathcal{B}$, and

$\frac{1}{1 - A(z, u)}$ is the OBGF associated with $SEQ(\mathcal{A})$.

Proof. Omitted. ■

Similarly, given a labelled class \mathcal{A} with a parameter defined as a cost function that is defined for every object in the class, we are interested in the exponential *bivariate* generating function (EBGF)

$$A(z, u) = \sum_{n \geq 0} \sum_{k \geq 0} a_{nk} \frac{z^n}{n!} u^k,$$

where a_{nk} is the number of objects of size n and cost k . Again, the fundamental identity

$$A(z, u) = \sum_{n \geq 0} \sum_{k \geq 0} a_{nk} \frac{z^n}{n!} u^k = \sum_{a \in \mathcal{A}} \frac{z^{|a|}}{|a|!} u^{\text{cost}(a)}$$

allows us to view the OGF as an analytic form representing the double counting sequence for size and cost (the left sum) or as a combinatorial form representing all the individual objects (the right sum).

Theorem 5.4 (Symbolic method for labelled class EBGFs). Let \mathcal{A} and \mathcal{B} be classes of labelled combinatorial objects. If $A(z, u)$ and $B(z, u)$ are the EBGFs associated with \mathcal{A} and \mathcal{B} , respectively, where z marks size and u marks a parameter, then

$A(z, u) + B(z, u)$ is the EBGF associated with $\mathcal{A} + \mathcal{B}$,

$A(z, u)B(z, u)$ is the EBGF associated with $\mathcal{A} \star \mathcal{B}$,

$\frac{1}{1 - A(z, u)}$ is the EBGF associated with $SEQ(\mathcal{A})$,

$e^{A(z, u)}$ is the EBGF associated with $SET(\mathcal{A})$, and

$\ln \frac{1}{1 - A(z, u)}$ is the EBGF associated with $CYC(\mathcal{A})$.

Proof. Omitted. ■

Exercise 5.10 Extend the proof of Theorem 5.1 to give a proof of Theorem 5.3.

Exercise 5.11 Extend the proof of Theorem 5.2 to give a proof of Theorem 5.4.

Note that taking $u = 1$ in these theorems transforms them to Theorem 5.1 and Theorem 5.2. Developing equations satisfied by BGFs describing parameters of combinatorial classes is often immediate from the same constructions that we used to derive GFs that enumerate the classes, slightly augmented to mark parameter values as well as size. To illustrate the process, we consider three classic examples.

Bitstrings. How many bitstrings of length N have k 1-bits? This well-known quantity, which we already discussed in Chapter 3, is the binomial coefficient $\binom{N}{k}$. The derivation is immediate with the symbolic method. In the construction

$$\mathcal{B} = \epsilon + (\mathcal{Z}_0 + \mathcal{Z}_1) \times \mathcal{B},$$

we use the BGF z for \mathcal{Z}_0 and the BGF zu for \mathcal{Z}_1 and then use Theorem 5.4 to translate directly to the BGF equation

$$B(z, u) = 1 + z(1 + u)B(z, u),$$

so that

$$B(z, u) = \frac{1}{1 - (1 + u)z} = \sum_{N \geq 0} (1 + u)^N z^N = \sum_{N \geq 0} \sum_{k \geq 0} \binom{N}{k} z^N u^k,$$

as expected. Alternatively, we could use the construction

$$\mathcal{B} = SEQ(\mathcal{Z}_0 + \mathcal{Z}_1)$$

to get the same result by the sequence rule of Theorem 5.4. This example is fundamental, and just a starting point.

Cycles in permutations. What is the average number of cycles in a permutation of length N ? From inspection of Figure 5.8, you can check that the cumulative counts for $N = 1, 2, 3$, and 4 are $1, 3, 11$, and 50 , respectively. Symbolically, we have the construction

$$\mathcal{P}^* = SET(CYC(\mathcal{Z})),$$

which, by the cycle and sequence rules in Theorem 5.4 (marking each cycle with u), gives the EGF

$$P^*(z) = \exp\left(u \ln \frac{1}{1-z}\right) = \frac{1}{(1-z)^u}.$$

From this explicit representation of the BGF, we can use the techniques described at length in Chapter 3 to analyze parameters. In this case,

$$P(z, 1) = \frac{1}{1-z},$$

as expected, and

$$P_u(z, 1) = \frac{1}{1-z} \ln \frac{1}{1-z},$$

so the average number of cycles in a random permutation is

$$\frac{N![z^N]P_u(z, 1)}{N![z^N]P(z, 1)} = H_N.$$

Leaves in binary trees. What proportion of the internal nodes in a binary tree of size N have two external children? Such nodes are called *leaves*. From inspection of Figure 5.2, you can check that the total numbers of such nodes for $N = 0, 1, 2, 3$, and 4 are $0, 1, 2, 6$, and 20 , respectively. Dividing by the Catalan numbers, the associated proportions are $0, 1, 1, 6/5$ and $10/7$. In terms of the BGF

$$T(z, u) = \sum_{t \in \mathcal{T}} z^{|t|} u^{\text{leaves}(t)}$$

the following are the coefficients of z^0, z^1, z^2, z^3, z^4 , respectively, and are reflected directly in the trees in Figure 5.2:

$$u^0$$

$$u^1$$

$$u^1 + u^1$$

$$u^1 + u^1 + u^2 + u^1 + u^1 + u^1 + u^2 + u^1 + u^2 + u^1 + u^1 + u^2 + u^2.$$

Adding these terms, we know that

$$T(z, u) = 1 + z^1 u + 2z^2 u + z^3 (4u + u^2) + z^4 (8u + 6u^2) + \dots.$$

Checking small values, we find that

$$T(z, 1) = 1 + z^1 + 2z^2 + 5z^3 + 14z^4 + \dots$$

and

$$T_u(z, 1) = z^1 + 2z^2 + 6z^3 + 20z^4 + \dots$$

as expected. To derive a GF equation with the symbolic method, we add \mathcal{Z}_\bullet to both sides of the standard recursive construction to get

$$\mathcal{T} + \mathcal{Z}_\bullet = \mathcal{E} + \mathcal{Z}_\bullet + \mathcal{Z}_\bullet \times \mathcal{T} \times \mathcal{T}.$$

This gives us a way to mark leaves (by using the BGF zu for the \mathcal{Z}_\bullet term on the right) and to balance the equation for the tree of size 1. Applying Theorem 5.3 (using the BGF z for the \mathcal{Z}_\bullet term on the left and the \mathcal{Z}_\bullet factor on the rightmost term, since neither corresponds to a leaf) immediately gives the functional equation

$$T(z, u) + z = 1 + zu + zT(z, u)^2.$$

Setting $u = 1$ gives the OGF for the Catalan numbers as expected and differentiating with respect to u and evaluating at $u = 1$ gives

$$\begin{aligned} T_u(z, 1) &= z + 2zT(1, z)T_u(z, 1) \\ &= \frac{z}{1 - 2zT(z, 1)} \\ &= \frac{z}{\sqrt{1 - 4z}}. \end{aligned}$$

Thus, by the standard BGF calculation shown in Table 3.6, the average number of internal nodes with both nodes external in a binary tree of size n is

$$\frac{[z^n] \frac{z}{\sqrt{1 - 4z}}}{\frac{1}{n+1} \binom{2n}{n}} = \frac{\binom{2n-2}{n-1}}{\frac{1}{n+1} \binom{2n}{n}} = \frac{(n+1)n}{2(2n-1)}$$

(see §3.4 and §3.8), which tends to $n/4$ in the limit. About $1/4$ of the internal nodes in a binary tree are leaves.

Exercise 5.12 Confirm that the average number of 1 bits in a random bitstring is $N/2$ by computing $B_u(z, 1)$.

Exercise 5.13 What is the average number of 1 bits in a random bitstring of length N having no 00?

Exercise 5.14 What is the average number of cycles in a random derangement?

Exercise 5.15 Find the average number of internal nodes in a binary tree of size n with both children internal.

Exercise 5.16 Find the average number of internal nodes in a binary tree of size n with one child internal and one child external.

Exercise 5.17 Find an explicit formula for $T(z, u)$ and compute the variance of the number of leaves in binary trees.

THIS BRIEF INTRODUCTION ONLY scratches the surface of what is known about the symbolic method, which is one of the cornerstones of modern combinatorial analysis. The symbolic method summarized by Theorem 5.1, Theorem 5.2, Theorem 5.3, and Theorem 5.4 works for an ever-expanding set of structures, though it cannot naturally solve all problems: some combinatorial objects just have too much internal “cross-structure” to be amenable to this treatment. But it is a method of choice for combinatorial structures that have a nice decomposable form, such as trees (Chapter 6), the example *par excellence*; permutations (Chapter 7); strings (Chapter 8); and words or mappings (Chapter 9). When the method does apply, it can succeed spectacularly, especially in allowing quick analysis of variants of basic structures.

Much more information about the symbolic method may be found in Goulden and Jackson [9] or Stanley [13]. In [8], we give a thorough treatment of the method (see also [14] for more information in the context of the analysis of algorithms). The theory is sufficiently complete that it has been embodied in a computer program that can automatically determine generating functions for a structure from a simple recursive definition, as described in Flajolet, Salvy, and Zimmerman [7].

Next, we consider the second stage of analytic combinatorics, where we pivot from the symbolic to the analytic so that we may consider transfer theorems that take us from GF representations to coefficient approximations, with similar ease.

5.5 Generating Function Coefficient Asymptotics. The constructions associated with the symbolic method yield an extensive variety of generating function equations. The next challenge in analytic combinatorics is to transfer those GF equations to useful approximations of counting sequences for those classes. In this section, we briefly review examples of theorems that we have seen that are effective for such transfers and develop another theorem. While indicative of the power of analytic combinatorics and useful for many of the classes that we consider in this book, these theorems are only a starting point. In [8], we use complex-analytic techniques to develop remarkably general transfer theorems that, paired with the symbolic method, provide a basis for the assertion “if you can specify it, you can analyze it.”

Taylor's theorem. A first example of a transfer theorem is the first method that we considered in Chapter 3 for extracting GF coefficients. Simply put, Taylor's theorem says that

$$[z^n]f(z) = \frac{f^{(n)}(0)}{n!},$$

provided that the derivatives exist. As we have seen, it is an effective method for extracting coefficients for $1/(1 - 2z)$ (the OGF for bitstrings), e^z (the EGF for permutations), and many other elementary GF equations that derive from the symbolic method. While Taylor's theorem is effective in principle for a broad class of GF expansions, it specifies exact values that can involve detailed calculations, so we generally prefer transfer theorems that can directly give the asymptotic estimates that we ultimately seek.

Exercise 5.18 Use Taylor's theorem to find $[z^N] \frac{e^{-z}}{1-z}$.

Rational functions. A second example of a transfer theorem is Theorem 4.1, which gives asymptotics for coefficients of rational functions (of the form $f(z)/g(z)$, where $f(z)$ and $g(z)$ are polynomials). To review from §4.1, the growth of the coefficients depends on the root $1/\beta$ of $g(z)$ of largest modulus. If the multiplicity of $1/\beta$ is 1, then

$$[z^n] \frac{f(z)}{g(z)} = -\frac{\beta f(1/\beta)}{g'(\beta)} \beta^n.$$

For example, this is an effective method for extracting coefficients for $(1+z)/(1-z-z^2)$ (the OGF for bitstrings with no 00) and similar GFs. We study such applications in detail in Chapter 8.

Exercise 5.19 Use Theorem 4.1 to show that the number of bitstrings having no occurrence of 00 is $\sim \phi^N/\sqrt{5}$.

Exercise 5.20 Find an approximation for the number of bitstrings having no occurrence of 01.

Radius-of-convergence bounds. It has been known since Euler and Cauchy that knowledge of the *radius of convergence* of a generating function provides information on the rate of growth of its coefficients. Specifically, if $f(z)$ is a power series that has radius of convergence $R > 0$, then $[z^n]f(z) = O(r^{-n})$ for any positive $r < R$. This fact is easy to prove: Take any r such that $0 < r < R$ and let $f_n = [z^n]f(z)$. The series $\sum_n f_n r^n$ converges; hence its general term $f_n r^n$ tends to zero and in particular is bounded from above by a constant.

For example, the Catalan generating function converges for $|z| < 1/4$ since it involves $(1 - 4z)^{1/2}$ and the binomial series $(1 + u)^{1/2}$ converges for $|u| < 1$. This gives us the bound

$$[z^n] \frac{1 - \sqrt{1 - 4z}}{2} = O((4 + \epsilon)^n)$$

for any ϵ . This is weaker form of what we derived from Stirling's formula in §4.4.

In the case of combinatorial generating functions, we can considerably strengthen these bounds. More generally, let $f(z)$ have positive coefficients. Then

$$[z^n]f(z) \leq \min_{x \in (0, R)} \frac{f(x)}{x^n}.$$

This follows simply from $f_n x^n \leq f(x)$, as $f_n x^n$ is just one term in a convergent sum of positive quantities. In particular, we will again make use of this very general bounding technique in Chapter 8 when we discuss permutations with restricted cycle lengths.

Exercise 5.21 Prove that there exists a constant C such that

$$[z^n] \exp(z/(1 - z)) = O(\exp(C\sqrt{n})).$$

Exercise 5.22 Establish similar bounds for the OGF of integer partitions

$$[z^n] \prod_{k \geq 1} (1 - z^k)^{-1}.$$

More specifically, we can use convolution and partial fraction decomposition to develop coefficient asymptotics for GFs that involve powers of $1/(1-z)$, which arise frequently in derivations from combinatorial constructions. For instance, if $f(z)$ is a polynomial and r is an integer, then partial fraction decomposition (Theorem 4.1) yields

$$[z^n] \frac{f(z)}{(1-z)^r} \sim f(1) \binom{n+r-1}{n} \sim f(1) \frac{n^{r-1}}{(r-1)!},$$

provided of course $f(1) \neq 0$. A much more general result actually holds.

Theorem 5.5 (Radius-of-convergence transfer theorem). Let $f(z)$ have radius of convergence strictly larger than 1 and assume that $f(1) \neq 0$. For any real $\alpha \notin \{0, -1, -2, \dots\}$, there holds

$$[z^n] \frac{f(z)}{(1-z)^\alpha} \sim f(1) \binom{n+\alpha-1}{n} \sim \frac{f(1)}{\Gamma(\alpha)} n^{\alpha-1}.$$

Proof. Let $f(z)$ have radius of convergence $> r$, where $r > 1$. We know that $f_n \equiv [z^n]f(z) = O(r^{-n})$ from radius-of-convergence bounds, and in particular the sum $\sum_n f_n$ converges to $f(1)$ geometrically fast.

It is then a simple matter to analyze the convolution:

$$\begin{aligned} [z^n] \frac{f(z)}{(1-z)^\alpha} &= f_0 \binom{n+\alpha-1}{n} + f_1 \binom{n+\alpha-2}{n-1} + \cdots + f_n \binom{\alpha-1}{0} \\ &= \binom{n+\alpha-1}{n} \left(f_0 + f_1 \frac{n}{n+\alpha-1} \right. \\ &\quad \left. + f_2 \frac{n(n-1)}{(n+\alpha-1)(n+\alpha-2)} \right. \\ &\quad \left. + f_3 \frac{n(n-1)(n-2)}{(n+\alpha-1)(n+\alpha-2)(n+\alpha-3)} + \cdots \right). \end{aligned}$$

The term of index j in this sum is

$$f_j \frac{n(n-1)\cdots(n-j+1)}{(n+\alpha-1)(n+\alpha-2)\cdots(n+\alpha-j)},$$

which tends to f_j when $n \rightarrow +\infty$. From this, we deduce that

$$[z^n]f(z)(1-z)^{-\alpha} \sim \binom{n+\alpha-1}{n}(f_0 + f_1 + \cdots + f_n) \sim f(1)\binom{n+\alpha-1}{n},$$

since the partial sums $f_0 + \cdots + f_n$ converge to $f(1)$ geometrically fast. The approximation to the binomial coefficient follows from the Euler-Maclaurin formula (see Exercise 4.60). ■

In general, coefficient asymptotics are determined by the behavior of the GF near where it diverges. When the radius of convergence is not 1, we can rescale to a function for which the theorem applies. Such rescaling always introduces a multiplicative exponential factor.

Corollary Let $f(z)$ have radius of convergence strictly larger than ρ and assume that $f(\rho) \neq 0$. For any real $\alpha \notin \{0, -1, -2, \dots\}$, there holds

$$[z^n]\frac{f(z)}{(1-z/\rho)^\alpha} \sim \frac{f(\rho)}{\Gamma(\alpha)}\rho^n n^{\alpha-1}.$$

Proof. Let $g(z) = f(z/\rho)$. Then $[z^n]g(z) = \rho^n[z^n]g(\rho z) = \rho^n[z^n]f(z)$. ■

While it has limitations, Theorem 5.5 (and its corollary) is effective for extracting coefficients from many of the GFs that we encounter in this book. It is an outstanding example of the analytic transfer theorems that comprise the second phase of analytic combinatorics. We consider next three classic examples of applying the theorem.

Generalized derangements. In §5.3, we used the symbolic method to show that the probability that a given permutation has no cycles of length less than or equal to M is $[z^N]P_{>M}^*(z)$, where

$$P_{>M}^*(z) = \frac{e^{-z-z^2/2-\dots-z^M/M}}{1-z}$$

From this expression, Theorem 5.5 gives *immediately*

$$[z^N]P_{>M}^*(z) \sim \frac{1}{e^{H_M}}.$$

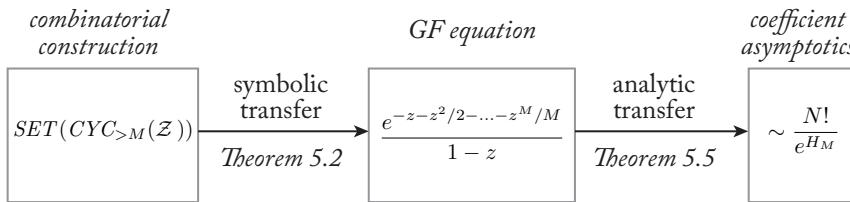


Figure 5.11 Analytic combinatorics to count generalized derangements

Figure 5.11 summarizes the ease with which analytic combinatorics gets to this result through two general transfer theorems. While it is possible to get to the same result with direct calculations along the lines of the proofs of the theorems, such calculations can be lengthy and complicated. One of the prime tenets of analytic combinatorics is that such detailed calculations are often not necessary, as general transfer theorems can provide accurate results for a great variety of combinatorial classes, as they did in this case.

Catalan numbers. Similarly, the corollary to Theorem 5.5 immediately provides a transfer from the Catalan GF

$$T(z) = \frac{1 - \sqrt{1 - 4z}}{2}$$

to the asymptotic form of its coefficients: Ignore the constant term, and take $\alpha = -1/2$ and $f(z) = -1/2$ to get the asymptotic result

$$T_N \sim \frac{4^N}{N\sqrt{\pi N}}.$$

While this simple derivation is very appealing, we note that is actually possible to derive the result *directly* from the form

$$T(z) = z + T(z)^2$$

using a general transfer theorem based on complex-analytic techniques that is beyond the scope of this book (see [8]).

Classic application. A famous example of the application of this technique (see Comtet [5]) is to the function

$$f(z) = \frac{e^{z/2+z^2/4}}{\sqrt{1-z}}$$

that is the EGF of the so-called 2-regular graphs. It involves a numerator $g(z) = \exp(z/2 + z^2/4)$ that has radius of convergence clearly equal to ∞ . Again Theorem 5.5 immediately gives the result

$$[z^n]f(z) \sim \frac{e^{3/4}}{\sqrt{\pi n}}.$$

The ease with the asymptotic form of the coefficients can be read from the GFs in these and many other applications is quite remarkable.

WITH RESULTS SUCH AS Theorem 5.5, the asymptotic form of coefficients is directly “transferred” from elements like $(1-z)^{-\alpha}$ (called “singular elements”) that play a role very similar to partial fraction elements in the analysis of rational functions. And deeper mathematical truths are at play. Theorem 5.5 is only the simplest of a whole set of similar results originating with Darboux in the last century (see [5] and [16]), and further developed by Pólya and Szegő, Bender, and others [1][6]. These methods are discussed in detail in [8]; unlike what we could do here, their full development requires the theory of functions of a complex variable. This approach to asymptotics is called *singularity analysis*.

The transfer theorems that we have considered in this section yield coefficient asymptotics from explicit GF formulae. It is also possible to work directly with *implicit* GF representations, like the $T(z) = z + T(z)^2$ equation that we get for binary trees. In §6.12, we will consider the Lagrange Inversion Theorem, a very powerful tool for extracting coefficients from such implicit representations. General transfer theorems based on inversion play a central role in analytic combinatorics (see [8]).

Exercise 5.23 Show that the probability that all the cycles are of odd length in a random permutation of length N is $1/\sqrt{\pi N/2}$ (see Exercise 5.7).

Exercise 5.24 Give a more precise version of Theorem 5.5—extend the asymptotic series to three terms.

Exercise 5.25 Show that

$$[z^n]f(z)\ln\frac{1}{1-z} \sim \frac{f(1)}{n}.$$

Exercise 5.26 Give a transfer theorem like Theorem 5.5 for

$$[z^n]f(z)\frac{1}{1-z}\ln\frac{1}{1-z}.$$

ANALYTIC combinatorics is a calculus for the quantitative study of large combinatorial structures. It has been remarkably successful as a general approach for analyzing classical combinatorial structures of all sorts, including those that arise in the analysis of algorithms. Table 5.3, which summarizes just the results derived in this chapter, indicates the breadth of applicability of analytic combinatorics, even though we have considered only a few basic transfer theorems. Interested readers may find extensive coverage of advanced transfer theorems and many more applications in our book [8].

More important, analytic combinatorics remains an active and vibrant field of research. New transfer theorems, an ever-widening variety of combinatorial structures, and applications in a variety of scientific disciplines continue to be discovered. The full story of analytic combinatorics certainly remains to be written.

The remainder of this book is devoted to surveying classic combinatorial structures and their relationship to a variety of important computer algorithms. Since we are primarily studying fundamental structures, we will often use analytic combinatorics. But another important theme is that plenty of important problems stemming from the analysis of algorithms are still not fully understood. Indeed, some of the most challenging problems in analytic combinatorics derive from structures defined in classic computer algorithms that are simple, elegant, and broadly useful.

	construction	symbolic transfer	GF	analytic transfer	coefficient asymptotics
Unlabelled classes					
integers	$SEQ(\mathcal{Z})$	5.1	$\frac{1}{1-z}$	Taylor	1
bistings	$SEQ(\mathcal{Z}_0 + \mathcal{Z}_1)$	5.1	$\frac{1}{1-2z}$	Taylor	2^N
bitstrings with no 00	$\mathcal{G} = \epsilon + \mathcal{Z}_0 + (\mathcal{Z}_1 + \mathcal{Z}_{01}) \times \mathcal{G}$	5.1	$\frac{1+z}{1-z-z-z^2}$	4.1	$\sim \frac{\phi^N}{\sqrt{5}}$
binary trees	$\mathcal{T} = \square + \mathcal{T} \times \mathcal{T}$	5.1	$\frac{1-\sqrt{1-4z}}{2}$	5.5 corollary	$\frac{4^N}{N\sqrt{\pi N}}$
bytestings	$SEQ(\mathcal{Z}_0 + \dots + \mathcal{Z}_{M-1})$	5.1	$\frac{1}{1-Mz}$	Taylor	M^N
Labelled classes					
urns	$SET(\mathcal{Z})$	5.2	e^z	Taylor	1
permutations	$SEQ(\mathcal{Z})$	5.2	$\frac{1}{1-z}$	Taylor	$N!$
cycles	$CYC(\mathcal{Z})$	5.2	$\ln \frac{1}{1-z}$	Taylor	$(N-1)!$
derangements	$SET(CYC_{>1}(\mathcal{Z}))$	5.2	$\frac{e^{-z}}{1-z}$	5.5	$\sim \frac{N!}{e}$
generalized derangements	$SET(CYC_{>M}(\mathcal{Z}))$	5.2	$\frac{e^{-z\dots-z^{M/M}}}{1-z}$	5.5	$\sim \frac{N!}{e^{H_M}}$

Table 5.3 Analytic combinatorics examples in this chapter

References

1. E. A. BENDER. "Asymptotic methods in enumeration," *SIAM Review* **16**, 1974, 485–515.
2. E. A. BENDER AND J. R. GOLDMAN. "Enumerative uses of generating functions," *Indiana University Mathematical Journal* **2**, 1971, 753–765.
3. F. BERGERON, G. LABELLE, AND P. LEROUX. *Combinatorial Species and Tree-like Structures*, Cambridge University Press, 1998.
4. N. CHOMSKY AND M. P. SCHÜTZENBERGER. "The algebraic theory of context-free languages," in *Computer Programming and Formal Languages*, P. Braffort and D. Hirschberg, eds., North Holland, 1963, 118–161.
5. L. COMTET. *Advanced Combinatorics*, Reidel, Dordrecht, 1974.
6. P. FLAJOLET AND A. ODLYZKO. "Singularity analysis of generating functions," *SIAM Journal on Discrete Mathematics* **3**, 1990, 216–240.
7. P. FLAJOLET, B. SALVY, AND P. ZIMMERMAN. "Automatic average-case analysis of algorithms," *Theoretical Computer Science* **79**, 1991, 37–109.
8. P. FLAJOLET AND R. SEDGEWICK. *Analytic Combinatorics*, Cambridge University Press, 2009.
9. I. GOULDEN AND D. JACKSON. *Combinatorial Enumeration*, John Wiley, New York, 1983.
10. G. PÓLYA. "On picture-writing," *American Mathematical Monthly* **10**, 1956, 689–697.
11. G. PÓLYA, R. E. TARJAN, AND D. R. WOODS. *Notes on Introductory Combinatorics*, Progress in Computer Science, Birkhäuser, 1983.
12. V. N. SACHKOV. *Combinatorial Methods in Discrete Mathematics*, volume 55 of *Encyclopedia of Mathematics and its Applications*, Cambridge University Press, 1996.
13. R. P. STANLEY. *Enumerative Combinatorics*, Wadsworth & Brooks/Cole, 1986, 2nd edition, Cambridge, 2011.
14. 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.
15. E. T. WHITTAKER AND G. N. WATSON. *A Course of Modern Analysis*, Cambridge University Press, 4th edition, 1927.

16. H. Wilf. *Generatingfunctionology*, Academic Press, San Diego, 1990,
2nd edition, A. K. Peters, 2006.

CHAPTER SIX

TREES

TREES are fundamental structures that arise implicitly and explicitly in many practical algorithms, and it is important to understand their properties in order to be able to analyze these algorithms. Many algorithms construct trees explicitly; in other cases trees assume significance as models of programs, especially recursive programs. Indeed, trees are the quintessential nontrivial recursively defined objects: a tree is either empty or a root node connected to a sequence (or a multiset) of trees. We will examine in detail how the recursive nature of the structure leads directly to recursive analyses based upon generating functions.

We begin with *binary trees*, a particular type of tree first introduced in Chapter 3. Binary trees have many useful applications and are particularly well suited to computer implementations. We then consider trees in general, including a correspondence between general and binary trees. Trees and binary trees are also directly related to several other combinatorial structures such as paths in lattices, triangulations, and ruin sequences. We discuss several different ways to represent trees, not only because alternate representations often arise in applications, but also because an analytic argument often may be more easily understood when based upon an alternate representation.

We consider binary trees both from a purely combinatorial point of view (where we enumerate and examine properties of all possible different structures) and from an algorithmic point of view (where the structures are built and used by algorithms, with the probability of occurrence of each structure induced by the input). The former is important in the analysis of recursive structures and algorithms, and the most important instance of the latter is a fundamental algorithm called *binary tree search*. Binary trees and binary tree search are so important in practice that we study their properties in considerable detail, expanding upon the approach begun in Chapter 3.

As usual, after considering enumeration problems, we move on to analysis of parameters. We focus on the analysis of *path length*, a basic parameter of trees that is natural to study and useful to know about, and we consider *height* and some other parameters as well. Knowledge of basic facts about

path length and height for various kinds of trees is crucial for us to be able to understand a variety of fundamental computer algorithms. Our analysis of these problems is prototypical of the relationship between classical combinatoric and modern algorithmic studies of fundamental structures, a recurring theme throughout this book.

As we will see, the analysis of path length in trees flows naturally from the basic tools that we have developed, and generalizes to provide a way to study a broad variety of tree parameters. We will also see that, by contrast, the analysis of tree height presents significant technical challenges. Though there is not necessarily any relationship between the ease of describing a problem and the ease of solving it, this disparity in ease of analysis is somewhat surprising at first look, because path length and height are both parameters that have simple recursive descriptions that are quite similar.

As discussed in Chapter 5, analytic combinatorics can unify the study of tree enumeration problems and analysis of tree parameters, leading to very straightforward solutions for many otherwise unapproachable problems. For best effect, this requires a certain investment in some basic combinatorial machinery (see [15] for full details). We provide direct analytic derivations for many of the important problems considered in this chapter, alongside symbolic arguments or informal descriptions of how they might apply. Detailed study of this chapter might be characterized as an exercise in appreciating the value of the symbolic method.

We consider a number of different types of trees, and some classical combinatorial results about properties of trees, moving from the specifics of the binary tree to the general notion of a tree as an acyclic connected graph. Our goal is to provide access to results from an extensive literature on the combinatorial analysis of trees, while at the same time providing the groundwork for a host of algorithmic applications.

6.1 Binary Trees. In Chapter 3, we encountered *binary trees*, perhaps the simplest type of tree. Binary trees are recursive structures that are made up of two different types of nodes that are attached together according to a simple recursive definition:

Definition A *binary tree* is either an external node or an internal node attached to an ordered pair of binary trees called the left subtree and the right subtree of that node.

We refer to empty subtrees in a binary tree as *external nodes*. As such, they serve as placeholders. Unless in a context where both types are being considered, we refer to the internal nodes of a tree simply as the “nodes” of the tree. We normally consider the subtrees of a node to be connected to the node with two links, the left link and the right link.

For reference, Figure 6.1 shows three binary trees. By definition, each internal node has exactly two links; by convention, we draw the subtrees of each node below the node on the page, and represent the links as lines connecting nodes. Each node has exactly one link “to” it, except the node at the top, a distinguished node called the *root*. It is customary to borrow terminology from family trees: the nodes directly below a node are called its *children*; nodes farther down are called *descendants*; the node directly above each node is called its *parent*; nodes farther up are called *ancestors*. The root, drawn at the top of the tree by convention, has no parent.

External nodes, represented as open boxes in Figure 6.1, are at the bottom of the tree and have no children. To avoid clutter, we often refrain from drawing the boxes representing external nodes in figures with large trees or large numbers of trees. A *leaf* in a binary tree is an internal node with no children (both subtrees empty), which is *not* the same thing as an external node (a placeholder for an empty subtree).

We have already considered the problem of enumerating binary trees. Figures 3.1 and 5.2 show all the binary trees with 1, 2, 3, and 4 internal nodes, and the following result is described in detail in Chapters 3 and 5.

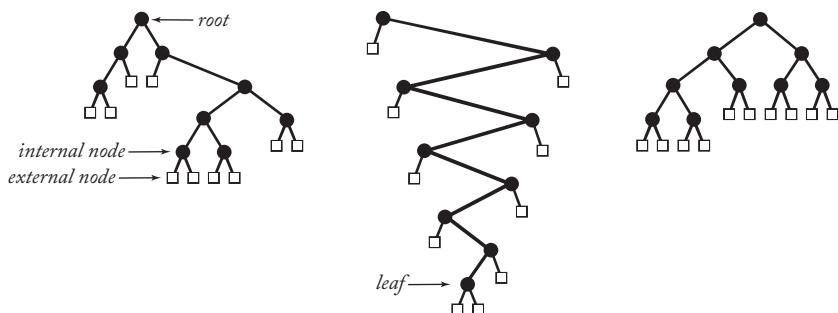


Figure 6.1 Three binary trees

Theorem 6.1 (Enumeration of binary trees). The number of binary trees with N internal nodes is given by the Catalan numbers:

$$T_N = \frac{1}{N+1} \binom{2N}{N} \sim \frac{4^N}{\sqrt{\pi N^3}}.$$

Proof. See §3.8 and §5.2. Figure 6.2 summarizes the analytic combinatorics of the proof. ■

By the following lemma (and as noted in the analytic proof in Chapter 5), the Catalan numbers also count the number of binary trees with $N + 1$ external nodes. In this chapter, we consider other basic parameters, in the context of comparisons with other types of trees.

Lemma The number of external nodes in any binary tree is exactly one greater than the number of internal nodes.

Proof. Let e be the number of external nodes and i the number of internal nodes. We count the links in the tree in two different ways. Each internal node has exactly two links “from” it, so the number of links is $2i$. But the number of links is also $i + e - 1$, since each node but the root has exactly one link “to” it. Equating these two gives $2i = i + e - 1$, or $i = e - 1$. ■

Exercise 6.1 Develop an alternative proof of this result using induction.

Exercise 6.2 What proportion of the binary trees with N internal nodes have both subtrees of the root nonempty? For $N = 1, 2, 3$, and 4 , the answers are $0, 0, 1/5$, and $4/14$, respectively (see Figure 5.2).

Exercise 6.3 What proportion of the binary trees with $2N + 1$ internal nodes have N internal nodes in each of the subtrees of the root?

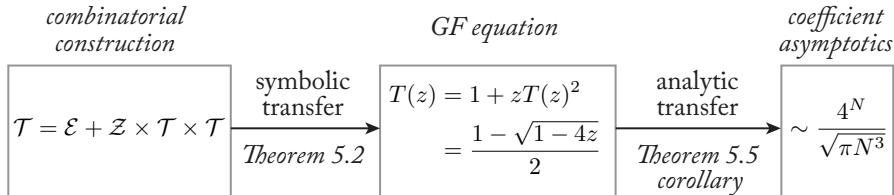


Figure 6.2 Enumerating binary trees via analytic combinatorics

6.2 Forests and Trees. In a binary tree, no node has more than two children. This characteristic makes it obvious how to represent and manipulate such trees in computer implementations, and it relates naturally to “divide-and-conquer” algorithms that divide a problem into two subproblems. However, in many applications (and in more traditional mathematical usage), we need to consider a more general kind of tree:

Definition A *forest* is a sequence of disjoint trees. A *tree* is a node (called the *root*) connected to the roots of trees in a forest.

This definition is recursive, as are tree structures. When called for in context, we sometimes use the term *general tree* to refer to trees. As for binary trees, the order of the trees in a forest is significant. When applicable, we use the same nomenclature as for binary trees: the subtrees of a node are its children, a root node has no parents, and so forth. Trees are more appropriate models than binary trees for certain computations.

For reference, Figure 6.3 shows a three-tree forest. Again, roots of trees have no parents and are drawn at the top by convention. There are no external nodes; instead, a node at the bottom with no children is called a *leaf*. Figure 6.4 depicts all the forests of 1 through 4 nodes and all the trees of 1 through 5 nodes. The number of forests with N nodes is the same as the number of trees with $N + 1$ nodes—just add a root and make its children the roots of the trees in the forest. Moreover, the Catalan numbers are immediately apparent in Figure 6.4. A well-known 1:1 correspondence with binary trees is one way to enumerate binary trees. Before considering that correspondence, we consider an analytic proof using the symbolic method.

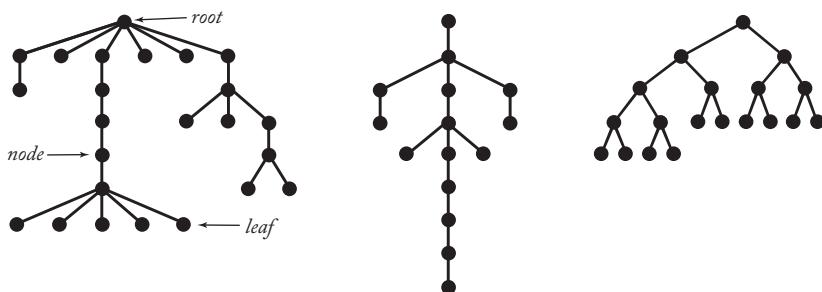
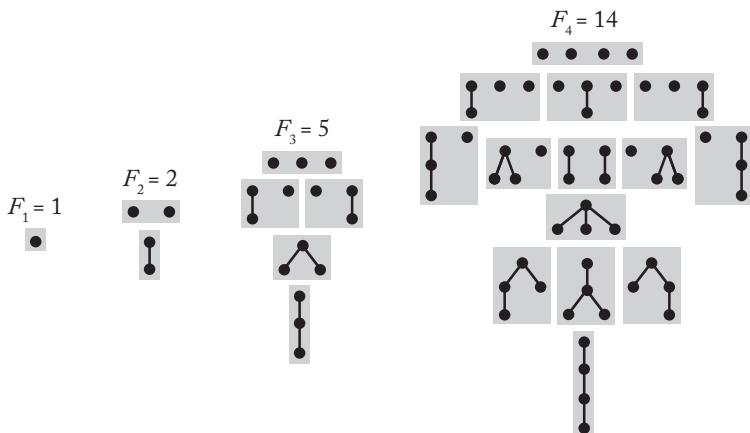


Figure 6.3 A three-tree forest

Forests (with N nodes, $1 \leq N \leq 4$)



General trees (with N nodes, $1 \leq N \leq 5$)

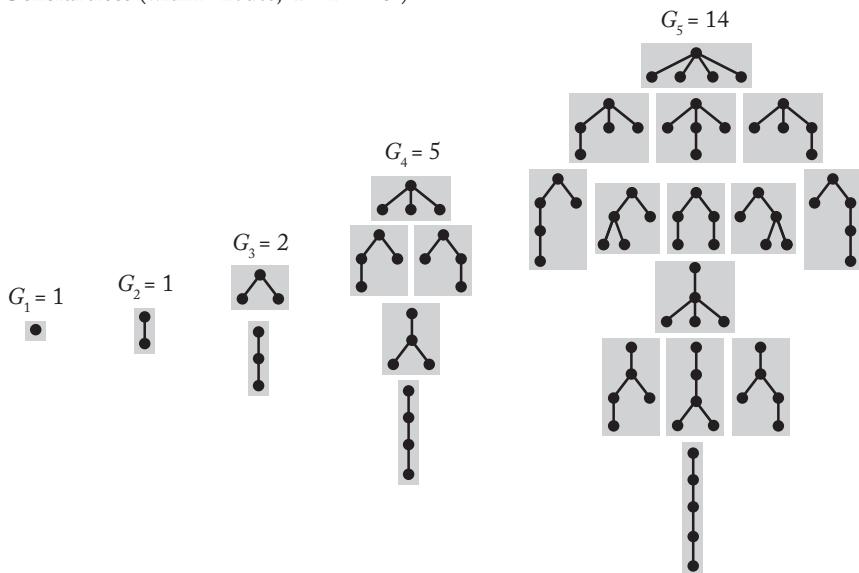


Figure 6.4 Forests and trees

Theorem 6.2 (Enumeration of forests and trees). Let F_N be the number of forests with N nodes and G_N be the number of trees with N nodes. Then G_N is exactly equal to F_{N-1} and F_N is exactly equal to the number of binary trees with N internal nodes and is given by the Catalan numbers:

$$F_N = T_N = \frac{1}{N+1} \binom{2N}{N} \sim \frac{4^N}{\sqrt{\pi N^3}}.$$

Proof. Immediate via the symbolic method. A forest is a sequence of trees and a tree is a node connected to a forest

$$\mathcal{F} = SEQ(\mathcal{G}) \quad \text{and} \quad \mathcal{G} = \mathcal{Z} \times \mathcal{F},$$

which translates directly (see Theorem 5.1) into

$$F(z) = \frac{1}{1 - G(z)} \quad \text{and} \quad G(z) = zF(z),$$

which implies that $G_N = F_{N-1}$ and

$$F(z) - zF(z)^2 = 1.$$

so $F_N = T_N$ because their GFs satisfy the same functional equation (see Section 5.2).

Exercise 6.4 For what proportion of the trees with N internal nodes does the root have a single child? For $N = 1, 2, 3, 4$, and 5 , the answers are $0, 1, 1/2, 2/5$, and $5/14$, respectively (see Figure 6.4).

Exercise 6.5 Answer the previous questions for the root having t children, for $t = 2, 3$, and 4 .

Exercise 6.6 What proportion of the forests with N nodes have no trees consisting of a single node? For $N = 1, 2, 3$, and 4 , the answers are $0, 1/2, 2/5$, and $3/7$, respectively (see Figure 6.4).

6.3 Combinatorial Equivalences to Trees and Binary Trees. In this section, we address the broad reach of trees and binary trees as combinatorial models. We begin by showing that trees and binary trees may be viewed as two specific ways to represent the same combinatorial objects (which are enumerated by the Catalan numbers). Then we summarize other combinatorial objects that arise in numerous applications and for which similar correspondences have been developed (see, for example, [31]).

Rotation correspondence. A fundamental one-to-one correspondence between forests and binary trees provides a direct proof that $F_N = T_N$. This correspondence, called the *rotation correspondence*, is illustrated in Figure 6.5. Given a forest, we construct the corresponding binary tree as follows: the root of the binary tree is the root of the first tree in the forest; its right link points to the representation of the remainder of the forest (not including the first tree); its left link points to the representation for the forest comprising the subtrees of the root of the first tree. In other words, each node has a left link to its first child and a right link to its next sibling in the forest (the correspondence is also often called the “first child, next sibling correspondence”). As illustrated in the figure, the nodes in the binary tree appear to be placed by rotating the general tree 45 degrees clockwise.

Conversely, given a binary tree with root x , construct the corresponding forest as follows: the root of the first tree in the forest is x ; the children of that node are the trees in the forest constructed from the left subtree of x ; and the rest of the forest comprises the trees in the forest constructed from the right subtree of x .

This correspondence is of interest in computing applications because it provides an efficient way to represent forests (with binary trees). Next, we see



Figure 6.5 Rotation correspondence between trees and binary trees

that many other types of combinatorial objects not only have this property, but also can provide alternative representations of trees and binary trees.

Parenthesis systems. Each forest with N nodes corresponds to a set of N pairs of parentheses: from the definition, there is a sequence of trees, each consisting of a root and a sequence of subtrees with the same structure. If we consider that each tree should be enclosed in parentheses, then we are led immediately to a representation that uses only parentheses: for each tree in the forest, write a left parenthesis, followed by the parenthesis system for the forest comprising the subtrees (determined recursively), followed by a right parenthesis. For example, this system represents the forest at the left in Figure 6.5:

$$((()) () ()) () (() ()) (()).$$

You can see the relationship between this and the tree structure by writing parentheses at the levels corresponding to the root nodes of the tree they enclose, as follows:

$$\begin{array}{ccccccccc} (& &) & (&) & (&) & (&) \\ (&) & (&) & (&) & (&) & (&) \\ (&) \end{array}$$

Collapsing this structure gives the parenthesis system representation.

Cast in terms of tree traversal methods, we can find the parenthesis system corresponding to a tree by recursively traversing the tree, writing “(” when going “down” an edge and “)” when going “up” an edge. Equivalently, we may regard “(” as corresponding to “start a recursive call” and “)” as corresponding to “finish a recursive call.”

In this representation, we are describing only the shape of the tree, not any information that might be contained in the nodes. Next we consider representations that are appropriate if nodes may have an associated key or other additional information.

Exercise 6.7 Give parenthesis systems that correspond to the forests in Figure 6.2.

Space-efficient representations of tree shapes. The parenthesis system encodes a tree of size N with a sequence of $2N + O(1)$ bits. This is appreciably smaller than standard representations of trees with pointers. Actually, it

comes close to the information-theoretic optimal encoding length, the logarithm of the Catalan numbers:

$$\lg T_N = 2N - O(\log N).$$

Such representations of tree shapes are useful in applications where very large trees must be stored (e. g., index structures in databases) or transmitted (e. g., representations of Huffman trees; see [33]).

Preorder and postorder representations of trees. In §6.5, we discuss basic *tree traversal* algorithms that lead immediately to various tree representations. Specifically, we extend the parenthesis system to include information at the nodes. As with tree traversal, the root can be listed either before the subtrees (preorder) or after the subtrees (postorder). Thus the preorder representation of the forest on the left in Figure 6.5 is

$$(\bullet (\bullet (\bullet)) (\bullet) (\bullet)) (\bullet) (\bullet (\bullet) (\bullet)) (\bullet (\bullet))$$

and the postorder representation is

$$(((\bullet) \bullet) (\bullet) (\bullet) \bullet) (\bullet) ((\bullet) (\bullet) \bullet) ((\bullet) \bullet).$$

When we discuss refinements below, we do so in terms of preorder, though of course the representations are essentially equivalent, and the refinements apply to postorder as well.

Preorder degree representation. Another way to represent the shape of the forest in Figure 6.5 is the string of integers

$$3 \ 1 \ 0 \ 0 \ 0 \ 0 \ 2 \ 0 \ 0 \ 1 \ 0.$$

This is simply a listing of the numbers of children of the nodes, in preorder. To see why it is a unique representation, it is simpler to consider the same sequence, but subtracting 1 from each term:

$$2 \ 0 \ -1 \ -1 \ -1 \ -1 \ 1 \ -1 \ -1 \ 0 \ -1.$$

Moving from left to right, this can be divided into subsequences that have the property that (i) the sum of the numbers in the subsequence is -1 , and (ii) the sum of any prefix of the numbers in the subsequence is greater than or equal to -1 . Delimiting the subsequences by parentheses, we have

$$(2 \ 0 \ -1 \ -1 \ -1) \ (-1) \ (1 \ -1 \ -1) \ (0 \ -1).$$

This gives a correspondence to parenthesis systems: each of these subsequences corresponds to a tree in the forest. Deleting the first number in each sequence and recursively decomposing gives the parenthesis system. Now, each parenthesized sequence of numbers not only sums to -1 but also has the property that the sum of any prefix is nonnegative. It is straightforward to prove by induction that conditions (i) and (ii) are necessary and sufficient to establish a direct correspondence between sequences of integers and trees.

Binary tree traversal representations. Binary tree representations are simpler because of the marked distinction between external (degree 0) and internal (degree 2) nodes. For example, in §6.5 we consider binary tree representations of arithmetic expressions (see Figure 6.11). The familiar representation corresponds to to list the subtrees, parenthesized, with the character associated with the root in between:

$$((x + y) * z) - (w + ((a - (v + y)) / ((z + y) * x))).$$

This is called *inorder* or *infix* when referring specifically to arithmetic expressions. It corresponds to an inorder tree traversal where we write “(,” then traverse the left subtree, then write the character at the root, then traverse the right subtree, then write “).”

Representations corresponding to preorder and postorder traversals can be defined in an analogous manner. But for preorder and postorder, parentheses are not needed: external (operands) and internal (operators) nodes are identified; thus the preorder node degree sequence is implicit in the representation, which determines the tree structure, in the same manner as discussed earlier. In turn, the preorder, or *prefix*, listing of the above sequence is

$$- * + x \ y \ z + w / - a + v \ y * + z \ y \ x,$$

and the postorder, or *postfix*, listing is

$$x \ y + z * w \ a \ v \ y + - z \ y + x * / + -.$$

Exercise 6.8 Given an (ordered) tree, consider its representation as a binary tree using the rotation correspondence. Discuss the relationship between the preorder and postorder representations of the ordered tree and the preorder, inorder, and postorder representations of the corresponding binary tree.

Gambler's ruin and lattice paths. For binary trees, nodes have either two or zero children. If we list, for each node, one less than the number of children, in preorder, then we get either $+1$ or -1 , which we abbreviate simply as $+$ or $-$. Thus a binary tree can be uniquely represented as a string of $+$ and $-$ symbols. The tree structure in Figure 6.11 is

$$+ + + - - - + - + + - + - - + + - - - .$$

This encoding is a special case of the preorder degree representation. Which strings of $+$ and $-$ symbols correspond to binary trees?

Gambler's ruin sequences. These strings correspond exactly to the following situation. Suppose that a gambler starts with $\$0$ and makes a $\$1$ bet. If he loses, he has $-\$1$ and is ruined, but if he wins, he has $\$1$ and bets again. The plot of his holdings is simply the plot of the partial sums of the plus-minus sequence (number of pluses minus number of minuses). Any path that does not cross the $\$0$ point (except at the last step) represents a possible path to ruin for the gambler, and such paths are also in direct correspondence with binary trees. Given a binary tree, we produce the corresponding path as just described: it is a gambler's ruin path by the same inductive reasoning as we used earlier to prove the validity of the preorder degree representation of ordered trees. Given a gambler's ruin path, it can be divided in precisely one way into two subpaths with the same characteristics by deleting the first step and splitting the path at the *first* place that it hits the $\$0$ axis. This division (inductively) leads to the corresponding binary tree.

Ballot problems. A second way of looking at this situation is to consider an election where the winner has $N + 1$ votes and the loser has N votes. A plus-minus sequence then corresponds to the set of ballots, and plus-minus sequences corresponding to binary trees are those where the winner is never behind as the ballots are counted.

Paths in a lattice. A third way of looking at the situation is to consider paths in an N -by- N square lattice that proceed from the upper left corner down to the lower right corner using “right” and “down” steps. There are $\binom{2N}{N}$ such paths, but only one out of every $N + 1$ starts “right” and does not cross the diagonal, because there is a direct correspondence between such paths and binary trees, as shown in Figure 6.6. This is obviously a graph of the gambler's holdings (or of the winner's margin as the ballots are counted) rotated 45

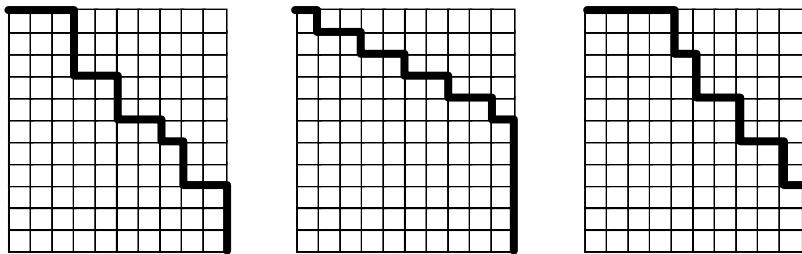


Figure 6.6 Lattice-path representations of binary trees in Figure 6.1

degrees. It also is a graph of the stack size if the corresponding tree is traversed in preorder.

We will study properties of these *gambler's ruin sequences*, or *ballot sequences*, in Chapters 7 and 8. They turn out to be relevant in the analysis of sorting and merging algorithms (see §7.6), and they can be studied using general tools related to string enumeration (see §8.5).

Exercise 6.9 Find and prove the validity of a correspondence between N -step gambler's ruin paths and ordered forests of $N - 1$ nodes.

Exercise 6.10 How many N -bit binary strings have the property that the number of ones in the first k bits does not exceed the number of zeros, for all k ?

Exercise 6.11 Compare the parenthesis representation of an ordered forest to the plus-minus representation of its associated binary tree. Explain your observation.

Planar subdivision representations. We mention another classical correspondence because it is so well known in combinatorics: the “triangulated N -gon” representation, shown in Figure 6.7 for the binary tree at the left in Figure 6.1. Given a convex N -gon, how many ways are there to divide it into triangles with noncrossing “diagonal” lines connecting vertices? The answer is a Catalan number, because of the direct correspondence with binary trees. This application marked the first appearance of Catalan numbers, in the work of Euler and Segner in 1753, about a century before Catalan himself. The correspondence is plain from Figure 6.7: given a triangulated N -gon, put an internal node on each diagonal and one (the root) on one exterior edge and an external node on each remaining exterior edge. Then connect the root to

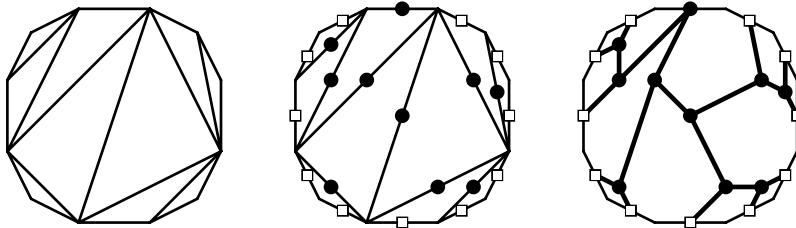


Figure 6.7 Binary tree corresponding to a triangulated N -gon

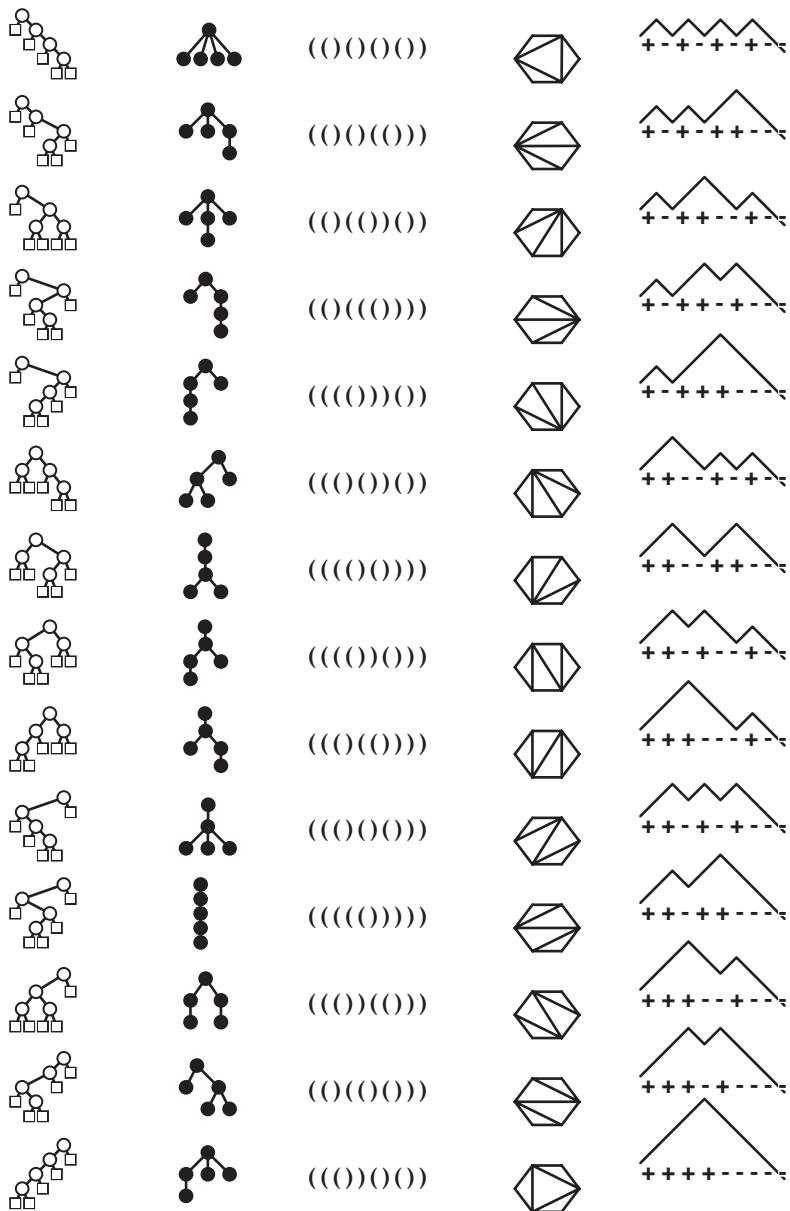
the other two nodes in its triangle and continue connecting in the same way down to the bottom of the tree.

This particular correspondence is classical in combinatorics, and there are other planar subdivisions that have been more recently developed and are of importance in the design and analysis of some geometric algorithms. For example, Bentley's *2D-tree* data structure [4] is based on dividing a rectangular region in the plane with horizontal lines, then further dividing the resulting regions with vertical lines, and so on, continuing to divisions as fine as desired, alternating horizontal and vertical lines. This recursive division corresponds to a tree representation. Many planar subdivisions of this sort have been devised to subdivide multidimensional spaces for point location and other applications.

Figure 6.8 summarizes the most well-known tree representations that we have discussed in this section, for five-node trees. We dwell on these representations to underscore the ubiquity of trees in combinatorics and—since trees arise explicitly as data structures and implicitly as models of recursive computation—the analysis of algorithms, as well. Familiarity with various representations is useful because properties of a particular algorithm can sometimes be more clearly seen in one of the equivalent representations than in another.

Exercise 6.12 Give a method for representing a tree with a subdivided rectangle when the ratio of the height to the width of any rectangle is between α and $1/\alpha$ for constant $\alpha > 1$. Find a solution for α as small as you can.

Exercise 6.13 There is an obvious correspondence where left-right symmetry in triangulations is reflected in left-right symmetry in trees. What about rotations? Is

**Figure 6.8** Binary trees, trees, parentheses, triangulations, and ruin sequences

there any relationship among the N trees corresponding to the N rotations of an asymmetric triangulation?

Exercise 6.14 Consider strings of N integers with two properties: first, if $k > 1$ is in the string, then so is $k - 1$, and second, some larger integer must appear somewhere between any two occurrences of any integer. Show that the number of such strings of length N is described by the Catalan numbers, and find a direct correspondence with trees or binary trees.

6.4 Properties of Trees. Trees arise naturally in a variety of computer applications. Our primary notion of *size* is generally taken to be the number of nodes for general trees and, depending on context, the number of internal nodes or the number of external nodes for binary trees. For the analysis of algorithms, we are primarily interested in two basic properties of trees of a given size: *path length* and *height*.

To define these properties, we introduce the notion of the *level* of a node in a tree: the root is at level 0, children of the root are at level 1, and in general, children of a node at level k are at level $k + 1$. Another way of thinking of the level is as the distance (number of links) we have to traverse to get from the root to the node. We are particularly interested in the sum of the distances from each node to the root:

Definition Given a tree or forest t , the *path length* is the sum of the levels of each of the nodes in t and the *height* is the maximum level among all the nodes in t .

We use the notation $|t|$ to refer to the number of nodes in a tree t , the notation $\text{pl}(t)$ to refer to the path length, and the notation $\text{h}(t)$ to refer to the height. These definitions hold for forests as well. In addition, the path length of a forest is the sum of the path lengths of the constituent trees and the height of a forest is the maximum of the heights of the constituent trees.

Definition Given a binary tree t , the *internal path length* is the sum of the levels of each of the internal nodes in t , the *external path length* is the sum of the levels of each of the external nodes in t , and the *height* is the maximum level among all the external nodes in t .

We use the notation $\text{ipl}(t)$ to refer to the internal path length of a binary tree, $\text{xpl}(t)$ to refer to the external path length, and $\text{h}(t)$ to refer to the height.

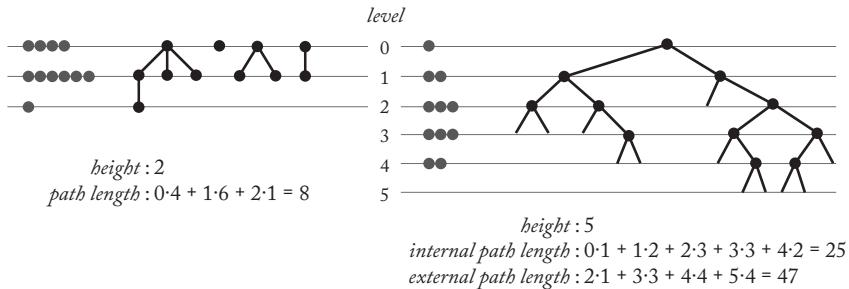


Figure 6.9 Path length and height in a forest and in a binary tree

We use $|t|$ to refer to the number of internal nodes in a binary tree unless specifically noted in contexts where it is more appropriate to count by external nodes or all nodes.

Definition In general trees, *leaves* are the nodes with no children. In binary trees, leaves are the (internal) nodes with both children external.

Figure 6.9 gives examples for the purpose of reinforcing these definitions. The forest on the left has height 2 and path length 8, with 7 leaves; the binary tree on the right has height 5, internal path length 25, and external path length 47, with 4 leaves. To the left of each tree is a *profile*—a plot of the number of nodes on each level (internal nodes for the binary tree), which facilitates calculating path lengths.

Recursive definitions and elementary bounds. It is often convenient to work with recursive definitions for tree parameters. In a binary tree t , the parameters we have defined are all 0 if t is an external node; otherwise, if the root of t is an internal node and the left and right subtrees, respectively, are denoted by t_l and t_r , we have following recursive formulae:

$$\begin{aligned} |t| &= |t_l| + |t_r| + 1 \\ \text{ipl}(t) &= \text{ipl}(t_l) + \text{ipl}(t_r) + |t| - 1 \\ \text{xpl}(t) &= \text{xpl}(t_l) + \text{xpl}(t_r) + |t| + 1 \\ h(t) &= 1 + \max(h(t_l), h(t_r)). \end{aligned}$$

These are equivalent to the definitions given earlier. First, the internal node count of a binary tree is the sum of the node counts for its subtrees plus 1 (the

root). Second, the internal path length is the sum of the internal path lengths of the subtrees plus $|t| - 1$ because each of the $|t| - 1$ nodes in the subtrees is moved down exactly one level when the subtrees are attached to the tree. The same argument holds for external path length, noting that there are $|t| + 1$ external nodes in the two subtrees of a binary tree with $|t|$ internal nodes. The result for height again follows from the fact that the levels of all the nodes in the subtrees are increased by exactly 1.

Exercise 6.15 Give recursive formulations describing path length and height in general trees.

Exercise 6.16 Give recursive formulations for the number of leaves in binary trees and in general trees.

These definitions will serve as the basis for deriving functional equations on associated generating functions when we analyze the parameters below. Also, they can be used for inductive proofs about relationships among the parameters.

Lemma Path lengths in any binary tree t satisfy $xpl(t) = ipl(t) + 2|t|$.

Proof. Subtracting the recursive formula for $xpl(t)$ from the recursive formula for $ipl(t)$, we have

$$ipl(t) - xpl(t) = ipl(t_l) - xpl(t_l) + ipl(t_r) - xpl(t_r) + 2$$

and the lemma follows directly by induction. ■

Path length and height are not independent parameters: if the height is very large, then so must be the path length, as shown by the following bounds, which are relatively crude, but useful.

Lemma The height and internal path length of any nonempty binary tree t satisfy the inequalities

$$ipl(t) \leq |t|h(t) \quad \text{and} \quad h(t) \leq \sqrt{2ipl(t)} + 1.$$

Proof. If $h(t) = 0$ then $ipl(t) = 0$ and the stated inequalities hold. Otherwise, we must have $ipl(t) < |t|h(t)$, since the level of each internal node must be strictly smaller than the tree height. Furthermore, there is at least one internal node at each level less than the height, so we must have $0 + 1 + 2 + \dots + h(t) - 1 \leq ipl(t)$. Hence $2ipl(t) \geq h(t)^2 - h(t) \geq (h(t) - 1)^2$ (subtract the quantity $h(t) - 1$, which is nonnegative, from the right-hand side), and thus $h(t) \leq \sqrt{2ipl(t)} + 1$. ■

Exercise 6.17 Prove that the height of a binary tree with N external nodes has to be at least $\lg N$.

Exercise 6.18 [Kraft equality] Let k_j be the number of external nodes at level j in a binary tree. The sequence $\{k_0, k_1, \dots, k_h\}$ (where h is the height of the tree) describes the profile of the tree. Show that a vector of integers describes the profile of a binary tree if and only if $\sum_j 2^{-k_j} = 1$.

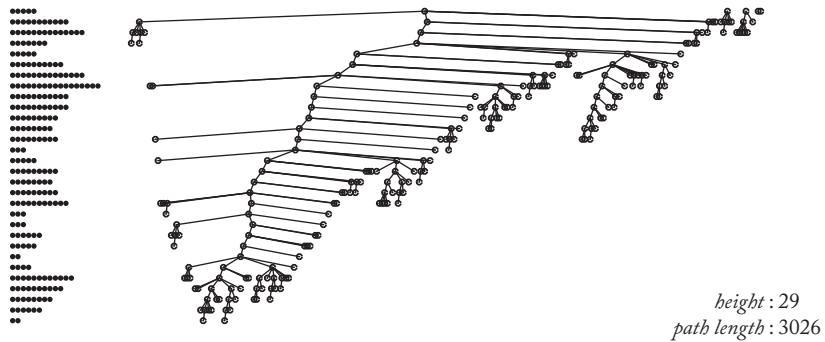
Exercise 6.19 Give tight upper and lower bounds on the path length of a general tree with N nodes.

Exercise 6.20 Give tight upper and lower bounds on the internal and external path lengths of a binary tree with N internal nodes.

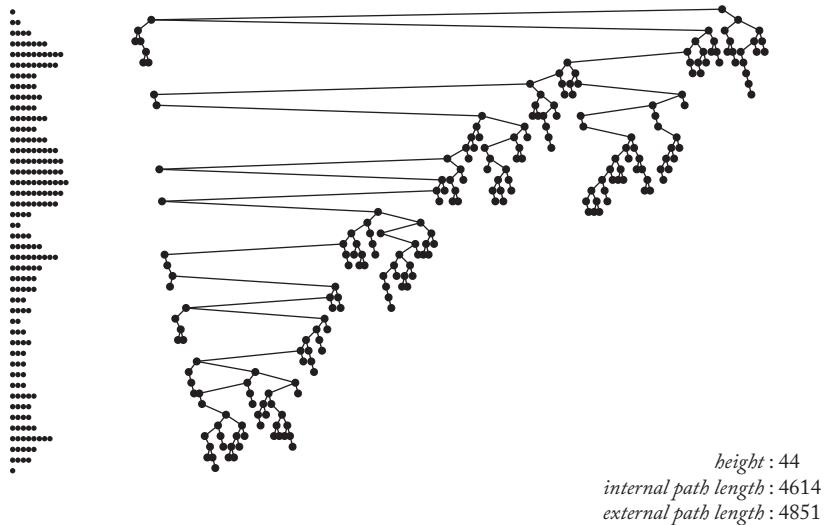
Exercise 6.21 Give tight upper and lower bounds on the number of leaves in a binary tree with N nodes.

IN THE ANALYSIS OF algorithms, we are particularly interested in knowing the *average* values of these parameters, for various types of “random” trees. One of our primary topics of discussion for this chapter is how these quantities relate to fundamental algorithms and how we can determine their expected values. Figure 6.10 gives some indication of how these differ for different types of trees. At the top is a random forest, drawn from the distribution where each forest with the same number of nodes is equally likely to occur. At the bottom is a random binary tree, drawn from the distribution where each binary tree with the same number of nodes is considered to be equally likely to occur. The figure also depicts a profile for each tree (the number of nodes at each level), which makes it easier to calculate the height (number of levels) and path length (sum over i of i times the number of nodes at level i). The random binary tree clearly has larger values for both path length and height than the random forest. One of the prime objectives of this chapter is to quantify these and similar observations, precisely.

Random forest (with 237 nodes)



Random binary tree (with 237 internal nodes)

**Figure 6.10** A random forest and a random binary tree

6.5 Examples of Tree Algorithms. Trees are relevant to the study of analysis of algorithms not only because they implicitly model the behavior of recursive programs but also because they are involved explicitly in many basic algorithms that are widely used. We will briefly describe a few of the most fundamental such algorithms here. This brief description certainly cannot do justice to the general topic of the utility of tree structures in algorithm design, but we can indicate that the study of tree parameters such as path length and height provides the basic information needed to analyze a host of important algorithms.

Traversal. In a computer representation, one of the fundamental operations on trees is *traversal*: systematically processing each of the nodes of the tree. This operation also is of interest combinatorially, as it represents a way to establish a correspondence between (two-dimensional) tree structures and various (one-dimensional) linear representations.

The recursive nature of trees gives rise to a simple recursive procedure for traversal. We “visit” the root of the tree and recursively “visit” the subtrees. Depending on whether we visit the root before, after, or (for binary trees) in between the subtrees, we get one of three different traversal methods:

To visit all the nodes of a tree in *preorder*:

- Visit the root
- Visit the subtrees (in preorder)

To visit all the nodes of a tree in *postorder*:

- Visit the subtrees (in postorder)
- Visit the root

To visit all the nodes of a binary tree in *inorder*:

- Visit the left subtree (in inorder)
- Visit the root
- Visit the right subtree (in inorder)

In these methods, “visiting” the root might imply any procedure at all that should be systematically applied to nodes in the tree. Program 6.1 is an implementation of preorder traversal of binary trees.

To implement a recursive call, the system uses a *pushdown stack* to save the current “environment,” to be restored upon return from the procedure. The maximum amount of memory used by the pushdown stack when traversing a tree is directly proportional to tree height. Though this memory usage

may be hidden from the programmer in this case, it certainly is an important performance parameter, so we are interested in analyzing tree height.

Another way to traverse a tree is called *level order*: first list all the nodes on level 0 (the root); then list all the nodes on level 1, left to right; then list all the nodes on level 2 (left to right); and so on. This method is not suitable for a recursive implementation, but it is easily implemented just as shown earlier using a queue (first-in-first-out data structure) instead of a stack.

Tree traversal algorithms are fundamental and widely applicable. For many more details about them and relationships among recursive and nonrecursive implementations, see Knuth [24] or Sedgewick [32].

Expression evaluation. Consider arithmetic expressions consisting of *operators*, such as $+$, $-$, $*$, and $/$, and *operands*, denoted by numbers or letters. Such expressions are typically parenthesized to indicate precedence between operations. Expressions can be represented as binary trees called *parse trees*. For example, consider the case where an expression uses only binary operators. Such an expression corresponds to a binary tree, with operators in the internal nodes and operands in the external nodes, as shown in Figure 6.11. The operands corresponding to each operator are the expressions represented by the left and right subtrees of its corresponding internal node.

Given a parse tree, we can develop a simple recursive program to compute the value of the corresponding expression: (recursively) evaluate the two subtrees, then apply the operator to the computed values. Evaluation of an external node gives the current value of the associated variable. This program

```
private void preorder(Node x)
{
    if (x == null) return;
    process(x.key);
    preorder(x.left);
    preorder(x.right);
}
```

Program 6.1 Preorder traversal of a binary tree

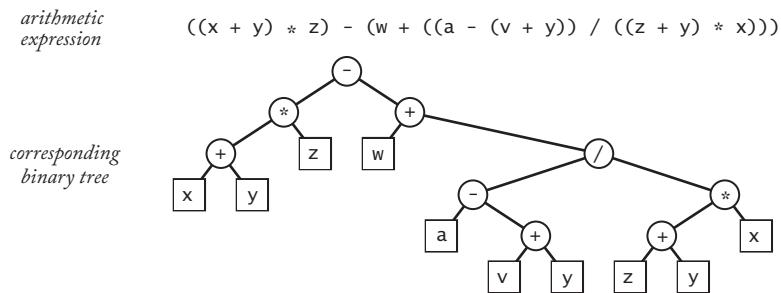


Figure 6.11 Binary tree representation of an arithmetic expression

is equivalent to a tree traversal such as Program 6.1 (but in postorder). As with tree traversal, the space consumed by this program will be proportional to the tree height. This approach is often used to evaluate arithmetic expressions in computer applications systems.

From the advent of computing, one of the main goals was to evaluate arithmetic expressions by translating them into machine code that can efficiently do the job. To this end, one approach that is typically used by programming-language compilers is to first build an expression tree associated with part of a program, then convert the expression tree into a list of instructions for evaluating an expression at execution time, such as the following:

```

r1 ← x+y
r2 ← r1*z
r3 ← v+y
r4 ← a-r3
r5 ← z+y
r6 ← r5*x
r7 ← r4/r6
r8 ← w+r7
r9 ← r2-r8

```

These instructions are close to machine instructions, for example, involving binary arithmetic operations using machine registers (indicated by the temporary variables $r1$ through $r9$), a (limited) machine resource for holding results of arithmetic operations. Generally, a reasonable goal is to use as few

registers as possible. For example, we could replace the last two instructions by the instructions

$$\begin{aligned} r7 &\leftarrow w+r7 \\ r7 &\leftarrow r2-r7 \end{aligned}$$

and use two fewer registers. Similar savings are available at other parts of the expression. The minimum number of registers needed to evaluate an expression is a tree parameter of direct practical interest. This quantity is bounded from above by tree height, but it is quite different. For example, the degenerate binary tree where all nodes but one have exactly one null link has height N but the corresponding expression can be evaluated with one register. Determining the minimum number of registers needed (and how to use them) is known as the *register allocation* problem. Expression evaluation is of interest in its own right, and it is also indicative of the importance of trees in the process of translating computer programs from higher-level languages to machine languages. Compilers generally first “parse” programs into tree representations, then process the tree representation.

Exercise 6.22 What is the minimum number of registers needed to evaluate the expression in Figure 6.11?

Exercise 6.23 Give the binary tree corresponding to the expressions $(a + b) * d$ and $((a + b) * (d - e)) * (f + g)) - h * i$. Also give the preorder, inorder, and postorder traversals of those trees.

Exercise 6.24 An expression where operators have varying numbers of operands corresponds to a tree, with operands in leaves and operators in nonleaves. Give the preorder and postorder traversals of the tree corresponding to the expression $((a^2 + b + c) * (d^4 - e^2)) * (f + g + h)) - i * j$, then give the binary tree representation of that tree and the preorder, inorder, and postorder traversals of the binary tree.

TREE TRAVERSAL AND EXPRESSION manipulation are representative of many applications where the study of parameters such as the path length and height of trees is of interest. To consider the average value of such parameters, of course, we need to specify a model defining what is meant by a “random” tree. As a starting point, we study so-called *Catalan models*, where each of the T_N binary trees of size N or general trees of size $N + 1$ are taken with equal probability. This is not the only possibility—in many situations, the trees are induced by external data, and other models of randomness are appropriate. Next, we consider a particularly important example of this situation.

6.6 Binary Search Trees. One of the most important applications of binary trees is the *binary tree search* algorithm, a method based on explicitly constructing binary trees to provide an efficient solution to a fundamental problem that arises in numerous applications. The analysis of binary tree search illustrates the distinction between models where all trees are equally likely to occur and models where the underlying distribution is determined by other factors. This juxtaposition of models is an essential concept in this chapter.

The *dictionary*, *symbol table*, or simply *search* problem is a fundamental one in computer science: a set of distinct keys is to be organized so that client queries whether or not a given key is in the set can be efficiently answered. More generally, with distinct keys, we can use binary search trees to implement an *associative array*, where we associate information with each key and can use the key to store or retrieve such information. The binary search method discussed in §2.6 is one basic method for solving this problem, but that method is of limited use because it requires a preprocessing step where all the keys are first put into sorted order, while typical applications intermix the operations of searching for keys and inserting them.

A binary tree structure can be used to provide a more flexible solution to the dictionary problem, by assigning a key to each node and keeping things arranged so that the key in every node is larger than any key in the left subtree and smaller than any key in the right subtree.

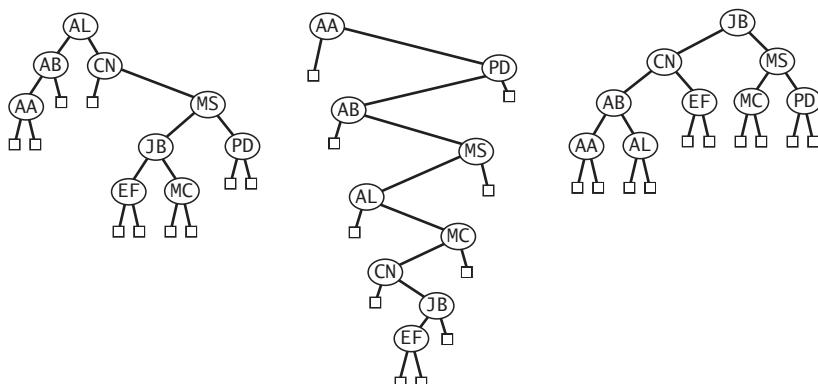


Figure 6.12 Three binary search trees

Definition A *binary search tree* is a binary tree with keys associated with the internal nodes, satisfying the constraint that the key in every node is greater than all the keys in its left subtree and smaller than all the keys in its right subtree.

Binary search trees can be built from any type of data for which a total order is defined. Typically, keys are numbers in numerical order or strings in alphabetical order. Many different binary search trees may correspond to a given set of keys. For reference, consider Figure 6.12, which shows three different binary search trees containing the same set of two-character keys AA AB AL CN EF JB MC MS PD.

Program 6.2 demonstrates the utility of binary search trees in solving the dictionary problem. It assumes that the set of keys is stored in a binary search tree and uses a recursive implementation of a “search” procedure that determines whether or not a given key is somewhere in the binary search tree. To search for a node with a key v , terminate the search (unsuccessfully) if the tree is empty and terminate the search (successfully) if the key in the root node is v . Otherwise, look in the left subtree if v is less than the key in the root node and look in the right subtree if v is greater than the key in the root node. It is a simple matter to verify that, if started on the root of a valid binary search tree with search key `key`, Program 6.2 returns `true` if and only if there is a node containing `key` in the tree; otherwise, it returns `false`.

Indeed, *any* given set of N ordered keys can be associated with any of the T_N binary tree shapes—to make a binary search tree, visit the nodes of

```
private boolean search(Node x, Key key)
{
    if (x == null) return false;
    if (key < x.key) return search(x.left, key);
    if (key > x.key) return search(x.right, key);
    return true;
}
```

Program 6.2 Binary tree search

the tree in postorder, assigning the next key in the order when visiting each node. The search algorithm works properly for any such binary search tree.

How do we construct a binary search tree containing a given set of keys? One practical approach is to add keys one by one to an initially empty tree, using a recursive strategy similar to `search`. We assume that we have first done a search to determine that the new key is not in the tree, to maintain the property that the keys in the tree are all different. To insert a new key into an empty tree, create a node containing the key and make its left and right pointers `null`, and return a reference to the node. (We use the value `null` to represent all external nodes.) If the tree is nonempty, insert the key into the left subtree if it is less than the key at the root, and into the right subtree if it is greater than the key at the root, resetting the link followed to the reference returned. This is equivalent to doing an unsuccessful search for the key, then inserting a new node containing the key in place of the external node where the search ends. The shape of the tree and the cost of building it and searching are dependent on the order in which the keys are inserted.

Program 6.3 is an implementation of this method. For example, if the key `DD` were to be inserted into any of the trees in Figure 6.9, Program 6.3 would create a new node with the key `DD` as the left child of `EF`.

In the present context, our interest is that this insertion algorithm defines a mapping from permutations to binary trees: Given a permutation,

```
private Node insert(Key key)
{
    if (x == null)
        { x = new Node(); x.key = key; }
    if (key < x.key)
        x.left = insert(x.left, key);
    else if (key > x.key)
        x.right = insert(x.right, key);
    return x;
}
```

Program 6.3 Binary search tree insertion

build a binary tree by inserting the elements in the permutation into an initially empty tree, proceeding from left to right. Figure 6.13 illustrates this correspondence for the three examples we started with. This correspondence is important in the study of search algorithms. Whatever the data type, we can consider the permutation defined by the relative order of the keys when they are inserted into the data structure, which tells us which binary tree is built by successive insertions.

In general, many different permutations may map to the same tree. Figure 6.14 shows the mapping between the permutations of four elements and the trees of four nodes. Thus, for example, it is not true that each tree is equally likely to occur if keys are inserted in random order into an initially empty tree. Indeed, it is fortunately the case that the more “balanced” tree structures, for which search and construction costs are low, are more likely to occur than tree structures for which the costs are high. In the analysis, we will quantify this observation.

Some trees are much more expensive to construct than others. In the worst case, a degenerate tree where each node has at least one external child, $i - 1$ internal nodes are examined to insert the i th node for each i between 1 and N , so a total of $N(N - 1)/2$ nodes are examined in order to construct the

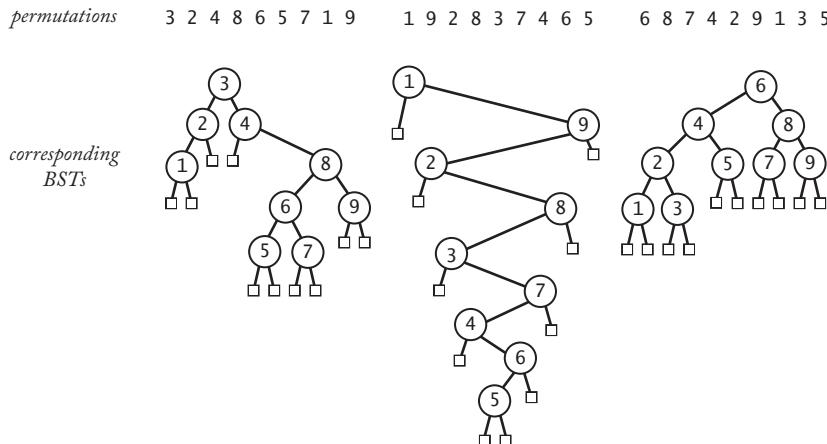


Figure 6.13 Permutations and binary search trees

tree. In the best case, the middle node will be at the root for every subtree, with about $N/2$ nodes in each subtree, so the standard divide-and-conquer recurrence $T_N = 2T_{N/2} + N$ holds, which implies that a total of about $N \lg N$ steps are required to construct the tree (see §2.6).

The cost of constructing a particular tree is directly proportional to its internal path length, since nodes are not moved once inserted, and the level of a node is exactly the number of compares required to insert it. Thus, the cost of constructing a tree is the same for each of the insertion sequences that could lead to its construction.

We could obtain the average construction cost by computing the sum of the internal path lengths of the trees resulting from all $N!$ permutations (the cumulated cost) and then dividing by $N!$. Or, we could compute the cumulated cost by adding, for all trees, the product of the internal path length

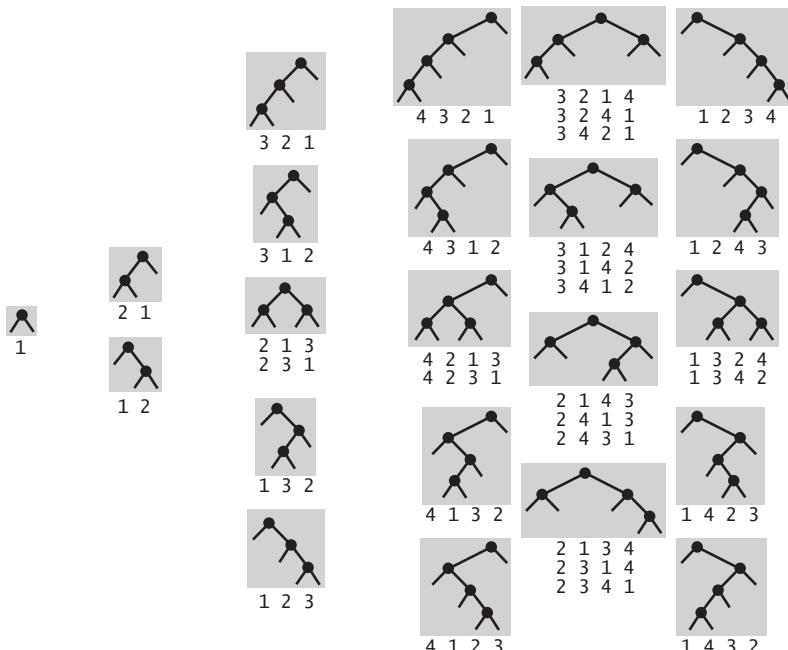


Figure 6.14 Permutations associated with N -node BSTs, $1 \leq N \leq 4$

and the number of permutations that lead to the tree being constructed. The result of this computation is the average internal path length that we expect after N random insertions into an initially empty binary search tree, but it is *not* the same as the average internal path length of a random binary tree, under the model where all trees are equally likely. Instead, it assumes that all *permutations* are equally likely.

The differences in the models are evident even for the 3-node and 4-node trees shown in Figure 6.14. There are five different 3-node trees, four with internal path length 3 and one with internal path length 2. Of the six permutations of size 3, four correspond to the trees with larger path length and two correspond to the balanced tree. Therefore, if Q_N is the average internal path length of a binary tree and C_N is the average internal path length of a binary search tree built from a random permutation, then

$$Q_3 = (3+3+2+3+3)/5 = 2.8 \quad \text{and} \quad C_3 = (3+3+2\cdot2+3+3)/6 \doteq 2.667.$$

For 4-node trees, the corresponding calculations are

$$Q_4 = (6+6+5+6+6+4+4+4+4+6+6+5+6+6)/14 \doteq 5.286$$

for random 4-node binary trees and

$$C_4 = (6+6+5\cdot2+6+6+4\cdot3+4\cdot3+4\cdot3+4\cdot3+6+6+5\cdot2+6+6)/24 \doteq 4.833$$

for binary search trees built from random permutations of size 4. In both cases, the average path length for the binary search trees is smaller because more permutations map to the balanced trees. This difference is fundamental. We will consider a full analysis for the “random tree” case in the next section and for the “binary search trees built from a random permutation” case in §6.8.

Exercise 6.25 Compute Q_5 and C_5 .

Exercise 6.26 Show that two different permutations cannot give the same degenerate tree structure. If all $N!$ permutations are equally likely, what is the probability that a degenerate tree structure will result?

Exercise 6.27 For $N = 2^n - 1$, what is the probability that a perfectly balanced tree structure (all 2^n external nodes on level n) will be built, if all $N!$ key insertion sequences are equally likely?

Exercise 6.28 Show that traversing a binary search tree in preorder and inserting the keys into an initially empty tree results in the original tree. Is the same true for postorder and/or level order? Prove your answer.

6.7 Average Path Length in Random Catalan Trees. To begin our analysis of tree parameters, we consider the model where each tree is equally likely to occur. To avoid confusion with other models, we add the modifier *Catalan* to refer to random trees under this assumption, since the probability that a particular tree occurs is the inverse of a Catalan number. This model is a reasonable starting point for many applications, and the combinatorial tools developed in Chapters 3 and 5 are directly applicable in the analysis.

Binary Catalan trees. What is the *average* (internal) path length of a binary tree with N internal nodes, if each N -node tree is considered to be equally likely? Our analysis of this important question is prototypical of the general approach to analyzing parameters of combinatorial structures that we considered in Chapters 3 and 5:

- Define a bivariate generating function (BGF), with one variable marking the size of the tree and the other marking the internal path length.
- Derive a functional equation satisfied by the BGF, or its associated cumulative generating function (CGF).
- Extract coefficients to derive the result.

We will start with a recurrence-based argument for the second step, because the underlying details are of interest and related to familiar problems. We know from Chapter 5 that direct generating-function-based arguments are available for such problems. We shall consider two such derivations in the next subsection.

To begin, we observe that the probability that the left subtree has k nodes (and the right subtree has $N - k - 1$ nodes) in a random binary Catalan tree with N nodes is $T_k T_{N-k-1} / T_N$ (where $T_N = \binom{2N}{N} / (N+1)$ is the N th Catalan number). The denominator is the number of possible N -node trees and the numerator counts the number of ways to make an N -node tree by using any tree with k nodes on the left and any tree with $N - k - 1$ nodes on the right. We refer to this probability distribution as the *Catalan distribution*.

Figure 6.14 shows the Catalan distribution as N grows. One of the striking facts about the distribution is that the probability that one of the subtrees is empty tends to a constant as N grows: it is $2T_{N-1}/T_N \sim 1/2$. Random binary trees are not particularly well balanced.

One approach to analyzing path length in a random binary tree is to use the Catalan distribution to write down a recurrence very much like the one that we have studied for quicksort: the average internal path length in a

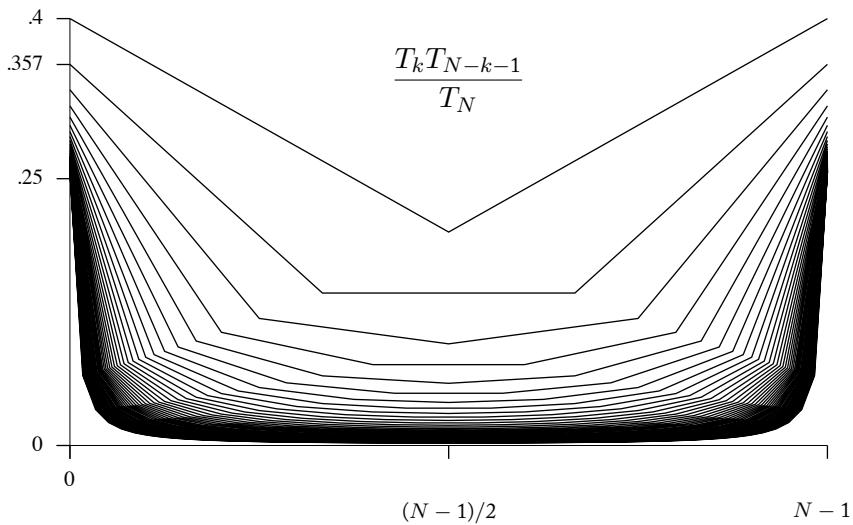


Figure 6.15 Catalan distribution (subtree sizes in random binary trees) (k -axes scaled to N)

random binary Catalan tree is described by the recurrence

$$Q_N = N - 1 + \sum_{1 \leq k \leq N} \frac{T_{k-1} T_{N-k}}{T_N} (Q_{k-1} + Q_{N-k}) \quad \text{for } N > 0$$

with $Q_0 = 0$. The argument underlying this recurrence is general, and can be used to analyze random binary tree structures under other models of randomness, by substituting other distributions for the Catalan distribution. For example, as discussed later, the analysis of binary search trees leads to the uniform distribution (each subtree size occurs with probability $1/N$) and the recurrence becomes like the quicksort recurrence of Chapter 1.

Theorem 6.3 (Path length in binary trees). The average internal path length in a random binary tree with N internal nodes is

$$\frac{(N+1)4^N}{\binom{2N}{N}} - 3N - 1 = N\sqrt{\pi N} - 3N + O(\sqrt{N}).$$

Proof. We develop a BGF as in §3.10. First, the probability generating function $Q_N(u) = \sum_{k \geq 0} q_{Nk} u^k$ with q_{Nk} the probability that k is the total internal path length satisfies the recurrence relation

$$Q_N(u) = u^{N-1} \sum_{1 \leq k \leq N} \frac{T_{k-1} T_{N-k}}{T_N} Q_{k-1}(u) Q_{N-k}(u) \quad \text{for } N > 0$$

with $Q_0(u) = 1$. To simplify this recurrence, we move to an enumerative approach, where we work with $p_{Nk} = T_N q_{Nk}$ (the number of trees of size N with internal path length k) instead of the probabilities. These satisfy, from the above recurrence,

$$\sum_{k \geq 0} p_{Nk} u^k = u^{N-1} \sum_{1 \leq k \leq N} \sum_{r \geq 0} p_{(k-1)r} u^r \sum_{s \geq 0} p_{(N-k)s} u^s \quad \text{for } N > 0.$$

To express this in terms of the bivariate generating function

$$P(z, u) = \sum_{N \geq 0} \sum_{k \geq 0} p_{Nk} z^N u^k,$$

we multiply the above by z^N and sum on N to get

$$\begin{aligned} P(z, u) &= \sum_{N \geq 1} \sum_{1 \leq k \leq N} \sum_{r \geq 0} p_{(k-1)r} u^r \sum_{s \geq 0} p_{(N-k)s} u^s z^N u^{N-1} + 1 \\ &= z \sum_{k \geq 0} \sum_{r \geq 0} p_{kr} (zu)^k u^r \sum_{N \geq k} \sum_{s \geq 0} p_{(N-k)s} (zu)^{N-k} u^s + 1 \\ &= z \sum_{k \geq 0} \sum_{r \geq 0} p_{kr} (zu)^k u^r \sum_{N \geq 0} \sum_{s \geq 0} p_{Ns} (zu)^N u^s + 1 \\ &= zP(zu, u)^2 + 1. \end{aligned}$$

Later, we will also see a simple direct argument for this equation. Now, we can use Theorem 3.11 to get the desired result: setting $u = 1$ gives the familiar functional equation for the generating function for the Catalan numbers, so $P(z, 1) = T(z) = (1 - \sqrt{1 - 4z})/(2z)$. The partial derivative $P_u(z, 1)$ is the generating function for the cumulative total if we add the internal path lengths of all binary trees. From Theorem 3.11, the average that we seek is $[z^N]P_u(z, 1)/[z^N]P(z, 1)$.

Differentiating both sides of the functional equation for the BGF with respect to u (using the chain rule for partial derivatives) gives

$$P_u(z, u) = 2zP(zu, u)(P_u(zu, u) + zP_z(zu, u)).$$

Evaluating this at $u = 1$ gives a functional equation for the CGF:

$$P_u(z, 1) = 2zT(z)(P_u(z, 1) + zT'(z)),$$

which yields the solution

$$P_u(z, 1) = \frac{2z^2T(z)T'(z)}{1 - 2zT(z)}.$$

Now, $T(z) = (1 - \sqrt{1 - 4z})/(2z)$, so $1 - 2zT(z) = \sqrt{1 - 4z}$ and $zT'(z) = -T(z) + 1/\sqrt{1 - 4z}$. Substituting these gives the explicit expression

$$zP_u(z, 1) = \frac{z}{1 - 4z} - \frac{1 - z}{\sqrt{1 - 4z}} + 1,$$

which expands to give the stated result. ■

This result is illustrated by the large random binary tree in Figure 6.10: asymptotically, a large tree roughly fits into a \sqrt{N} -by- \sqrt{N} square.

Direct combinatorial argument for the BGF. The proof of Theorem 6.3 involves the bivariate generating function

$$P(z, u) = \sum_{N \geq 0} \sum_{k \geq 0} p_{Nk} u^k z^N$$

where p_{Nk} is the number of trees with N nodes and internal path length k . As we know from Chapter 5, this may be expressed equivalently as

$$P(z, u) = \sum_{t \in \mathcal{T}} z^{|t|} u^{\text{ipl}(t)}.$$

Now the recursive definitions in §6.4 lead immediately to

$$P(z, u) = \sum_{t_l \in \mathcal{T}} \sum_{t_r \in \mathcal{T}} z^{|t_l| + |t_r| + 1} u^{\text{ipl}(t_l) + \text{ipl}(t_r) + |t_l| + |t_r|} + 1.$$

The number of nodes is 1 plus the number of nodes in the subtrees, and the internal path length is the sum of the internal path lengths of the subtrees plus 1 for each node in the subtrees. Now, it is easy to rearrange this double sum to make two independent sums:

$$\begin{aligned} P(z, u) &= z \sum_{t_l \in \mathcal{T}} (zu)^{|t_l|} u^{\text{ipl}(t_l)} \sum_{t_r \in \mathcal{T}} (zu)^{|t_r|} u^{\text{ipl}(t_r)} + 1 \\ &= zP(zu, u)^2 + 1, \end{aligned}$$

as before. The reader may wish to study this example carefully, to appreciate both its simplicity and its subtleties. It is also possible to directly derive equations of this form via the symbolic method (see [15]).

Cumulative generating function. An even simpler path to the same result is to derive the functional equation for the CGF directly. We define the CGF

$$C_T(z) \equiv P_u(z, 1) = \sum_{t \in \mathcal{T}} \text{ipl}(t) z^{|t|}.$$

The average path length is $[z^n]C_T(z)/[z^n]T(z)$. In precisely the same manner as above, the recursive definition of binary trees leads immediately to

$$\begin{aligned} C_T(z) &= \sum_{t_l \in \mathcal{T}} \sum_{t_r \in \mathcal{T}} (\text{ipl}(t_l) + \text{ipl}(t_r) + |t_l| + |t_r|) z^{|t_l|+|t_r|+1} \\ &= 2zC_T(z)T(z) + 2z^2T(z)T'(z), \end{aligned}$$

which is the same as the functional equation derived for Theorem 6.3.

Exercise 6.29 Derive this equation from the recurrence for path length.

The three derivations just considered are based on the same combinatorial decomposition of binary trees, but the CGF suppresses the most detail and is certainly the preferred method for finding the average. The contrast between the complex recurrence given in the proof to Theorem 6.3 and this “two-line” derivation of the same result given here is typical, and we will see many other problems throughout this book where the amount of detail suppressed using CGFs is considerable.

General Catalan trees. We can proceed in the same manner to find the expected path length in a random general tree via BGFs. Readers not yet convinced of the utility of BGFs and CGFs are invited to go through the exercise of deriving this result from a recurrence.

Theorem 6.4 (Path length in general trees). The average internal path length in a random general tree with N internal nodes is

$$\frac{N}{2} \left(\frac{4^{N-1}}{\binom{2N-2}{N-1}} - 1 \right) = \frac{N}{2} (\sqrt{\pi N} - 1) + O(\sqrt{N}).$$

Proof. Proceed as described earlier:

$$\begin{aligned} Q(z, u) &\equiv \sum_{t \in \mathcal{G}} z^{|t|} u^{\text{ipl}(t)} \\ &= \sum_{k \geq 0} \sum_{t_1 \in \mathcal{G}} \dots \sum_{t_k \in \mathcal{G}} u^{\text{ipl}(t_1) + \dots + \text{ipl}(t_k) + |t_1| + \dots + |t_k|} z^{|t_1| + \dots + |t_k| + 1} \\ &= z \sum_{k \geq 0} Q(zu, u)^k \\ &= \frac{z}{1 - Q(zu, u)}. \end{aligned}$$

Setting $u = 1$, we see that $Q(z, 1) = G(z) = zT(z) = (1 - \sqrt{1 - 4z})/2$ is the Catalan generating function, which enumerates general trees, as we found in §6.2. Differentiating the BGF derived above with respect to u and evaluating at $u = 1$ gives the CGF

$$C_G(z) \equiv Q_u(z, 1) = \frac{zC_G(z) + z^2G'(z)}{(1 - G(z))^2}.$$

This simplifies to give

$$C_G(z) = \frac{1}{2} \frac{z}{1 - 4z} - \frac{1}{2} \frac{z}{\sqrt{1 - 4z}}.$$

Next, as before, we use Theorem 3.11 and compute $[z^N]C_G(z)/[z^N]G(z)$, which immediately leads to the stated result. ■

Exercise 6.30 Justify directly the equation given in the proof of Theorem 6.4 for the CGF for path length in general trees (as we did for binary trees).

Exercise 6.31 Use the rotation correspondence between general trees and binary trees to derive the average path length in random general trees from the corresponding result on random binary trees.

6.8 Path Length in Binary Search Trees. As we have noted, the analysis of path length in binary search trees is actually the study of a property of *permutations*, not trees, since we start with a random permutation. In Chapter 7, we discuss properties of permutations as combinatorial objects in some detail. We consider the analysis of path length in BSTs here not only because it is interesting to compare it with the analysis just given for random trees, but also because we have already done all the work, in Chapters 1 and 3.

Figure 6.14 indicates—and the analysis proves—that the binary search tree insertion algorithm maps more permutations to the more balanced trees with small internal path length than to the less balanced trees with large internal path length. Binary search trees are widely used because they accommodate intermixed searches, insertions, and other operations in a uniform and flexible manner, and they are primarily useful because the search itself is efficient. In the analysis of the costs of any searching algorithm, there are two quantities of interest: the *construction cost* and the *search cost*, and, for the latter, it is normally appropriate to consider separately the cases where the search is successful or unsuccessful. In the case of binary search trees, these cost functions are closely related to path length.

Construction cost. We assume that a binary search tree is built by successive insertions, drawing from a random source of keys (for example, independent and uniformly distributed random numbers between 0 and 1). This implies that all $N!$ key orderings are equally likely, and is thus equivalent to assuming that the keys are a random permutation of the integers 1 to N . Now, observe that the trees are formed by a splitting process: the first key inserted becomes the node at the root, then the left and right subtrees are built independently. The probability that the k th smallest of the N keys is at the root is $1/N$ (independent of k), in which case subtrees of size $k - 1$ and $N - k$ are built on the left and right, respectively. The total cost of building the subtrees is one larger for each node (a total of $k - 1 + N - k = N - 1$) than if the subtree

were at the root, so we have the recurrence

$$C_N = N - 1 + \frac{1}{N} \sum_{1 \leq k \leq N} (C_{k-1} + C_{N-k}) \quad \text{for } N > 0 \text{ with } C_0 = 0.$$

Of course, as mentioned in §6.6, this recurrence also describes the average internal path length for binary search trees. This is also the recurrence solved in Chapter 1 for the number of comparisons taken by quicksort, except with $N - 1$ instead of $N + 1$.

Thus, we have already done the analysis of the cost of constructing a binary search tree, in §1.5 and in §3.10.

Theorem 6.5 (Construction cost of BSTs). The average number of comparisons involved in the process of constructing a binary search tree by inserting N distinct keys in random order into an initially empty tree (the average internal path length of a random binary search tree) is

$$2(N + 1)(H_{N+1} - 1) - 2N \approx 1.386N\lg N - 2.846N$$

with variance asymptotic to $(7 - 2\pi^2/3)N^2$.

Proof. From the earlier discussion, the solution for the average follows directly from the proof and discussion of Theorem 1.2.

The variance follows precisely as in the proof of Theorem 3.12. Define the BGF

$$Q(z, u) = \sum_{p \in \mathcal{P}} \frac{z^{|p|}}{|p|!} u^{\text{ipl}(p)}$$

where \mathcal{P} denotes the set of all permutations and $\text{ipl}(p)$ denotes the internal path length of the binary search tree constructed when the elements of p are inserted into an initially empty tree using the standard algorithm. By virtually the same computation as in §3.10, this BGF must satisfy the functional equation

$$\frac{\partial}{\partial z} Q(z, u) = Q^2(zu, u) \quad \text{with} \quad Q(0, u) = 1.$$

This equation differs from the corresponding equation for quicksort only in that it lacks a u^2 factor (which originates in the difference between $N + 1$ and $N - 1$ in the recurrences). To compute the variance, we proceed just as in §3.10, with exactly the same result (the u^2 factor does not contribute to the variance). ■

Thus, a “random” binary search tree (a tree built from a random permutation) costs only about 40% more than a perfectly balanced tree. Figure 6.16 shows a large random binary search tree, which is quite well balanced by comparison with the bottom tree in Figure 6.10, a “random” binary tree under the assumption that all trees are equally likely.

The relationship to the quicksort recurrence highlights a fundamental reason why trees are important to study in the analysis of algorithms: recursive programs involve implicit tree structures. For example, the tree on the left in Figure 6.12 can also be viewed as a precise description of the process of sorting the keys with Program 1.2: we view the key at the root as the partitioning element; the left subtree as a description of the sorting of the left subfile; and the right subtree as a description of the sorting of the right subfile. Binary trees could also be used to describe the operation of mergesort, and other types of trees are implicit in the operation of other recursive programs.

Exercise 6.32 For each of the trees in Figure 6.12, give permutations that would cause Program 1.2 to partition as described by the tree.

Search costs. A *successful search* is a search where a previously inserted key is found. We assume that each key in the tree is equally likely to be sought. An *unsuccessful search* is a search for a key that has *not* been previously inserted. That is, the key sought is not in the tree, so the search terminates at an external node. We assume that each external node is equally likely to be reached. For example, this is the case for each search in our model, where new keys are drawn from a random source.

We want to analyze the costs of searching in the tree, apart from its construction. This is important in applications because we normally expect a tree to be involved in a very large number of search operations, and the construction costs are small compared to the search costs for many applications. To do the analysis, we adopt the probabilistic model that the tree was built by

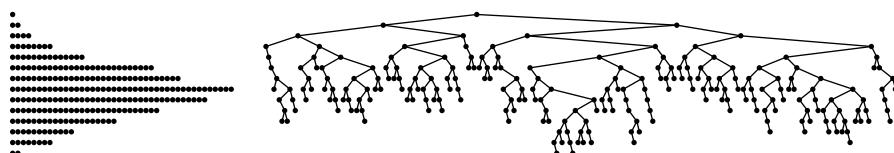


Figure 6.16 A binary search tree built from 237 randomly ordered keys

random insertions and that the searches are “random” in the tree, as described in the previous paragraph. Both costs are directly related to path length.

Theorem 6.6 (Search costs in BSTs). In a random binary search tree of N nodes, the average cost of a successful search is $2H_N - 3 - 2H_N/N$ and the average cost of an unsuccessful search is $2H_{N+1} - 2$. In both cases, the variance is $\sim 2H_N$.

Proof. The number of comparisons needed to find a key in the tree is exactly one greater than the number that was needed to insert it, since keys never move in the tree. Thus, the result for successful search is obtained by dividing the cost of constructing the tree (the internal path length, given in Theorem 6.5) by N and adding 1.

Since the level of an external node is precisely the cost of reaching it during an unsuccessful search, the average cost of an unsuccessful search is exactly the external path length divided by $N + 1$, so the stated result follows directly from the first lemma in §6.3 and Theorem 6.5.

The variances require a different calculation, discussed below. ■

Analysis with PGFs. The proof of Theorem 6.6 is a convenient application of previously derived results to give average costs; however, it does not give a way to calculate, for example, the standard deviation.

This is true because of differences in the probabilistic models. For internal path length (construction cost), there are $N!$ different possibilities to be accounted for, while for successful search cost, there are $N \cdot N!$ possibilities. Internal path length is a quantity that varies between $N \lg N$ and N^2 (roughly), while successful search cost varies between 1 and N . For a particular tree, we get the *average* successful search cost by dividing the internal path length by N , but characterizing the *distribution* of search costs is another matter. For example, the probability that the successful search cost is 1 is $1/N$, which is not at all related to the probability that the internal path length is N , which is 0 for $N > 1$.

Probability generating functions (or, equivalently in this case, the symbolic method) provide an alternative derivation for search costs and also can allow calculation of moments. For example, the PGF for the cost of an unsuccessful search satisfies

$$p_N(u) = \left(\frac{N-1}{N+1} + \frac{2u}{N+1} \right) p_{N-1}(u),$$

since the N th insertion contributes 1 to the cost of an unsuccessful search if the search terminates at one of its two external nodes, which happens with probability $2/(N + 1)$; or 0 otherwise. Differentiating and evaluating at 1 gives a simple recurrence for the average that telescopes directly to the result of Theorem 6.6, and the variance follows in a similar manner. These calculations are summarized in the exercises that follow.

Exercise 6.33 What is the probability that the successful search cost is 2?

Exercise 6.34 Construct a random 1000-node binary search tree by inserting 1000 random keys into an initially empty tree, then do 10,000 random searches in that tree and plot a histogram of the search costs, for comparison with Figure 1.4.

Exercise 6.35 Do the previous exercise, but generate a new tree for each trial.

Exercise 6.36 [Lynch, cf. Knuth] By calculating $p''_N(1) + p'_N(1) - p'(1)^2$. show that the variance of unsuccessful search cost is $2H_{N+1} - 4H_{N+1}^{(2)} + 2$.

Exercise 6.37 [Knott, cf. Knuth] Using a direct argument with PGFs, find the average and variance for the cost of a *successful* search.

Exercise 6.38 Express the PGF for successful search in terms of the PGF for unsuccessful search. Use this to express the average and variance for successful search in terms of the average and variance for unsuccessful search.

6.9 Additive Parameters of Random Trees. The CGF-based method that we used earlier to analyze path length in Catalan trees and binary search trees generalizes to cover a large class of parameters that are defined additively over subtrees. Specifically, define an *additive parameter* to be any parameter whose cost function satisfies the linear recursive schema

$$c(t) = e(t) + \sum_s c(s)$$

where the sum is over all the subtrees of the root of t . The function e is called the “toll,” the portion of the cost associated with the root. The following table gives examples of cost functions and associated tolls:

<i>toll function</i> $e(t)$	<i>cost function</i> $c(t)$
1	size $ t $
$ t - 1$	internal path length
$\delta_{ t 1}$	number of leaves

We normally take the toll function to be 0 for the empty binary tree.

It is possible to develop a fully general treatment of the average-case analysis of any additive parameter for both of the Catalan tree models and for the BST model. Indeed, this encompasses all the theorems about properties of trees that we have seen to this point.

Theorem 6.7 (Additive parameters in random trees). Let $C_T(z)$, $C_G(z)$, and $C_B(z)$ be the CGFs of an additive tree parameter $c(t)$ for the binary Catalan, general Catalan, and binary search tree models, respectively, and let $E_T(z)$, $E_G(z)$, and $E_B(z)$ be the CGFs for the associated toll function $e(t)$. (For the binary search tree case, use exponential CGFs.) These functions are related by the equations

$$C_T(z) = \frac{E_T(z)}{\sqrt{1 - 4z}} \quad (\text{binary Catalan trees})$$

$$C_G(z) = \frac{1}{2}E_G(z)\left(1 + \frac{1}{\sqrt{1 - 4z}}\right) \quad (\text{general Catalan trees})$$

$$C_B(z) = \frac{1}{(1 - z)^2}\left(E_B(0) + \int_0^z (1 - x)^2 E'_B(x) dx\right) \quad (\text{binary search trees}).$$

Proof. The proofs follow precisely the same lines as the arguments that we have given for path length.

First, let \mathcal{T} be the set of all binary Catalan trees. Then, just as in §6.6, we have

$$\begin{aligned} C_T(z) &\equiv \sum_{t \in \mathcal{T}} c(t)z^{|t|} \\ &= \sum_{t \in \mathcal{T}} e(t)z^{|t|} + \sum_{t_l \in \mathcal{T}} \sum_{t_r \in \mathcal{T}} (c(t_l) + c(t_r))z^{|t_l|+|t_r|+1} \\ &= E_T(z) + 2zT(z)C_T(z), \end{aligned}$$

where $T(z) = (1 - \sqrt{1 - 4z})/(2z)$ is the OGF for the Catalan numbers T_N . This leads directly to the stated result.

Next, for general Catalan trees, let \mathcal{G} be the set of trees. Again, just as in §6.6, we have

$$\begin{aligned} C_G(z) &\equiv \sum_{t \in \mathcal{G}} c(t) z^{|t|} \\ &= \sum_{t \in \mathcal{G}} e(t) z^{|t|} + \sum_{k \geq 0} \sum_{t_1 \in \mathcal{G}} \dots \sum_{t_k \in \mathcal{G}} (c(t_1) + \dots + c(t_k)) z^{|t_1|+...+|t_k|+1} \\ &= E_G(z) + z \sum_{k \geq 0} k C_G(z) G^{k-1}(z) \\ &= E_G(z) + \frac{z C_G(z)}{(1 - G(z))^2} \end{aligned}$$

where $G(z) = zT(z) = (1 - \sqrt{1 - 4z})/2$ OGF for the Catalan numbers T_{N-1} , enumerating general trees. Again, substituting this and simplifying leads directly to the stated result.

For binary search trees, we let c_N and e_N , respectively, denote the expected values of $c(t)$ and $e(t)$ over random BSTs of size N . Then the exponential CGFs $C(z)$ and $E(z)$ are the same as the OGFs for these sequences, and we follow the derivation in §3.3. We have the recurrence

$$c_N = e_N + \frac{2}{N} \sum_{1 \leq k \leq N} c_{k-1} \quad \text{for } N \geq 1 \text{ with } c_0 = e_0,$$

which leads to the differential equation

$$C'_B(z) = E'_B(z) + 2 \frac{C_B(z)}{1 - z} \quad \text{with } C_B(0) = E_B(0),$$

which can be solved precisely as in §3.3 to yield the stated solution. ■

Corollary The mean values of the additive parameters are given by

$$[z^N]C_T(z)/T_N \quad (\text{binary Catalan trees})$$

$$[z^N]C_G(z)/T_{N-1} \quad (\text{general Catalan trees})$$

$$[z^N]C_B(z) \quad (\text{binary search trees}).$$

Proof. These follow directly from the definitions and Theorem 3.11. ■

This vastly generalizes the counting and path length analyses that we have done and permits us to analyze many important parameters. The counting and path length results that we have derived in the theorems earlier in this chapter all follow from a simple application of this theorem. For example, to compute average path length in binary Catalan trees, we have

$$E_T(z) = 1 + \sum_{t \in \mathcal{T}} (|t| - 1)z^{|t|} = 1 + zT'(z) - T(z)$$

and therefore

$$C_T(z) = \frac{zT'(z) - T(z) + 1}{\sqrt{1 - 4z}},$$

which is equivalent to the expression derived in the proof of Theorem 6.3.

Leaves. As an example of the use of Theorem 6.7 for a new problem, we consider the analysis of the average number of leaves for each of the three models. This is representative of an important class of problems related to memory allocation for recursive structures. For example, in a binary tree, if space is at a premium, we might seek a representation that avoids the null pointers in leaves. How much space could be saved in this way? The answer to this question depends on the tree model: determining the average number of leaves is a straightforward application of Theorem 6.7, using $e(t) = \delta_{|t|=1}$ and therefore $E_T(z) = E_G(z) = E_B(z) = z$.

First, for binary Catalan trees, we have $C_T(z) = z/\sqrt{1 - 4z}$. This matches the result derived in §3.10.

Second, for general Catalan trees, we have

$$C_G(z) = \frac{z}{2} + \frac{z}{2\sqrt{1 - 4z}},$$

which leads to the result that the average is $N/2$ exactly for $N > 1$.

Third, for binary search trees, we get

$$C_B(z) = \frac{1}{3} \frac{1}{(1-z)^2} + \frac{1}{3}(z-1),$$

so the mean number of leaves is $(N+1)/3$ for $N > 1$.

Corollary For $N > 1$, the average number of leaves is given by

$$\frac{N(N+1)}{2(2N-1)} \sim \frac{N}{4} \text{ in a random binary Catalan tree with } N \text{ nodes,}$$

$$\frac{N}{2} \quad \text{in a random general Catalan tree with } N \text{ nodes, and}$$

$$\frac{N+1}{3} \quad \text{in a binary search tree built from } N \text{ random keys.}$$

Proof. See the discussion provided earlier. ■

The techniques that we have been considering are clearly quite useful in analyzing algorithms involving trees, and they apply in some other situations, as well. For example, in Chapter 7 we analyze properties of permutations via a correspondence with trees (see §7.5).

Exercise 6.39 Find the average number of children of the root in a random Catalan tree of N nodes. (From Figure 6.3, the answer is 2 for $N = 5$.)

Exercise 6.40 In a random Catalan tree of N nodes, find the proportion of nodes with one child.

Exercise 6.41 In a random Catalan tree of N nodes, find the proportion of nodes with k children for $k = 2, 3$, and higher.

Exercise 6.42 Internal nodes in binary trees fall into one of three classes: they have either two, one, or zero external children. What fraction of the nodes are of each type, in a random binary Catalan tree of N nodes?

Exercise 6.43 Answer the previous question for random binary search trees.

Exercise 6.44 Set up BGFs for the number of leaves and estimate the variance for each of the three random tree models.

Exercise 6.45 Prove relationships analogous to those in Theorem 6.7 for BGFs.

6.10 Height. What is the average height of a tree? Path length analysis (using the second lemma in §6.4) suggests lower and upper bounds of order $N^{1/2}$ and $N^{3/4}$ for Catalan trees (either binary or general) and of order $\log N$ and $\sqrt{N \log N}$ for binary search trees. Developing more precise estimates for the average height turns out to be a more difficult question to answer, even though the recursive definition of height is as simple as the recursive definition of path length. The height of a tree is 1 plus the maximum of the heights of the subtrees; the path length of a tree is 1 plus the sum of the path lengths of the subtrees plus the number of nodes in the subtrees. As we have seen, the latter decomposition can correspond to “constructing” trees from subtrees, and additivity is mirrored in the analysis (by the linearity of the cost GF equations). No such treatment applies to the operation of taking the maximum over subtrees.

Generating functions for binary Catalan trees. We begin with the problem of finding the height of a binary Catalan tree. Attempting to proceed as for path length, we start with the bivariate generating function

$$P(z, u) = \sum_{N \geq 0} \sum_{h \geq 0} P_{Nh} z^N u^h = \sum_{t \in \mathcal{T}} z^{|t|} u^{h(t)}.$$

Now the recursive definition of height leads to

$$P(z, u) = \sum_{t_l \in \mathcal{T}} \sum_{t_r \in \mathcal{T}} z^{|t_l| + |t_r| + 1} u^{\max(h(t_l), h(t_r))}.$$

For path length, we were able to rearrange this into independent sums, but the “max” precludes this.

In contrast, using the “vertical” formulation for bivariate sequences that is described in §3.10, we can derive a simple functional equation. Let \mathcal{T}_h be the class of binary Catalan trees of height no greater than h , and

$$T^{[h]}(z) = \sum_{t \in \mathcal{T}_h} z^{|t|}.$$

Proceeding in precisely the same manner as for enumeration gives a simple functional equation for $T^{[h]}(z)$: any tree with height no greater than $h + 1$ is

either empty or a root node and two subtrees with height no greater than h , so

$$\begin{aligned} T^{[h+1]}(z) &= 1 + \sum_{t_L \in \mathcal{T}_h} \sum_{t_R \in \mathcal{T}_h} z^{|t_L|+|t_R|+1} \\ &= 1 + zT^{[h]}(z)^2. \end{aligned}$$

This result is also available via the symbolic method: it corresponds to the symbolic equation

$$\mathcal{T}_{h+1} = \square + \bullet \times \mathcal{T}_h \times \mathcal{T}_h.$$

Iterating this recurrence, we have

$$\begin{aligned} T^{[0]}(z) &= 1 \\ T^{[1]}(z) &= 1 + z \\ T^{[2]}(z) &= 1 + z + 2z^2 + z^3 \\ T^{[3]}(z) &= 1 + z + 2z^2 + 5z^3 + 6z^4 + 6z^5 + 4z^6 + z^7 \\ &\vdots \\ T^{[\infty]}(z) &= 1 + z + 2z^2 + 5z^3 + 14z^4 + 42z^5 + 132z^6 + \dots = T(z). \end{aligned}$$

The reader may find it instructive to check these against the initial values for the small trees given in Figure 5.2. Next, the corollary to Theorem 3.11 tells us that the cumulated cost (the sum of the heights of all trees of N nodes) is given by

$$[z^N] \sum_{h \geq 0} (T(z) - T^{[h]}(z)).$$

But now our analytic task is much harder. Rather than estimating coefficients in an expansion on one function for which we have a defining functional equation, we need to estimate coefficients in an entire series of expansions of functions defined by interrelated functional equations. This turns out to be an extremely challenging task for this particular problem.

Theorem 6.8 (Binary tree height). The average height of a random binary Catalan tree with N nodes is $2\sqrt{\pi N} + O(N^{1/4+\epsilon})$ for any $\epsilon > 0$.

Proof. Omitted, though see the comments above. Details may be found in Flajolet and Odlyzko [12]. ■

Average height of Catalan trees. For general Catalan trees, the problem of determining the average height is still considerably more difficult than analyzing path length, but we can sketch the solution. (*Warning:* This “sketch” involves a combination of many of the advanced techniques from Chapters 2 through 5, and should be approached with caution by novice readers.)

First, we construct \mathcal{G}_{h+1} , the set of trees of height $\leq h + 1$, by

$$\mathcal{G}_{h+1} = \{\bullet\} \times (\epsilon + \mathcal{G}_h + (\mathcal{G}_h \times \mathcal{G}_h) + (\mathcal{G}_h \times \mathcal{G}_h \times \mathcal{G}_h) + (\mathcal{G}_h \times \mathcal{G}_h \times \mathcal{G}_h \times \mathcal{G}_h) + \dots),$$

which translates by the symbolic method to

$$G^{[h+1]}(z) = z(1 + G^{[h]}(z) + G^{[h]}(z)^2 + G^{[h]}(z)^3 + \dots) = \frac{z}{1 - G^{[h]}(z)}.$$

Iterating this recurrence, we see that

$$G^{[0]}(z) = z$$

$$G^{[1]}(z) = z \frac{1}{1-z}$$

$$G^{[2]}(z) = \frac{z}{1 - \frac{z}{1-z}} = z \frac{1-z}{1-2z}$$

$$G^{[3]}(z) = \frac{z}{1 - \frac{z}{1 - \frac{z}{1-z}}} = z \frac{1-2z}{1-3z+z^2}$$

⋮

$$G^{[\infty]}(z) = z + z^2 + 2z^3 + 5z^4 + 14z^5 + 42z^6 + 132z^7 + \dots = zT(z)$$

These are rational functions with enough algebraic structure that we can derive exact enumerations for the height and obtain asymptotic estimates.

Theorem 6.9 (Catalan tree height GF). The number of Catalan trees with $N + 1$ nodes and height greater than or equal to $h - 1$ is

$$G_{N+1} - G_{N+1}^{[h-2]} = \sum_{k \geq 1} \left(\binom{2N}{N+1-kh} - 2 \binom{2N}{N-kh} + \binom{2N}{N-1-kh} \right).$$

Proof. From the basic recurrence and initial values given previously, it follows that $G^{[h]}(z)$ can be expressed in the form $G^{[h]}(z) = zF_{h+1}(z)/F_{h+2}(z)$, where $F_h(z)$ is a family of polynomials

$$\begin{aligned} F_0(z) &= 0 \\ F_1(z) &= 1 \\ F_2(z) &= 1 \\ F_3(z) &= 1 - z \\ F_4(z) &= 1 - 2z \\ F_5(z) &= 1 - 3z + z^2 \\ F_6(z) &= 1 - 4z + 3z^2 \\ F_7(z) &= 1 - 5z + 6z^2 - z^3 \\ &\vdots \end{aligned}$$

that satisfy the recurrence

$$F_{h+2}(z) = F_{h+1}(z) - zF_h(z) \quad \text{for } h \geq 0 \text{ with } F_0(z) = 0 \text{ and } F_1(z) = 1.$$

These functions are sometimes called *Fibonacci polynomials*, because they generalize the Fibonacci numbers, to which they reduce when $z = -1$.

When z is kept fixed, the Fibonacci polynomial recurrence is simply a linear recurrence with constant coefficients (see §2.4). Thus its solutions are expressible in terms of the solutions

$$\beta = \frac{1 + \sqrt{1 - 4z}}{2} \quad \text{and} \quad \hat{\beta} = \frac{1 - \sqrt{1 - 4z}}{2}$$

of the characteristic equation $y^2 - y + z = 0$. Solving precisely as we did for the Fibonacci numbers in §2.4, we find that

$$F_h(z) = \frac{\beta^h - \hat{\beta}^h}{\beta - \hat{\beta}} \quad \text{and therefore} \quad G^{[h]}(z) = z \frac{\beta^{h+1} - \hat{\beta}^{h+1}}{\beta^{h+2} - \hat{\beta}^{h+2}}.$$

Notice that the roots are closely related to the Catalan GF:

$$\hat{\beta} = G(z) = zT(z) \quad \text{and} \quad \beta = z/\hat{\beta} = z/G(z) = 1/T(z)$$

and that we have the identities $z = \beta(1 - \beta) = \widehat{\beta}(1 - \widehat{\beta})$.

In summary, the GF for trees of bounded height satisfies the formula

$$G^{[h]}(z) = 2z \frac{(1 + \sqrt{1 - 4z})^{h+1} - (1 - \sqrt{1 - 4z})^{h+1}}{(1 + \sqrt{1 - 4z})^{h+2} - (1 - \sqrt{1 - 4z})^{h+2}},$$

and a little algebra shows that

$$G(z) - G^{[h]}(z) = \sqrt{1 - 4z} \frac{u^{h+2}}{1 - u^{h+2}}$$

where $u \equiv \widehat{\beta}/\beta = G^2(z)/z$. This is a function of $G(z)$, which is implicitly defined by $z = G(z)(1 - G(z))$, so the Lagrange inversion theorem (see §6.12) applies, leading (after some calculation) to the stated result for $[z^{N+1}](G(z) - G^{[h-2]}(z))$. ■

Corollary The average height of a random Catalan tree with N nodes is $\sqrt{\pi N} + O(1)$.

Proof Sketch. By the corollary to Theorem 3.11, the average height is given by

$$\sum_{h \geq 1} \frac{[z^N](G(z) - G^{[h-1]}(z))}{G_N}.$$

For Theorem 6.9, this reduces to three sums that are very much like Catalan sums, and can be treated in a manner similar to the proof of Theorem 4.9. From asymptotic results on the tails of the binomial coefficients (the corollary to Theorem 4.6), the terms are exponentially small for large h . We have

$$[z^N](G(z) - G^{[h-1]}(z)) = O(N 4^N e^{-(\log 2 N)})$$

for $h > \sqrt{N} \log N$ by applying tail bounds to each term in the binomial sum in Theorem 6.9. This already shows that the expected height is itself $O(N^{1/2} \log N)$.

For smaller values of h , the normal approximation of Theorem 4.6 applies nicely. Using the approximation termwise as we did in the proof of Theorem 4.9, it is possible to show that

$$\frac{[z^N](G(z) - G^{[h-1]}(z))}{G_N} \sim H(h/\sqrt{N})$$

where

$$H(x) \equiv \sum_{k \geq 1} (4k^2 x^2 - 2) e^{-k^2 x^2}.$$

Like the trie sum diagrammed in Figure 4.7, the function $H(h/\sqrt{N})$ is close to 1 when h is small and close to 0 when h is large, with a transition from 1 to 0 when h is close to \sqrt{N} . Then, the expected height is approximately

$$\sum_{h \geq 1} H(h/\sqrt{N}) \sim \sqrt{N} \int_0^\infty H(x) dx \sim \sqrt{\pi N}$$

by Euler-Maclaurin summation and by explicit evaluation of the integral.

In the last few steps, we have ignored the error terms, which must be kept suitably uniform. As usual for such problems, this is not difficult because the tails are exponentially small, but we leave the details for the exercises below. Full details for a related but different approach to proving this result are given in De Bruijn, Knuth, and Rice [8]. ■

The analyses of tree height in binary trees and binary Catalan trees are the hardest nuts that we are cracking in this book. While we recognize that many readers may not be expected to follow a proof of this scope and complexity without very careful study, we have sketched the derivation in some detail because height analysis is extremely important to understanding basic properties of trees. Still, this sketch allows us to appreciate (i) that analyzing tree height is not an easy task, but (ii) that it is *possible* to do so, using the basic techniques that we have covered in Chapters 2 through 5.

Exercise 6.46 Prove that $F_{h+1}(z) = \sum_j \binom{h-j}{j} (-z)^j$.

Exercise 6.47 Show the details of the expansion of $G(z) - G^{[h-2]}(z)$ with the Lagrange inversion theorem (see §6.12).

Exercise 6.48 Provide a detailed proof of the corollary, including proper attention to the error terms.

Exercise 6.49 Draw a plot of the function $H(x)$.

Height of binary search trees. For binary search trees built from random permutations, the problem of finding the average height is also quite difficult. Since the average path length is $O(N \log N)$, we would expect the average height of a binary search tree to be $\sim c \log N$, for some constant c ; this is in fact the case.

Theorem 6.10 (Binary search tree height). The expected height of a binary search tree built from N random keys is $\sim c \log N$, where $c \approx 4.31107\dots$ is the solution $c > 2$ of $c \ln(2e/c) = 1$.

Proof. Omitted; see Devroye [9] or Mahmoud [27]. ■

Though the complete analysis is at least as daunting as the Catalan tree height analysis provided earlier, it is easy to derive functional relationships among the generating functions. Let $q_N^{[h]}$ be the probability that a BST built with N random keys has height no greater than h . Then, using the usual splitting argument, and noting that the subtrees have height no greater than $h - 1$, we have the recurrence

$$q_N^{[h]} = \frac{1}{N} \sum_{1 \leq k \leq N} q_{k-1}^{[h-1]} q_{N-1-k}^{[h-1]},$$

which leads immediately to the schema

$$\frac{d}{dz} q^{[h]}(z) = (q^{[h-1]}(z))^2.$$

Stack height. Tree height appears frequently in the analysis of algorithms. Fundamentally, it measures not only the size of the stack needed to traverse a tree, but also the space used when a recursive program is executed. For example, in the expression evaluation algorithm discussed earlier, the tree height $\eta(t)$ measures the maximum depth reached by the recursive stack when the expression represented by t is evaluated. Similarly, the height of a binary search tree measures the maximum stack depth reached by a recursive inorder traversal to sort the keys, or the implicit stack depth used when a recursive quicksort implementation is used.

Tree traversal and other recursive algorithms also can be implemented without using recursion by directly maintaining a pushdown stack (last-in-first-out data structure). When there is more than one subtree to visit, we

save all but one on the stack; when there are no subtrees to visit, we pop the stack to get a tree to visit. This uses fewer stack entries than are required in a stack supporting a recursive implementation, because nothing is put on the stack if there is only one subtree to visit. (A technique called *end recursion removal* is sometimes used to get equivalent performance for recursive implementations.) The maximum stack size needed when a tree is traversed using this method is a tree parameter called the *stack height*, which is similar to height. It can be defined by the recursive formula:

$$s(t) = \begin{cases} 0, & \text{if } t \text{ is an external node;} \\ s(t_l), & \text{if } t_r \text{ is an external node;} \\ s(t_r), & \text{if } t_l \text{ is an external node;} \\ 1 + \max(s(t_l), s(t_r)) & \text{otherwise.} \end{cases}$$

Because of the rotation correspondence, it turns out that the stack height of binary Catalan trees is essentially distributed like the height of general Catalan trees. Thus, the average stack height for binary Catalan trees is also studied by De Bruijn, Knuth, and Rice [8], and shown to be $\sim \sqrt{\pi N}$.

Exercise 6.50 Find a relationship between the stack height of a binary tree and the height of the corresponding forest.

Register allocation. When the tree represents an arithmetic expression, the minimum number of registers needed to evaluate the expression can be described by the following recursive formula:

$$r(t) = \begin{cases} 0, & \text{if } t \text{ is an external node;} \\ r(t_l), & \text{if } t_r \text{ is an external node;} \\ r(t_r), & \text{if } t_l \text{ is an external node;} \\ 1 + r(t_l), & \text{if } r(t_l) = r(t_r); \\ \max(r(t_l), r(t_r)) & \text{otherwise.} \end{cases}$$

This quantity was studied by Flajolet, Raoult, and Vuillemin [14] and by Kemp [23]. Though this recurrence seems quite similar to the corresponding recurrences for height and stack height, the average value is *not* $O(\sqrt{N})$ in this case, but rather $\sim (\lg N)/2$.

6.11 Summary of Average-Case Results on Properties of Trees. We have discussed three different tree structures (binary trees, trees, and binary search trees) and two basic parameters (path length and height), giving a total of six theorems describing the average values of these parameters in these structures. Each of these results is fundamental, and it is worthwhile to consider them in concert with one another.

As indicated in §6.9, the basic analytic methodology for these parameters extends to cover a wide variety of properties of trees, and we can place new problems in proper context by examining relationships among these parameters and tree models. At the same time, we briefly sketch the history of these results, which are summarized in Tables 6.1 and 6.2.

For brevity in this section, we refer to binary Catalan trees simply as “binary trees,” Catalan trees as “trees,” and binary search trees as “BSTs,” recognizing that a prime objective in the long series of analyses that we have discussed has been to justify these distinctions in terminology and quantify differences in the associated models of randomness.

Figures 6.10 and 6.16 show a random forest (random tree with its root removed), binary tree, and binary search tree, respectively. These reinforce the analytic information given in Tables 6.1 and 6.2: heights for binary trees and trees are similar (and proportional to \sqrt{N}), with trees about half as high as binary trees; and paths in binary search trees are much shorter (proportional to $\log N$). The probability distribution imposed on binary search tree structures is biased toward trees with short paths.

	functional equation on GF	asymptotic estimate of $[z^N]$
tree	$Q(z, u) = \frac{z}{1 - Q(zu, u)}$	$\frac{N}{2}\sqrt{\pi N} - \frac{N}{2} + O(\sqrt{N})$
binary tree	$Q(z, u) = zQ(zu, u)^2 + 1$	$N\sqrt{\pi N} - 3N + O(\sqrt{N})$
BST	$\frac{\partial}{\partial z}Q(z, u) = Q(zu, z)^2$	$2N\ln N + (2\gamma - 4)N + O(\log N)$

Table 6.1 Expected path length of trees

Perhaps the easiest problem on the list is the analysis of path length in binary search trees. This is available with elementary methods, and dates back at least to the invention of quicksort in 1960 [22]. The variance for tree construction costs (the same as the variance for quicksort) was evidently first published by Knuth [25]; Knuth indicates that recurrence relations describing the variance and results about search costs were known in the 1960s. By contrast, the analysis of the average height of binary search trees is a quite challenging problem, and was the last problem on the list to be completed, by Devroye in 1986 [9][10]. Path length in random trees and random binary trees is also not difficult to analyze, though it is best approached with generating-function-based or symbolic combinatorial tools. With such an approach, analysis of this parameter (and other additive parameters) is not much more difficult than counting.

The central role of tree height in the analysis of computer programs based on trees and recursive programs was clear as such programs came into widespread use, but it was equally clear that the analysis of nonadditive parameters in trees such as height can present significant technical challenges. The analysis of the height of trees (and stack height for binary trees)—published in 1972 by De Bruijn, Knuth, and Rice [8]—showed that such challenges could be overcome, with known analytic techniques, as we have sketched in §6.10. Still, developing new results along these lines can be a daunting task, even for experts. For example, the analysis of height of binary trees was not completed until 1982, by Flajolet and Odlyzko [12].

	functional equation on GF	asymptotic estimate of mean
tree	$q^{[h+1]}(z) = \frac{z}{1 - q^{[h]}(z)}$	$\sqrt{\pi N} + O(1)$
binary tree	$q^{[h+1]}(z) = z(q^{[h]}(z))^2 + 1$	$2\sqrt{\pi N} + O(N^{1/4+\epsilon})$
BST	$\frac{d}{dz}q^{[h+1]}(z) = (q^{[h]}(z))^2$	$(4.3110 \dots) \ln N + o(\log N)$

Table 6.2 Expected height of trees

Path length and height in random trees are worthy of careful study because they illustrate the power of generating functions, and the contrasting styles in analysis that are appropriate for “additive” and “nonadditive” parameters in recursive structures. As we saw in §6.3, trees relate directly to a number of classical problems in probability and combinatorics, so some of the problems that we consider have a distinguished heritage, tracing back a century or two. But the motivation for developing precise asymptotic results for path length and height as we have been doing certainly can be attributed to the importance of trees in the analysis of algorithms (see Knuth [8][24][25]).

6.12 Lagrange Inversion. Next, we turn to the study of other types of trees, using analytic combinatorics. The symbolic method often leaves us with the need to extract coefficients from generating functions that are implicitly defined through functional equations. The following transfer theorem is available for this task, and is of particular importance for tree enumeration.

Theorem 6.11 (Lagrange inversion theorem). Suppose that a generating function $A(z) = \sum_{n \geq 0} a_n z^n$ satisfies the functional equation $z = f(A(z))$, where $f(z)$ satisfies $f(0) = 0$ and $f'(0) \neq 0$. Then

$$a_n \equiv [z^n]A(z) = \frac{1}{n}[u^{n-1}]\left(\frac{u}{f(u)}\right)^n.$$

Also,

$$[z^n](A(z))^m = \frac{m}{n}[u^{n-m}]\left(\frac{u}{f(u)}\right)^n$$

and

$$[z^n]g(A(z)) = \frac{1}{n}[u^{n-1}]g'(u)\left(\frac{u}{f(u)}\right)^n.$$

Proof. Omitted; see, for example, [6]. There is a vast literature on this formula, dating back to the 18th century. ■

The *functional inverse* of a function f is the function f^{-1} that satisfies

$$f^{-1}(f(z)) = f(f^{-1}(z)) = z.$$

Applying f^{-1} to both sides of the equation $z = f(A(z))$, we see that the function $A(z)$ is the functional inverse of $f(z)$. The Lagrange theorem is a

general tool for inverting power series, in this sense. Its surprising feature is to provide a direct relation between the coefficients of the functional inverse of a function and the powers of that function. In the present context, Lagrange inversion is a very useful tool for extracting coefficients for implicit GFs. Below we show how it applies to binary trees and then give two examples that emphasize the formal manipulations and motivate the utility of the theorem, which will prepare us for the study of many other types of trees.

Binary trees. Let $T^{[2]}(z) = zT(z)$ be the OGF for binary trees, counted by external nodes. Rewriting the functional equation $T^{[2]}(z) = z + T^{[2]}(z)^2$ as

$$z = T^{[2]}(z) - T^{[2]}(z)^2,$$

we can apply Lagrange inversion with $f(u) = u - u^2$. This gives the result

$$[z^n]T^{[2]}(z) = \frac{1}{n}[u^{n-1}]\left(\frac{u}{u-u^2}\right)^n = \frac{1}{n}[u^{n-1}]\left(\frac{1}{1-u}\right)^n.$$

Now, from Table 3.1, we know that

$$\frac{u^{n-1}}{(1-u)^n} = \sum_{k \geq n-1} \binom{k}{n-1} u^k$$

so that, considering the term $k = 2n-2$,

$$[u^{n-1}]\left(\frac{1}{1-u}\right)^n = \binom{2n-2}{n-1}$$

which leads to the Catalan numbers, as expected.

Ternary trees. One way to generalize binary trees is to consider *ternary trees* where every node is either external or has three subtrees (left, middle, and right). Note that the number of external nodes in a ternary tree is odd. The sequence of counts for ternary trees with n external nodes for $n = 1, 2, 3, 4, 5, 6, 7, \dots$ is $1, 0, 1, 0, 3, 0, 3, \dots$. The symbolic method immediately gives the GF equation

$$z = T^{[3]}(z) - T^{[3]}(z)^3.$$

Proceeding as in §3.8 for binary trees does not succeed easily because this is a cubic equation, not a quadratic. But applying Lagrange inversion with $f(u) = u - u^3$ immediately gives the result

$$[z^n]T^{[3]}(z) = \frac{1}{n}[u^{n-1}]\left(\frac{1}{1-u^2}\right)^n.$$

Proceeding in the same manner as used earlier, we know from Table 3.1 that

$$\frac{u^{2n-2}}{(1-u^2)^n} = \sum_{k \geq n-1} \binom{k}{n-1} u^{2k}.$$

Considering the term $2k = 3n - 3$ (which only exists when n is odd) gives

$$[z^n]T^{[3]}(z) = \frac{1}{n} \binom{(3n-3)/2}{n-1}$$

for n odd and 0 for n even, which is OEIS [A001764](#) [34] alternating with 0s.

Forests of binary trees. Another way to generalize binary trees is to consider sets of them, or so-called forests. A k -forest of binary trees is simply an ordered sequence of k binary trees. By Theorem 5.1, the OGF for k -forests is just $(zT(z))^k$, where $T(z)$ is the OGF for binary trees, and by Lagrange inversion (using the second case in Theorem 6.11), the number of k -forests of binary trees with n external nodes is therefore

$$[z^n] \left(\frac{1 - \sqrt{1 - 4z}}{2} \right)^k = \frac{k}{n} \binom{2n-k-1}{n-1}.$$

These numbers are also known as the *ballot numbers* (see Chapter 8).

Exercise 6.51 Find $[z^n]A(z)$ when $A(z)$ is defined by $z = A(z)/(1 - A(z))$.

Exercise 6.52 What is the functional inverse of $e^z - 1$? What do we get in terms of power series by applying Lagrange inversion?

Exercise 6.53 Find the number of n -node 3-forests of ternary trees.

Exercise 6.54 Find the number of 4-ary trees, where every node either is external or has a sequence of four subtrees.

6.13 Rooted Unordered Trees. An essential aspect of the definition of trees and forests given previously is the notion of a *sequence* of trees: the order in which individual trees appear is considered significant. Indeed, the trees that we have been considering are also called *ordered* trees. This is natural when we consider various computer representations or, for example, when we draw a tree on paper, because we must somehow put down one tree after another. Forests with trees in differing orders look different and they are typically processed differently by computer programs. In some applications, however, the sequence is actually irrelevant. We will see examples of such algorithms as we consider the basic definitions, then we will consider enumeration problems for unordered trees.

Definition An *unordered tree* is a node (called the root) attached to a multiset of unordered trees. (Such a multiset is called an *unordered forest*.)

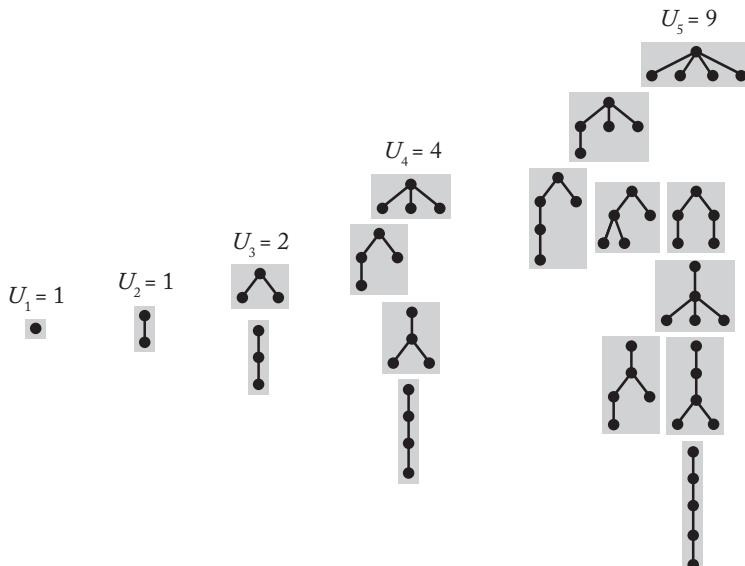


Figure 6.17 Rooted unordered trees with N nodes, $1 \leq N \leq 5$

Figure 6.17 shows the small rooted unordered trees, derived from Figure 6.4 by deleting every tree that can be transformed to a tree to its left by interchanging the order of subtrees at any node.

Sample application. As an example of an algorithm where rooted unordered trees are an appropriate underlying data structure, we consider the *union–find* problem: The goal is to process a sequence of “union–find” operations on pairs of N distinct items. Each operation combines a “find” operation that returns T if the two items are equivalent or F if they are not equivalent with a “union” operation that makes the two items equivalent by taking the union of their equivalence classes. A familiar application is a social network, where a new link between two friends merges their sets of friends. For example, given the 16 items (again, we use two initials for brevity) MS, JL, HT, JG, JB, GC, PL, PD, MC, AB, AA, HF, EF, CN, AL, and JC, the sequence of operations

$$\begin{array}{llllll} \text{MS} \equiv \text{JL} & \text{MS} \equiv \text{HT} & \text{JL} \equiv \text{HT} & \text{AL} \equiv \text{EF} & \text{AB} \equiv \text{MC} & \text{JB} \equiv \text{JG} \\ \text{AL} \equiv \text{CN} & \text{PL} \equiv \text{MS} & \text{JB} \equiv \text{GC} & \text{JL} \equiv \text{JG} & \text{AL} \equiv \text{JC} & \text{GC} \equiv \text{MS} \end{array}$$

should result in the sequence of return values

$$\text{F} \quad \text{F} \quad \text{T} \quad \text{F} \quad \text{T}$$

because the first two instructions make MS, JL, and HT equivalent, then the third finds JL and HT to be already equivalent, and so on.

```
public boolean unionfind(int p, int q)
{
    int i = p; while (id[i] != i) i = id[i];
    int j = q; while (id[j] != j) j = id[j];
    if (i == j) return true;
    id[i] = j; // Union operation
    return false;
}
```

Program 6.4 Union-find

Program 6.4 gives a solution to this problem. As it stands, the code is opaque, but it is easy to understand in terms of an explicit *parent link* representation of unordered forests, where each tree in the forest represents an equivalence class. First, we use a symbol table to associate each item with an integer between 0 and $N - 1$. Then we represent the forest as an item-indexed array: the entry corresponding to each node is the index of its parent in the tree containing them, where a root has its own index. The algorithm uses the roots to determine whether or not two items are equivalent.

Given the index corresponding to an item, the `unionfind` method in Program 6.4 finds its corresponding root by following parent links until it reaches a root. Accordingly, `unionfind` starts by finding the roots corresponding to the two given items. If both items correspond to the same root, then they belong to the same equivalence class; otherwise, the relation connects heretofore disconnected components. The forest depicted in Figure 6.18 is the one built for the sequence of operations in the example given earlier. The shape of the forest depends on the relations seen so far and the order in which they are presented.

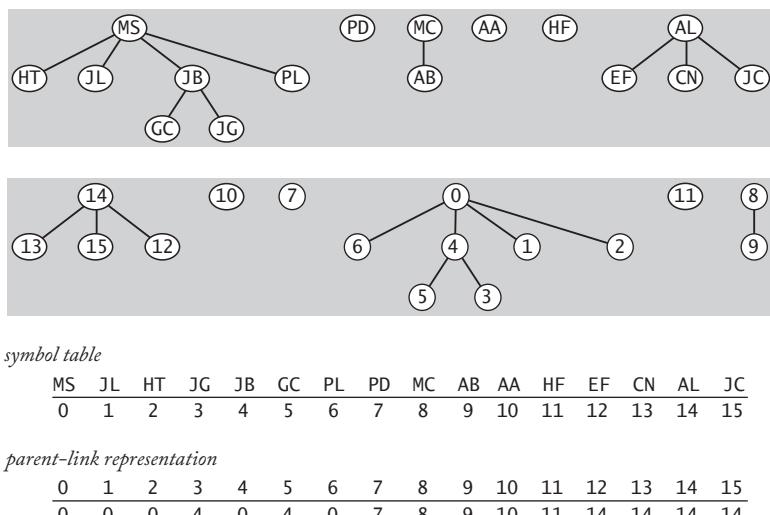


Figure 6.18 Representations of a rooted (unordered) forest

The algorithm moves up through the tree and never examines the subtrees of a node (or even tests how many there are). Combinatorially, the union-find algorithm is a mapping from permutations of relations to unordered forests. Program 6.4 is quite simple, and a variety of improvements to the basic idea have been suggested and analyzed. The key point to note in the present context is that the order of appearance of children of a node is not significant to the algorithm, or to the internal representation of the associated tree—this algorithm provides an example of unordered trees naturally occurring in a computation.

Unrooted (free) trees. Still more general is the concept of a tree where no root node is distinguished. Figure 6.18 depicts all such trees with less than 7 nodes. To properly define “unrooted, unordered trees,” or “free trees,” or just “trees,” it is convenient to move from the general to the specific, starting with *graphs*, the fundamental structure underlying all combinatorial objects based on sets of nodes and connections between them.

Definition A *graph* is a set of nodes together with a set of edges that connect pairs of distinct nodes (with at most one edge connecting any pair of nodes).

We can envision starting at some node and “following” an edge to the constituent node for the edge, then following an edge to another node, and

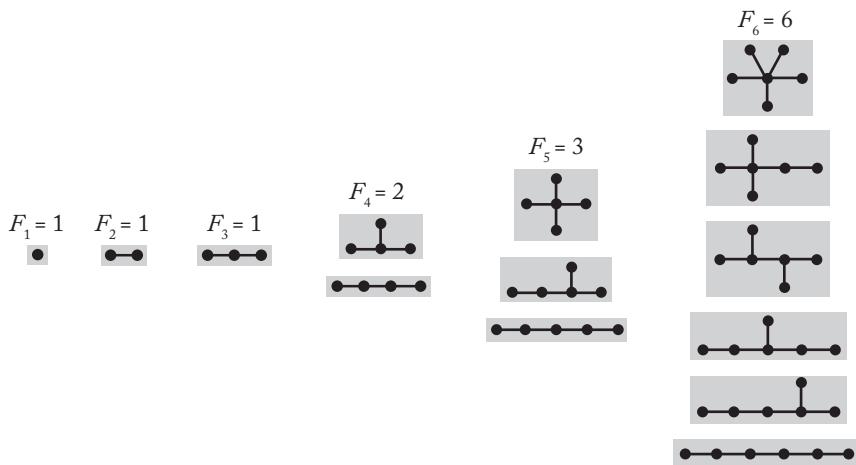


Figure 6.19 Unrooted unordered (free) trees with N nodes, $1 \leq N \leq 6$

so on. The shortest sequence of edges leading from one node to another in this way is called a *simple path*. A graph is *connected* if there is a simple path connecting any pair of nodes. A simple path from a node back to itself is called a *cycle*.

Every tree is a graph—but which graphs are trees? It is well known that any one of the following four conditions is necessary and sufficient to ensure that a graph G with N nodes is an (unrooted unordered) tree:

- (i) G has $N - 1$ edges and no cycles.
- (ii) G has $N - 1$ edges and is connected.
- (iii) Exactly one simple path connects each pair of vertices in G .
- (iv) G is connected, but does not remain connected if any edge is removed.

That is, we could use any one of these conditions to define free trees. To be concrete, we choose the following descriptive combination:

Definition A *tree* is a connected acyclic graph.

As an example of an algorithm where free trees arise in a natural way, consider perhaps the most basic question that we can ask about a graph: is it connected? That is, is there some path connecting every pair of vertices? If so, then there is a minimal set of edges comprising such paths called the *spanning tree* of the graph. If the graph is not connected, then there is a spanning forest, one tree for each connected component. Figure 6.20 gives examples of two spanning trees of a large graph.

Definition A *spanning tree* of a graph of N vertices is a set of $N - 1$ of the edges of the graph that form a tree.

By the basic properties of trees, a spanning tree must include all of the nodes, and its existence demonstrates that all pairs of nodes are connected by some path. In general, a spanning tree is an unrooted, unordered tree.

One well-known algorithm for finding a spanning tree is to consider each edge in turn, checking whether adding the next edge to the set comprising the partial spanning tree built so far would cause a cycle. If not, add it to the spanning tree and go on to consider the next edge. When the set has $N - 1$ edges in it, the edges represent an unordered, unrooted tree; indeed, it is a spanning tree for the graph. One way to implement this algorithm is to use the union-find algorithm given earlier for the cycle test. That is, we just run unionfind until getting a single component. If the edges have

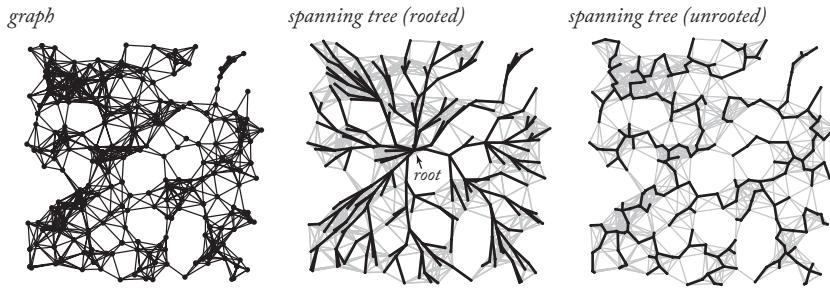


Figure 6.20 A large graph and two of its spanning trees

lengths that satisfy the triangle equality and we consider the edges in order of their length, we get *Kruskal's algorithm*, which computes a *minimal* spanning tree (no other spanning tree has smaller total edge length), shown on the right in Figure 6.20. The key point to note now is that Kruskal's algorithm is an example of free trees naturally occurring in a computation. Many other algorithms for finding spanning trees have been devised and analyzed—for example, the *breadth-first search* algorithm picks a root and considers vertices in order of their distance from the root, thereby computing a rooted spanning tree, shown in the middle in Figure 6.20.

The combinatorics literature contains a vast amount of material on the theory of graphs, including many textbooks, and the computer science literature contains a vast amount of material about algorithms on graphs, also including many textbooks. Full coverage of this material is beyond the scope of this book, but understanding the simpler structures and algorithms that we do cover is good preparation for addressing more difficult questions about properties of random graphs and the analysis of algorithms on graphs. Examples of graph problems where the techniques we have been considering apply directly may be found in [15] and the classical reference Harary and Palmer [21]. We will consider some special families of graphs again in Chapter 9, but let us return now to our study of various types of trees.

Exercise 6.55 How many of the $2^{\binom{N}{2}}$ graphs on N labelled vertices are free trees?

Exercise 6.56 For each of the four properties listed earlier, show that the other three are implied. (This is 12 exercises in disguise!)

Tree hierarchy. The four major types of trees that we have defined form a hierarchy, as summarized and illustrated in Figure 6.21. (i) The *free tree* is the most general, simply an acyclic connected graph. (ii) The *rooted tree* has a distinguished root node. (iii) The *ordered tree* is a rooted tree where the order of the subtrees of a node is significant. (iv) The *binary tree* is an ordered tree with the further restriction that every node has degree 0 or 2. In the nomenclature that we use, the adjective describes the characteristic that separates each type of tree from the one above it in the hierarchy. It is also common to use nomenclature that separates each type from the one *below* it in the hierarchy. Thus, we sometimes refer to free trees as unrooted trees, rooted trees as unordered trees, and ordered trees as general Catalan trees.

A few more words on nomenclature are appropriate because of the variety of terms found in the literature. Ordered trees are often called *plane* or *planar* trees and unordered trees are referred to as *nonplane* trees. The term *plane* is used because the structures can be transformed to one another with continuous deformations in the plane. Though this terminology is widely used, we prefer *ordered* because of its natural implications with regard to computer representations. The term *oriented* in Figure 6.21 refers to the fact that the

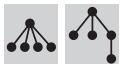
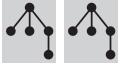
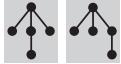
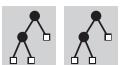
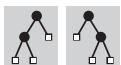
	other names	basic properties	identical trees	different trees
free tree	unrooted tree tree	connected acyclic		
rooted tree	planted tree oriented tree unordered tree	specified root node		
ordered tree	planar tree tree Catalan tree	significant subtree order		
binary tree	binary Catalan tree	rooted, ordered 2-ary internal nodes 0-ary external nodes		

Figure 6.21 Summary of tree nomenclature

root is distinguished, so there is an orientation of the edges toward the root; we prefer the term *rooted*, and we omit even that modifier when it is obvious from the context that there is a root involved.

As the definitions get more restrictive, the number of trees that are regarded as different gets larger, so, for a given size, there are more rooted trees than free trees and more ordered trees than rooted trees. It turns out that the ratio between the number of rooted trees and the number of free trees is proportional to N ; the corresponding ratio of ordered trees to rooted trees grows exponentially with N . It is also the case that the ratio of the number of binary trees to the number of ordered trees with the same number of nodes is a constant. The rest of this section is devoted to a derivation of analytic results that quantify these distinctions. The enumeration results are summarized in Table 6.3.

Figure 6.22 is an illustration of the hierarchy for trees with five nodes. The 14 different five-node ordered trees are depicted in the figure, and they are further organized into equivalence classes using a meta-forest where all the trees equivalent to a given tree are its children. There are 3 different five-node free trees (hence three trees in the forest), 9 different five-node rooted trees (those at level 1 in the forest), and 14 different five-node ordered trees (those at the bottom level in the forest). Note that the counts just given for Figure 6.22 correspond to the fourth column ($N = 5$) in Table 6.3.

From a combinatorial point of view, we perhaps might be more interested in free trees because they differentiate structures at the most essential

	2	3	4	5	6	7	8	9	10	N
free	1	1	2	3	6	11	23	47	106	$\sim c_1 \alpha^N / N^{5/2}$
rooted	1	2	4	9	20	48	115	286	719	$\sim c_2 \alpha^N / N^{3/2}$
ordered	1	2	5	14	42	132	429	1430	4862	$\sim c_3 4^N / N^{3/2}$
binary	2	5	14	42	132	429	1430	4862	16796	$\sim (4c_3) 4^N / N^{3/2}$

$$\alpha \approx 2.9558, c_1 \approx .5350, c_2 \approx .4399, c_3 = 1/4\sqrt{\pi} \approx .1410$$

Table 6.3 Enumeration of unlabelled trees

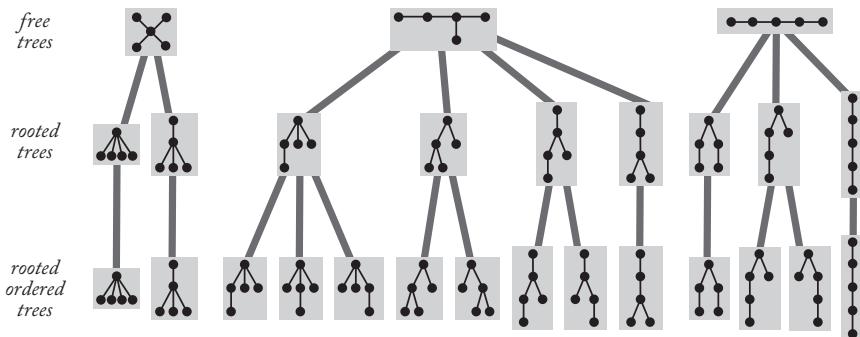


Figure 6.22 Trees with five nodes (ordered, unordered, and unrooted)

level. From the point of view of computer applications, we are perhaps more interested in binary trees and ordered trees because they have the property that the standard computer representation uniquely determines the tree, and in rooted trees because they are the quintessential recursive structure. In this book, we consider all these types of trees not only because they all arise in important computer algorithms, but also because their analysis illustrates nearly the full range of analytic techniques that we present. But we maintain our algorithmic bias, by reserving the word *tree* for ordered trees, which arise in perhaps the most natural way in computer applications. Combinatorics texts more typically reserve the unmodified “tree” to describe unordered or free trees.

Exercise 6.57 Which free tree structure on six nodes appears most frequently among all ordered trees on six nodes? (Figure 6.22 shows that the answer for five nodes is the tree in the middle.)

Exercise 6.58 Answer the previous exercise for seven, eight, and more nodes, going as high as you can.

In binary tree search and other algorithms that use binary trees, we directly represent the ordered pair of subtrees with an ordered pair of links to subtrees. Similarly, a typical way to represent the subtrees in general Catalan trees is an ordered list of links to subtrees. There is a one-to-one correspondence between the trees and their computer representation. Indeed, in §6.3,

we considered a number of different ways to represent trees and binary trees in an unambiguous manner. The situation is different when it comes to representing rooted trees and free trees, where we are faced with several ways to represent the same tree. This has many implications in algorithm design and analysis. A typical example is the “tree isomorphism” problem: given two different tree representations, determine whether they represent the same rooted tree, or the same free tree. Not only are no efficient algorithms known for this problem, but also it is one of the few problems whose difficulty remains unclassified (see [16]).

The first challenge in analyzing algorithms that use trees is to find a probabilistic model that realistically approximates the situation at hand. Are the trees random? Is the tree distribution induced by some external randomness? How does the tree representation affect the algorithm and analysis? These lead to a host of analytic problems. For example, the “union-find” problem mentioned earlier has been analyzed using a number of different models (see Knuth and Schönhage [26]). We can assume that the sequence of equivalence relations consists of random node pairs, or that they correspond to random edges from a random forest, and so on. As we have seen with binary search trees and binary Catalan trees, the fundamental recursive decomposition leads to similarities in the analysis, but the induced distributions lead to significant differences in the analysis.

For various applications we may be interested in values of parameters that measure fundamental characteristics of the various types of trees, so we are faced with a host of analytic problems to consider. The enumeration results are classical (see [21], [30], and [24]), and are summarized in Table 6.3. The derivation of some of these results is discussed next. Functional equations on the generating functions are easily available through the symbolic method, but asymptotic estimates of coefficients are slightly beyond the scope of this book, in some cases. More details may be found in [15].

Exercise 6.59 Give an efficient algorithm that takes as input a set of edges that represents a tree and produces as output a parenthesis system representation of that tree.

Exercise 6.60 Give an efficient algorithm that takes as input a set of edges that represents a tree and produces as output a binary tree representation of that tree.

Exercise 6.61 Give an efficient algorithm that takes as input two binary trees and determines whether they are different when considered as unordered trees.

Exercise 6.62 [cf. Aho, Hopcroft, and Ullman] Give an efficient algorithm that takes as input two parenthesis systems and determines whether they represent the same rooted tree.

Counting rooted unordered trees. This sequence is OEIS A000081 [34]. Let \mathcal{U} be the set of all rooted (unordered) trees with associated OGF

$$U(z) = \sum_{u \in \mathcal{U}} z^{|u|} = \sum_{N \geq 0} U_N z^N,$$

where U_N is the number of rooted trees with N nodes. Since each rooted tree comprises a root and a multiset of rooted trees, we can also express this generating function as an infinite product, in two ways:

$$U(z) = z \prod_{u \in \mathcal{U}} (1 - z^{|u|})^{-1} = z \prod_N (1 - z^N)^{-U_N}.$$

The first product is an application of the “multiset” construction associated with the symbolic method (see Exercise 5.6): for each tree u , the term $(1 - z^{|u|})^{-1}$ allows for the presence of an arbitrary number of occurrences of u in the set. The second product follows by grouping the U_N terms corresponding to the trees with N nodes.

Theorem 6.12 (Enumeration of rooted unordered trees). The OGF that enumerates unordered trees satisfies the functional equation

$$U(z) = z \exp \left\{ U(z) + \frac{1}{2} U(z^2) + \frac{1}{3} U(z^3) + \dots \right\}.$$

Asymptotically,

$$U_N \equiv [z^N]U(z) \sim c\alpha^N/N^{3/2}$$

where $c \approx 0.4399237$ and $\alpha \approx 2.9557649$.

Proof. Continuing the discussion above, take the logarithm of both sides:

$$\begin{aligned} \ln \frac{U(z)}{z} &= - \sum_{N \geq 1} U_N \ln(1 - z^N) \\ &= \sum_{N \geq 1} U_N (z^N + \frac{1}{2}z^{2N} + \frac{1}{3}z^{3N} + \frac{1}{4}z^{4N} + \dots) \\ &= U(z) + \frac{1}{2}U(z^2) + \frac{1}{3}U(z^3) + \frac{1}{4}U(z^4) + \dots \end{aligned}$$

The stated functional equation follows by exponentiating both sides.

The asymptotic analysis is beyond the scope of this book. It depends on complex analysis methods related to the direct generating function asymptotics that we introduced in Chapter 4. Details may be found in [15], [21], and [30]. ■

This result tells us several interesting lessons. First, the OGF admits no explicit form in terms of elementary functions of analysis. However, it is fully determined by the functional equation. Indeed, the same reasoning shows that the OGF of trees of height $\leq h$ satisfies

$$U^{[0]}(z) = z; \quad U^{[h+1]}(z) = z \exp\left(U^{[h]}(z) + \frac{1}{2}U^{[h]}(z^2) + \frac{1}{3}U^{[h]}(z^3) + \dots\right).$$

Moreover, $U^{[h]}(z) \rightarrow U(z)$ as $h \rightarrow \infty$, and both series agree to $h + 1$ terms. This provides a way to compute an arbitrary number of initial values:

$$U(z) = z + z^2 + 2z^3 + 4z^4 + 9z^5 + 20z^6 + 48z^7 + 115z^8 + 286z^9 + \dots$$

It is also noteworthy that a precise asymptotic analysis can be effected even though the OGF admits no closed form. Actually, this analysis is the historical source of the so-called Darboux-Polya method of asymptotic enumeration, which we introduced briefly in §5.5. Polya in 1937 realized in this way the asymptotic analysis of a large variety of tree types (see Polya and Read [30], a translation of Polya's classic paper), especially models of chemical isomers of hydrocarbons, alcohols, and so forth.

Exercise 6.63 Write a program to compute all the values of U_N that are smaller than the maximum representable integer in your machine, using the method suggested in the text. Estimate how many (unlimited precision) arithmetic operations would be required for large N , using this method.

Exercise 6.64 [cf. Harary-Palmer] Show that

$$NU_{N+1} = \sum_{1 \leq k \leq N} \left(kU_k \sum_{k \leq kl \leq N} T_{N+1-kl} \right)$$

and deduce that U_N can be determined in $O(N^2)$ arithmetic operations. (*Hint:* Differentiate the functional equation.)

Exercise 6.65 Give a polynomial-time algorithm to generate a random rooted tree of size N .

Counting free trees. This sequence is OEIS A000055 [34]. Without a root to fix upon, the combinatorial argument is more sophisticated, though it has been known at least since 1889 (see Harary and Palmer [21]). The asymptotic estimate follows via a generating function argument, using the asymptotic formula for rooted trees just derived. We leave details for exercises.

Exercise 6.66 Show that the number of rooted trees of N nodes is bounded below by the number of free trees of N nodes and bounded above by N times the number of free trees of N nodes. (Thus, the exponential order of growth of the two quantities is the same.)

Exercise 6.67 Let $F(z)$ be the OGF for free trees. Show that

$$F(z) = U(z) - \frac{1}{2}U(z)^2 + \frac{1}{2}U(z^2).$$

Exercise 6.68 Derive the asymptotic formula for free trees given in Table 6.3, using the formula given in Theorem 6.12 for rooted (unordered) trees and the previous exercise.

6.14 Labelled Trees. The counting results above assume that the nodes in the trees are indistinguishable. If, on the contrary, we assume that the nodes have distinct identities, then there are many more ways to organize them into trees. For example, different trees result when different nodes are used for the root. As mentioned earlier, the number of “different” trees increases when we specify a root and when we consider the order of subtrees significant. As in Chapter 5, we are using “labels” as a combinatorial device to distinguish nodes. This, of course, has nothing to do with keys in binary search trees, which are application data associated with nodes.

The different types of labelled trees are illustrated in Figure 6.23, which corresponds to Figure 6.22. The trees at the bottom level are different rooted, ordered, and labelled trees; those in the middle level are different unordered labelled trees; and those at the top level are different unrooted, unordered labelled trees. As usual, we are interested in knowing how many labelled trees there are, of each of the types that we have considered. Table 6.4 gives small values and asymptotic estimates for the counts of the various labelled trees. The second column ($N = 3$) in Table 6.4 corresponds to the trees in Figure 6.23.

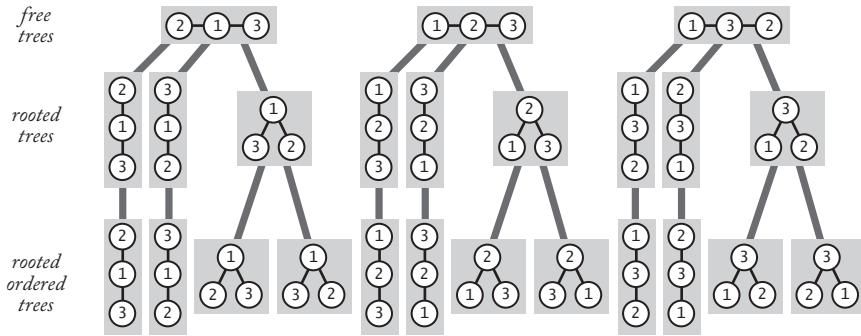


Figure 6.23 Labelled trees with three nodes
(ordered, unordered, and unrooted)

As discussed in Chapter 5, EGFs are the appropriate tool for approaching the enumeration of labelled trees, not just because there are so many more possibilities, but also because the basic combinatorial manipulations that we use on labelled structures are naturally understood through EGFs.

Exercise 6.69 Which tree of four nodes has the most different labellings? Answer this question for five, six, and more nodes, going as high as you can.

Counting ordered labelled trees. An unlabelled tree is uniquely determined by a preorder traversal, and any of the $N!$ permutations can be used with the preorder traversal to assign labels to an ordered tree with N nodes, so the number of labelled trees is just $N!$ times the number of unlabelled trees. Such

	2	3	4	5	6	7	N
<i>ordered</i>	2	12	120	1680	46656	665280	$\frac{(2N - 2)!}{(N - 1)!}$
<i>rooted</i>	2	9	64	625	7976	117649	N^{N-1}
<i>free</i>	1	3	16	125	1296	16807	N^{N-2}

Table 6.4 Enumeration of labelled trees

an argument is clearly general. For ordered trees, the labelled and unlabelled varieties are closely related and their counts differ only by a factor of $N!$. These simple combinatorial arguments are appealing and instructive, but it is also instructive to use the symbolic method.

Theorem 6.13 (Enumeration of ordered labelled trees). The number of ordered rooted labelled trees with N nodes is $(2N - 2)!/(N - 1)!$.

Proof. An ordered labelled forest is either empty or a sequence of ordered labelled trees, so we have the combinatorial construction

$$\mathcal{L} = \mathcal{Z} \times \text{SEQ}(\mathcal{L})$$

and by the symbolic method we have

$$L(z) = \frac{z}{1 - L(z)}.$$

This is virtually the same argument as that used previously for ordered (unlabelled) trees, but we are now working with EGFs (Theorem 5.2) for labelled objects, where before we were using OGFs for unlabelled objects (Theorem 5.1). Thus $L(z) = (1 - \sqrt{1 - 4z})/2$ and the number of ordered rooted labelled trees with N nodes is given by

$$N![z^N]L(z) = N! \frac{1}{N} \binom{2N - 2}{N - 1} = \frac{(2N - 2)!}{(N - 1)!}.$$

This sequence is OEIS A001813 [34].

Counting unordered labelled trees. Unordered (rooted) labelled trees are also called *Cayley trees*, because they were enumerated by A. Cayley in the 19th century. A Cayley forest is either empty or a set of Cayley trees, so we have the combinatorial construction

$$\mathcal{C} = \mathcal{Z} \times \text{SET}(\mathcal{C})$$

and by the symbolic method we have

$$C(z) = ze^{C(z)}.$$

Theorem 6.14 (Enumeration of unordered labelled trees). The EGF that enumerates unordered labelled trees satisfies the functional equation

$$C(z) = ze^{C(z)}.$$

The number of such trees of size N is

$$C_N = N![z^N]C(z) = N^{N-1}$$

and the number of unordered k -forests of such trees is

$$C_N^{[k]} = N![z^N] \frac{(C(z))^k}{k!} = \binom{N-1}{k-1} N^{N-k}.$$

Proof. Following the derivation of the EGF via the symbolic method, Lagrange inversion (see §6.12) immediately yields the stated results. This sequence is OEIS A000169 [34]. ■

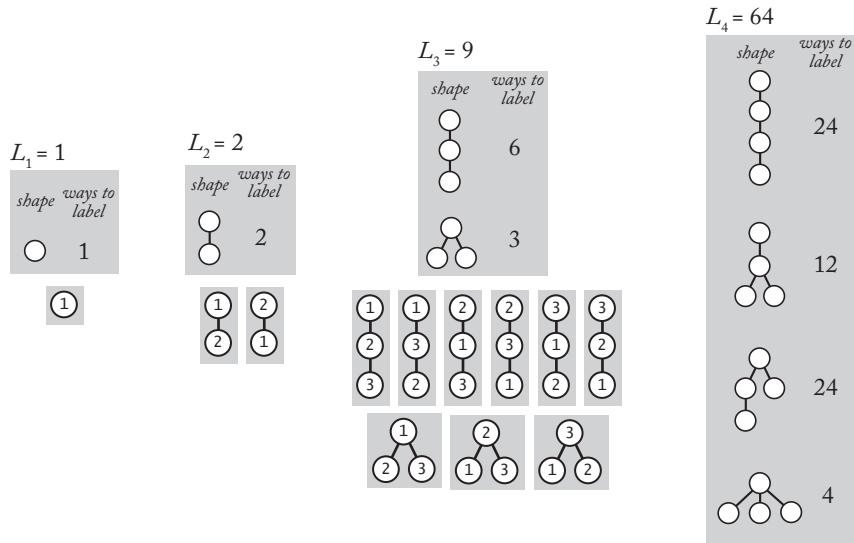


Figure 6.24 Cayley (labelled rooted unordered) trees, $1 \leq N \leq 4$.

Combinatorial proof? As illustrated in Figure 6.24, Cayley tree enumeration is a bit magical. Adding the ways to label each unordered labelled tree shape (each count needing a separate argument) gives a very simple expression. Is there a simple combinatorial proof? The answer to this question is a classic exercise in elementary combinatorics: Devise a 1:1 correspondence between N -node Cayley trees and sequences of $N - 1$ integers, all between 1 and N . Readers are encouraged to think about this problem before finding a solution in a combinatorics text (or [15] or [24]). Such constructions are interesting and appealing, but they perhaps underscore the importance of general approaches that can solve a broad variety of problems, such as the symbolic method and Lagrange inversion.

For reference, the enumeration generating functions for both unlabelled trees and labelled trees are given in Table 6.5. The values of the coefficients for labelled trees are given in Table 6.4.

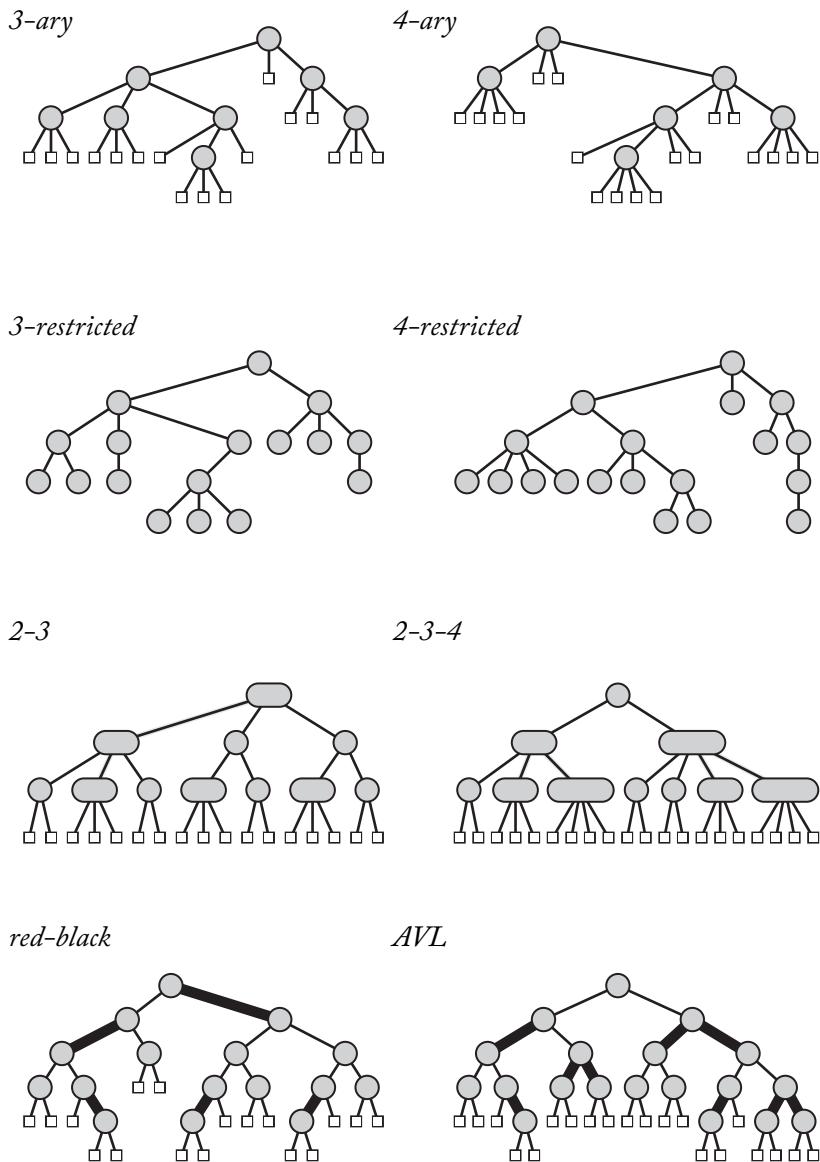
Exercise 6.70 What is the number of labelled rooted forests of N nodes?

Exercise 6.71 Show that the EGF that enumerates labelled free trees is equal to $C(z) - C(z)^2/2$.

6.15 Other Types of Trees. It is often convenient to place various local and global restrictions on trees—for example, to suit requirements of a particular application or to try to rule out degenerate cases. From a combinatorial standpoint, any restriction corresponds to a new class of tree, and a new collection of problems need to be solved to enumerate the trees and to learn their

	unlabelled (OGF)	labelled (EGF)
ordered	$G(z) = \frac{z}{1 - G(z)}$	$L(z) = \frac{z}{1 - L(z)}$
rooted	$U(z) = z \exp\left\{\sum_{i \geq 1} U(z^i)/i\right\}$	$C(z) = ze^{C(z)}$
free	$U(z) - U(z)^2/2 + U(z^2)/2$	$C(z) - C(z)^2/2$

Table 6.5 Tree enumeration generating functions

**Figure 6.25** Examples of various other types of trees

statistical properties. In this section, we catalog many well-known and widely used special types of trees, for reference. Examples are drawn in Figure 6.25, and definitions are given in the discussion below. (*Note on nomenclature:* In this section, we use $T(z)$ to denote the OGF for various generalizations of the Catalan OGF to emphasize similarities in the analysis, while sparing the reader from excessive notational baggage.)

Definition A t -ary tree is either an external node or an internal node attached to an ordered sequence of t subtrees, all of which are t -ary trees.

This is the natural generalization of binary trees that we considered as an example when looking at Lagrange inversion in §6.12. We insist that every node have exactly t descendants. These trees are normally considered to be ordered—this matches a computer representation where t links are reserved for each node, to point to its descendants. In some applications, keys might be associated with internal nodes; in other cases, internal nodes might correspond to sequences of $t - 1$ keys; in still other cases data might be associated with external nodes. One important tree of this type is the *quad tree*, where information about geometric data is organized by decomposing an area into four quadrants, proceeding recursively.

Theorem 6.15 (Enumeration of t -ary trees). The OGF that enumerates t -ary trees (by external nodes) satisfies the functional equation

$$T(z) = z + (T(z))^t.$$

The number of t -ary trees with N internal nodes and $(t - 1)N + 1$ external nodes is

$$\frac{1}{(t - 1)N + 1} \binom{tN}{N} \sim c_t (\alpha_t)^N / N^{3/2}$$

where $\alpha_t = t^t / (t - 1)^{t-1}$ and $c_t = 1 / \sqrt{(2\pi)(t - 1)^3/t}$.

Proof. We use Lagrange inversion, in a similar manner as for the solution given in §6.12 for the case $t = 3$. By the symbolic method, the OGF with size measured by external nodes satisfies

$$T(z) = z + T(z)^3.$$

This can be subjected to the Lagrange theorem, since $z = T(z)(1 - T(z)^2)$, so we have an expression for the number of trees with $2N + 1$ external nodes (N internal nodes):

$$\begin{aligned}[z^{2N+1}]T(z) &= \frac{1}{2N+1}[u^{2N}]\frac{1}{(1-u^2)^{2N+1}} \\ &= \frac{1}{2N+1}[u^N]\frac{1}{(1-u)^{2N+1}} \\ &= \frac{1}{2N+1}\binom{3N}{N}.\end{aligned}$$

This is equivalent to the expression given in §6.12, and it generalizes immediately to give the stated result. The asymptotic estimate also follows directly when we use the same methods as for the Catalan numbers (see §4.3). ■

Exercise 6.72 Find the number of k -forests with a total of N internal nodes.

Exercise 6.73 Derive the asymptotic estimate given in Theorem 6.15 for the number of t -ary trees with N internal nodes.

Definition A *t -restricted tree* is a node (called the root) containing links to t or fewer t -restricted trees.

The difference between t -restricted trees and t -ary trees is that not every internal node must have t links. This has direct implications in the computer representation: for t -ary trees, we might as well reserve space for t links in all internal nodes, but t -restricted trees might be better represented as binary trees, using the standard correspondence. Again, we normally consider these to be ordered, though we might also consider unordered and/or unrooted t -restricted trees. Every node is linked to at most $t + 1$ other nodes in a t -restricted tree, as shown in Figure 6.25.

The case $t = 2$ corresponds to the so-called *Motzkin numbers*, for which we can get an explicit expression for the OGF $M(z)$ by solving the quadratic equation. We have

$$M(z) = z(1 + M(z) + M(z)^2)$$

so that

$$M(z) = \frac{1 - z - \sqrt{1 - 2z - 3z^2}}{2z} = \frac{1 - z - \sqrt{(1+z)(1-3z)}}{2z}.$$

Now, Theorem 4.11 provides an immediate proof that $[z^N]M(z)$ is $O(3^N)$, and methods from complex asymptotics yield the more accurate asymptotic estimate $3^N/\sqrt{3/4\pi N^3}$. Actually, with about the same amount of work, we can derive a much more general result.

Theorem 6.16 (Enumeration of t -restricted trees). Let $\theta(u) = 1+u+u^2+\dots+u^t$. The OGF that enumerates t -restricted trees satisfies the functional equation

$$T(z) = z\theta(T(z))$$

and the number of t -restricted trees is

$$[z^N]T(z) = \frac{1}{N}[u^{N-1}](\theta(u))^N \sim c_t \alpha_t^N / N^{3/2}$$

where τ is the smallest positive root of $\theta(\tau) - \tau\theta'(\tau) = 0$ and the constants α_t and c_t are given by $\alpha_t = \theta'(\tau)$ and $c_t = \sqrt{\theta(\tau)/2\pi\theta''(\tau)}$.

Proof. The first parts of the theorem are immediate from the symbolic method and Lagrange inversion. The asymptotic result requires singularity analysis, using an extension of Theorem 4.12 (see [15]).

This result follows from a theorem proved by Meir and Moon in 1978 [28], and it actually holds for a large class of polynomials $\theta(u)$ of the form $1+a_1u+a_2u^2+\dots$, subject to the constraint that the coefficients are positive and that a_1 and at least one other coefficient are nonzero. ■

The asymptotic estimates of the number of t -restricted trees for small t are given in the following table:

t	c_t	α_t
2	.4886025119	3.0
3	.2520904538	3.610718613
$c_t \alpha_t^N / N^{3/2}$.1932828341	3.834437249
4	.1691882413	3.925387252
5	.1571440515	3.965092635
6	.1410473965	4.0
∞		

For large t , the values of α_t approaches 4, which is perhaps to be expected, since the trees are then like general Catalan trees.

Exercise 6.74 Use the identity $1 + u + u^2 + \dots + u^t = (1 - u^{t+1})/(1 - u)$ to find a sum expression for the number of t -restricted trees with N nodes.

Exercise 6.75 Write a program that, given t , will compute the number of t -restricted trees for all values of N for which the number is smaller than the maximum representable integer in your machine.

Exercise 6.76 Find the number of “even” t -restricted trees, where all nodes have an even number of, and less than t , children.

Height-restricted trees. Other types of trees involve restrictions on height. Such trees are important because they can be used as binary search tree replacements that provide a guaranteed $O(\log N)$ search time. This was first shown in 1960 by Adel'son-Vel'skii and Landis [1], and such trees have been widely studied since (for example, see Bayer and McCreight [3] or Guibas and Sedgewick [20]). Balanced trees are of practical interest because they combine the simplicity and flexibility of binary tree search and insertion with good worst-case performance. They are often used for very large database applications, so asymptotic results on performance are of direct practical interest.

Definition An *AVL tree* of height 0 or height -1 is an external node; an AVL tree of height $h > 0$ is an internal node linked to a left and a right subtree, both of height $h - 1$ or $h - 2$.

Definition A *B-tree* of height 0 is an external node; a B-tree of order M and height $h > 0$ is an internal node connected to a sequence of between $\lceil M/2 \rceil$ and M B-trees of order M and height $h - 1$.

B-trees of order 3 and 4 are normally called 2-3 trees and 2-3-4 trees, respectively. Several methods, known as *balanced tree* algorithms, have been devised using these and similar structures, based on the general theme of mapping permutations into tree structures that are guaranteed to have no long paths. More details, including relationships among the various types, are given by Guibas and Sedgewick [20], who also show that many of the structures (including AVL trees and B-trees) can be mapped into binary trees with marked edges, as in Figure 6.25.

Exercise 6.77 Without solving the enumeration problem in detail, try to place the following classes of trees in increasing order of their cardinality for large N : 3-ary, 3-restricted, 2-3, and AVL.

Exercise 6.78 Build a table giving the number of AVL and 2-3 trees with fewer than 15 nodes that are different when considered as unordered trees.

Balanced tree structures illustrate the variety of tree structures that arise in applications. They lead to a host of analytic problems of interest, and they fall at various points along the continuum between purely combinatoric structures and purely “algorithmic” structures. None of the binary tree structures has been precisely analyzed under random insertions for statistics such as path length, despite their importance. It is even challenging to enumerate them (for example, see Aho and Sloane [2] or Flajolet and Odlyzko [13]).

For each of these types of structures, we are interested in knowing how many essentially different structures there are of each size, plus statistics about various important parameters. For some of the structures, developing functional equations for enumeration is relatively straightforward, because they are recursively defined. (Some of the balanced tree structures cannot even be easily defined and analyzed recursively, but rather need to be defined in terms of the algorithm that maps permutations into them.) As with tree height, the functional equation is only a starting point, and further analysis of these structures turns out to be quite difficult. Functional equations for generating functions for several of the types we have discussed are given in Table 6.6.

Exercise 6.79 Prove the functional equations on the generating functions for AVL and 2-3 trees given in Table 6.6.

More important, just as we analyzed both binary trees (uniformly distributed) and binary search trees (binary trees distributed as constructed from random permutations by the algorithm), we often need to know statistics on various classes of trees according to a distribution induced by an algorithm that transforms some other combinatorial object into a tree structure, which leads to more analytic problems. That is, several of the basic tree structures that we have defined serve many algorithms. We used the term *binary search tree* to distinguish the combinatorial object (the binary tree) from the algorithm that maps permutations into it; balanced tree and other algorithms need to be distinguished in a similar manner.

Indeed, AVL trees, B-trees, and other types of search trees are *primarily* of interest when distributed as constructed from random permutations. The “each tree equally likely” combinatorial objects have been studied both because the associated problems are more amenable to combinatorial analysis and because knowledge of their properties may give some insight into solv-

ing problems that arise when analyzing them as data structures. Even so, the basic problem of just enumerating the balanced tree structures is still quite difficult (for example, see [29]). None of the associated algorithms has been analyzed under the random permutation model, and the average-case analysis of balanced tree algorithms is one of the outstanding problems in the analysis of algorithms.

Figure 6.26 gives some indication of the complexity of the situation. It shows the distribution of the subtree sizes in random AVL trees (all trees equally likely) and may be compared with Figure 6.10, the corresponding figure for Catalan trees. The corresponding figure for BSTs is a series of straight lines, at height $1/N$.

Where Catalan trees have an asymptotically constant probability of having a fixed number of nodes in a subtree for any tree size N , the balance condition for AVL trees means that small subtrees cannot occur for large N . Indeed, we might expect the trees to be “balanced” in the sense that the subtree sizes might cluster near the middle for large N . This does seem to be

tree type (size measure)	functional equation on generating function from symbolic method
3-ary (external nodes)	$T(z) = z + T(z)^3$
3-ary (internal nodes)	$T(z) = 1 + zT(z)^3$
3-restricted (nodes)	$T(z) = z(1 + T(z) + T(z)^2 + T(z)^3)$
AVL of height h (internal nodes)	$A_h(z) = \begin{cases} 1 & h < 0 \\ 1 \\ zA_{h-1}(z)^2 + 2zA_{h-1}(z)A_{h-2}(z) & h > 0 \end{cases}$
2-3 of height h (external nodes)	$B_h(z) = \begin{cases} z & h = 0 \\ B_{h-1}(z^2 + z^3) & h > 0 \end{cases}$

Table 6.6 Generating functions for other types of trees

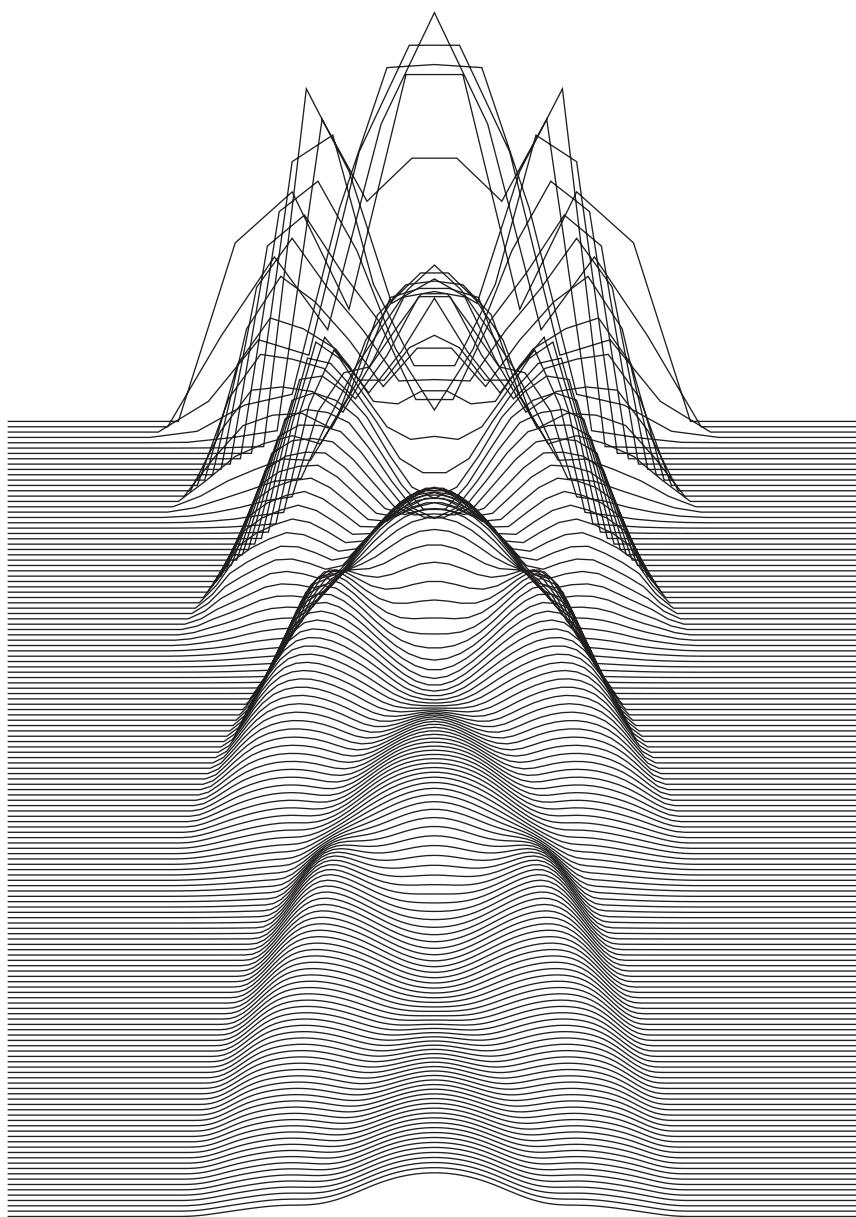


Figure 6.26 AVL distribution (subtree sizes in random AVL trees)
(scaled and translated to separate curves)

the case for some N , but it also is true that for some other N , there are *two* peaks in the distribution, which means that a large fraction of the trees have significantly fewer than half of the nodes on one side or the other. Indeed, the distribution exhibits an oscillatory behavior, roughly between these two extremes. An analytic expression describing this has to account for this oscillation and so may not be as concise as we would like. Presumably, similar effects are involved when balanced trees are built from permutations in searching applications, but this has not yet been shown.

TREES are pervasive in the algorithms we consider, either as explicit structures or as models of recursive computations. Much of our knowledge of properties of our most important algorithms can be traced to properties of trees.

We will encounter other types of trees in later chapters, but they all share an intrinsic recursive nature that makes their analysis natural using generating functions as just described: the recursive structure leads directly to an equation that yields a closed-form expression or a recursive formulation for the generating function. The second part of the analysis, extracting the desired coefficients, requires advanced techniques for some types of trees.

The distinction exhibited by comparing the analysis of tree path length with tree height is essential. Generally, we can describe combinatorial parameters recursively, but “additive” parameters such as path length are much simpler to handle than “nonadditive” parameters such as height, because generating function constructions that correspond to combinatorial constructions can be exploited directly in the former case.

Our first theme in this chapter has been to introduce the history of the analysis of trees as combinatorial objects. In recent years, general techniques have been found that help to unify some of the classical results and make it possible to learn characteristics of ever more complicated new tree structures. We discuss this theme in detail and cover many examples in [15], and Drmota’s book [11] is a thorough treatment that describes the extensive amount of knowledge about random trees that has been developed in the years since the early breakthroughs that we have described here.

Beyond classical combinatorics and specific applications in algorithmic analysis, we have endeavored to show how algorithmic applications lead to a host of new mathematical problems that have an interesting and intricate structure in their own right. The binary search tree algorithm is prototypical

of many of the problems that we know how to solve: an algorithm transforms some input combinatorial object (permutations, in the case of binary search trees) into some form of tree. Then we are interested in analyzing the combinatorial properties of trees, not under the uniform model, but under the distribution induced by the transformation. Knowing detailed properties of the combinatorial structures that arise and studying effects of such transformations are the bases for our approach to the analysis of algorithms.

	construction	GF equation	approximate asymptotics
Unlabelled classes			
binary trees	$\mathcal{T} = \mathcal{Z} + \mathcal{T}^2$	$T(z) = z + T^2(z)$.56 $\frac{4^N}{N^{3/2}}$
3-ary trees	$\mathcal{T} = \mathcal{Z} + \mathcal{T}^3$	$T(z) = z + T^3(z)$.24 $\frac{6.75^N}{N^{3/2}}$
trees	$\mathcal{G} = \mathcal{Z} \times \text{SEQ}(\mathcal{G})$	$G(z) = \frac{z}{1 - G(z)}$.14 $\frac{4^N}{N^{3/2}}$
unordered trees	$\mathcal{U} = \mathcal{Z} \times \text{MSET}(\mathcal{U})$	$U(z) = ze^{U(z)+U(z)^2/2+\dots}$.54 $\frac{2.96^N}{N^{3/2}}$
Motzkin trees	$\mathcal{T} = \mathcal{Z} \times (\mathcal{E} + \mathcal{T} + \mathcal{T}^2)$	$T(z) = z(1 + T(z) + T(z)^2)$.49 $\frac{3^N}{N^{3/2}}$

Labelled classes

trees	$\mathcal{L} = \mathcal{Z} \times \text{SEQ}(\mathcal{L})$	$L(z) = \frac{z}{1 - L(z)}$	$\frac{(2N - 2)!}{(N - 1)!}$
Cayley trees	$\mathcal{C} = \mathcal{Z} \times \text{SET}(\mathcal{C})$	$C(z) = ze^{C(z)}$	N^{N-1}

Table 6.7 Analytic combinatorics examples in this chapter

References

1. G. ADEL'SON-VEL'SKII AND E. LANDIS. *Doklady Akademii Nauk SSR* **146**, 1962, 263–266. English translation in *Soviet Math* **3**.
2. A. V. AHO AND N J. A. SLOANE. “Some doubly exponential sequences,” *Fibonacci Quarterly* **11**, 1973, 429–437.
3. R. BAYER AND E. MCCREIGHT. “Organization and maintenance of large ordered indexes,” *Acta Informatica* **3**, 1972, 173–189.
4. J. BENTLEY. “Multidimensional binary search trees used for associative searching,” *Communications of the ACM* **18**, 1975, 509–517.
5. B. BOLLOBÁS. *Random Graphs*, Academic Press, London, 1985.
6. L. COMTET. *Advanced Combinatorics*, Reidel, Dordrecht, 1974.
7. T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, AND C. STEIN. *Introduction to Algorithms*, MIT Press, New York, 3rd edition, 2009.
8. N. G. DE BRUIJN, D. E. KNUTH, AND S. O. RICE. “The average height of planted plane trees,” in *Graph Theory and Computing*, R. C. Read, ed., Academic Press, New York, 1971.
9. L. DEVROYE. “A note on the expected height of binary search trees,” *Journal of the ACM* **33**, 1986, 489–498.
10. L. DEVROYE. “Branching processes in the analysis of heights of trees,” *Acta Informatica* **24**, 1987, 279–298.
11. M. DRMOTA. *Random Trees: An Interplay Between Combinatorics and Probability*, Springer Wein, New York, 2009.
12. P. FLAJOLET AND A. ODLYZKO. “The average height of binary trees and other simple trees,” *Journal of Computer and System Sciences* **25**, 1982, 171–213.
13. P. FLAJOLET AND A. ODLYZKO. “Limit distributions for coefficients of iterates of polynomials with applications to combinatorial enumerations,” *Mathematical Proceedings of the Cambridge Philosophical Society* **96**, 1984, 237–253.
14. P. FLAJOLET, J.-C. RAOULT, AND J. VUILLEMIN. “The number of registers required to evaluate arithmetic expressions,” *Theoretical Computer Science* **9**, 1979, 99–125.
15. P. FLAJOLET AND R. SEDGEWICK. *Analytic Combinatorics*, Cambridge University Press, 2009.

16. M. R. GAREY AND D. S. JOHNSON. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, New York, 1979.
17. G. H. GONNET AND R. BAEZA-YATES. *Handbook of Algorithms and Data Structures in Pascal and C*, 2nd edition, Addison-Wesley, Reading, MA, 1991.
18. I. GOULDEN AND D. JACKSON. *Combinatorial Enumeration*, John Wiley, New York, 1983.
19. R. L. GRAHAM, D. E. KNUTH, AND O. PATASHNIK. *Concrete Mathematics*, 1st edition, Addison-Wesley, Reading, MA, 1989. Second edition, 1994.
20. L. GUIBAS AND R. SEDGEWICK. “A dichromatic framework for balanced trees,” in *Proceedings 19th Annual IEEE Symposium on Foundations of Computer Science*, 1978, 8–21.
21. F. HARARY AND E. M. PALMER. *Graphical Enumeration*, Academic Press, New York, 1973.
22. C. A. R. HOARE. “Quicksort,” *Computer Journal* **5**, 1962, 10–15.
23. R. KEMP. “The average number of registers needed to evaluate a binary tree optimally,” *Acta Informatica* **11**, 1979, 363–372.
24. D. E. KNUTH. *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, 1st edition, Addison-Wesley, Reading, MA, 1968. Third edition, 1997.
25. D. E. KNUTH. *The Art of Computer Programming. Volume 3: Sorting and Searching*, 1st edition, Addison-Wesley, Reading, MA, 1973. Second edition, 1998.
26. D. E. KNUTH AND A. SCHÖNHAGE. “The expected linearity of a simple equivalence algorithm,” *Theoretical Computer Science* **6**, 1978, 281–315.
27. H. MAHMOUD. *Evolution of Random Search Trees*, John Wiley, New York, 1992.
28. A. MEIR AND J. W. MOON. “On the altitude of nodes in random trees,” *Canadian Journal of Mathematics* **30**, 1978, 997–1015.
29. A. M. ODLYZKO. “Periodic oscillations of coefficients of power series that satisfy functional equations,” *Advances in Mathematics* **44**, 1982, 180–205.

30. G. PÓLYA AND R. C. READ. *Combinatorial Enumeration of Groups, Graphs, and Chemical Compounds*, Springer-Verlag, New York, 1987. (English translation of original paper in *Acta Mathematica* **68**, 1937, 145–254.)
31. R. C. READ. “The coding of various kinds of unlabelled trees,” in *Graph Theory and Computing*, R. C. Read, ed., Academic Press, New York, 1971.
32. R. SEDGEWICK. *Algorithms*, 2nd edition, Addison-Wesley, Reading, MA, 1988.
33. R. SEDGEWICK AND K. WAYNE. *Algorithms*, 4th edition, Addison-Wesley, Boston, 2011.
34. N. SLOANE AND S. PLLOUFFE. *The Encyclopedia of Integer Sequences*, Academic Press, San Diego, 1995. Also accessible as *On-Line Encyclopedia of Integer Sequences*, <http://oeis.org>.
35. 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.

CHAPTER SEVEN

PERMUTATIONS

COMBINATORIAL algorithms often deal only with the relative order of a sequence of N elements; thus we can view them as operating on the numbers 1 through N in some order. Such an ordering is called a *permutation*, a familiar combinatorial object with a wealth of interesting properties. We have already encountered permutations: in Chapter 1, where we discussed the analysis of two important comparison-based sorting algorithms using random permutations as an input model; and in Chapter 5, where they played a fundamental role when we introduced the symbolic method for labelled objects. In this chapter, we survey combinatorial properties of permutations and use probability, cumulative, and bivariate generating functions (and the symbolic method) to analyze properties of random permutations.

From the standpoint of the analysis of algorithms, permutations are of interest because they are a suitable model for studying sorting algorithms. In this chapter, we cover the analysis of basic sorting methods such as insertion sort, selection sort, and bubble sort, and discuss several other algorithms that are of importance in practice, including shellsort, priority queue algorithms, and rearrangement algorithms. The correspondence between these methods and basic properties of permutations is perhaps to be expected, but it underscores the importance of fundamental combinatorial mechanisms in the analysis of algorithms.

We begin the chapter by introducing several of the most important properties of permutations and considering some examples as well as some relationships between them. We consider both properties that arise immediately when analyzing basic sorting algorithms and properties that are of independent combinatorial interest.

Following this, we consider numerous different ways to represent permutations, particularly representations implied by inversions and cycles and a two-dimensional representation that exposes relationships between a permutation and its inverse. This representation also helps define explicit relationships between permutations, binary search trees, and “heap-ordered trees”

and reduces the analysis of certain properties of permutations to the study of properties of trees.

Next, we consider enumeration problems on permutations, where we want to count permutations having certain properties, which is equivalent to computing the probability that a random permutation has the property. We attack such problems using generating functions (including the symbolic method on labelled objects). Specifically, we consider properties related to the “cycle structure” of the permutations in some detail, extending the analysis based on the symbolic method that we began in Chapter 5.

Following the same general structure as we did for trees in Chapter 6, we proceed next to analysis of parameters. For trees, we considered path length, height, number of leaves and other parameters. For permutations, we consider properties such as the number of runs and the number of inversions, many of which can be easily analyzed. As usual, we are interested in the expected “cost” of permutations under various measures relating to their properties, assuming all permutations equally likely. For such analyses, we emphasize shortcuts based on generating functions, like the use of CGFs.

We consider the analysis of parameters in the context of two fundamental sorting methods, insertion sort and selection sort, and their relationship to two fundamental characteristics of permutations, inversions and left-to-right minima. We show how CGFs lead to relatively straightforward analyses of these algorithms. We also consider the problem of permuting an array in place and its relationship to the cycle structure of permutations. Some of these analyses lead to familiar generating functions for special numbers from Chapter 3—for example, Stirling and harmonic numbers.

We also consider problems analogous to height in trees in this chapter, including the problems of finding the average length of the shortest and longest cycles in a random permutation. As with tree height, we can set up functional equations on indexed “vertical” generating functions, but asymptotic estimates are best developed using more advanced tools.

The study of properties of permutations illustrates that there is a fine dividing line indeed between trivial and difficult problems in the analysis of algorithms. Some of the problems that we consider can be easily solved with elementary arguments; other (similar) problems are not elementary but can be studied with generating functions and the asymptotic methods we have been considering; still other (still similar) problems require advanced complex analysis or probabilistic methods.

7.1 Basic Properties of Permutations. Permutations may be represented in many ways. The most straightforward, introduced in Chapter 5, is simply a rearrangement of the numbers 1 through N :

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
permutation	9	14	4	1	12	2	10	13	5	6	11	3	8	15	7

In §5.3, we saw that one way to think of a permutation is as a specification of a rearrangement: “1 goes to 9, 2 goes to 14, 3 goes to 4,” and so on. In this section, we introduce a number of basic characteristics of permutations that not only are of inherent interest from a combinatorial standpoint, but also are significant in the study of a number of important algorithms. We also present some analytic results—in later sections we discuss how the results are derived and relate them to the analysis of algorithms.

We will be studying inversions, left-to-right minima and maxima, cycles, rises, runs, falls, peaks, valleys, and increasing subsequences in permutations; inverses of permutations; and special types of permutations called involutions and derangements. These are all explained, in terms of a permutation $p_1 p_2 p_3 \dots p_N$ of the integers 1 to N , in the definitions and text that follow, also with reference to the sample permutation.

Definition An *inversion* is a pair $i < j$ with $p_i > p_j$. If q_j is the number of $i < j$ with $p_i > p_j$, then $q_1 q_2 \dots q_N$ is called the *inversion table* of $p_1 p_2 \dots p_N$. We use the notation $\text{inv}(p)$ to denote the number of inversions in a permutation p , the sum of the entries in the inversion table.

The sample permutation given above has 49 inversions, as evidenced by adding the elements in its inversion table.

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
permutation	9	14	4	1	12	2	10	13	5	6	11	3	8	15	7
inversion table	0	0	2	3	1	4	2	1	5	5	3	9	6	0	8

By definition, the entries in the inversion table $q_1 q_2 \dots q_N$ of a permutation satisfy $0 \leq q_j < j$ for all j from 1 to N . As we will see in §7.3, a unique permutation can be constructed from any sequence of numbers satisfying these constraints. That is, there is a 1:1 correspondence between inversion tables of size N and permutations of N elements (and there are $N!$ of each). That correspondence will be exploited later in this chapter in the analysis of basic sorting methods such as insertion sort and bubble sort.

Definition A *left-to-right minimum* is an index i with $p_j > p_i$ for all $j < i$. We use the notation $\text{lrm}(p)$ to refer to the number of left-to-right minima in a permutation p .

The first element in every permutation is a left-to-right minimum; so is the smallest element. If the smallest element is the first, then it is the only left-to-right minimum; otherwise, there are at least two (the first and the smallest). In general, there could be as many as N left-to-right minima (in the permutation $1 \ 2 \ \dots \ N$). There are three in our sample permutation, at positions 1, 3, and 4. Note that each left-to-right minimum corresponds to an entry $q_k = k - 1$ in the inversion table (all entries to the left are smaller), so counting left-to-right minima in permutations is the same as counting such entries in inversion tables. Left-to-right maxima and right-to-left minima and maxima are defined analogously. In probability theory, left-to-right minima are also known as *records* because they represent new “record” low values that are encountered when moving from left to right through the permutation.

Exercise 7.1 Explain how to compute the number of left-to-right maxima, right-to-left minima, and right-to-left maxima from the inversion table.

Definition A *cycle* is an index sequence $i_1 i_2 \dots i_t$ with $p_{i_1} = i_2$, $p_{i_2} = i_3$, \dots , $p_{i_t} = i_1$. An element in a permutation of length N belongs to a unique cycle of length from 1 to N ; permutations of length N are sets of from 1 to N cycles. A *derangement* is a permutation with no cycles of length 1.

We use the notation $(i_1 \ i_2 \ \dots \ i_t)$ to specify a cycle, or simply draw a circular directed graph, as in §5.3 and Figure 7.1. Our sample permutation is made up of four cycles. One of the cycles is of length 1 so the permutation is not a derangement.

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
permutation	9	14	4	1	12	2	10	13	5	6	11	3	8	15	7
cycles	(1	9	5	12	3	4)	(2	14	15	7	10	6)	(8	13)	(11)

The cycle representation might be read as “1 goes to 9 goes to 5 goes to 12 goes to 3 goes to 4 goes to 1,” and so on. The longest cycle in this permutation is of length 6 (there are two such cycles); the shortest is of length 1. There are t equivalent ways to list any cycle of length t , and the cycles constituting a permutation may be themselves listed in any order. In §5.3 we proved the fundamental bijection between permutations and sets of cycles.

Foata's correspondence. Figure 7.1 also illustrates that if we choose to list the smallest element in each cycle (the *cycle leaders*) first and then take the cycles in decreasing order of their leaders, then we get a canonical form that has an interesting property: *the parentheses are unnecessary*, since each left-to-right minimum in the canonical form corresponds to a new cycle (everything in the same cycle is larger by construction). This constitutes a combinatorial proof that the number of cycles and the number of left-to-right minima are identically distributed for random permutations, a fact we also will verify analytically in this chapter. In combinatorics, this is known as “Foata’s correspondence,” or the “fundamental correspondence.”

Exercise 7.2 How many different ways are there to write the sample permutation in cycle notation?

Exercise 7.3 How many permutations of $2N$ elements have exactly two cycles, each of length N ? How many have N cycles, each of length 2?

Exercise 7.4 Which permutations of N elements have the maximum number of different representations with cycles?

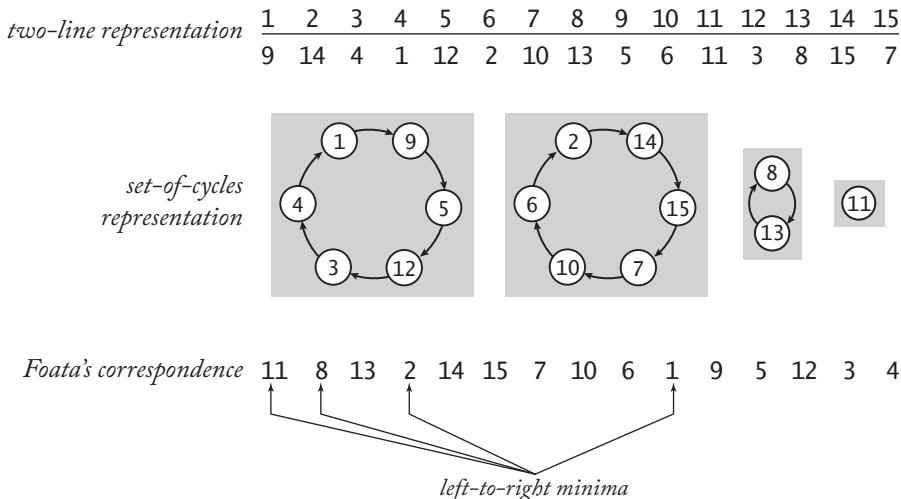


Figure 7.1 Two-line, cycle, and Foata representations of a permutation

Definition The *inverse* of a permutation $p_1 \dots p_N$ is the permutation $q_1 \dots q_N$ with $q_{p_i} = p_{q_i} = i$. An *involution* is a permutation that is its own inverse: $p_{p_i} = i$.

For our sample permutation, the 1 is in position 4, the 2 in position 6, the 3 in position 12, the 4 in position 3, and so forth.

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
permutation	9	14	4	1	12	2	10	13	5	6	11	3	8	15	7
inverse	4	6	12	3	9	10	15	13	1	7	11	5	8	2	14

By the definition, every permutation has a unique inverse, and the inverse of the inverse is the original permutation. The following example of an involution and its representation in cycle form expose the important properties of involutions.

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
involution	9	2	12	4	7	10	5	13	1	6	11	3	8	15	14
cycles	(1 9)	(2)	(3 12)	(4)	(5 7)	(6 10)	(8 13)	(11)	(14 15)						

Clearly, a permutation is an involution if and only if all its cycles are of length 1 or 2. Determining a precise estimate for the number of involutions of length N turns out to be an interesting problem that illustrates many of the tools that we consider in this book.

Definition A *rise* is an occurrence of $p_i < p_{i+1}$. A *fall* is an occurrence of $p_{i-1} > p_i$. A *run* is a maximal increasing contiguous subsequence in the permutation. A *peak* is an occurrence of $p_{i-1} < p_i > p_{i+1}$. A *valley* is an occurrence of $p_{i-1} > p_i < p_{i+1}$. A *double rise* is an occurrence of $p_{i-1} < p_i < p_{i+1}$. A *double fall* is an occurrence of $p_{i-1} > p_i > p_{i+1}$. We use the notation $\text{runs}(p)$ to refer to the number of runs in a permutation p .

In any permutation, the number of rises plus the number of falls is the length minus 1. The number of runs is one greater than the number of falls, since every run except the last one in a permutation must end with a fall. These facts and others are clear if we consider a representation of $N - 1$ plus signs and minus signs corresponding to the sign of the difference between successive elements in the permutation; falls correspond to + and rises correspond to -.

permutation	9	14	4	1	12	2	10	13	5	6	11	3	8	15	7
rises/falls	-	+	+	-	+	-	-	+	-	-	+	-	-	-	+

Counting + and - characters, it is immediately clear that there are eight rises and six falls. Also, the plus signs mark the ends of runs (except the last), so there are seven runs. Double rises, valleys, peaks, and double falls correspond to occurrences of - -, + -, - +, and + + respectively. This permutation has three double rises, four valleys, five peaks, and one double fall.

Figure 7.2 is an intuitive graphical representation that also illustrates these quantities. When we draw a line connecting (i, p_i) to $(i + 1, p_{i+1})$ for $1 \leq i < N$, then rises go up, falls go down, peaks point upward, valleys point downward, and so forth. The figure also has an example of an “increasing subsequence”—a dotted line that connects points on the curve and rises as it moves from left to right.

Definition An *increasing subsequence* in a permutation is an increasing sequence of indices i_1, i_2, \dots, i_k with $p_{i_1} < p_{i_2} < \dots < p_{i_k}$.

By convention, the empty subsequence is considered to be “increasing.” For example, the increasing permutation 1 2 3 … N has 2^N increasing subsequences, one corresponding to every set of the indices, and the decreasing permutation N N-1 N-2 … 1 has just $N + 1$ increasing subsequences. We may account for the increasing subsequences in a permutation as we did for inversions: we keep a table $s_1 s_2 \dots s_N$ with s_i the number of increasing

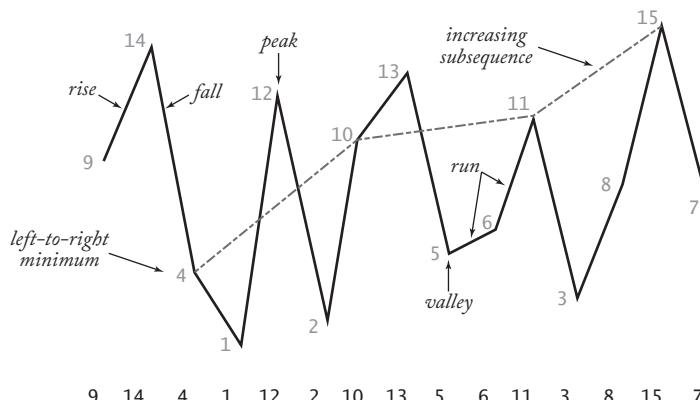


Figure 7.2 Anatomy of a permutation

subsequences that begin at position i . Our sample permutation has 9 increasing subsequences starting at position 1, 2 starting at position 2, and so forth.

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
permutation	9	14	4	1	12	2	10	13	5	6	11	3	8	15	7
subseq. table	9	2	33	72	4	34	5	2	8	7	2	5	2	1	1

For example, the fifth entry in this table corresponds to the four increasing subsequences 12, 12 13, 12 15, and 12 13 15. Adding the entries in this table (plus one for the empty subsequence) shows that the number of increasing subsequences in our sample permutation is 188.

Exercise 7.5 Write a program that computes the number of increasing subsequences in a given permutation in polynomial time.

Table 7.1 gives values for all of these properties for several random permutations of nine elements, and Table 7.2 gives their values for all of the permutations of four elements. Close examination of Tables 7.1 and 7.2 will reveal characteristics of these various properties of permutations and relationships among them that we will be proving in this chapter. For example, we have already mentioned that the distribution for the number of left-to-right maxima is the same as the distribution for the number of cycles.

Intuitively, we would expect rises and falls to be equally likely, so that there should be about $N/2$ of each in a random permutation of length N . Similarly, we would expect about half the elements to the left of each element to be larger, so the number of inversions should be about $\sum_{1 \leq i \leq N} i/2$, which

permutation	inversions	left-right minima	cycles	runs	longest cycle	inversion table	inverse
961534872	21	3	2	6	7	012233127	395642871
412356798	4	2	5	3	4	011100001	234156798
732586941	19	4	4	6	3	012102058	932846157
236794815	15	2	2	3	7	000003174	812693475
162783954	13	1	4	5	4	001003045	136982457
259148736	16	2	2	4	4	000321253	418529763

Table 7.1 Basic properties of some random permutations of nine elements

permutation	subseqs	inversions	left-right minima	cycles	longest cycle	runs	inversion table	inverse
1234	16	0	1	4	1	1	0000	1234
1243	14	1	1	3	2	2	0001	1243
1324	13	1	1	3	2	2	0010	1324
1342	10	2	1	2	3	2	0002	1423
1423	10	2	1	2	3	2	0011	1342
1432	8	3	1	3	3	3	0012	1432
2134	12	1	2	2	3	2	0100	2134
2143	10	2	2	2	2	3	0101	2143
2314	10	2	2	2	3	2	0020	3124
2341	9	3	2	1	4	2	0003	4123
2413	8	4	2	1	4	2	0013	3142
2431	7	4	2	2	3	3	0013	4132
3124	9	1	2	2	3	2	0010	2314
3142	8	3	2	1	4	3	0002	2413
3214	8	3	3	3	2	3	0120	3214
3241	7	4	3	2	3	3	0003	4213
3412	7	4	2	2	2	2	0022	3412
3421	6	5	3	1	4	3	0023	4312
4123	9	3	2	1	4	2	0111	2341
4132	8	4	2	2	3	3	0112	2431
4213	7	4	3	2	3	3	0122	3142
4231	6	5	3	3	2	3	0113	4231
4312	6	5	3	1	4	3	0122	3421
4321	5	6	4	2	2	4	0123	4321

Table 7.2 Basic properties of all permutations of four elements

is about $N^2/4$. We will see how to quantify these arguments precisely, how to compute other moments for these quantities, and how to study left-to-right minima and cycles using similar techniques.

Of course, if we ask more detailed questions, then we are led to more difficult analytic problems. For example, what proportion of permutations are involutions? Derangements? How many permutations have no cycles with more than three elements? How many have no cycles with fewer than three elements? What is the average value of the *maximum* element in the inversion table? What is the expected number of increasing subsequences in a permutation? What is the average length of the longest cycle in a permutation? The longest run? Such questions arise in the study of specific algorithms and have also been addressed in the combinatorics literature.

In this chapter, we answer many of these questions. Some of the average-case results that we will develop are summarized in Table 7.3. Some of these analyses are quite straightforward, but others require more advanced tools, as we will see when we consider the use of generating functions to derive these and other results throughout this chapter. We also will consider relationships to sorting algorithms in some detail.

	2	3	4	5	6	7	exact average	asymptotic estimate
permutations	2	6	24	120	720	5040	1	1
inversions	1	9	72	600	5400	52,920	$\frac{N(N-1)}{4}$	$\sim \frac{N^2}{4}$
left-right minima	3	11	50	274	1764	13,068	H_N	$\sim \ln N$
cycles	3	11	50	274	1764	13,068	H_N	$\sim \ln N$
rises	6	36	48	300	2160	17,640	$\frac{N-1}{2}$	$\sim \frac{N}{2}$
increasing subsequences	5	27	169	1217	7939	72,871	$\sum_{k \geq 0} \frac{1}{k!} \binom{N}{k}$	$\sim \frac{1}{2\sqrt{\pi e}} \frac{e^{2\sqrt{N}}}{N^{1/4}}$

Table 7.3 Cumulative counts and averages for properties of permutations

7.2 Algorithms on Permutations. Permutations, by their very nature, arise directly or indirectly in the analysis of a wide variety of algorithms. Permutations specify the way data objects are ordered, and many algorithms need to process data in some specified order. Typically, a complex algorithm will invoke a sorting procedure at some stage, and the direct relationship to sorting algorithms is motivation enough for studying properties of permutations in detail. We also consider a number of related examples.

Sorting. As we saw in Chapter 1, we very often assume that the input to a sorting method is a list of randomly ordered records with distinct keys. Keys in random order will in particular be produced by any process that draws them independently from an arbitrary continuous distribution. With this natural model, the analysis of sorting methods is essentially equivalent to the analysis of properties of permutations. Beginning with the comprehensive coverage in Knuth [10], there is a vast literature on this topic. A broad variety of sorting algorithms have been developed, appropriate for differing situations, and the analysis of algorithms has played an essential role in our understanding of their comparative performance. For more information, see the books by Knuth [10], Gonnet and Baeza-Yates [5], and Sedgewick and Wayne [15]. In this chapter, we will study direct connections between some of the most basic properties of permutations and some fundamental elementary sorting methods.

Exercise 7.6 Let a_1 , a_2 , and a_3 be “random” numbers between 0 and 1 produced independently as values of a random variable X satisfying the continuous distribution $F(x) = \Pr\{X \leq x\}$. Show that the probability of the event $a_1 < a_2 < a_3$ is $1/3!$. Generalize to any ordering pattern and any number of keys.

Rearrangement. One way to think of a permutation is as a specification of a rearrangement to be put into effect. This point of view leads to a direct connection with the practice of sorting. Sorting algorithms are often implemented to refer to the array being sorted indirectly: rather than moving elements around to put them in order, we compute the permutation that would put the elements in order. Virtually any sorting algorithm can be implemented in this way: for the methods we have seen, we maintain an “index” array $p[]$ that will contain the permutation.

For simplicity in this discussion, we maintain compatibility with our convention for specifying permutations by working with arrays of N items

indexed from 1 to N , even though modern programming languages index arrays of N items from 0 to $N-1$.

Initially, we set $p[i]=i$; then we modify the sorting code to refer to $a[p[i]]$ instead of $a[i]$ for any comparison, but to refer to p instead of a when doing any data movement. These changes ensure that, at any point during the execution of the algorithm, $a[p[1]], a[p[2]], \dots, a[p[N]]$ is identical to $a[1], a[2], \dots, a[N]$ in the original algorithm.

For example, if a sorting method is used in this way to put the sample input file

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys	29	41	77	26	58	59	97	82	12	44	63	31	53	23	93

into increasing order, it produces the permutation

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
permutation	9	14	4	1	12	2	10	13	5	6	11	3	8	15	7

as the result. One way of interpreting this is as instructions that the original input can be printed out (or accessed) in sorted order by first printing out the ninth element (12), then the fourteenth (23), then the fourth (26), then the first (29), and so on. In the present context, we note that the permutation computed is the inverse of the permutation that represents the initial ordering of the keys. For our example the following permutation results:

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
inverse	4	6	12	3	9	10	15	13	1	7	11	5	8	2	14

With this approach, sorting amounts to computing the inverse of a permutation. If an output array $b[1] \dots b[N]$ is available, the program that actually finishes a sort is trivial:

```
for (int i = 1; i <= N; i++) b[i] = a[p[i]]
```

For sorting applications where data movement is expensive (for example, when records are much larger than keys), this change can be very important. If space is not available for the output array, it is still possible to do the rearrangement “in place”—we will examine an algorithm for doing so later in this chapter.

It is amusing to note that, of course, the result of the sort can be an involution. For example, consider the input file

58 23 77 29 44 59 31 82 12 41 63 26 53 97 93.

For this file, the same permutation not only represents the initial ordering of the input file but also is the permutation that specifies how to rearrange the file to sort it. That is, the permutation

9 2 12 4 7 10 5 13 1 6 11 3 8 15 11

can be interpreted in two ways: not only is it the case that 58 is the ninth smallest element, 23 the second smallest, 77 the twelfth smallest, and so on, but it is also the case that the ninth element in the file (12) is the smallest, the second element (26) is the second smallest, the twelfth element (29) is the third smallest, and so on.

Random permutations. Many important applications involve *randomization*. For example, if the assumption that the inputs to a sorting algorithm are not randomly ordered is not necessarily justified, we can use the method shown in Program 7.1 to create a random ordering, and then sort the array. This randomization technique assumes a procedure that can produce a “random” integer in a given range (such programs are well studied; see Knuth [9]). Each of the $N!$ orderings of the input is equally likely to occur: the i th time through the loop, any one of i distinct rearrangements could happen, for a grand total of $2 \cdot 3 \cdot 4 \cdots N = N!$.

As mentioned in Chapter 1, from the standpoint of the analysis of algorithms, randomizing the order of the inputs in this way turns any sorting algorithm into a “probabilistic algorithm” whose performance characteristics are precisely described by the average-case results that we study. Indeed, this is one of the very first randomized algorithms, as it was proposed for quicksort by Hoare in 1960 (see Chapter 1).

```
public void exch(Item[] a, int i, int j)
{ Item t = a[i]; a[i] = a[j]; a[j] = t; }

for ( int i = 1; i < N; i++)
    exch(a, i, StdRandom.uniform(0, i));
```

Program 7.1 Randomly permuting an array

Priority queues. It is not always necessary that an algorithm rearrange its inputs into sorted order before it examines any of them. Another widely used option is to develop a data structure, consisting of records with keys, that supports two operations: *insert* a new item into the data structure and *remove smallest*, which retrieves the record with the smallest key from the data structure. Such a data structure is called a *priority queue*.

Priority queue algorithms are closely related to sorting algorithms: for example, we could use any priority queue algorithm to implement a sorting algorithm simply by inserting all the records, then removing them all. (They will come out in increasing order.) But priority queue algorithms are much more general, and they are widely used, both because the insert and remove operations can be intermixed, and because several other operations can be supported as well. The study of the properties of sophisticated priority queue algorithms is among the most important and most challenging areas of research in the analysis of algorithms. In this chapter, we will examine the relationship between a fundamental priority queue structure (heap-ordered trees) and binary search trees, which is made plain by the association of both with permutations.

7.3 Representations of Permutations. While it is normally most convenient to represent permutations as we have been doing—as rearrangements of the numbers 1 through N —many other ways to represent permutations are often appropriate. We will see that various different representations can show relationships among basic properties of permutations and can expose essential properties germane to some particular analysis or algorithm. Since there are $N!$ different permutations of N elements, any set of $N!$ different combinatorial objects might be used to represent permutations—we will examine several useful ones in this section.

Cycle structure. We have already considered the representation of permutations as a directed graph with sets of cycles, as illustrated in Figure 7.2. This representation is of interest because it makes the cycle structure obvious. Also, extending this representation leads to structures for general functions that have many interesting properties, which we will study in Chapter 9.

Foata's representation. We also have already considered Foata's representation, also illustrated in Figure 7.2, where we write cycles starting with their

smallest element and put the cycles in decreasing order of these elements. This representation establishes an important relationship between two properties of permutations (number of cycles and number of left-right minima) that would not otherwise seem to be related.

Exercise 7.7 Alternatively, we might put the cycles in decreasing order of their smallest elements, writing the smallest element in the cycle first. Give this representation for our sample permutation.

Exercise 7.8 Write a programs that will compute Foata's representation of a given permutation, and vice versa.

Inversion tables. A one-to-one correspondence between permutations and lists of N integers $q_1 q_2 \dots q_N$ with $0 \leq q_i < i$ is easy to establish. Given a permutation, its inversion table is such a list; and given such a list, the corresponding permutation can be constructed from right to left: for i decreasing from N down to 1, set p_i to be the q_i th largest of the integers not yet used. Consider the example:

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
inversion table	0	0	2	3	1	4	2	1	5	5	3	9	6	0	8
permutation	9	14	4	1	12	2	10	13	5	6	11	3	8	15	7

The permutation can be constructed from the inversion table by moving right to left: 7 is the eighth largest of the integers from 1 to 15, 15 is the largest of the remaining integers, 8 is the sixth largest of what's left, and so forth. There are $N!$ inversion tables (and permutations!) since there are i possibilities for the i th entry.

This correspondence is important in the analysis of algorithms because a random permutation is equivalent to a "random" inversion table, built by making its j th entry a random integer in the range 0 to $j - 1$. We will make use of this fact when setting up GFs for inversions and left-to-right maxima that nicely decompose in product form.

Exercise 7.9 Give an *efficient* algorithm for computing the inversion table corresponding to a given permutation, and another algorithm for computing the permutation corresponding to a given inversion table.

Exercise 7.10 Another way to define inversion tables is with q_i equal to the number of integers to the left of i in the permutation that are greater. Prove the one-to-one correspondence for this kind of inversion table.

Lattice representation. Figure 7.3 shows a two-dimensional representation that is useful for studying a number of properties of permutations: the permutation $p_1 p_2 \dots p_N$ is represented by labelling the cell at row p_i and column i with the number p_i for each i . Reading these numbers from right to left gives back the permutation. There is one label in each row and in each column, so each cell in the lattice corresponds to a unique pair of labels: the one in its row and the one in its column. If one member of the pair is below and the other to the right, then that pair is an inversion in the permutation, and the corresponding cell is marked in Figure 7.3. Note in particular that the diagram for the permutation and its inverse are simply transposes of each other—this is an elementary proof that every permutation has the same number of inversions as does its inverse.

Exercise 7.11 Show that the number of ways to place k mutually nonattacking rooks in an N -by- N chessboard is $\binom{N}{k}^2 k!$.

Exercise 7.12 Suppose that we mark cells whose marks are above and to the left in the lattice representation. How many cells are marked? Answer the same question for the other two possibilities (“above and right” and “below and left”).

Exercise 7.13 Show that the lattice representation of an involution is symmetric about the main diagonal.

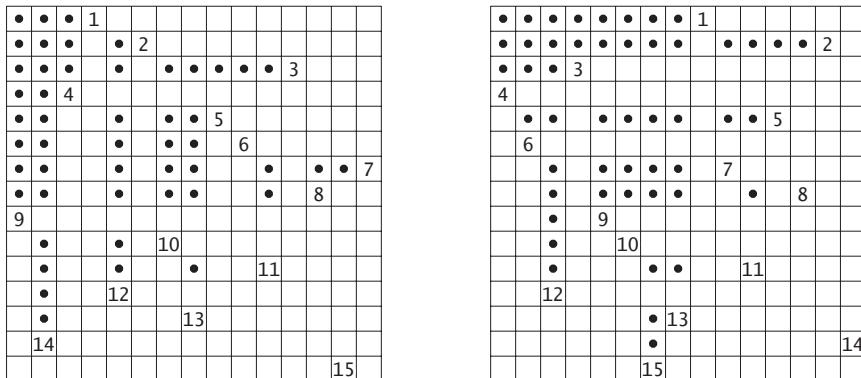


Figure 7.3 Lattice representation of a permutation and its inverse

Binary search trees. In Chapter 6, we analyzed properties of binary trees built by using Program 6.3 to insert keys from a random permutation successively into an initially empty tree, which implies a correspondence between permutations and BSTs (see Figure 6.14).

Figure 7.4 illustrates the correspondence in terms of a lattice representation: each label corresponds to a node with its row number as the key value, and left and right subtrees built from the parts of the lattice above and below the label, respectively. Specifically, the binary search tree corresponding to rows $l, l+1, \dots, r$, with the leftmost (lowest column number) mark in row k , is defined recursively by a node with key k , left subtree corresponding to rows $l, l+1, \dots, k-1$, and right subtree corresponding to rows $k+1, k+2, \dots, r$. Note that many permutations might correspond to the same binary search tree: interchanging columns corresponding to a node above and a node below any node will change the permutation, not the tree. Indeed, we know that the number of different binary tree structures is counted by the Catalan numbers, which are relatively small (about $4^N/N\sqrt{\pi N}$) compared to $N!$, so a large number of permutations certainly must correspond to each tree. The analytic results on BSTs in the previous chapter might be characterized as a study of the nature of this relationship.

Exercise 7.14 List five permutations that correspond to the BST in Figure 7.4.

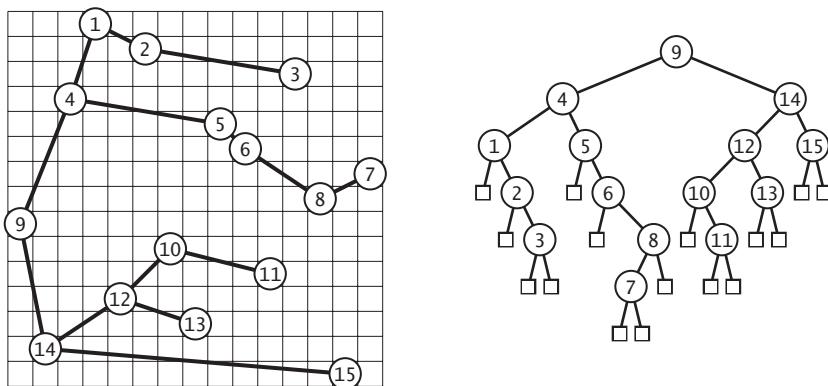


Figure 7.4 Binary search tree corresponding to a permutation

Heap-ordered trees. A tree can also be built from the lattice representation in a similar manner involving columns, as illustrated in Figure 7.5. These trees, in which the key at the root has a smaller key than the keys in the subtrees, are called *heap-ordered trees* (HOTs). In the present context, they are important because properties of permutations that are of interest are easily seen to be tree properties. For example, a node with two nonempty subtrees in the tree corresponds to a valley in the permutation, and leaves correspond to peaks.

Such trees are also of interest as explicit data structures, to implement priority queues. The smallest key is at the root, so the “remove smallest” operation can be implemented by (recursively) replacing the node at the root with the node at the root of the subtree with the smaller key. The “insert” operation can be implemented by (recursively) inserting the new node in the right subtree of the root, unless it has a smaller key than the node at the root, in which case it becomes the root, with the old root as its left subtree and a null right subtree.

Combinatorially, it is important to observe that an HOT is a complete encoding of a unique permutation (contrary to BSTs). See Vuillemin [20] for detailed information on HOTs, including other applications.

Direct counts of HOTs. To complete the cycle, it is instructive to consider the problem of enumerating HOTs directly (we already know that there are

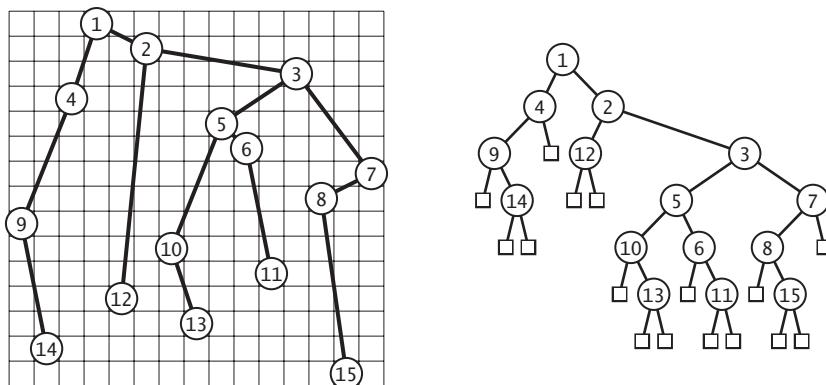


Figure 7.5 Heap-ordered tree corresponding to a permutation

$N!$ of them). Let \mathcal{H} be the set of HOTs and consider the EGF

$$H(z) = \sum_{t \in \mathcal{H}} \frac{z^{|t|}}{|t|!}.$$

Now, every HOT with $|t_L| + |t_R| + 1$ nodes can be constructed in a unique way by combining any HOT of size $|t_L|$ on the left with any HOT of size $|t_R|$ on the right, assigning the label “1” to the root, and assigning labels to the subtrees by dividing the $|t_L| + |t_R|$ labels into one set of size $|t_L|$ and one set of size $|t_R|$, sorting each set, and assigning them to the subtrees in order of their index. This is essentially a restatement of the labelled product construct described in Chapter 3. In terms of the EGF, this leads to the equation

$$H(z) = \sum_{t_L \in \mathcal{H}} \sum_{t_R \in \mathcal{H}} \binom{|t_L| + |t_R|}{|t_L|} \frac{z^{|t_L| + |t_R| + 1}}{(|t_L| + |t_R| + 1)!}.$$

Differentiating both sides immediately gives

$$H'(z) = H^2(z) \quad \text{or} \quad H(z) = 1 + \int_0^z H^2(t) dt.$$

This formula is also available directly via the symbolic method for labelled objects (see Chapter 3 and [4]). Two basic operations explain the formula: labelling the root 1 corresponds to integration, and combining two subtrees corresponds to a product. Now, the solution to the differential equation is $H(z) = 1/(1 - z)$, which checks with our knowledge that there are $N!$ HOTs on N nodes.

Similar computations will give access to statistics on parameters of permutations that are characterized on the HOT representation (e.g., peaks, valleys, rises, and falls), as introduced in the following exercises and discussed further in §7.5.

Exercise 7.15 Characterize the nodes in an HOT that correspond to rises, double rises, falls, and double falls in permutations.

Exercise 7.16 How many permutations are strictly alternating, with p_{i-1} and p_{i+1} either both less than or both greater than p_i for $1 < i < N$?

Exercise 7.17 List five permutations corresponding to the HOT in Figure 7.5.

Exercise 7.18 Let $K(z) = z/(1 - z)$ be the EGF for nonempty HOTs. Give a direct argument showing that $K'(z) = 1 + 2K(z) + K^2(z)$.

Exercise 7.19 Use an argument similar to HOT enumeration by EGF to derive the differential equation for the exponential CGF for internal path length in binary search trees (see the proof of Theorem 5.5 and §6.6).

How many permutations correspond to a given binary tree shape via the HOT or BST construction? This turns out to be expressed with a simple formula. Given a tree t , let $f(t)$ be the number of ways to label it as an HOT. By the argument just given, this function satisfies the recursive equation

$$f(t) = \binom{|t_L| + |t_R|}{|t_L|} f(t_L) f(t_R).$$

The same formula holds if f is counting the number of permutations corresponding to a BST since there is a unique label for the root ($|t_L| + 1$) and an unrestricted partition of the rest of the labels for the subtrees. Noting that the number of nodes in t is $|t_L| + |t_R| + 1$, and dividing both sides by that quantity, we get

$$\frac{f(|t_L| + |t_R| + 1)}{(|t_L| + |t_R| + 1)!} = \frac{1}{|t_L| + |t_R| + 1} \frac{f(t_L)}{|t_L|!} \frac{f(t_R)}{|t_R|!}.$$

This leads directly to the following theorem:

Theorem 7.1 (Frequency of HOTs and BSTs). The number of permutations corresponding to a tree t , as either an HOT or BST shape, is

$$f(t) = \frac{|t|!}{|u_1||u_2| \cdots |u_{|t|}|}$$

where $u_1, \dots, u_{|t|}$ are the subtrees rooted at each of the nodes of $|t|$. In other words, $f(t)$ is the number of ways to label t as an HOT using the labels $1 \dots |t|$ and the number of permutations of $1 \dots |t|$ that lead to a BST with the shape of t when built with the standard algorithm.

Proof. Iterate the recursive formula just given, using the initial condition that $f(t) = 1$ for $|t| = 1$. ■

For example, the number of ways to label the HOT in Figure 7.5 is $15!/(15 \cdot 3 \cdot 2 \cdot 11 \cdot 9 \cdot 5 \cdot 2 \cdot 2 \cdot 3) = 1223040$.

Notice that the fact that frequencies for HOTs and BSTs coincide is also to be expected combinatorially. The two correspondences exemplified by Figures 7.4 and 7.5 are structurally the same, with only the axes being interchanged. In other words, a stronger property holds: the BST corresponding to a permutation has the same shape as the HOT corresponding to the inverse permutation (but not the same labels).

Exercise 7.20 How many permutations correspond to the BST in Figure 7.4?

Exercise 7.21 Give the number of permutations that correspond to each of the BST shapes in Figure 6.1.

Exercise 7.22 Characterize the binary trees of size N to which the smallest number of permutations correspond and those to which the largest number of permutations correspond.

7.4 Enumeration Problems. Many of the problems listed in the introductory section of this chapter are enumeration problems. That is, we want to know the number of permutations that satisfy a certain property for various properties of interest. Equivalently, dividing by $N!$ gives the probability that a random permutation satisfies the property of interest. We work with the EGF that enumerates the permutations, which is equivalent to the OGF for the probability.

A variety of interesting properties of permutations can be expressed in terms of simple restrictions on the cycle lengths, so we begin by considering such problems in detail. We are interested in knowing, for a given parameter k , exact counts of the number of permutations having (i) cycles only of length equal to k , (ii) no cycles of length greater than k , and (iii) no cycles of length less than k . In this section, we develop analytic results for each of these enumeration problems. We already considered (ii) in §5.3; we will briefly review that analysis in context here. Table 7.4 gives the counts for $N \leq 10$ and $k \leq 4$. For $N \leq 4$, you can check the values against Figure 5.8.

Cycles of equal length. How many permutations of size N consist solely of cycles of length k ? We start with a simple combinatorial argument for $k = 2$. There are zero permutations consisting solely of doubleton cycles if N is odd, so we consider N even. We choose half the elements for the first element of

cycle lengths	1	2	3	4	5	6	7	8	9
all = 1	1	1	1	1	1	1	1	1	1
= 2	0	1	0	3	0	15	0	105	0
= 3	0	0	2	0	0	40	0	0	2240
= 4	0	0	0	6	0	0	0	1260	0
< 3 (involutions)	1	2	4	10	26	76	232	764	2620
< 4	1	2	6	18	66	276	1212	5916	31,068
< 5	1	2	6	24	96	456	2472	14,736	92,304
> 1 (derangements)	0	1	2	9	44	265	1854	14,833	133,496
> 2	0	0	2	6	24	160	1140	8988	80,864
> 3	0	0	0	6	24	120	720	6300	58,464
no restrictions	1	2	6	24	120	720	5040	40,320	362,880

Table 7.4 Enumeration of permutations with cycle length restrictions

each cycle, then assign the second element in each of $(N/2)!$ possible ways. The order of the elements is immaterial, so this counts each permutation $2^{N/2}$ times. This gives a total of

$$\binom{N}{N/2} (N/2)! / 2^{N/2} = \frac{N!}{(N/2)! 2^{N/2}}$$

permutations made up of cycles of length 2. Multiplying by z^N and dividing by $N!$ we get the EGF

$$\sum_{N \text{ even}} \frac{z^N}{(N/2)! 2^{N/2}} = e^{z^2/2}.$$

As we saw in Chapter 5, the symbolic method also gives this EGF directly: the combinatorial construction $SET(CYC_2(\mathcal{Z}))$ immediately translates to $e^{z^2/2}$ via Theorem 5.2 (and its proof). This argument immediately extends to show that the EGF for the number of permutations consisting solely of cycles of length k is $e^{z^k/k}$.

Derangements and lower bounds on cycle length. Perhaps the most famous enumeration problem for permutations is the *derangement problem*. Suppose that N students throw their hats in the air, then each catches a random hat. What is the probability that none of the hats go to their owners? This problem is equivalent to counting derangements—permutations with no singleton cycles—and immediately generalizes to the problem of counting the number of permutations with no cycles of length M or less. As we saw in §5.3 and §5.5, solving this problem is straightforward with analytic combinatorics. For completeness, we restate the result here.

Theorem 7.2 (Minimal cycle lengths). The probability that a random permutation of length N has no cycles of length M or less is $\sim e^{-H_M}$.

Proof. (Quick recap of the proof discussed at length in §5.3). Let $\mathcal{P}_{>M}$ be the class of permutations with no cycles of length M or less. The symbolic equation for permutations

$$\mathcal{P} = SET(CYC_1(\mathcal{Z})) \times SET(CYC_2(\mathcal{Z})) \times \dots \times SET(CYC_M(\mathcal{Z})) \times \mathcal{P}_{>M}$$

transfers immediately to the generating function equation

$$\frac{1}{1-z} = e^z e^{z^2/2} \cdots e^{z^M/M} P_{>M}(z).$$

which immediately gives the following EGF:

$$P_{>M}(z) = \frac{1}{1-z} e^{-z-z^2/2-z^3/3-\dots-z^M/M}.$$

The stated asymptotic result is a direct consequence of Theorem 5.5. ■

In particular, this theorem answers our original question: the probability that a random permutation of length N is a derangement is

$$[z^N] P_{>0}(z) = \sum_{0 \leq k \leq N} \frac{(-1)^k}{k!} \sim 1/e \approx .36787944.$$

Upper bounds on cycle length and involutions. Continuing in this manner, developing the EGF for the number of permutations that have a specified upper bound on cycle length is straightforward.

Theorem 7.3 (Maximal cycle lengths). The EGF that enumerates the number of permutations with no cycles of length greater than M is

$$\exp(z + z^2/2 + z^3/3 + \dots + z^M/M).$$

Proof. Immediate via the symbolic method: Let $\mathcal{P}_{\leq M}$ be the class of permutations with no cycle lengths larger than M . The symbolic equation for permutations

$$\mathcal{P}_{\leq M} = SET(CYC_1(\mathcal{Z})) \times SET(CYC_2(\mathcal{Z})) \times \dots \times SET(CYC_M(\mathcal{Z}))$$

transfers immediately to the stated EGF. ■

As mentioned briefly in the previous section, involutions can be characterized by cycle length restrictions, and thus enumerated by the argument just given. If $p_i = j$ in a permutation, then $p_j = i$ in the inverse: both of these must hold for $i \neq j$ in an involution, or the cycle (i, j) must be present. This observation implies that involutions consist of cycles of length 2 ($p_{p_i} = i$) or 1 ($p_i = i$). Thus involutions are precisely those permutations composed solely of singleton and doubleton cycles, and the EGF for involutions is $e^{z+z^2/2}$.

Theorem 7.4 (Involutions). The number of involutions of length N is

$$\sum_{0 \leq k \leq N/2} \frac{N!}{(N-2k)!2^kk!} \sim \frac{1}{\sqrt{2\sqrt{e}}} \left(\frac{N}{e}\right)^{N/2} e^{\sqrt{N}}.$$

Proof. From the preceding discussion, the associated EGF is $e^{z+z^2/2}$. The general analytic transfer theorem from analytic combinatorics to extract coefficients from such functions is the saddle point method from complex analysis, as described in [4]. The following sketch, using real analysis, is from Knuth [10]. First, the summation follows from the convolution

$$e^{z+z^2/2} = \sum_{j \geq 0} \frac{1}{j!} \left(z + \frac{z^2}{2}\right)^j = \sum_{j,k \geq 0} \frac{1}{j!} \binom{j}{k} z^{j-k} \left(\frac{z^2}{2}\right)^k$$

and collecting $[z^N]$. Next, use the Laplace method of Chapter 4. Taking the ratio of successive terms in the sum, we have

$$\frac{N!}{(N-2k)!2^kk!} / \frac{N!}{(N-2k-2)!2^{k+1}(k+1)!} = \frac{2(k+1)}{(N-2k)(N-2k-1)},$$

which shows that the terms in the sum increase until k is approximately $(N - \sqrt{N})/2$, then decrease. Using Stirling's approximation to estimate the dominant contribution near the peak and a normal approximation to bound the tails, the result follows in the same way as in various examples in Chapter 4. Details are worked out in [10]. ■

Direct derivation of involution EGF. It is also instructive to derive the exponential generating function for involutions directly. Every involution of length $|p|$ corresponds to (i) one involution of length $|p| + 1$, formed by appending the singleton cycle consisting of $|p| + 1$; and (ii) $|p| + 1$ involutions of length $|p| + 2$, formed by, for each k from 1 to $|p| + 1$, adding 1 to permutation elements greater than k , then appending the doubleton cycle consisting of k and $|p| + 2$. This implies that the EGF must satisfy

$$B(z) \equiv \sum_{\substack{p \in \mathcal{P} \\ p \text{ involution}}} \frac{z^{|p|}}{|p|!} = \sum_{\substack{p \in \mathcal{P} \\ p \text{ involution}}} \frac{z^{|p|+1}}{(|p|+1)!} + \sum_{\substack{p \in \mathcal{P} \\ p \text{ involution}}} (|p|+1) \frac{z^{|p|+2}}{(|p|+2)!}.$$

Differentiating, this simplifies to the differential equation

$$B'(z) = (1 + z)B(z)$$

which has the solution

$$B(z) = e^{z+z^2/2}$$

as expected. The differential equation is also available via the symbolic method (see [4]).

FOR REFERENCE, TABLE 7.5 LISTS THE solutions that are discussed here to enumeration problems for permutations with cycle length restrictions. All of the EGFs are easy to derive with the symbolic method; the asymptotic estimates require a range of techniques. Next, we move on to consider properties of permutations, using bivariate and cumulative generating functions.

EGF	asymptotic estimate of $N![z^N]$
singleton cycles	e^z 1
cycles of length M	$e^{z^M/M}$ —
all permutations	$\frac{1}{1-z}$ $N! \sim \sqrt{2\pi N} \left(\frac{N}{e}\right)^N$
derangements	$\frac{e^{-z}}{1-z}$ $\sim \frac{N!}{e}$
all cycles $> M$	$\frac{e^{-z-z^2/2-\dots-z^M/M}}{1-z}$ $\sim \frac{N!}{e^{H_M}}$
involutions	$e^{z+z^2/2}$ $\sim \frac{1}{\sqrt{2\sqrt{e}}} e^{\sqrt{N}} \left(\frac{N}{e}\right)^{N/2}$
all cycles $\leq M$	$e^{z+z^2/2+\dots+z^M/M}$ —

Table 7.5 EGFs for permutations with cycle length restrictions

Exercise 7.23 Show that the number of involutions of size N satisfies the recurrence

$$b_{N+1} = b_N + Nb_{N-1} \quad \text{for } N > 0 \text{ with } b_0 = b_1 = 1.$$

(This recurrence can be used, for example, to compute the entries on the row corresponding to involutions in Table 7.4.)

Exercise 7.24 Derive a recurrence that can be used to compute the number of permutations that have no cycle of length > 3 .

Exercise 7.25 Use the methods from §5.5 to derive a bound involving $N^{N(1-1/k)}$ for the number of permutations with no cycle of length greater than k .

Exercise 7.26 Find the EGF for the number of permutations that consist only of cycles of even length. Generalize to find the EGF for the number of permutations that consist only of cycles of length divisible by t .

Exercise 7.27 By differentiating the relation $(1 - z)D(z) = e^z$ and setting coefficients equal, obtain a recurrence satisfied by the number of derangements of N elements.

Exercise 7.28 Write a program to, given k , print a table of the number of permutations of N elements with no cycles of length $< k$ for $N < 20$.

Exercise 7.29 An *arrangement* of N elements is a sequence formed from a subset of the elements. Prove that the EGF for arrangements is $e^z/(1 - z)$. Express the coefficients as a simple sum and give a combinatorial interpretation of that sum.

7.5 Analyzing Properties of Permutations with CGFs. In this section, we outline the basic method that we will use for the analysis of properties of permutations for many of the problems given in this chapter, using cumulative generating functions (CGFs). Introduced in Chapter 3, this method may be summarized as follows:

- Define an exponential CGF of the form $B(z) = \sum_{p \in \mathcal{P}} \text{cost}(p)z^{|p|}/|p|!$.
- Identify a combinatorial construction and use it to derive a functional equation for $B(z)$.
- Solve the equation or use analytic techniques to find $[z^N]B(z)$.

The second step is accomplished by finding a correspondence among permutations, most often one that associates $|p| + 1$ permutations of length $|p| + 1$ with each of the $|p|!$ permutations of length $|p|$. Specifically, if \mathcal{P}_q is the set of permutations of length $|q| + 1$ that correspond to a given permutation q , then the second step corresponds to rewriting the CGF as follows:

$$B(z) = \sum_{q \in \mathcal{P}} \sum_{p \in \mathcal{P}_q} \text{cost}(p) \frac{z^{|q|+1}}{(|q|+1)!}.$$

The permutations in \mathcal{P}_q are typically closely related, and then the inner sum is easy to evaluate, which leads to an alternative expression for $B(z)$. Often, it is also convenient to differentiate to be able to work with $z^{|q|}/|q|!$ on the right-hand side.

For permutations, the analysis is also somewhat simplified because of the circumstance that the factorial for the exponential CGF also counts the total number of permutations, so the *exponential* CGF is also an *ordinary* GF for the average value sought. That is, if

$$B(z) = \sum_{p \in \mathcal{P}} \text{cost}(p)z^{|p|}/|p|!,$$

then it follows that

$$[z^N]B(z) = \sum_k k \{ \# \text{ of perms of length } N \text{ with cost } k \} / N!,$$

which is precisely the average cost. For other combinatorial structures it is necessary to divide the cumulative count obtained from the CGF by the total count to get the average, though there are other cases where the cumulative count has a sufficiently simple form that the division can be incorporated into the generating function. We will see another example of this in Chapter 8.

Combinatorial constructions. We consider several combinatorial constructions for permutations. Typically, we use these to derive a recurrence, a CGF or even a full BGF. BGFs give a stronger result since explicit knowledge about the BGF for a parameter implies knowledge about the distribution of values. CGFs lead to simpler computations because they are essentially average values, which can be manipulated without full knowledge of the distribution.

“First” or “last” construction. Given any one of the $N!$ different permutations of length N , we can identify $N + 1$ different permutations of length $N + 1$ by, for each k from 1 to N , prepending k and then incrementing all numbers larger than or equal to k . This defines the “first” correspondence. It is equivalent to the $\mathcal{P} = \mathcal{Z} \star \mathcal{P}$ construction that we used in §5.3 (see Figure 5.7). Alternatively, we can define a “last” construction that puts the new item at the end and is equivalent to $\mathcal{P} = \mathcal{P} \star \mathcal{Z}$ (see Figure 7.6).

“Largest” or “smallest” construction. Given any permutation p of length N , we can identify $N + 1$ different permutations of length $N + 1$ by putting the largest element in each of the $N + 1$ possible positions between elements of p . This defines the “largest” construction, illustrated in Figure 7.6. The figure uses a different type of star to denote the construction, to distinguish it from the \star that we use in analytic combinatorics. It is often possible to embrace such constructions within the symbolic method, but we will avoid confusion by refraining from doing so. We will also refrain from associating unique symbols with all the constructions that we consider, since there are so many possibilities. For example, we can base a similar construction on any other element, not just the largest, by renumbering the other elements appropriately. Using the smallest element involves adding one to each other element, then placing 1 in each possible position. We refer to this one as the “smallest” construction.

Binary search tree construction. Given two permutations p_l, p_r , we can create a total of

$$\binom{|p_l| + |p_r|}{|p_l|}$$

permutations of size $|p_l| + |p_r| + 1$ by (i) adding $|p_l| + 1$ to each element of p_r ; (ii) intermixing p_l and p_r in all possible ways; and (iii) prefixing each permutation so obtained by $|p_l| + 1$. As we saw in §6.3, all the permutations obtained in this way lead to the construction of the same binary search tree

“Last” construction

$$\begin{array}{l}
 \begin{array}{c} (1\ 2\ 3) \star (1) = \\ \quad \quad \quad \end{array} \quad \begin{array}{c} (1\ 2\ 3\ 2) \star (1) = \\ \quad \quad \quad \end{array} \\
 \begin{array}{c} (2\ 3\ 4\ 1) \\ (1\ 3\ 4\ 2) \\ (1\ 2\ 4\ 3) \\ (1\ 2\ 3\ 4) \end{array} \quad \quad \quad \begin{array}{c} (2\ 4\ 3\ 1) \\ (1\ 4\ 3\ 2) \\ (1\ 4\ 2\ 3) \\ (1\ 3\ 2\ 4) \end{array} \\
 \begin{array}{c} (2\ 1\ 3) \star (1) = \\ \quad \quad \quad \end{array} \quad \begin{array}{c} (2\ 3\ 1) \star (1) = \\ \quad \quad \quad \end{array} \\
 \begin{array}{c} (3\ 2\ 4\ 1) \\ (3\ 1\ 4\ 2) \\ (2\ 1\ 4\ 3) \\ (2\ 1\ 3\ 4) \end{array} \quad \quad \quad \begin{array}{c} (3\ 4\ 2\ 1) \\ (3\ 4\ 1\ 2) \\ (2\ 4\ 1\ 3) \\ (2\ 3\ 1\ 4) \end{array} \\
 \begin{array}{c} (3\ 1\ 2) \star (1) = \\ \quad \quad \quad \end{array} \quad \begin{array}{c} (3\ 2\ 1) \star (1) = \\ \quad \quad \quad \end{array} \\
 \begin{array}{c} (4\ 2\ 3\ 1) \\ (4\ 1\ 3\ 2) \\ (4\ 1\ 2\ 3) \\ (3\ 1\ 2\ 4) \end{array} \quad \quad \quad \begin{array}{c} (4\ 3\ 2\ 1) \\ (4\ 3\ 1\ 2) \\ (4\ 2\ 1\ 3) \\ (3\ 2\ 1\ 4) \end{array}
 \end{array}$$

“Largest” construction

$$\begin{array}{l}
 \begin{array}{c} (1\ 2\ 3\ 4) \\ (1\ 2\ 4\ 3) \\ (1\ 4\ 2\ 3) \\ (4\ 1\ 2\ 3) \end{array} \quad \begin{array}{c} (1\ 3\ 2\ 4) \\ (1\ 3\ 4\ 2) \\ (1\ 4\ 3\ 2) \\ (4\ 1\ 3\ 2) \end{array} \\
 \begin{array}{c} (2\ 1\ 3\ 4) \\ (2\ 1\ 4\ 3) \\ (2\ 4\ 1\ 3) \\ (4\ 2\ 1\ 3) \end{array} \quad \begin{array}{c} (2\ 3\ 1\ 4) \\ (2\ 3\ 4\ 1) \\ (2\ 4\ 3\ 1) \\ (4\ 2\ 3\ 1) \end{array} \\
 \begin{array}{c} (3\ 1\ 2\ 4) \\ (3\ 1\ 4\ 2) \\ (3\ 4\ 1\ 2) \\ (4\ 3\ 1\ 2) \end{array} \quad \begin{array}{c} (3\ 2\ 1\ 4) \\ (3\ 2\ 4\ 1) \\ (3\ 4\ 2\ 1) \\ (4\ 3\ 2\ 1) \end{array}
 \end{array}$$

Figure 7.6 Two combinatorial constructions for permutations

using the standard algorithm. Therefore this correspondence can be used as a basis for analyzing BST and related algorithms. (See Exercise 6.19.)

Heap-ordered tree construction. The combinatorial construction $\mathcal{P} \star \mathcal{P}$, used recursively, is useful for parameters that have a natural interpretation in terms of heap-ordered trees, as follows. The construction identifies “left” and “right” permutations. Adding 1 to each element and placing a 1 between the left permutation and the right permutation corresponds to an HOT.

NOTE THAT ALL OF these decompositions lead to differential equations in the CGFs because adding an element corresponds to shifting the counting sequences, which translates into differentiating the generating function (see Table 3.4). Next, we consider several examples.

Runs and rises. As a first example, consider the average number of runs in a random permutation. Elementary arguments given previously show that the average number of runs in a permutation of length N is $(N + 1)/2$, but the full distribution is interesting to study, as discovered by Euler (see [1], [7], and [10] for many details). Figure 7.7 illustrates this distribution for small values of N and k . The sequence for $k = 2$ is OEIS A000295 [18]; the full sequence is OEIS A008292. We start with the (exponential) CGF

$$A(z) = \sum_{p \in \mathcal{P}} \text{runs}(p) \frac{z^{|p|}}{|p|!}$$

and use the “largest” construction: if the largest element is inserted at the end of a run in p , there is no change in the number of runs; otherwise, the number of runs is increased by 1. The total number of runs in the permutations corresponding to a given permutation p is

$$(|p| + 1)\text{runs}(p) + |p| + 1 - \text{runs}(p) = |p|\text{runs}(p) + |p| + 1.$$

This leads to the alternative expression

$$A(z) = \sum_{p \in \mathcal{P}} (|p|\text{runs}(p) + |p| + 1) \frac{z^{|p|+1}}{(|p| + 1)!},$$

which simplifies considerably if we differentiate:

$$\begin{aligned} A'(z) &= \sum_{p \in \mathcal{P}} (|p|\text{runs}(p) + |p| + 1) \frac{z^{|p|}}{|p|!} \\ &= zA'(z) + \frac{z}{(1-z)^2} + \frac{1}{1-z}. \end{aligned}$$

Therefore $A'(z) = 1/(1-z)^3$, so, given the initial conditions,

$$A(z) = \frac{1}{2(1-z)^2} - \frac{1}{2}$$

and we have the anticipated result $[z^N]A(z) = (N+1)/2$.

We will be doing several derivations of this form in this chapter, because the CGF usually gives the desired results without much calculation. Still, it is instructive to note that the same construction often yields an explicit equation for the BGF, either directly through the symbolic method or indirectly via a recurrence. Doing so is worthwhile because the BGF carries full information about the distribution—in this case it is subject to “perturbation” methods from analytic combinatorics that ultimately show it to be asymptotically normal, as is apparent in Figure 7.7.

Theorem 7.5 (Eulerian numbers). Permutations of N elements with k runs are counted by the Eulerian numbers, A_{Nk} , with exponential BGF

$$A(z, u) \equiv \sum_{N \geq 0} \sum_{k \geq 0} A_{Nk} \frac{z^N}{N!} u^k = \frac{1-u}{1-ue^{z(1-u)}}.$$

Proof. The argument given for the CGF generalizes to provide a partial differential equation for the BGF using the “largest” construction. We leave

N	\downarrow	$k \rightarrow 1$	2	3	4	5	6	7	8	9	10
1		1									
2		1	1								
3		1	4	1							
4		1	11	11	1						
5		1	26	66	26	1					
6		1	57	302	302	57	1				
7		1	120	1191	2416	1191	120	1			
8		1	247	4293	15,619	15,619	4293	247	1		
9		1	502	14,608	88,234	156,190	88,234	14,608	502	1	
10		1	1013	47,840	455,192	1,310,354	1,310,354	455,192	47,840	1013	1

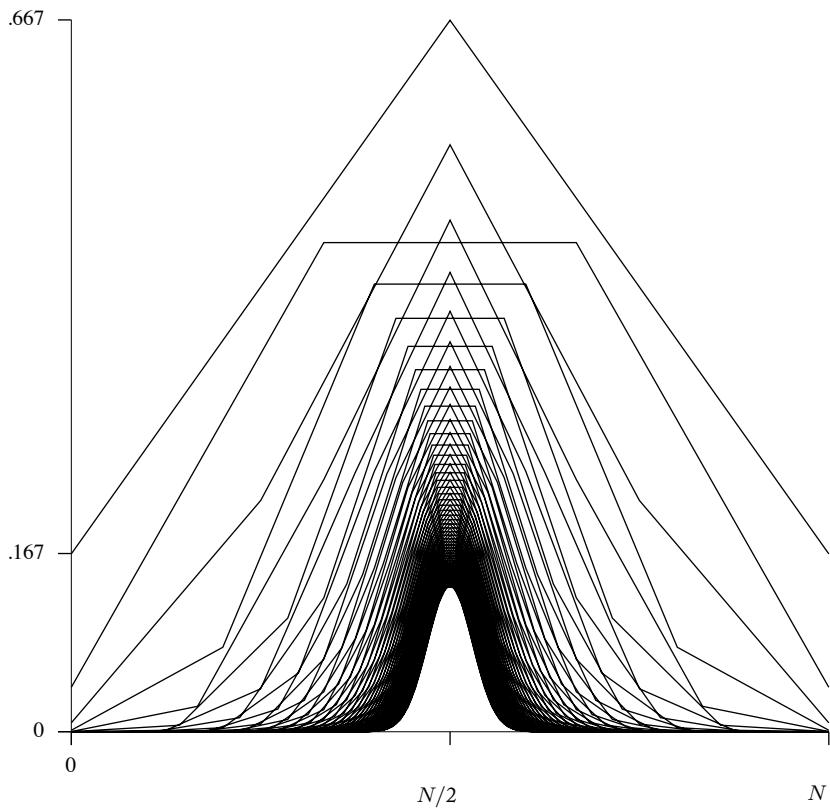


Figure 7.7 Distribution of runs in permutations (Eulerian numbers)

that for an exercise and consider the recurrence-based derivation, derived from the same construction. To get a permutation with k runs, there are k possibilities that the largest element is inserted at the end of an existing run in a permutation with k runs, and $N - k + 1$ possibilities that the largest element “breaks” an existing run of a permutation with $k - 1$ runs, thereby increasing the number of runs to k . This leads to

$$A_{Nk} = (N - k + 1)A_{(N-1)(k-1)} + kA_{(N-1)k},$$

which, together with the initial conditions $A_{00} = 1$ and $A_{N0} = 0$ for $N \geq 0$ or $k > N$, fully specifies the A_{Nk} .

Multiplying by $z^N u^k$ and summing on N and k leads directly to the partial differential equation

$$A_z(z, u) = \frac{1}{1 - uz}(uA(z, u) + u(1 - u)A_u(z, u)).$$

It is then easily checked that the stated expression for $A(z, u)$ satisfies this equation.

Corollary The average number of runs in a permutation of size $N > 1$ is $(N + 1)/2$ with variance $(N + 1)/12$.

Proof. We calculate the mean and variance as in Table 3.6, but taking into account that using an exponential BGF automatically includes division by $N!$, as usual for permutations. Thus, the mean is given by

$$[z^N] \frac{\partial A(z, u)}{\partial u} \Big|_{u=1} = [z^N] \frac{1}{2(1 - z)^2} = \frac{N + 1}{2},$$

and we can compute the variance in a similar manner. ■

As noted above, all runs but the last in a permutation are terminated by a fall, so Theorem 7.5 also implies that the number of falls in a permutation has mean $(N - 1)/2$ and variance $(N + 1)/12$. The same result also applies to the number of rises.

Exercise 7.30 Give a simple noncomputational proof that the mean number of rises in a permutation of N elements is $(N - 1)/2$. (*Hint:* For every permutation $p_1 p_2 \dots p_N$, consider the “complement” $q_1 q_2 \dots q_N$ formed by $q_i = N + 1 - p_i$.)

Exercise 7.31 Generalize the CGF argument given earlier to provide an alternative direct proof that the BGF $A(z, u) = \sum_{p \in \mathcal{P}} u^{\text{runs}(p)} z^{|p|}$ satisfies the partial differential equation given in the proof of Theorem 7.5.

Exercise 7.32 Prove that

$$A_{Nk} = \sum_{0 \leq j \leq k} (-1)^j \binom{N+1}{j} (k-j)^N.$$

Exercise 7.33 Prove that

$$x^N = \sum_{1 \leq k \leq N} A_{Nk} \binom{x+k-1}{N} \quad \text{for } N \geq 1.$$

Increasing subsequences. Another way to develop an explicit formula for a CGF is to find a recurrence on the cumulative cost. For example, let

$$S(z) = \sum_{p \in \mathcal{P}} \{ \# \text{ increasing subsequences in } p \} \frac{z^{|p|}}{|p|!} = \sum_{N \geq 0} S_N \frac{z^N}{N!}$$

so S_N represents the total number of increasing subsequences in all permutations of length N . Then, from the “largest” correspondence, we find that

$$S_N = NS_{N-1} + \sum_{0 \leq k < N} \binom{N-1}{k} (N-1-k)! S_k \quad \text{for } N > 0 \text{ with } S_0 = 1.$$

This accounts for the N copies of the permutation of length $N - 1$ (all the increasing subsequences in that permutation appear N times) *and* in a separate accounting, all the increasing subsequences ending in the largest element. If the largest element is in position $k + 1$, then all the permutations for each of the choices of elements for the first k positions appear $(N - 1 - k)!$ times (one for each arrangement of the larger elements), each contributing S_k to the total. This argument assumes that the empty subsequence is counted as “increasing,” as in our definition. Dividing by $N!$ and summing on N , we get the functional equation

$$(1 - z)S'(z) = (2 - z)S(z)$$

which has the solution

$$S(z) = \frac{1}{1-z} \exp\left(\frac{z}{1-z}\right).$$

The appropriate transfer theorem for extracting coefficients from such GFs involves complex-analytic methods (see [4]), but, as with involutions, it is a single convolution that we can handle with real analysis.

Theorem 7.6 (Increasing subsequences). The average number of increasing subsequences in a random permutation of N elements is

$$\sum_{0 \leq k \leq N} \binom{N}{k} \frac{1}{k!} \sim \frac{1}{2\sqrt{\pi e}} \frac{e^{2\sqrt{N}}}{N^{1/4}}.$$

Proof. The exact formula follows directly from the previous discussion, computing $[z^N]S(z)$ by convolving the two factors in the explicit formula just given for the generating function.

The Laplace method is effective for the asymptotic estimate of the sum. Taking the ratio of successive terms, we have

$$\binom{N}{k} \frac{1}{k!} / \binom{N}{k+1} \frac{1}{(k+1)!} = \frac{(k+1)^2}{N-k},$$

which shows that a peak occurs when k is about \sqrt{N} . As in several examples in Chapter 4, Stirling's formula provides the local approximations and the tails are bounded via a normal approximation. Details may be found in Lifschitz and Pittel [13]. ■

Exercise 7.34 Give a direct combinatorial derivation of the exact formula for S_N . (*Hint:* Consider all places at which an increasing subsequence may appear.)

Exercise 7.35 Find the EGF and an asymptotic estimate for the number of increasing subsequences of length k in a random permutation of length N (where k is fixed relative to N).

Exercise 7.36 Find the EGF and an asymptotic estimate for the number of increasing subsequences of length at least 3 in a random permutation of length N .

Peaks and valleys. As an example of the use of the heap-ordered tree decomposition of permutations, we now will derive results that refine the rise and run statistics. The nodes in an HOT are of three types: leaves (nodes with both children external), unary nodes (with one child internal and one external), and binary nodes (with both children internal). The study of the different types of nodes is directly relevant to the study of peaks and valleys in permutations (see Exercise 6.18). Moreover, these statistics are of independent interest because they can be used to analyze the storage requirements for HOTs and BSTs.

Given a heap-ordered tree, its associated permutation is obtained by simply listing the node labels in infix (left-to-right) order. In this correspondence, it is clear that a binary node in the HOT corresponds to a peak in the permutation: in a left-to-right scan, a binary node is preceded by a smaller element from its left subtree and followed by another smaller element from its right subtree. Thus the analysis of peaks in random permutations is reduced to the analysis of the number of binary nodes in random HOTs.

Binary nodes in HOTs. Using the symbolic method to analyze heap-ordered trees requires an additional construction that we have not covered (see [4], where HOTs are called “increasing binary trees”), but they are also easily handled with familiar tree recurrences. A random HOT of size N is composed of a left subtree of size k and a right subtree of size $N - k - 1$, where all values of k between 0 and $N - 1$ are equally likely and hence have probability $1/N$. This can be seen directly (the minimum of a permutation assumes each possible rank with equal likelihood) or via the HOT-BST equivalence. Mean values are thus computed by the same methods as those developed for BSTs in Chapter 6.

For example, the average number of binary nodes in a random HOT satisfies the recurrence

$$V_N = \frac{1}{N} \sum_{0 \leq k \leq N-1} (V_k + V_{N-k-1}) + \frac{N-2}{N} \quad \text{for } N \geq 3$$

since the number of binary nodes is the sum of the number of binary nodes in the left and right subtrees plus 1 unless the minimal element is the first or last in the permutation (an event that has probability $2/N$). We have seen this type of recurrence on several occasions, starting in §3.3. Multiplying by z^{N-1} and summing leads to the differential equation

$$V'(z) = 2 \frac{V(z)}{1-z} + \frac{z^2}{(1-z)^2},$$

which has the solution

$$V(z) = \frac{1}{3} \frac{z^3}{(1-z)^2} \quad \text{so that} \quad V_N = \frac{N-2}{3}.$$

Thus, the average number of valleys in a random permutation is $(N-2)/3$, and similar results about related quantities follow immediately.

Theorem 7.7 (Local properties of permutations and nodes in HOTs/BSTs). In a random permutation of N elements, the average numbers of valleys, peaks, double rises, and double falls are, respectively,

$$\frac{N-2}{3}, \quad \frac{N-2}{3}, \quad \frac{N+1}{6}, \quad \frac{N+1}{6}.$$

In a random HOT or BST of size N , the average numbers of binary nodes, leaves, left-branching, and right-branching nodes are, respectively,

$$\frac{N-2}{3}, \quad \frac{N+1}{3}, \quad \frac{N+1}{6}, \quad \frac{N+1}{6}.$$

Proof. These results are straightforward by arguments similar to that given above (or just applying Theorem 5.7) and using simple relationships among the various quantities. See also Exercise 6.15.

For example, a fall in a permutation is either a valley or a double fall, so the average number of double falls is

$$\frac{N-1}{2} - \frac{N-2}{3} = \frac{N+1}{6}.$$

For another example, we know that the expected number of leaves in a random BST is $(N+1)/3$ (or a direct proof such as the one cited for HOTs could be used) and the average number of binary nodes is $(N-2)/3$ by the argument above. Thus the average number of unary nodes is

$$N - \frac{N-2}{3} - \frac{N+1}{3} = \frac{N+1}{3},$$

with left- and right-branching nodes equally likely. ■

Table 7.7 summarizes the results derived above and some of the results that we will derive in the next three sections regarding the average values of various parameters for random permutations. Permutations are sufficiently simple combinatorial objects that we can derive some of these results in several ways, but, as the previous examples make clear, combinatorial proofs with BGFs and CGFs are particularly straightforward.

Exercise 7.37 Suppose that the space required for leaf, unary, and binary nodes is proportional to c_0 , c_1 , and c_2 , respectively. Show that the storage requirement for random HOTs and for random BSTs is $\sim (c_0 + c_1 + c_2)N/3$.

Exercise 7.38 Prove that valleys and peaks have the same distribution for random permutations.

Exercise 7.39 Under the assumption of the previous exercise, prove that the storage requirement for random binary Catalan trees is $\sim (c_0 + 2c_1 + c_2)N/4$.

Exercise 7.40 Show that a sequence of N random real numbers between 0 and 1 (uniformly and independently generated) has $\sim N/6$ double rises and $\sim N/6$ double

	exponential CGF	average ($[z^N]$)
left-to-right minima	$\frac{1}{1-z} \ln \frac{1}{1-z}$	H_N
cycles	$\frac{1}{1-z} \ln \frac{1}{1-z}$	H_N
singleton cycles	$\frac{z}{1-z}$	1
cycles = k	$\frac{z^k}{k} \frac{1}{1-z}$	$\frac{1}{k}$ ($N \geq k$)
cycles $\leq k$	$\frac{1}{1-z} \left(z + \frac{z^2}{2} + \dots + \frac{z^k}{k} \right)$	H_k ($N \geq k$)
runs	$\frac{1}{2(1-z)^2} - \frac{1}{2}$	$\frac{N+1}{2}$
inversions	$\frac{z^2}{2(1-z)^3}$	$\frac{N(N-1)}{4}$
increasing subsequences	$\frac{1}{1-z} \exp\left(\frac{z}{1-z}\right)$	$\sim \frac{1}{2\sqrt{\pi e}} \frac{e^{2\sqrt{N}}}{N^{1/4}}$
peaks, valleys	$\frac{z^3}{3(1-z)^2}$	$\frac{N-2}{3}$

Table 7.7 Analytic results for properties of permutations (average case)

falls, on the average. Deduce a direct continuous-model proof of this asymptotic result.

Exercise 7.41 Generalize Exercise 6.18 to show that the BGF for right-branching nodes and binary nodes in HOTs satisfies

$$K_z(z, u) = 1 + (1 + u)K(z, u) + K^2(z, u)$$

and therefore

$$K(z, u) = \frac{1 - e^{(u-1)z}}{u - e^{(u-1)z}}.$$

(Note: This provides an alternative derivation of the BGF for Eulerian numbers, since $A(z, u) = 1 + uK(z, u)$.)

7.6 Inversions and Insertion Sorts. Program 7.2 is an implementation of *insertion sort*, a simple sorting method that is easily analyzed. In this method, we “insert” each element into its proper position among those previously considered, moving larger elements over one position to make room. The left portion of Figure 7.8 shows the operation of Program 7.2 on a sample array of distinct keys, mapped to a permutation. The highlighted elements in the i th line in the figure are the elements moved to do the i th insertion.

The running time of insertion sort is proportional to $c_1N + c_2B + c_3$, where c_1 , c_2 , and c_3 are appropriate constants that depend on the implementation and B , a function of the input permutation, is the number of exchanges. The number of exchanges to insert each element is the number of larger elements to the left, so we are led directly to consider inversion tables. The right portion of Figure 7.8 is the inversion table for the permutation as the sort proceeds. After the i th insertion (shown on the i th line), the first i elements in the inversion table are zero (because the first i elements of the permutation

```
for (int i = 1; i < N; i++)
    for (int j = i; j >= 1 && a[j-1] > a[j]; j--)
        exch(a, j, j-1);
```

Program 7.2 Insertion sort

JC	PD	CN	AA	MC	AB	JG	MS	EF	HF	JL	AL	JB	PL	HT
9	14	4	1	12	2	10	13	5	6	11	3	8	15	7
9	14	4	1	12	2	10	13	5	6	11	3	8	15	7
4	9	14	1	12	2	10	13	5	6	11	3	8	15	7
1	4	9	14	12	2	10	13	5	6	11	3	8	15	7
1	4	9	12	14	2	10	13	5	6	11	3	8	15	7
1	2	4	9	12	14	10	13	5	6	11	3	8	15	7
1	2	4	9	10	12	14	13	5	6	11	3	8	15	7
1	2	4	9	10	12	13	14	5	6	11	3	8	15	7
1	2	4	5	9	10	12	13	14	6	11	3	8	15	7
1	2	4	5	6	9	10	12	13	14	11	3	8	15	7
1	2	4	5	6	9	10	11	12	13	14	3	8	15	7
1	2	3	4	5	6	8	9	10	11	12	13	14	15	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
AA	AB	AL	CN	EF	HF	HT	JB	JC	JG	JL	MC	MS	PD	PL

Figure 7.8 Insertion sort and inversions

are sorted), and the next element in the inversion table specifies how many elements are going to be moved in the next insertion, because it specifies the number of larger elements to the left of the $(i + 1)$ st element. The only effect of the i th insertion on the inversion table is to zero its i th entry. This implies that the value of the quantity B when insertion sort is run on a permutation is equal to the sum of the entries in the inversion table—the total number of inversions in the permutation.

Exercise 7.42 How many permutations of N elements have exactly one inversion? Two? Three?

Exercise 7.43 Show how to modify insertion sort to also compute the inversion table for the permutation associated with the original ordering of the elements.

As mentioned previously, there is a one-to-one correspondence between permutations and inversion tables. In any inversion table $q_1 q_2 \dots q_N$, each entry q_i must be between 0 and $i - 1$ (in particular, q_1 is always 0). There are i possible values for each of the q_i , so there are $N!$ different inversion tables. Inversion tables are simpler to use in the analysis because their entries are independent: each q_i takes on its i different values independent of the values of the other entries.

Theorem 7.8 (Inversion distribution). The number of permutations of size N with k inversions is

$$[u^k] \prod_{1 \leq k \leq N} \frac{1 - u^k}{1 - u} = [u^k](1 + u)(1 + u + u^2) \cdots (1 + u + \dots + u^{N-1}).$$

A random permutation of N elements has $N(N - 1)/4$ inversions on the average, with standard deviation $N(N - 1)(2N + 5)/72$.

Proof. We present the derivation using PGFs; a combinatorial derivation would follow along (almost) identical lines.

In the inversion table for a random permutation, the i th entry can take on each value between 0 and $i - 1$ with probability $1/i$, independently of the other entries. Thus, the probability generating function for the number of inversions involving the N th element is $(1 + u + u^2 + \dots + u^{N-1})/N$, independent of the arrangement of the previous elements. As discussed in Chapter 3, the PGF for the sum of independent random variables is the product of the individual PGFs, so the generating function for the total number of inversions in a random permutation of N elements satisfies

$$b_N(u) = \frac{1 + u + u^2 + \dots + u^{N-1}}{N} b_{N-1}(u).$$

That is, the number of inversions is the sum, for j from 1 to N , of independent uniformly distributed random variables with OGF $(1 + u + u^2 + \dots + u^{j-1})/j$. The counting GF of the theorem statement equals $N!$ times $b_N(u)$. The average is the sum of the individual averages $(j - 1)/2$, and the variance is the sum of the individual variances $(j^2 - 1)/12$. ■

The full distribution $[u^k]b_N(u)$ is shown in Figure 7.9. The curves are symmetric about $N(N - 1)/4$, and they shrink toward the center (albeit slowly) as N grows. This curve can be characterized as the distribution for the sum of independent random variables. Though they are not identically distributed, it can be shown by the classical Central Limit Theorem of probability theory that the distribution is normal in the limit (see, for example, David and Barton [2]). This outcome is not atypical in the analysis of algorithms (see, for example, Figure 7.7); indeed, such results are common in the BGF-based limit laws of analytic combinatorics (see [4]).

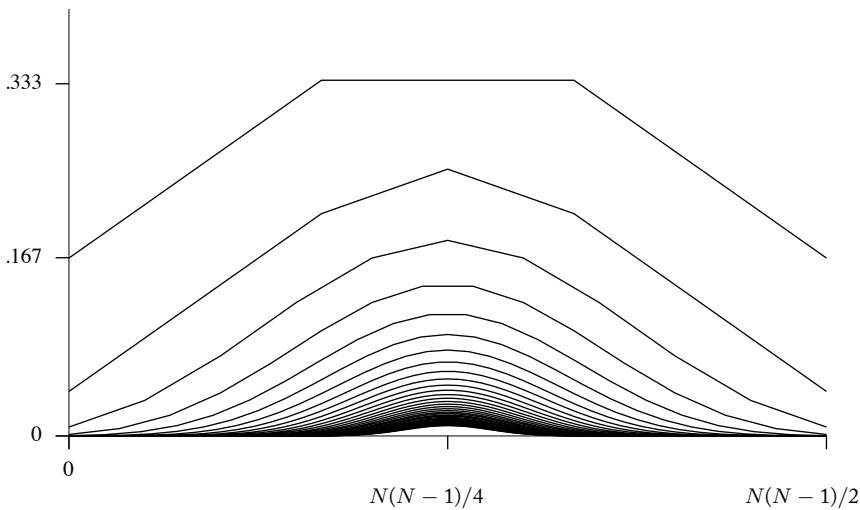


Figure 7.9 Distribution of inversions, $3 \leq N \leq 60$ (k -axes scaled to $\binom{N}{2}$)

Corollary Insertion sort performs $\sim N^2/4$ comparisons and $\sim N^2/4$ moves, on the average, to sort a file of N randomly ordered records with distinct keys.

Solution with CGFs. The proof of Theorem 7.8 calculates the cumulated cost from the “horizontal” generating functions for inversions; we now consider an alternative derivation that uses the CGF directly. Consider the CGF

$$B(z) = \sum_{p \in \mathcal{P}} \text{inv}(p) \frac{z^{|p|}}{|p|!}.$$

As mentioned previously, the coefficient of $z^N/N!$ in $B(z)$ is the total number of inversions in all permutations of length N , so that $B(z)$ is the OGF for the average number of inversions in a permutation.

Using the “largest” construction, every permutation of length $|p|$ corresponds to $|p| + 1$ permutations of length $|p| + 1$, formed by putting element $|p| + 1$ between the k th and $(k+1)$ st element, for k between 0 and $|p|$. Such a

permutation has $|p| - k$ more inversions than p , which leads to the expression

$$B(z) = \sum_{p \in \mathcal{P}} \sum_{0 \leq k \leq |p|} (\text{inv}(p) + |p| - k) \frac{z^{|p|+1}}{(|p|+1)!}.$$

The sum on k is easily evaluated, leaving

$$B(z) = \sum_{p \in \mathcal{P}} \text{inv}(p) \frac{z^{|p|+1}}{|p|!} + \sum_{p \in \mathcal{P}} \binom{|p|+1}{2} \frac{z^{|p|+1}}{(|p|+1)!}.$$

The first sum is $zB(z)$, and the second is simple to evaluate because it depends only on the length of the permutation, so the $k!$ permutations of length k can be collected for each k , leaving

$$B(z) = zB(z) + \frac{z}{2} \sum_{k \geq 0} kz^k = zB(z) + \frac{1}{2} \frac{z^2}{(1-z)^2},$$

so $B(z) = z^2/(2(1-z)^3)$, the GF for $N(N-1)/4$, as expected.

WE WILL BE STUDYING other properties of inversion tables later in the chapter, since they can describe other properties of permutations that arise in the analysis of some other algorithms. In particular, we will be concerned with the number entries in the inversion table that are at their maximum value (for selection sort) and the value of the largest element (for insertion sort).

Exercise 7.44 Derive a recurrence relation satisfied by p_{Nk} , the probability that a random permutation of N elements has exactly k inversions.

Exercise 7.45 Find the CGF for the total number of inversions in all involutions of length N . Use this to find the average number of inversions in an involution.

Exercise 7.46 Show that $N!p_{Nk}$ is a fixed polynomial in N for any fixed k , when N is sufficiently large.

Shellsort. Program 7.3 gives a practical improvement to insertion sort, called shellsort, which reduces the running time well below N^2 by making several passes through the file, each time sorting h independent subfiles (each of size about N/h) of elements spaced by h . The sequence of “increments” $h[t], h[t-1], \dots, h[1]$ that control the sort is usually chosen to be decreasing and must end in 1. Though it is a simple extension to insertion sort, shellsort has proved to be extremely difficult to analyze (see [16]).

In principle, mathematical analysis should guide us in choosing an increment sequence, but the average-case analysis of shellsort remains an unsolved problem, even for simple increment sequences that are widely used in practice such as $\dots, 364, 121, 40, 13, 4, 1$. Yao [22] has done an analysis of $(h, k, 1)$ shellsort using techniques similar to those we used for insertion sort, but the results and methods become much more complicated. For general shellsort, not even the functional form of the order of growth of the running time is known for any practical increment sequence.

Two-ordered permutations. The analysis of shellsort for the case where h takes on only the values 2 and 1 is interesting to consider because it is closely related to the analysis of path length in trees of Chapter 6. This is equivalent to a merging algorithm: the files in odd- and even-numbered positions are sorted independently (with insertion sort), then the resulting permutation is sorted with insertion sort. Such a permutation, which consists of two interleaved sorted permutations, is said to be *2-ordered*. Properties of 2-ordered permutations are of interest in the study of other merging algorithms as well. Since the final pass of shellsort, with $h=1$, is just insertion sort, its average

```

for (int k = 0; k < incs.length; k++)
{
    int h = incs[k];
    for (int i = h; i < N; i++)
        for (int j = i; j >= h && a[j-h] > a[j]; j--)
            exch(a, j, j-h);
}

```

Program 7.3 Shellsort

running time will depend on the average number of inversions in a 2-ordered permutation. Three sample 2-ordered permutations, and their inversion tables, are given in Table 7.8.

Let $S(z)$ be the OGF that enumerates 2-ordered permutations. It is obvious that

$$S(z) = \sum_{N \geq 0} \binom{2N}{N} z^N = \frac{1}{\sqrt{1 - 4z}},$$

but we will consider an alternative method of enumeration to expose the structure. Figure 7.10 illustrates the fact that 2-ordered permutations correspond to paths in an N -by- N lattice, similar to those described for the “gambler’s ruin” representation of trees in Chapter 5. Starting at the upper left corner, move right if i is in an odd-numbered position and down if i is in an even-numbered position. Since there are N moves to the right and N moves down, we end up in the lower right corner.

Now, the lattice paths that do not touch the diagonal correspond to trees, as discussed in Chapter 5, and are enumerated by the generating function

$$zT(z) = G(z) = (1 - \sqrt{1 - 4z})/2.$$

For 2-ordered permutations, the restriction on touching the diagonal is removed. However, any path through the lattice must touch the diagonal for the first time, which leads to the symbolic equation

$$S(z) = 2G(z)S(z) + 1$$

for the enumerating OGF for 2-ordered permutations. That is, any path through the lattice can be uniquely constructed from an initial portion that

4	1	5	2	6	3	9	7	10	8	13	11	15	12	16	14	19	17	20	18
0	1	0	2	0	3	0	1	0	2	0	1	0	2	0	2	0	1	0	2
1	4	2	5	3	6	8	7	9	12	10	13	11	14	17	15	18	16	19	20
0	0	1	0	2	0	0	1	0	0	1	0	2	0	0	1	0	2	0	0
4	1	5	2	6	3	7	8	12	9	13	10	14	11	15	17	16	18	20	19
0	1	0	2	0	3	0	0	1	0	2	0	3	0	0	1	0	0	1	

Table 7.8 Three 2-ordered permutations, with inversion tables

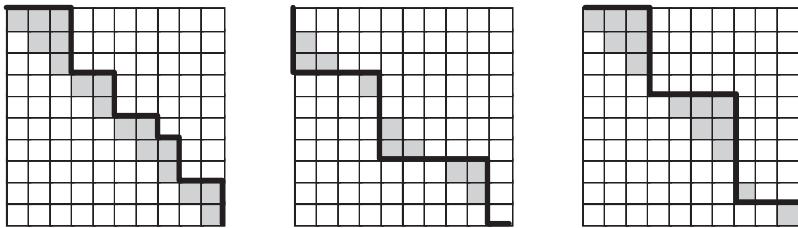


Figure 7.10 Lattice paths for 2-ordered permutations in Table 7.8

does not touch the diagonal except at the endpoints, followed by a general path. The factor of 2 accounts for the fact that the initial portion may be either above or below the diagonal. This simplifies to

$$S(z) = \frac{1}{1 - 2G(z)} = \frac{1}{1 - (1 - \sqrt{1 - 4z})} = \frac{1}{\sqrt{1 - 4z}},$$

as expected. Knuth [10] (see also Vitter and Flajolet [19]) shows that this same general structure can be used to write explicit expressions for the BGF for inversions, with the eventual result that the cumulative cost (total number of inversions in all 2-ordered permutations of length $2N$) is simply $N4^{N-1}$. The argument is based on the observation that the number of inversions in a 2-ordered permutation is equal to the number of lattice squares between the corresponding lattice path and the “down-right-down-right...” diagonal.

Theorem 7.9 (Inversions in 2-ordered permutations). The average number of inversions in a random 2-ordered permutation of length $2N$ is

$$N4^{N-1} / \binom{2N}{N} \sim \sqrt{\pi/128} (2N)^{3/2}.$$

Proof. The calculations that lead to this simple result are straightforward but intricate and are left as exercises. We will address this problem again, in a more general setting, in §8.5. ■

Corollary The average number of comparisons used by (2, 1) shellsort on a file of N elements is $N^2/8 + \sqrt{\pi/128} N^{3/2} + O(N)$.

Proof. Assume that N is even. The first pass consists of two independent sorts of $N/2$ elements and therefore involves $2((N/2)(N/2 - 1)/4) = N^2/8 + O(N)$ comparisons, and leaves a random 2-ordered file. Then an additional $\sqrt{\pi/128} N^{3/2}$ comparisons are used during the second pass.

The same asymptotic result follows for the case when N is odd. Thus, even though it requires two passes over the file, (2, 1) shellsort uses a factor of 2 fewer comparisons than insertion sort. ■

Exercise 7.47 Show that the number of inversions in a 2-ordered permutation is equal to the number of lattice squares between the path and the “down-right-down-right...” diagonal.

Exercise 7.48 Let \mathcal{T} be the set of all 2-ordered permutations, and define the BGF

$$P(z, u) = \sum_{p \in \mathcal{T}} u^{\{\# \text{ inversions in } p\}} \frac{z^{|p|}}{|p|!}.$$

Define $Q(z, u)$ in the same way, but restricted to the set of 2-ordered permutations whose corresponding lattice paths do not touch the diagonal except at the endpoints. Moreover, define $S(z, u)$ and $T(z, u)$ similarly, but restricted to 2-ordered permutations whose corresponding lattice paths lie entirely above the diagonal except at the endpoints. Show that $P(z, u) = 1/(1 - Q(z, u))$ and $S(z, u) = 1/(1 - T(z, u))$.

Exercise 7.49 Show that $T(z, u) = uzS(uz, u)$ and $Q(uz, u) = T(uz, u) + T(z, u)$.

Exercise 7.50 Using the result of the previous two exercises, show that

$$S(z, u) = uzS(uz, u)S(uz, u) + 1$$

and

$$P(z, u) = (uzS(uz, u) + zS(z, u))P(z, u) + 1.$$

Exercise 7.51 Using the result of the previous exercise, show that

$$P_u(1, z) = \frac{z}{(1 - 4z)^2}.$$

Exercise 7.52 Give an asymptotic formula for the average number of inversions in a 3-ordered permutation, and analyze shellsort for the case when the increments are 3 and 1. Generalize to estimate the leading term of the cost of $(h, 1)$ shellsort, and the asymptotic cost when the best value of h is used (as a function of N).

Exercise 7.53 Analyze the following sorting algorithm: given an array to be sorted, sort the elements in odd positions and in even positions recursively, then sort the resulting 2-ordered permutation with insertion sort. For which values of N does this algorithm use fewer comparisons, on the average, than the pure recursive quicksort of Chapter 1?

7.7 Left-to-Right Minima and Selection Sort. The trivial algorithm for finding the minimum element in an array is to scan through the array, from left to right, keeping track of the minimum found so far. By successively finding the minimum, we are led to another simple sorting method called *selection sort*, shown in Program 7.4. The operation of selection sort on our sample file is diagrammed in Figure 7.11: again, the permutation is shown on the left and the corresponding inversion table on the right.

Finding the minimum. To analyze selection sort, we first need to analyze the algorithm for “finding the minimum” in a random permutation: the first ($i = 0$) iteration of the outer loop of Program 7.4. As for insertion sort, the running time of this algorithm can be expressed in terms of N and a quantity whose value depends on the particular permutation—in this case the number of times the “current minimum” is updated (the number of exchanges in Program 7.4 while $i = 0$). This is precisely the number of left-to-right minima in the permutation.

Foata’s correspondence gives a 1-1 correspondence between left-to-right minima and cycles, so our analysis of cycles in §5.4 tells us that the average

```

for (int i = 0; i < N-1; i++)
{
    int min = i;
    for (int j = i+1; j < N; j++)
        if (a[j] < a[min]) min = j;
        exch(a, i, min);
}

```

Program 7.4 Selection sort

JC	PD	CN	AA	MC	AB	JG	MS	EF	HF	JL	AL	JB	PL	HT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
9	14	4	1	12	2	10	13	5	6	11	3	8	15	7	0	0	2	3	1	4	2	1	5	5	3	9	6	0	8	
1	14	4	9	12	2	10	13	5	6	11	3	8	15	7	0	0	1	1	4	2	1	5	5	3	9	6	0	8		
1	2	4	9	12	14	10	13	5	6	11	3	8	15	7	0	0	0	0	0	2	1	5	5	3	9	6	0	8		
1	2	3	9	12	14	10	13	5	6	11	4	8	15	7	0	0	0	0	0	2	1	5	5	3	8	6	0	8		
1	2	3	4	12	14	10	13	5	6	11	9	8	15	7	0	0	0	0	0	2	1	4	4	3	4	6	0	8		
1	2	3	4	5	14	10	13	12	6	11	9	8	15	7	0	0	0	0	0	1	1	2	4	3	4	6	0	8		
1	2	3	4	5	6	10	13	12	14	11	9	8	15	7	0	0	0	0	0	0	0	1	0	3	5	6	0	8		
1	2	3	4	5	6	7	13	12	14	11	9	8	15	10	0	0	0	0	0	0	0	0	1	0	3	4	5	0	8	
1	2	3	4	5	6	7	8	12	14	11	9	13	15	10	0	0	0	0	0	0	0	0	0	0	1	2	3	1	0	5
1	2	3	4	5	6	7	8	9	14	11	12	13	15	10	0	0	0	0	0	0	0	0	0	1	1	1	0	5	0	
1	2	3	4	5	6	7	8	9	10	11	12	13	15	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	
1	2	3	4	5	6	7	8	9	10	11	12	13	15	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	
1	2	3	4	5	6	7	8	9	10	11	12	13	15	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	
1	2	3	4	5	6	7	8	9	10	11	12	13	15	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	
1	2	3	4	5	6	7	8	9	10	11	12	13	15	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	
AA	AB	AL	CN	EF	HF	HT	JB	JC	JG	JL	MC	MS	PD	PL	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Figure 7.11 Selection sort and left-to-right minima

number of left-to-right minima in a random permutation of N elements is H_N . The following direct derivation is also of interest.

The number of left-to-right minima is not difficult to analyze with the help of the inversion table: each entry in the inversion table $q_1 q_2 \dots q_N$ for which $q_i = i - 1$ corresponds to a left-to-right minimum, since all elements to the left are larger (for example, this condition holds for q_1 , q_3 , and q_4 in the first line of the right part of Figure 7.11). Therefore, each entry in the inversion table is a left-to-right minimum with probability $1/i$, independent of the other entries, so the average is $\sum_{1 \leq i \leq N} 1/i = H_N$. A slight generalization of this argument gives the PGF.

Theorem 7.10 (Left-to-right minima distribution). Permutations of N elements with k left-to-right minima are counted by the Stirling numbers of the first kind:

$$\begin{bmatrix} N \\ k \end{bmatrix} = [u^k] u(u+1) \dots (u+N-1).$$

A random permutation of N elements has H_N left-to-right minima on the average, with variance $H_N - H_N^{(2)}$.

Proof. Consider the probability generating function $P_N(u)$ for the number of left-to-right minima in a random permutation of N elements. As earlier, we can decompose this into two independent random variables: one for a random permutation of $N - 1$ elements (with PGF $P_{N-1}(u)$) and one for the contribution of the last element (with PGF $(N - 1 + u)/N$, since the last element adds 1 to the number of left-to-right minima with probability $1/N$, 0 otherwise). Thus we must have

$$P_N(u) = \frac{N - 1 + u}{N} P_{N-1}(u),$$

and, as earlier, we find the mean and variance of the number of left-to-right minima by summing the means and variance from the simple probability generating functions $(z + k - 1)/k$. The counting GF equals $N! p_N(u)$. ■

Solution with CGFs. As usual, we introduce the exponential CGF

$$B(z) = \sum_{p \in \mathcal{P}} \text{lrm}(p) \frac{z^{|p|}}{|p|!}$$

so that $[z^N]B(z)$ is the average number of left-to-right minima in a random permutation of N elements. As before, we can directly derive a functional equation, in this case using the “last” construction. Of the $|p|+1$ permutations of size $|p| + 1$ that we construct from a given permutation p , one of them ends in 1 (and so has one more left-to-right minimum than p), and $|p|$ do not end in 1 (and so have the same number of left-to-right minima as p). This observation leads to the formulation

$$\begin{aligned} B(z) &= \sum_{p \in \mathcal{P}} (\text{lrm}(p) + 1) \frac{z^{|p|+1}}{(|p| + 1)!} + \sum_{p \in \mathcal{P}} |p| \text{lrm}(p) \frac{z^{|p|+1}}{(|p| + 1)!} \\ &= \sum_{p \in \mathcal{P}} \text{lrm}(p) \frac{z^{|p|+1}}{|p|!} + \sum_{p \in \mathcal{P}} \frac{z^{|p|+1}}{(|p| + 1)!} \\ &= zB(z) + \sum_{k \geq 0} \frac{z^{k+1}}{(k+1)!} = zB(z) + \ln \frac{1}{1-z}, \end{aligned}$$

which leads to the solution

$$B(z) = \frac{1}{1-z} \ln \frac{1}{1-z},$$

the generating function for the harmonic numbers, as expected. This derivation can be extended, with just slightly more work, to give an explicit expression for the exponential BGF describing the full distribution.

Stirling numbers of the first kind. Continuing this discussion, we start with

$$B(z, u) = \sum_{p \in \mathcal{P}} \frac{z^{|p|}}{|p|!} u^{\text{lrm}(p)} = \sum_{N \geq 0} \sum_{k \geq 0} p_{Nk} z^N u^k$$

where p_{Nk} is the probability that a random permutation of N elements has k left-to-right minima. The same combinatorial construction as used earlier leads to the formulation

$$B(z, u) = \sum_{p \in \mathcal{P}} \frac{z^{|p|+1}}{(|p|+1)!} u^{\text{lrm}(p)+1} + \sum_{p \in \mathcal{P}} \frac{z^{|p|+1}}{(|p|+1)!} |p| u^{\text{lrm}(p)}.$$

Differentiating with respect to z , we have

$$\begin{aligned} B_z(z, u) &= \sum_{p \in \mathcal{P}} \frac{z^{|p|}}{|p|!} u^{\text{lrm}(p)+1} + \sum_{p \in \mathcal{P}} \frac{z^{|p|}}{(|p|-1)!} u^{\text{lrm}(p)} \\ &= uB(z, u) + zB_z(z, u). \end{aligned}$$

Solving for $B_z(z, u)$, we get a simple first-order differential equation

$$B_z(z, u) = \frac{u}{1-z} B(z, u),$$

which has the solution

$$B(z, u) = \frac{1}{(1-z)^u}$$

(since $B(0, 0) = 1$). Differentiating with respect to u , then evaluating at $u = 1$ gives the OGF for the harmonic numbers, as expected. Expanding

$$B(z, u) = 1 + \frac{u}{1!} z + \frac{u(u+1)}{2!} z^2 + \frac{u(u+1)(u+2)}{3!} z^3 + \dots$$

gives back the expression for the Stirling numbers of the first kind in the statement of Theorem 7.10.

The BGF $B(z, u) = (1 - z)^{-u}$ is a classical one that we saw in §5.4. As we know from Foata's correspondence, the number of permutations of N elements with exactly k left-to-right minima is the same as the number of permutations of N elements with exactly k cycles. Both are counted by the Stirling numbers of the first kind, which are therefore sometimes called the Stirling "cycle" numbers. This distribution is OEIS A130534 [18], and is shown in Figure 7.12.

Selection sort. The leading term in the running time of selection sort is the number of comparisons, which is $(N + 1)N/2$ for every input permutation, and the number of exchanges is $N - 1$ for every input permutation. The only quantity whose value is dependent on the input in the running time of Program 7.4 is the total number of left-to-right minima encountered during the sort: the number of times the `if` statement succeeds.

Theorem 7.11 (Selection sort). Selection sort performs $\sim N^2/2$ comparisons, $\sim N \ln N$ minimum updates, and moves $\sim N$ moves, on the average, to sort a file of N records with randomly ordered distinct keys.

Proof. See the preceding discussion for comparisons and exchanges. It remains to analyze B_N , the expected value of the total number of left-to-right minima encountered during the sort for a random permutation of N elements. In Figure 7.12, it is obvious that the leftmost i elements of the inversion table are zero after the i th step, but the effect on the rest of the inversion table is more difficult to explain. The reason for this is that the passes in selection sort are *not* independent: after we complete one pass, the part of the permutation that we process in the next pass is very similar (certainly not random), as it differs only in one position, where we exchanged away the minimum.

We can use the following construction to find B_N : given a permutation p of N elements, increment each element and prepend 1 to construct a permutation of $N + 1$ elements, then construct N additional permutations by exchanging the 1 with each of the other elements. Now, if any of these $N + 1$ permutations is the initial input to the selection sort algorithm, the result will be equivalent to p for subsequent iterations. This correspondence implies that

$$B_N = B_{N-1} + H_N = (N + 1)H_N - N.$$

N	$k \rightarrow$	1	2	3	4	5	6	7	8	9	10
1		1									
2		1	1								
3		2	3	1							
4		6	11	6	1						
5		24	50	35	10	1					
6		120	274	225	85	15	1				
7		720	1764	1624	735	175	21	1			
8		5040	13,068	13,132	6769	1960	322	28	1		
9		40,320	109,584	118,124	67,284	22,449	4536	546	36	1	
10		362,880	1,026,576	1,172,700	723,680	269,325	63,273	9450	870	45	1

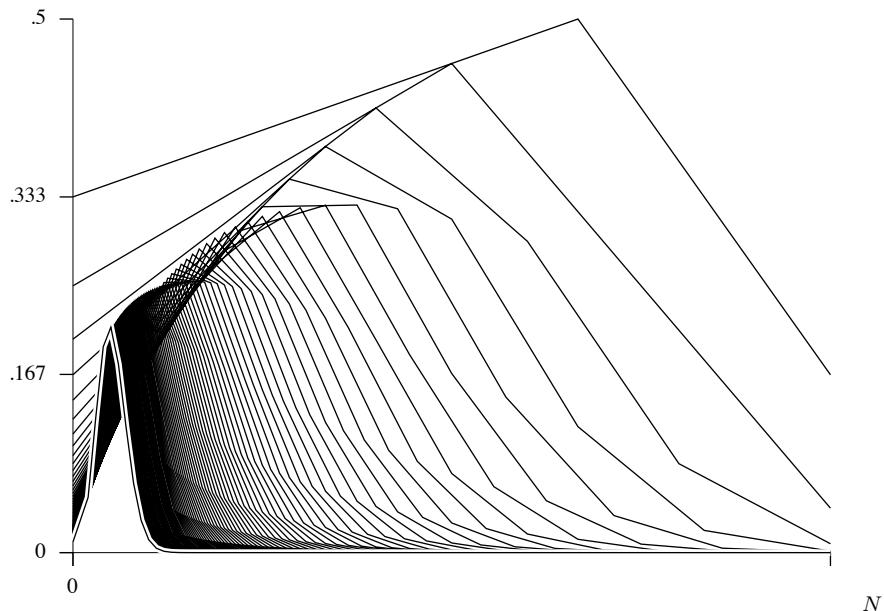


Figure 7.12 Distribution of left-to-right minima and cycles
(Stirling numbers of the first kind)

More specifically, let $\text{cost}(p)$ denote the total number of left-to-right minima encountered during the sort for a given permutation p , and consider the CGF

$$B(z) = \sum_{N \geq 0} B_N z^N = \sum_{p \in \mathcal{P}} \text{cost}(p) \frac{z^{|p|}}{|p|!}.$$

The construction defined above says that the algorithm has uniform behavior in the sense that each permutation costs $\text{lrm}(p)$ for the first pass; then, if we consider the result of the first pass (applied to all $|p|!$ possible inputs), each permutation of size $|p| - 1$ appears the same number of times. This leads to the solution

$$\begin{aligned} B(z) &= \sum_{p \in \mathcal{P}} \text{lrm}(p) \frac{z^{|p|}}{|p|!} + \sum_{p \in \mathcal{P}} (|p| + 1) \text{cost}(p) \frac{z^{|p|+1}}{(|p| + 1)!} \\ &= \frac{1}{1-z} \ln \frac{1}{1-z} + zB(z). \end{aligned}$$

Therefore,

$$B(z) = \frac{1}{(1-z)^2} \ln \frac{1}{1-z},$$

which is the generating function for partial sums of the harmonic numbers. Thus $B_N = (N+1)(H_{N+1}-1)$ as in Theorem 3.4 and therefore $B_N \sim N \ln N$, completing the proof. ■

This proof does *not* extend to yield the variance or other properties of this distribution. This is a subtle but important point. For left-to-right minima and other problems, we are able to transform the CGF derivation easily into a derivation for the BGF (which yields the variance), but the above argument does not extend in this way because the behavior of the algorithm on one pass may provide information about the next pass (for example, a large number of left-to-right minima on the first pass would imply a large number of left-to-right minima on the second pass). The lack of independence seems to make this problem nearly intractable: it remained open until 1988, when a delicate analysis by Yao [21] showed the variance to be $O(N^{3/2})$.

Exercise 7.54 Let p_{Nk} be the probability that a random permutation of N elements has k left-to-right minima. Give a recurrence relation satisfied by p_{Nk} .

Exercise 7.55 Prove directly that $\sum_k k \binom{N}{k} = N!H_N$.

Exercise 7.56 Specify and analyze an algorithm that determines, in a left-to-right scan, the *two* smallest elements in an array.

Exercise 7.57 Consider a situation where the cost of accessing records is 100 times the cost of accessing keys, and both are large by comparison with other costs. For which values of N is selection sort preferred over insertion sort?

Exercise 7.58 Answer the previous question for quicksort versus selection sort, assuming that an “exchange” costs twice as much as a “record access.”

Exercise 7.59 Consider an implementation of selection sort for linked lists, where on each iteration, the smallest remaining element is found by scanning the “input” list, but then it is *removed* from that list and appended to an “output” list. Analyze this algorithm.

Exercise 7.60 Suppose that the N items to be sorted actually consist of arrays of N words, the first of which is the sort key. Which of the four comparison-based methods that we have seen so far (quicksort, mergesort, insertion sort, and selection sort) adapts best to this situation? What is the complexity of this problem, in terms of the amount of input data?

7.8 Cycles and In Situ Permutation. In some situations, an array might need to be permuted “in place.” As described in §7.2, a sorting program can be organized to refer to records indirectly, computing a permutation that specifies how to do the arrangement instead of actually rearranging them. Here, we consider how the rearrangement might be done in place, in a second phase. Here is an example:

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
input keys	CN	HF	MC	AL	JC	JG	PL	MS	AA	HT	JL	EF	JB	AB	PD
permutation	9	14	4	1	12	2	10	13	5	6	11	3	8	15	7

That is, to put the array in sorted order, $a[9]$ has to be moved to position 1, $a[14]$ to position 2, $a[4]$ to position 3, and so on. One way to do this is to start by saving $a[1]$ in a register, replace it by $a[p[1]]$, set j to $p[1]$, and continue until $p[j]$ becomes 1, when $a[j]$ can be set to the saved value. This process is then repeated for each element not yet moved, but if we permute the p array in the same way, then we can easily identify elements that need to be moved, as in Program 7.5.

In the example above, first $a[9] = AA$ is moved to position 1, then $a[5] = JC$ is moved to position 9, then $a[12] = EF$ is moved to position 5, then $a[3] = MC$ is moved to position 12, then $a[4] = AL$ is moved to

```

for (int i = 1; i <= N; i++)
    if (p[i] != i)
    {
        int j, t = a[i], k = i;
        do
        {
            j = k; a[j] = a[p[j]];
            k = p[j]; p[j] = j;
        } while (k != i);
        a[j] = t;
    }
}

```

Program 7.5 In situ permutation

position 3, then CN, which was originally in position 1, is put in position 4. At the same time, $p[1]$ is set to 1, $p[9]$ is set to 9, and so on, to reflect in the permutation the fact that all of these elements are now in place. Then, the program increments i to find elements to move, ignoring elements that are already in place, with $p[i] = i$. This program moves each element at most once.

Clearly, the running time of Program 7.5 depends on the cycle structure of the permutation—the program itself is actually a succinct demonstration of the concept of cycles. Note also that every item that is on a cycle of length greater than 1 is moved precisely once, so the number of data moves is N minus the number of singleton cycles. We also need to know the number of cycles, since the overhead of the inner loop is incurred once for every cycle. Now, we know from §5.4 that the average number of cycles is H_N and we know the full distribution from our earlier analysis of left-to-right minima, by Foata's correspondence (see §7.1), but a direct derivation is also instructive.

Theorem 7.12 (Cycle distribution). The distribution of the number of cycles in a random permutation is the same as the distribution of the number of left-to-right minima, and is given by the Stirling numbers of the first kind. The average number of cycles in a random permutation is H_N , with standard deviation $H_N - H_N^{(2)}$.

Proof. As usual, we can directly derive a functional equation for the bivariate generating function

$$B(z, u) = \sum_{p \in \mathcal{P}} z^{|p|} u^{\text{cyc}(p)}$$

where $\text{cyc}(p)$ is the number of cycles in p . from a combinatorial construction. Given a permutation p , we can create $|p| + 1$ permutations of size $|p| + 1$ by adding element $|p| + 1$ at every position in every cycle (including the “null” cycle). Of these permutations, one has one more cycle than p and $|p|$ have the same number of cycles as p . This correspondence is structurally the same as the correspondence that we established for left-to-right minima. Using precisely the same argument as given previously, we of course find that the generating function is identical to the generating function for the number of left-to-right minima:

$$B(z, u) = \frac{1}{(1-z)^u}.$$

Therefore the distributions are identical: the average is as stated; the number of permutations of N elements with exactly k cycles is given by the Stirling numbers of the first kind, and so on. ■

Cycle lengths. To analyze Program 7.5 in detail, we have noted that we need to know the number of singleton cycles. We know from the enumeration results on permutations with cycle length restrictions in §7.4 that the probability that a random permutation has at least one singleton cycle is $1 - 1/e$, but how many might we expect to find? It turns out that the number of singleton cycles in a random permutation is 1, on the average, and that we can analyze other facts about the cycle length distribution.

Singleton cycles. For comparison with other derivations, we find the average number of singleton cycles using the combinatorial construction given earlier and the CGF

$$B(z) = \sum_{p \in \mathcal{P}} \text{cyc}_1(p) \frac{z^{|p|}}{|p|!}$$

where $\text{cyc}_1(p)$ is the number of singleton cycles in a permutation p , so that the desired answer is $[z^N]B(z)$. By the earlier construction, the permutations of length $|p|$ can be grouped into groups of size $|p|$, each of which corresponds to a permutation q of size $|p| - 1$. In the group corresponding to q , one has $\text{cyc}_1(q) + 1$ singleton cycles, $\text{cyc}_1(q)$ have $\text{cyc}_1(q) - 1$ singleton cycles, and the rest have $\text{cyc}_1(q)$ singleton cycles, for a total of

$\text{cyc}_1(q) + 1 + \text{cyc}_1(q)(\text{cyc}_1(q) - 1) + (|p| - 1 - \text{cyc}_1(q))\text{cyc}_1(q) = 1 + |q|\text{cyc}_1(q)$ singleton cycles for all of the permutations corresponding to q . Therefore,

$$B(z) = \sum_{q \in \mathcal{P}} (1 + |q|\text{cyc}_1(q)) \frac{z^{|q|+1}}{(|q| + 1)!}.$$

Differentiating this formula leads to the simple form

$$\begin{aligned} B'(z) &= \sum_{q \in \mathcal{P}} \frac{z^{|q|}}{|q|!} + \sum_{q \in \mathcal{P}} |q|\text{cyc}_1(q) \frac{z^{|q|}}{|q|!} \\ &= \frac{1}{1-z} + zB'(z), \end{aligned}$$

so $B'(z) = 1/(1-z)^2$ and $B(z) = 1/(1-z)$, as expected.

Cycles of length k . The symbolic method for parameters gives average number of cycles of length k in a random permutation. By adapting the arguments in §7.4, we can write down the (exponential) BGF for the number of cycles of length k :

$$\exp\left(z + \frac{z^2}{2} + \frac{z^3}{3} + \dots + \frac{z^{k-1}}{k-1} + \frac{z^k}{k}u + \frac{z^{k+1}}{k+1} + \dots\right).$$

When this is expanded, each term represents a permutation, where the exponent of u counts the number of times the term z^k/k is used, or the number of cycles of length k in the corresponding permutation. Now, the BGF can be rewritten in the form

$$\begin{aligned} & \exp\left(z + \frac{z^2}{2} + \dots + \frac{z^k}{k} + \dots\right) \exp\left((u-1)\frac{z^k}{k}\right) \\ &= \frac{1}{1-z} \exp((u-1)z^k/k). \end{aligned}$$

This form allows calculation of the quantities of interest.

Theorem 7.13 (Singleton cycle distribution). The probability that a permutation of N elements has j singleton cycles is asymptotic to $e^{-1}/j!$. The average number of cycles of length k in a random permutation of size $N \geq k$ is $1/k$, with variance $1/k$.

Proof. The probability sought is given by the coefficients of the BGF derived in the above discussion:

$$\begin{aligned} [u^j z^N] \frac{e^{(u-1)z}}{1-z} &= \frac{1}{j!} [z^{N-j}] \frac{e^{-z}}{1-z} \\ &\sim \frac{e^{-1}}{j!}, \end{aligned}$$

by Theorem 7.3.

The computation of the average is a simple application of Theorem 3.6: differentiating with respect to u and evaluating at 1 gives

$$[z^N] \frac{z^k}{k} \frac{1}{1-z} = \frac{1}{k} \quad \text{for } N \geq k,$$

and the variance follows from a similar calculation. ■

We might also sum on j to generalize Theorem 7.11 and derive the result that the average number of cycles of length $\leq k$ in a random permutation of N elements is H_k . Of course, this result holds only as long as k is not larger than N .

Exercise 7.61 Use asymptotics from generating functions (see §5.5) or a direct argument to show that the probability for a random permutation to have j cycles of length k is asymptotic to the Poisson distribution $e^{-\lambda} \lambda^j / j!$ with $\lambda = 1/k$.

Exercise 7.62 For a permutation of length 100, what is the probability that the loop in Program 7.5 never iterates more than 50 times?

Exercise 7.63 [Knuth] Consider a situation where the permutation array cannot be modified and no other extra memory is available. An algorithm to perform in situ permutation can be developed as follows: the elements in each cycle will be permuted when the smallest index in the cycle is encountered. For j from 1 to N , test each index to see if it is the smallest in the cycle by starting with $k = j$ and setting $k = p[k]$ while $k > j$. If it is, then permute the cycle as in Program 7.5. Show that the BGF for the number of times this $k = p[k]$ instruction is executed satisfies the functional equation

$$B_u(z, u) = B(z, u)B(z, zu).$$

From this, find the mean and variance for this parameter of random permutations. (See [12].)

7.9 Extremal Parameters. In Chapter 5, we found that tree height was much more difficult to analyze than path length because calculating the height involves taking the maximum subtree values, whereas path length involves just enumeration and addition, and the latter operations correspond more naturally to operations on generating functions. In this section, we consider analogous parameters on permutations. What is the average length of the longest or shortest cycle in a permutation? What is the average length of the longest run? The longest increasing subsequence? What is the average value of the largest element in the inversion table of a random permutation? This last question arises in the analysis of yet another elementary sorting algorithm, to which we now turn.

Bubble sort. This method is simple to explain: to sort an array, pass through it repeatedly, exchanging each element with the next to put them in order, if necessary. When a pass through the array is completed without any exchanges (each element is not larger than the next), the sort is completed. An implementation is given in Program 7.6. To analyze this algorithm, we need to count the exchanges and the passes.

Exchanges are straightforward: each exchange is with an adjacent element (as in insertion sort), and so removes exactly one inversion, so the total number of exchanges is exactly the number of inversions in the permutation. The number of passes used is also directly related to the inversion table, as shown in Figure 7.14: each pass actually reduces each nonzero entry in the inversion table by 1, and the algorithm terminates when there are no more nonzero entries. This implies immediately that the number of passes required to bubble sort a permutation is precisely equal to the largest element in the inversion table. The distribution of this quantity is shown in Figure 7.15. The sequence is OEIS A056151 [18].

```
for (int i = N-1; i > 1; i--)
    for (int j = 1; j <= i; j++)
        if (a[j-1] > a[j]) exch(a, j-1, j);
```

Program 7.6 Bubble sort

JC	PD	CN	AA	MC	AB	JG	MS	EF	HF	JL	AL	JB	PL	HT
9	14	4	1	12	2	10	13	5	6	11	3	8	15	7
9	4	1	12	2	10	13	5	6	11	3	8	14	7	15
4	1	9	2	10	12	5	6	11	3	8	13	7	14	15
1	4	2	9	10	5	6	11	3	8	12	7	13	14	15
1	2	4	9	5	6	10	3	8	11	7	12	13	14	15
1	2	4	5	6	9	3	8	10	7	11	12	13	14	15
1	2	4	5	6	3	8	9	7	10	11	12	13	14	15
1	2	4	5	3	6	8	7	9	10	11	12	13	14	15
1	2	4	3	5	6	7	8	9	10	11	12	13	14	15
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
AA	AB	AL	CN	EF	HF	HT	JB	JC	JG	JL	MC	MS	PD	PL

0	0	2	3	1	4	2	1	5	5	3	9	6	0	8
0	1	2	0	3	1	0	4	4	2	8	5	0	7	0
0	1	0	2	0	0	3	3	1	7	4	0	6	0	0
0	0	1	0	0	2	2	0	6	3	0	5	0	0	0
0	0	0	0	1	1	0	5	2	0	4	0	0	0	0
0	0	0	0	0	4	1	0	3	0	0	0	0	0	0
0	0	0	0	0	3	0	0	2	0	0	0	0	0	0
0	0	0	0	2	0	0	1	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 7.13 Bubble sort (permutation and associated inversion table)

Theorem 7.14 (Maximum inversion table entry). The largest element in the inversion table of a random permutation has mean value $\sim N - \sqrt{\pi N/2}$.

Proof. The number of inversion tables of length N with all entries less than k is simply $k!k^{N-k}$ since the i th entry can be anything between 0 and $i-1$ for $i \leq k$ and anything between 0 and $k-1$ for $i > k$. Thus, the probability that the maximum entry is less than k is $k!k^{N-k}/N!$, and the average value sought is

$$\sum_{0 \leq k \leq N} \left(1 - \frac{k!k^{N-k}}{N!} \right).$$

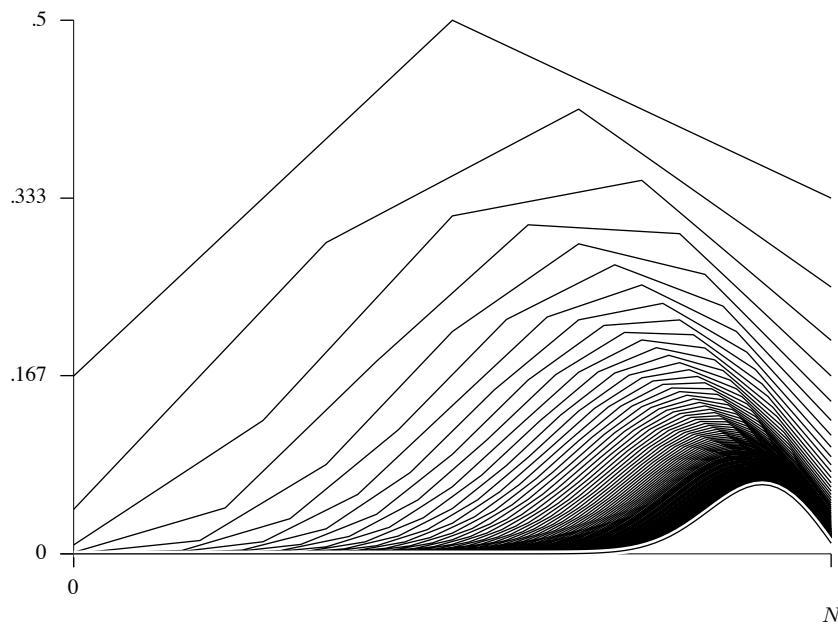
The second term in this sum is the “Ramanujan P -function” whose asymptotic value is given in Table 4.11. ■

Corollary Bubble sort performs $\sim N^2/2$ comparisons and $\sim N^2/2$ moves (in $\sim N - \sqrt{\pi N/2}$ passes), on the average, to sort a file of N randomly ordered records distinct keys.

Proof. See above discussion. ■

Exercise 7.64 Consider a modification of bubble sort where the passes through the array alternate in direction (right to left, then left to right). What is the effect of two such passes on the inversion table?

N	$\downarrow k \rightarrow 0$	1	2	3	4	5	6	7	8	9
1	1									
2	1	1								
3	1	3	2							
4	1	7	10	6						
5	1	15	38	42	24					
6	1	31	130	222	216	120				
7	1	63	422	1050	1464	1320	720			
8	1	127	1330	4686	8856	10920	9360	5040		
9	1	255	4118	20,202	50,424	80,520	91,440	75,600	40,320	
10	1	511	12610	85,182	27,6696	558,120	795,600	851,760	685,440	362,880

**Figure 7.14** Distribution of maximum inversion table entry

Longest and shortest cycles. What is the average length of the longest cycle in a permutation? We can immediately write down an expression for this. Earlier in this chapter, we derived the exponential GFs that enumerate permutations with no cycle of length $> k$ (see Theorem 7.2):

$$\begin{aligned}
 e^z &= 1 + z + \frac{z^2}{2!} + \frac{z^3}{3!} + \frac{z^4}{4!} + \frac{z^5}{5!} + \frac{z^6}{6!} + \dots \\
 e^{z+z^2/2} &= 1 + z + 2\frac{z^2}{2!} + 4\frac{z^3}{3!} + 10\frac{z^4}{4!} + 26\frac{z^5}{5!} + 76\frac{z^6}{6!} + \dots \\
 e^{z+z^2/2+z^3/3} &= 1 + z + 2\frac{z^2}{2!} + 6\frac{z^3}{3!} + 18\frac{z^4}{4!} + 66\frac{z^5}{5!} + 276\frac{z^6}{6!} + \dots \\
 e^{z+z^2/2+z^3/3+z^4/4} &= 1 + z + 2\frac{z^2}{2!} + 6\frac{z^3}{3!} + 24\frac{z^4}{4!} + 96\frac{z^5}{5!} + 456\frac{z^6}{6!} + \dots \\
 &\vdots \\
 e^{-\ln(1-z)} &= \frac{1}{1-z} = 1 + z + 2\frac{z^2}{2!} + 6\frac{z^3}{3!} + 24\frac{z^4}{4!} + 120\frac{z^5}{5!} + 720\frac{z^6}{6!} + \dots
 \end{aligned}$$

From these, we can write down the generating functions for the permutations with at least one cycle of length $> k$, or, equivalently, those for which the maximum cycle length is $> k$:

$$\begin{aligned}
 \frac{1}{1-z} - e^0 &= z + 2\frac{z^2}{2!} + 6\frac{z^3}{3!} + 24\frac{z^4}{4!} + 120\frac{z^5}{5!} + 720\frac{z^6}{6!} + \dots \\
 \frac{1}{1-z} - e^z &= \frac{z^2}{2!} + 5\frac{z^3}{3!} + 23\frac{z^4}{4!} + 119\frac{z^5}{5!} + 719\frac{z^6}{6!} + \dots \\
 \frac{1}{1-z} - e^{z+z^2/2} &= 2\frac{z^3}{3!} + 14\frac{z^4}{4!} + 94\frac{z^5}{5!} + 644\frac{z^6}{6!} + \dots \\
 \frac{1}{1-z} - e^{z+z^2/2+z^3/3} &= 6\frac{z^4}{4!} + 54\frac{z^5}{5!} + 444\frac{z^6}{6!} + \dots \\
 \frac{1}{1-z} - e^{z+z^2/2+z^3/3+z^4/4} &= 24\frac{z^5}{5!} + 264\frac{z^6}{6!} + \dots \\
 &\vdots
 \end{aligned}$$

From Table 3.6, the average length of the longest cycle in a random permutation is found by summing these and may be expressed as follows:

$$[z^N] \sum_{k \geq 0} \left(\frac{1}{1-z} - e^{z+z^2/2+z^3/3+\dots+z^k/k} \right).$$

As is typical for extremal parameters, derivation of an asymptotic result from this information is rather intricate. We can compute the exact value of this quantity for small N by summing the equations to get the initial terms of the exponential CGF for the length of the longest cycle:

$$1\frac{z^1}{1!} + 3\frac{z^2}{2!} + 13\frac{z^3}{3!} + 67\frac{z^4}{4!} + 411\frac{z^5}{5!} + \dots$$

It turns out that the length of the longest cycle in a random permutation is $\sim \lambda N$ where $\lambda \approx .62433 \dots$. This result was first derived by Golomb, Shepp, and Lloyd in 1966 [17]. The sequence is OEIS A028418 [18].

Exercise 7.65 Find the average length of the shortest cycle in a random permutation of length N , for all $N < 10$. (*Note:* Shepp and Lloyd show this quantity to be $\sim e^{-\gamma} \ln N$, where γ is Euler's constant.)

PERMUTATIONS are well studied as fundamental combinatorial objects, and we would expect that knowledge of their properties could help in the understanding of the performance characteristics of sorting algorithms. The direct correspondence between fundamental properties such as cycles and inversions and fundamental algorithms such as insertion sort, selection sort, and bubble sort confirms this expectation.

Research on new sorting algorithms and the analysis of their performance is quite active. Variants on sorting such as priority queues, merging algorithms, and sorting "networks" continue to be of practical interest. New types of computers and new applications demand new methods and better understanding of old ones, and the kind of analysis outlined in this chapter is an essential ingredient in designing and using such algorithms.

As suggested throughout this chapter, general tools are available [4] that can answer many of the more complicated questions raised in this chapter. We have emphasized the use of cumulative generating functions to analyze properties of permutations because they provide a straightforward "systematic"

path to the average value of quantities of interest. For analysis of properties of permutations, the cumulative approach can often yield results in a simpler, more direct manner than available with recurrences or BGFs. As usual, extremal parameters (those defined by a “maximum” or “minimum” rule as opposed to an additive rule) are more difficult to analyze, though “vertical” GFs can be used to compute small values and to start the analysis.

Despite the simplicity of the permutation as a combinatorial object, the wealth of analytic questions to be addressed is often quite surprising to the uninitiated. The fact that we can use a standard methodology to answer basic questions about the properties of permutations is encouraging, not only because many of these questions arise in important applications, but also because we can hope to be able to study more complicated combinatorial structures as well.

References

1. L. COMTET. *Advanced Combinatorics*, Reidel, Dordrecht, 1974.
2. F. N. DAVID AND D. E. BARTON. *Combinatorial Chance*, Charles Griffin, London, 1962.
3. W. FELLER. *An Introduction to Probability Theory and Its Applications*, John Wiley, New York, 1957.
4. P. FLAJOLET AND R. SEDGEWICK. *Analytic Combinatorics*, Cambridge University Press, 2009.
5. G. H. GONNET AND R. BAEZA-YATES. *Handbook of Algorithms and Data Structures in Pascal and C*, 2nd edition, Addison-Wesley, Reading, MA, 1991.
6. I. GOULDEN AND D. JACKSON. *Combinatorial Enumeration*, John Wiley, New York, 1983.
7. R. L. GRAHAM, D. E. KNUTH, AND O. PATASHNIK. *Concrete Mathematics*, 1st edition, Addison-Wesley, Reading, MA, 1989. Second edition, 1994.
8. D. E. KNUTH. *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, 1st edition, Addison-Wesley, Reading, MA, 1968. Third edition, 1997.
9. D. E. KNUTH. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*, 1st edition, Addison-Wesley, Reading, MA, 1969. Third edition, 1997.
10. D. E. KNUTH. *The Art of Computer Programming. Volume 3: Sorting and Searching*, 1st edition, Addison-Wesley, Reading, MA, 1973. Second edition, 1998.
11. D. E. KNUTH. *The Art of Computer Programming. Volume 4A: Combinatorial Algorithms, Part 1*, Addison-Wesley, Boston, MA, 2011.
12. D. E. KNUTH. “Mathematical Analysis of Algorithms,” *Information Processing 71*, Proceedings of the IFIP Congress, Ljubljana, 1971, 19–27.
13. V. LIFSCHITZ AND B. PITTEL. “The number of increasing subsequences of the random permutation,” *Journal of Combinatorial Theory (Series A)* **31**, 1981, 1–20.
14. B. F. LOGAN AND L. A. SHEPP. “A variational problem from random Young tableaux,” *Advances in Mathematics* **26**, 1977, 206–222.

15. R. SEDGEWICK. "Analysis of shellsort and related algorithms," *European Symposium on Algorithms*, 1986.
16. R. SEDGEWICK AND K. WAYNE. *Algorithms*, 4th edition, Addison-Wesley, Boston, 2011.
17. L. SHEPP AND S. P. LLOYD. "Ordered cycle lengths in a random permutation," *Transactions of the American Mathematical Society* **121**, 1966, 340–357.
18. N. SLOANE AND S. PLLOUFFE. *The Encyclopedia of Integer Sequences*, Academic Press, San Diego, 1995. Also accessible as *On-Line Encyclopedia of Integer Sequences*, <http://oeis.org>.
19. 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.
20. J. VUILLEMIN. "A unifying look at data structures," *Communications of the ACM* **23**, 1980, 229–239.
21. A. YAO. "An analysis of $(h, k, 1)$ shellsort," *Journal of Algorithms* **1**, 1980, 14–50.
22. A. YAO. "On straight selection sort," Technical report CS-TR-185-88, Princeton University, 1988.

This page intentionally left blank

CHAPTER EIGHT

STRINGS AND TRIES

SEQUENCES of characters or letters drawn from a fixed alphabet are called *strings*. Algorithms that process strings range from fundamental methods at the heart of the theory of computation to practical text-processing methods with a host of important applications. In this chapter, we study basic combinatorial properties of strings, some fundamental algorithms for searching for patterns in strings, and related data structures.

We use the term *bitstring* to refer to strings comprised of just two characters; if the alphabet is of size $M > 2$, we refer to the strings as *bytestrings*, or *words*, or M -ary strings. In this chapter, we assume that M is a small fixed constant, a reasonable assumption given our interest in text- and bit-processing algorithms. If M can grow to be large (for example, increasing with the length of the string), then we have a somewhat different combinatorial object, an important distinction that is one of the main subjects of the next chapter. In this chapter, our primary interest is in potentially long strings made from constant-size alphabets, and in their properties as sequences.

From an algorithmic point of view, not much generality is lost by focusing on bitstrings rather than on bytestrings: a string built from a larger alphabet corresponds to a bitstring built by encoding the individual characters in binary. Conversely, when an algorithm, data structure, or analysis of strings built from a larger alphabet depends in some way on the size of the alphabet, that same dependence can be reflected in a bitstring by considering the bits in blocks. This particular correspondence between M -ary strings and bitstrings is exact when M is a power of 2. It is also often very easy to generalize an algorithm or analysis from bitstrings to M -ary strings (essentially by changing “2” to “ M ” throughout), so we do so when appropriate.

Random bitstrings correspond precisely to sequences of independent Bernoulli trials, which are well studied in classical probability theory; in the analysis of algorithms such results are of interest because many algorithms naturally depend explicitly on properties of binary strings. We review some relevant classical results in this chapter and the next. As we have been doing for trees and permutations in the previous two chapters, we consider the prob-

lems from a computational standpoint and use generating functions as tools for combinatorial analysis. This approach yields simple solutions to some classical problems and gives a very general framework within which a surprising range of problems can be considered.

We consider basic algorithms for searching for the occurrence of a fixed pattern in a given string, which are best described in terms of pattern-specific finite-state automata (FSAs). Not only do FSAs lead to uniform, compact and efficient implementations, but also it turns out that the automata correspond precisely to generating functions associated with the patterns. In this chapter, we study some examples of this in detail.

Certain computational tasks require that we manipulate *sets* of strings. Sets of strings (generally, infinite sets) are called *languages* and are the basis of an extensive theory of fundamental importance in computer science. Languages are classified according to the difficulty of describing their constituent strings. In the present context, we will be most concerned with *regular* languages and *context-free* languages, which describe many interesting combinatorial structures. In this chapter, we revisit the symbolic method to illustrate the utility of generating functions in analyzing properties of languages. Remarkably, generating functions for both regular and context-free languages can be fully characterized and shown to be essentially different in nature.

A data structure called the *trie* is the basis for numerous efficient algorithms that process strings and sets of strings. Tries are treelike objects with structure determined by values in a set of strings. The trie is a combinatorial object with a wealth of interesting properties. Not found in classical combinatorics, it is the quintessential example of a new combinatorial object brought to the field by the analysis of algorithms. In this chapter, we look at basic trie algorithms, properties of tries, and associated generating functions. Not only are tries useful in a wide range of applications, but also their analysis exhibits and motivates many important tools for the analysis of algorithms.

8.1 String Searching. We begin by considering a basic algorithm for “string searching:” given a pattern of length P and some text of length N , look for occurrences of the pattern in the text. Program 8.1 gives the straightforward solution to this problem. For each position in the text, the program checks if there is a match by comparing the text, starting at this position, character-by-character with the pattern, starting at the beginning. The program assumes that two different sentinel characters are used, one at the end

of the pattern (the $(P + 1)$ st pattern character) and one at the end of the text (the $(N + 1)$ st text character). Thus, all string comparisons end on a character mismatch, and we tell whether the pattern was present in the text simply by checking whether the sentinel(s) caused the mismatch.

Depending on the application, one of a number of different variations of the basic algorithm might be of interest:

- Stop when the *first* match is found.
- Print out the position of *all* matches.
- Count the number of matches.
- Find the longest match.

The basic implementation given in Program 8.1 is easy to adapt to implement such variations, and it is a reasonable general-purpose method in many contexts.

Still, it is worthwhile to consider improvements. For example, we could search for a string of P consecutive 0s (a *run*) by maintaining a counter and scanning through the text, resetting the counter when a 1 is encountered, incrementing it when a 0 is encountered, and stopping when the counter reaches

```
public static int search(char[] pattern, char[] text)
{
    int P = pattern.length;
    int N = text.length;
    for (int i = 0; i <= N - P; i++)
    {
        int j;
        for (j = 0; j < P; j++)
            if (text[i+j] != pattern[j]) break;
        if (j == P) return i; // Found at offset i.
    }
    return N; // Not found.
}
```

Program 8.1 Basic method for string searching

P. By contrast, consider the action of Program 8.1 when searching for a string of *P* consecutive zeros and it encounters a small run of, say, five 0s followed by a 1. It examines all five 0s, determines there is a mismatch on finding the 1, then increments the text pointer *just by one* so that it finds a mismatch by checking four 0s and a 1, then checks three 0s and a 1, and so on. The program ends up checking $k(k + 1)/2$ extra bits for every run of *k* 0s. Later in the chapter, we will examine improvements to the basic algorithm that avoid such rechecking. For the moment, our interest is to examine how to analyze the basic method.

Analysis of “all matches” variant. We are interested in finding the average running time of Program 8.1 when searching for a given pattern in random text. Clearly, the running time is proportional to the number of characters examined during the search. Since each string comparison ends on a mismatch, its cost is 1 plus the number of characters that match the pattern at that text position. Table 8.1 shows this for each of the patterns of four bits

0 1 1 1 0 1 0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 1 1																total											
0000	1	0	0	0	1	0	3	2	1	0	4	4	3	2	1	0	2	1	0	4	4	3	2	1	0	0	39
0001	1	0	0	0	1	0	4	2	1	0	3	3	4	2	1	0	2	1	0	3	3	4	2	1	0	0	39
0010	1	0	0	0	1	0	2	4	1	0	2	2	2	4	1	0	4	1	0	2	2	2	3	1	0	0	35
0011	1	0	0	0	1	0	2	3	1	0	2	2	2	3	1	0	3	1	0	2	2	2	4	1	0	0	33
0100	2	0	0	0	4	0	1	4	0	1	1	1	1	4	0	1	4	0	1	1	1	1	2	0	0	31	
0101	2	0	0	0	3	0	1	3	0	1	1	1	1	3	0	1	3	0	1	1	1	1	2	0	0	27	
0110	3	0	0	0	2	0	1	1	2	0	1	1	1	2	0	1	2	0	1	1	1	1	3	0	0	22	
0111	4	0	0	0	2	0	1	1	2	0	1	1	1	2	0	1	2	0	1	1	1	1	3	0	0	25	
1000	0	1	1	2	0	4	0	0	0	4	0	0	0	0	3	0	0	4	0	0	0	0	0	1	1	21	
1001	0	1	1	2	0	3	0	0	0	3	0	0	0	0	0	4	0	0	3	0	0	0	0	0	1	1	19
1010	0	1	1	2	0	2	0	0	0	2	0	0	0	0	0	2	0	0	2	0	0	0	0	0	1	1	14
1011	0	1	1	2	0	2	0	0	0	2	0	0	0	0	2	0	0	2	0	0	0	0	0	1	1	1	14
1100	0	2	3	1	0	1	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	2	1	13
1101	0	2	4	1	0	1	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	2	1	14
1110	0	4	2	1	0	1	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	2	1	14
1111	0	3	2	1	0	1	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	2	1	13

Table 8.1 Cost of searching for 4-bit patterns (basic method)

in a sample text string. With each position in the text string we associate an integer—the number of character positions that match the pattern, starting at that text position. The total of these plus N (for the mismatches) is the number of times the inner loop in Program 8.1 is iterated, clearly the dominant term in the running time of the program. Using the cumulative method, we can calculate this with a simple counting argument.

Theorem 8.1 (Pattern occurrence enumeration). The expected number of occurrences of an arbitrary fixed pattern of length P in a random bitstring of length N is $(N - P + 1)/2^P$.

Proof. Using cumulative counting, we count the total number of occurrences of the pattern in all of the 2^N bitstrings of N bits. The P bits can start in any one of $N - P + 1$ positions, and, for each position, there are 2^{N-P} bitstrings with the pattern at that position. Collecting all these terms gives the total $(N - P + 1)2^{N-P}$, and dividing by 2^N gives the stated result. ■

Corollary The expected number of bit comparisons made by the basic string-searching algorithm when seeking all occurrences of an arbitrary fixed pattern of length P in a text string of length N is $N(2 - 2^{-P}) + O(1)$.

Proof. To find the number of bits examined by the algorithm in the search, we note that we can interpret the numbers in Table 8.1 another way: they count the number of *prefixes* of the pattern that occur starting at the corresponding position in the text. Now we can use the above formula to count the prefixes as well. This approach yields the expression

$$\sum_{1 \leq k \leq P} (N - k + 1)2^{N-k}$$

for the total number of occurrences of prefixes of the pattern in all of the 2^N bitstrings of N bits. Evaluating this sum, dividing by 2^N , and adding N (for the mismatch comparisons) gives the stated result. ■

Corollary The average number of bits examined by the basic string-searching algorithm to find the longest match with an arbitrary infinitely long pattern in a random bitstring is $\sim 2N$.

These results are independent of the pattern and seem to run counter to the intuition just given, where we know that the number of bits examined

during a search for a long string of 0s is quadratic in the length of each run of 0s in the text, whereas a search for a 1 followed by a long string of 0s does not have such evident quadratic behavior. This difference is perhaps explained by noting that prefixes of the latter pattern are *somewhere* in the text, but not bunched together as for strings of 0s. Since we use cumulated totals, we need not worry about independence among different instances of prefixes; we count them all. By contrast, the time required to find the *first* match does depend on the pattern, even for random text. This is directly related to another quantity of interest, the probability that the pattern does not occur in the text. In the next section, we will look at details of these analytic results.

Later in the chapter, we will look at algorithmic improvements to the basic string-searching algorithm. First, we will look at the Knuth-Morris-Pratt algorithm, a method that uses preprocessing time proportional to the length of the pattern to get an “optimal” search time, examining each character in the text at most once. At the end of the chapter, we will see that with (much) more investment in preprocessing, the text can be built into a data structure related to a general structure known as a *trie* that allows searches for patterns to be done in time proportional to the length of the pattern. Tries also provide efficient support for a number of other algorithms on bitstrings, but we postpone considering those until after covering basic analytic results.

8.2 Combinatorial properties of bitstrings. We are interested in studying the properties of random strings of 0 and 1 values (bitstrings) when we consider each of the 2^N bitstrings of length N to be equally likely. As we saw in §5.2, bitstrings are enumerated by the OGF

$$B(z) = \sum_{b \in \mathcal{B}} z^{|b|} = \sum_{N \geq 0} \{\# \text{ of bitstrings of length } N\} z^N$$

where \mathcal{B} denotes the set of all bitstrings. Bitstrings are either empty or begin with a 0 or a 1, so they are generated by the combinatorial construction

$$\mathcal{B} = \epsilon + (\mathcal{Z}_0 + \mathcal{Z}_1) \times \mathcal{B},$$

which immediately transfers to

$$B(z) = 1 + 2zB(z)$$

and therefore $B(z) = (1 - 2z)^{-1}$ and the number of bitstrings of length N is 2^N as expected. As with permutations and trees in the previous two chapters, we can modify this basic argument to study properties of bitstrings that are a bit more interesting.

For example, we have already encountered the problem of enumerating the 1s in a random bitstring as an example in §3.9 and §5.4: the associated bivariate generating function involves the binomial distribution

$$B(z, u) = \frac{1}{1 - z(1 + u)} = \sum_N \sum_k \binom{N}{k} u^k z^N.$$

We used Theorem 3.11 to calculate that the average number of 1 bits in a random N -bit string is

$$[z^N]B_u(z, 1)/2^N = N/2,$$

and so forth. We will consider “global” properties like this in much more detail in Chapter 9; in this chapter our focus is more on “local” properties involving bits that are near one another in the string.

Studying properties of random bitstrings is equivalent to studying properties of independent Bernoulli trials (perhaps coin flips), so classical results from probability theory are relevant. Our specific focus in this chapter is not just on the trials, but also on the *sequence* of events. There are also classical results related to this. In probability theory, we consider properties of a sequence of random trials and study “waiting times” (see Feller [9], for example); in the analysis of algorithms, we consider the equivalent problem of searching for patterns in strings by taking the bits in sequence, a natural way to think about the same problem. As we will see, it turns out that formal language theory and generating functions combine to provide a clear explanation of analytic phenomena relating to the study of sequences of Bernoulli trials.

Pattern occurrence enumeration. Any bitstring containing an arbitrary fixed pattern of length P is constructed by concatenating an arbitrary bitstring, the pattern, and another arbitrary bitstring. Therefore, by the symbolic method, the generating function enumerating such occurrences is

$$\frac{1}{1 - 2z} z^P \frac{1}{1 - 2z}.$$

A particular bitstring may be counted several times in this enumeration (once for each occurrence of the pattern), which is precisely what we seek. Thus, the number of occurrences of a fixed pattern of length P in all bitstrings of length N is

$$[z^N] \frac{1}{1-2z} z^P \frac{1}{1-2z} = [z^{N-P}] \frac{1}{(1-2z)^2} = (N-P+1)2^{N-P}$$

as in Theorem 8.1. Again, this is *not* the same as the number of bitstrings of length N containing (one or more occurrences of) an arbitrary fixed pattern.

Runs of 0s. Where should we expect to find the first run of P consecutive 0s in a random bitstring? Table 8.2 gives ten long random bitstrings and the positions where one, two, three, and four 0s first occur in each. It turns out that generating functions lead to a simple solution to this problem, and that it is representative of a host of similar problems.

	1	2	3	4
01010011101010000110011100010111011000110111111010	0	4	13	13
01110101010001010010011001010000100001010110100101	0	10	10	28
01011101011011110010001110000001010110010011110000	0	16	19	25
101010000101001110101011110000111110000111001001	1	5	5	5
1111111100100001001001100100110000000100000110001	8	8	11	11
00100010001100101110011100001100101000001011001111	0	0	3	24
010110111101100101100010010100101000010100111110	0	13	19	19
10011010000010011001010010100011000001111010011010	1	1	7	7
01100010011001101100010111110001001000111001111010	0	3	3	—
01011001000110000001000110010101011100111100100110	0	5	8	13
<hr/>				
average	1.0	6.5	9.8	—

Table 8.2 Position of first runs of 0s on sample bitstrings

Theorem 8.2 (Runs of 0s). The generating function enumerating the number of bitstrings with no runs of P consecutive 0s is given by

$$B_P(z) = \frac{1 - z^P}{1 - 2z + z^{P+1}}.$$

Proof. Let \mathcal{B}_P be the class of bitstrings having no runs of P consecutive 0s and consider the OGF

$$B_P(z) = \sum_{b \in \mathcal{B}_P} z^{|b|} = \sum_{N \geq 0} \{\# \text{ of } N\text{-bit strings having no runs of } P \text{ 0s}\} z^N.$$

Now, any bitstring without P consecutive 0s is either (i) null or consisting of from zero to $P - 1$ 0s; or (ii) a string of from zero to $P - 1$ 0s, followed by a 1, followed by any bitstring without P consecutive 0s. The symbolic method immediately gives the functional equation

$$S_P(z) = (1 + z + \dots + z^{P-1})(1 + zS_P(z)).$$

Noting that $1 + z + \dots + z^{P-1} = (1 + z^P)/(1 - z)$, we can calculate the following explicit expression for the OGF:

$$B_P(z) = \frac{\frac{1 - z^P}{1 - z}}{1 - z \frac{1 - z^P}{1 - z}} = \frac{1 - z^P}{1 - 2z + z^{P+1}}.$$
■

Each $B_P(z)$ is a rational function and is therefore easy to expand. Checking small values, we have the following expansions for $P = 1, 2, 3$:

$$\frac{1 - z}{1 - 2z + z^2} = 1 + z + z^2 + z^3 + z^4 + z^5 + z^6 + z^7 + \dots$$

$$\frac{1 - z^2}{1 - 2z + z^3} = 1 + 2z + 3z^2 + 5z^3 + 8z^4 + 13z^5 + 21z^6 + 34z^7 + \dots$$

$$\frac{1 - z^3}{1 - 2z + z^4} = 1 + 2z + 4z^2 + 7z^3 + 13z^4 + 24z^5 + 44z^6 + 81z^7 + \dots$$

For $P = 1$, the expansion confirms the fact that there is one string of length N with no runs of one 0 (the string that is all 1s). For $P = 2$, we have

$$B_2(z) = \frac{1+z}{1-z-z^2} \quad \text{so} \quad [z^N]B_2(z) = F_{N+1} + F_N = F_{N+2},$$

the Fibonacci numbers (see §2.4). Indeed, the representation

$$B_P(z) = \frac{1+z+z^2+\dots+z^{P-1}}{1-z-z^2-\dots-z^P}$$

shows that $[z^N]B_P(z)$ satisfies the same recurrence as the generalized Fibonacci numbers of Exercise 4.18, but with different initial values. Thus, we can use Theorem 4.1 to find asymptotic estimates for $[z^N]B_P(z)$.

Corollary The number of bitstrings of length N containing no runs of P consecutive 0s is asymptotic to $c\beta^N$, where β is the root of largest modulus of the polynomial $z^P - z^{P-1} - \dots - z - 1 = 0$ and $c = (\beta^P + \beta^{P-1} + \dots + \beta)/(\beta^{P-1} + 2\beta^{P-2} + 3\beta^{P-3} + \dots + (P-1)\beta + P)$.

Proof. Immediate from Exercise 4.18 and Theorem 4.1. Approximate values of c and β for small values of P are given in Table 8.3. ■

P	$B_P(z)$	c_P	β_P
2	$\frac{1-z^2}{1-2z+z^3}$	1.17082...	1.61803...
3	$\frac{1-z^3}{1-2z+z^4}$	1.13745...	1.83929...
4	$\frac{1-z^4}{1-2z+z^5}$	1.09166...	1.92756...
5	$\frac{1-z^5}{1-2z+z^6}$	1.05753...	1.96595...
6	$\frac{1-z^6}{1-2z+z^7}$	1.03498...	1.98358...

Table 8.3 Enumerating bitstrings with no runs of P 0s

Exercise 8.1 Give two recurrences satisfied by $[z^N]B_P(z)$.

Exercise 8.2 How long a string of random bits should be taken to be 99% sure that there are at least three consecutive 0s?

Exercise 8.3 How long a string of random bits should be taken to be 50% sure that there are at least 32 consecutive 0s?

Exercise 8.4 Show that

$$[z^N]B_P(z) = \sum_i (-1)^i 2^{N-(P+1)i} \left(\binom{N - Pi}{i} - 2^{-P} \binom{N - P(i+1)}{i} \right).$$

First run of 0s. A shortcut to computing the average position of the first run of P 0s is available because the OGF that enumerates bitstrings with no runs of P consecutive 0s is very closely related to the PGF for the position of the last bit (the end) in the first run of P consecutive 0s in a random bitstring, as shown by the following manipulations:

$$\begin{aligned} B_P(z) &= \sum_{b \in \mathcal{B}_P} z^{|b|} \\ &= \sum_{N \geq 0} \{\# \text{ of bitstrings of length } N \text{ with no runs of } P \text{ 0s}\} z^N \\ B_P(1/2) &= \sum_{N \geq 0} \{\# \text{ of bitstrings of length } N \text{ with no runs of } P \text{ 0s}\} / 2^N \\ &= \sum_{N \geq 0} \Pr \{ \text{1st } N \text{ bits of a random bitstring have no runs of } P \text{ 0s} \} \\ &= \sum_{N \geq 0} \Pr \{ \text{position of end of first run of } M \text{ 0s is } > N \}. \end{aligned}$$

This sum of cumulative probabilities is equal to the expectation.

Corollary The average position of the end of the first run of M 0s in a random bitstring is $B_P(1/2) = 2^{P+1} - 2$.

Generating functions simplify the computation of the expectation considerably; any reader still unconvinced of this fact is welcome, for example, to verify the result of this corollary by developing a direct derivation based on calculating probabilities. For permutations, we found that the count $N!$ of the number of permutations on N elements led us to EGFs; for bitstrings, the count 2^N of the number of bitstrings of N elements will lead us to functional equations involving $z/2$, as above.

Existence. The proof of the second corollary to Theorem 8.2 also illustrates that finding the first occurrence of a pattern is roughly equivalent to counting the number of strings that do *not* contain the pattern, and tells us that the probability that a random bitstring contains no run of P 0s is $[z^N]B_P(z/2)$. For example, for $P = 1$ this value is $1/2^N$, since only the bitstring that is all 1s contains no runs of P 0s. For $P = 2$ the probability is $O((\phi/2)^N)$ (with $\phi = (1 + \sqrt{5})/2 = 1.61803 \dots$), exponentially decreasing in N . For fixed P , this exponential decrease is always the case because the β_P 's in Table 8.3 remain strictly less than 2. A slightly more detailed analysis reveals that once N increases past 2^M , it becomes increasingly unlikely that some P -bit pattern does not occur. For example, a quick calculation from Table 8.3 shows that there is a 95% chance that a 10-bit string does not contain a run of six 0s, a 45% chance that a 100-bit string does not contain a run of six 0s, and a .02% chance that a 1000-bit string does not contain a run of six 0s.

Longest run. What is the average length of the longest run of 0s in a random bitstring? The distribution of this quantity is shown in Figure 8.1. As we did for tree height in Chapter 6 and cycle length in permutations in Chapter 7, we can sum the “vertical” GFs given previously to get an expression for the average length of the longest string of 0s in a random N -bit string:

$$\frac{1}{2^N}[z^N] \sum_{k \geq 0} \left(\frac{1}{1-2z} - \frac{1-z^k}{1-2z+z^{k+1}} \right).$$

Knuth [24] studied a very similar quantity for the application of determining carry propagation time in an asynchronous adder, and showed this quantity to be $\lg N + O(1)$. The constant term has an oscillatory behavior; close inspection of Figure 8.1 will give some insight into why this might be so. The function describing the oscillation turns out to be the same as one that we will study in detail for the analysis of tries at the end of this chapter.

Exercise 8.5 Find the bivariate generating function associated with the number of leading 1 bits in a random bitstring and use it to calculate the average and standard deviation of this quantity.

Exercise 8.6 By considering bitstrings with no runs of two consecutive 0s, evaluate the following sum involving Fibonacci numbers: $\sum_{j \geq 0} F_j/2^j$.

Exercise 8.7 Find the BGF for the length of the longest run of 0s in bitstrings.

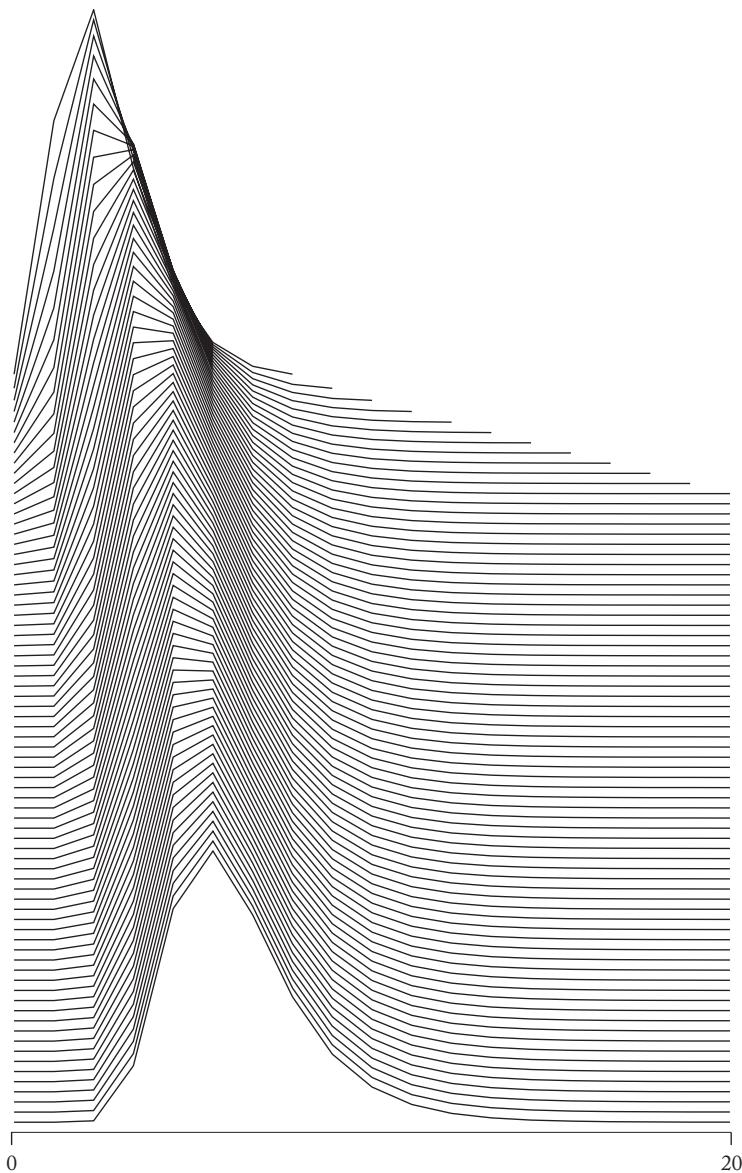


Figure 8.1 Distribution of longest run of 0s in a random bitstring
(horizontal axes translated to separate curves)

Exercise 8.8 What is the standard deviation of the random variable marking the first occurrence of a run of P 0s in a random bitstring?

Exercise 8.9 Use a computer algebra system to plot the average length of the longest run of 0s in a random bitstring of N bits, for $2 < N < 100$.

Exercise 8.10 How many bits are examined by the basic algorithm given in the previous section to find the first string of P 0s in a random bitstring?

Arbitrary patterns. At first, one might suspect that these results hold for *any* fixed pattern of P bits, but that is simply not true: the average position of the first occurrence of a fixed bit pattern in a random bitstring depends very much on the pattern itself. For example, it is easy to see that a pattern like 0001 tends to appear before 0000, on the average, by the following observation: once 000 has already been matched, in both cases a match occurs on the next character with probability $1/2$, but a mismatch for 0000 means that 0001 was in the text (and no match is possible for four more positions), while a mismatch for 0001 means that 0000 is in the text (and a match can happen at the next position). The dependence on the pattern turns out to be easily expressed in terms of a function matching the pattern against itself:

Definition The *autocorrelation* of a bitstring $b_0 b_1 \dots b_{P-1}$ is the bitstring $c_0 c_1 \dots c_{P-1}$ with c_i defined to be 1 if $b_j = b_{i+j}$ for $0 \leq j \leq P-1-i$, 0 otherwise. The corresponding autocorrelation polynomial is obtained by taking the bits as coefficients: $c(z) = c_0 + c_1 z + \dots + c_{P-2} z^{P-2} + c_{P-1} z^{P-1}$.

The autocorrelation is easily computed: the i th bit is determined by shifting left i positions, then putting 1 if the remaining bits match the original

	1	0	1	0	0	1	0	1	0
	1	0	1	0	0	1	0	1	0
	1	0	1	0	0	1	0	1	0
	1	0	1	0	0	1	0	1	0
	1	0	1	0	0	1	0	1	0
	1	0	1	0	0	1	0	1	0
	1	0	1	0	0	1	0	1	0
	1	0	1	0	0	1	0	1	0
	1	0	1	0	0	1	0	1	0
1	0	1	0	0	1	0	1	0	0

Table 8.4 Autocorrelation of 101001010

pattern, 0 otherwise. For example, Table 8.4 shows that the autocorrelation of 101001010 is 100001010, and the corresponding autocorrelation polynomial is $1 + z^5 + z^7$. Note that c_0 is always 1.

Theorem 8.3 (Pattern autocorrelation). The generating function for the number of bitstrings not containing a pattern $p_0 p_1 \dots p_{P-1}$ is given by

$$B_p(z) = \frac{c(z)}{z^P + (1 - 2z)c(z)},$$

where $c(z)$ is the autocorrelation polynomial for the pattern.

Proof. We use the symbolic method to generalize the proof given earlier for the case where the pattern is P consecutive 0s. We start with the OGF for \mathcal{S}_p , the set of bitstrings with no occurrence of p :

$$\begin{aligned} S_p(z) &= \sum_{s \in \mathcal{S}_p} z^{|s|} \\ &= \sum_{N \geq 0} \{\# \text{ of bitstrings of length } N \text{ with no occurrence of } p\} z^N. \end{aligned}$$

Similarly, we define \mathcal{T}_p to be the class of bitstrings that end with p but have no other occurrence of p , and name its associated generating function $T_p(z)$.

Now, we consider two symbolic relationships between \mathcal{S}_p and \mathcal{T}_p that translate to simultaneous equations involving $S_p(z)$ and $T_p(z)$. First, \mathcal{S}_p and \mathcal{T}_p are disjoint, and if we remove the last bit from a bitstring in either, we get a bitstring in \mathcal{S}_p (or the empty bitstring). Expressed symbolically, this means that

$$\mathcal{S}_p + \mathcal{T}_p = \epsilon + \mathcal{S}_p \times (\mathcal{Z}_0 + \mathcal{Z}_1),$$

which, since the OGF for $(\mathcal{Z}_0 + \mathcal{Z}_1)$ is $2z$, translates to

$$S_p(z) + T_p(z) = 1 + 2zS_p(z).$$

Second, consider the set of strings consisting of a string from \mathcal{S}_p followed by the pattern. For each position i in the autocorrelation for the pattern, this gives a string from \mathcal{T}_p followed by an i -bit “tail.” Expressed symbolically, this gives

$$\mathcal{S}_p \times \langle \text{pattern} \rangle = \mathcal{T}_p \times \sum_{c_i=1} \langle \text{tail} \rangle_i,$$

which, since the OGF for `<pattern>` is z^P and the OGF for `<tail>i` is z^i , translates to

$$S_p(z)z^P = T_p(z) \sum_{c_i=1} z^i = T_p(z)c(z).$$

The stated result follows immediately as the solution to the two simultaneous equations relating the OGFs $S(z)$ and $T(z)$. ■

For patterns consisting of P 0s (or P 1s), the autocorrelation polynomial is $1 + z + z^2 + \dots + z^{P-1} = (1 - z^P)/(1 - z)$, so Theorem 8.3 matches our previous result in Theorem 8.2.

Corollary The expected position of the end of the first occurrence of a bitstring with autocorrelation polynomial $c(z)$ is given by $2^P c(1/2)$.

Table 8.5 shows the generating functions for the number of bitstrings not containing each of the 16 patterns of four bits. The patterns group into four different sets of patterns with equal autocorrelation. For each set, the table also gives the dominant root of the polynomial in the denominator of the OGF, and the expected “wait time” (position of the first occurrence of the pattern), computed from the autocorrelation polynomial. We can develop these approximations using Theorem 4.1 and apply them to approximate the wait times using the corollaries to Theorem 8.2 in the same way we did for Table 8.3. That is, the probability that an N -bit string has no occurrence of the pattern 1000 is about $(1.83929/2)^N$, and so forth. Here, we are ignoring

pattern	autocorrelation	OGF	dominant root	wait
0000 1111	1111	$\frac{1 - z^4}{1 - 2z + z^5}$	1.92756...	30
0001 0011 0111	1000	$\frac{1}{1 - 2z + z^4}$	1.83929...	16
1000 1100 1110				
0010 0100 0110	1001	$\frac{1 + z^3}{1 - 2z + z^3 - z^4}$	1.86676...	18
1001 1011 1101				
0101 1010	1010	$\frac{1 + z^2}{1 - 2z + z^2 - 2z^3 + z^4}$	1.88320...	20

Table 8.5 Generating functions and wait times for 4-bit patterns

the constants like the ones in Table 8.3, which are close to, but not exactly, 1. Thus, for example, there is about a 43% chance that a 10-bit string does not contain 1000, as opposed to the 69% chance that a 10-bit string does not contain 1111.

It is rather remarkable that such results are so easily accessible through generating functions. Despite their fundamental nature and wide applicability, it was not until systematic analyses of string-searching algorithms were attempted that this simple way of looking at such problems became apparent. These and many more related results are developed fully in papers by Guibas and Odlyzko [17][18].

Exercise 8.11 Calculate the expected position of the first occurrence of each of the following patterns in a random bitstring: (i) $P - 1$ 0s followed by a 1; (ii) a 1 followed by $P - 1$ 0s; (iii) alternating 0-1 string of even length; (iv) alternating 0-1 string of odd length.

Exercise 8.12 Which bit patterns of length P are likely to appear the earliest in a random bitstring? Which patterns are likely to appear the latest?

Exercise 8.13 Does the standard deviation of the random variable marking the first position of a bit pattern of length P in a random bitstring depend on the pattern?

Larger alphabets. The methods just described apply directly to larger alphabets. For example, a proof virtually identical to the proof of Theorem 8.3 will show that the generating function for strings from an M -character alphabet that do not contain a run of P consecutive occurrences of a particular character is

$$\frac{1 - z^P}{1 - Mz + (M - 1)z^{P+1}}.$$

Similarly, as in the second corollary to Theorem 8.3, the average position of the end of the first run of P occurrences of a particular character in a random string taken from an M -character alphabet is $M(M^P - 1)/(M - 1)$.

Exercise 8.14 Suppose that a monkey types randomly at a 32-key keyboard. What is the expected number of characters typed before the monkey hits upon the phrase THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG?

Exercise 8.15 Suppose that a monkey types randomly at a 32-key keyboard. What is the expected number of characters typed before the monkey hits upon the phrase TO BE OR NOT TO BE?

8.3 Regular Expressions. The basic method using generating functions as described in the previous section generalizes considerably. To determine properties of random strings, we ended up deriving generating functions that count the cardinality of sets of strings with well-defined properties. But developing specific descriptions of sets of strings falls within the domain of *formal languages*, the subject of a vast literature. We use only basic principles, as described in any standard text—for example, Eilenberg [8]. The simplest concept from formal language theory is the *regular expression* (RE), a way to describe sets of strings based on the union, concatenation, and “star” operations, which are described later in this section. A set of strings (a language) is said to be regular if it can be described by a regular expression. For example, the following regular expression describes all bitstrings with no run of four consecutive 0s:

$$\mathcal{S}_4 = (1 + 01 + 001 + 0001)^*(\epsilon + 0 + 00 + 000).$$

In this expression, $+$ denotes unions of languages; the product of two languages is the language of strings formed by concatenating a string from the first with a string from the second; and $*$ is shorthand for concatenating a language with itself an arbitrary number of times (including zero). As usual, ϵ represents the empty string. Earlier, we derived the corresponding OGF

$$S_4(z) = \sum_{s \in \mathcal{S}_4} z^{|s|} = \frac{1 - z^4}{1 - 2z + z^5}$$

and deduced basic properties of the language by manipulating the OGF. Other problems that we have considered also correspond to languages that can be defined with REs and thus analyzed with OGFs, as we shall see.

There is a relatively simple mechanism for transforming the formal description of the sets of strings (the regular expression) into the formal analytic tool for counting them (the OGF). This is due to Chomsky and Schützenberger [3]. The sole requirement is that the regular expression be *unambiguous*: there must be only one way to derive any string in the language. This is not a fundamental restriction because it is known from formal language theory that any regular language can be specified by an unambiguous regular expression. In practice, however, it often is a restriction because the theoretically guaranteed unambiguous RE can be complicated, and checking for ambiguity or finding a usable unambiguous RE can be challenging.

Theorem 8.4 (OGFs for regular expressions). Let \mathcal{A} and \mathcal{B} be unambiguous regular expressions and suppose that $\mathcal{A} + \mathcal{B}$, $\mathcal{A} \times \mathcal{B}$, and \mathcal{A}^* are also unambiguous. If $A(z)$ is the OGF that enumerates \mathcal{A} and $B(z)$ is the OGF that enumerates \mathcal{B} , then

$A(z) + B(z)$ is the OGF that enumerates $\mathcal{A} + \mathcal{B}$,

$A(z)B(z)$ is the OGF that enumerates $\mathcal{A}\mathcal{B}$, and

$\frac{1}{1 - A(z)}$ is the OGF that enumerates \mathcal{A}^* .

Moreover, OGFs that enumerate regular languages are rational functions.

Proof. The first part is essentially the same as our basic theorem on the symbolic method for OGFs (Theorem 5.1), but it is worth restating here because of the fundamental nature of this application. If a_N is the number of strings of length N in \mathcal{A} and b_N is the number of strings of length N in \mathcal{B} , then $a_N + b_N$ is the number of strings of length N in $\mathcal{A} + \mathcal{B}$, since the requirement that the languages are unambiguous implies that $\mathcal{A} \cap \mathcal{B}$ is empty. Similarly, we can use a simple convolution to prove the translation of $\mathcal{A}\mathcal{B}$, and the symbolic representation

$$\mathcal{A}^* = \epsilon + \mathcal{A} + \mathcal{A}^2 + \mathcal{A}^3 + \mathcal{A}^4 + \dots$$

implies the rule for \mathcal{A}^* , exactly as for Theorem 5.1.

The second part of the theorem results from the remark that every regular language can be specified by an unambiguous regular expression. For the reader with knowledge of formal languages: if a language is regular, it can be recognized by a deterministic FSA, and the classical proof of Kleene's theorem associates an unambiguous regular expression to a deterministic automaton. We explore some algorithmic implications of associations with FSAs later in this chapter. ■

Thus, we have a simple and direct way to transform a regular expression into an OGF that counts the strings described by that regular expression, provided only that the regular expression is unambiguous. Furthermore, an important implication of the fact that the generating function that results when successively applying Theorem 8.4 is always rational is that asymptotic approximations for the coefficients are available, using general tools such as Theorem 4.1. We conclude this section by considering some examples.

Strings with no runs of k 0s. Earlier, we gave a regular expression for \mathcal{S}_k , the set of bitstrings with no occurrence of k consecutive 0s. Consider, for instance, \mathcal{S}_4 . From Theorem 8.4, we find immediately that the OGF for

$$1 + 01 + 001 + 0001 \text{ is } z + z^2 + z^3 + z^4$$

and the OGF for

$$\epsilon + 0 + 00 + 000 \text{ is } 1 + z + z^2 + z^3$$

so the construction for \mathcal{S}_4 given earlier immediately translates to the OGF equation

$$S_4(z) = \frac{1 + z + z^2 + z^3}{1 - (z + z^2 + z^3 + z^4)} = \frac{\frac{1 - z^4}{1 - z}}{1 - z \frac{1 - z^4}{1 - z}} = \frac{1 - z^4}{1 - 2z + z^5},$$

which matches the result that we derived in §8.2.

Multiples of three. The regular expression $(1(01^*0)^*10^*)^*$ generates the set of strings 11, 110, 1001, 1100, 1111, . . . , that are binary representations of multiples of 3. Applying Theorem 8.4, we find the generating function for the number of such strings of length N :

$$\begin{aligned} \frac{1}{1 - \frac{z^2}{1 - \frac{z^2}{1 - \frac{z^2}{1 - z}}\left(\frac{1}{1 - z}\right)}} &= \frac{1}{1 - \frac{z^2}{1 - z - z^2}} = \frac{1 - z - z^2}{1 - z - 2z^2} \\ &= 1 + \frac{z^2}{(1 - 2z)(1 + z)}. \end{aligned}$$

This GF is very similar to one of the first GFs encountered in §3.3: it expands by partial fractions to give the result $(2^{N-1} + (-1)^N)/3$. All the bitstrings start with 1: about a third of them represent numbers that are divisible by 3, as expected.

Height of a gambler's ruin sequence. Taking 1 to mean “up” and 0 to mean “down,” we draw a correspondence between bitstrings and random walks. If we restrict the walks to terminate when they first reach the start level (without ever going below it), we get walks that are equivalent to the gambler's ruin sequences that we introduced in §6.3.

We can use nested REs to classify these walks by height, as follows: To construct a sequence of height bounded by $h + 1$, concatenate any number of sequences of height bounded by h , each bracketed by a 1 on the left and a 0 on the right. Table 8.6 gives the resulting REs and corresponding OGFs for $h = 1, 2, 3$, and 4. Figure 8.2 (on the next page) gives examples that illustrate this construction. The OGF translation is immediate from Theorem 8.4, except that, since 0s and 1s are paired, we only translate one of them to z . These GFs match those involving the Fibonacci polynomials for the height of Catalan trees in §6.10, so the corollary to Theorem 6.9 tells us that the average height of a gambler's ruin sequence is $\sim \sqrt{\pi N}$.

	regular expression	generating function
height ≤ 1	$(10)^*$	$\frac{1}{1-z}$
height ≤ 2	$(1(10)^*0)^*$	$\frac{1}{1 - \frac{z}{1-z}} = \frac{1-z}{1-2z}$
height ≤ 3	$(1(1(10)^*0)^*0)^*$	$\frac{1}{1 - \frac{z}{1 - \frac{z}{1-z}}} = \frac{1-2z}{1-3z+z^2}$
height ≤ 4	$(1(1(1(10)^*0)^*0)^*0)^*$	$\frac{1}{1 - \frac{z}{1 - \frac{z}{1 - \frac{z}{1-z}}}} = \frac{1-3z+z^2}{1-4z+3z^2}$

Table 8.6 REs and OGFs for gambler's ruin sequences

Exercise 8.16 Give the OGFs and REs for gambler's ruin sequences with height no greater than 4, 5, and 6.

Exercise 8.17 Give a regular expression for the set of all strings having no occurrence of the pattern 101101. What is the corresponding generating function?

Exercise 8.18 What is the average position of the *second* disjoint string of P 0s in a random bitstring?

Exercise 8.19 Find the number of different ways to derive each string of N 0s with the RE 0^*00 . Answer the same question for the RE 0^*000^* .

Exercise 8.20 One way to generalize REs is to specify the number of copies implicit in the star operation. In this notation the first sequence in Figure 8.2 is $(10)^{22}$ and the second sequence is $(10)^3 1 (10)^5 0 (10)^3 1 (10)^7 0 (10)^2$, which better expose their structure. Give the generalized REs for the other two sequences in Figure 8.2.

Exercise 8.21 Find the average number of 0s appearing before the first occurrence of each of the bit patterns of length 4 in a random bitstring.

Exercise 8.22 Suppose that a monkey types randomly at a 2-key keyboard. What is the expected number of bits typed before the monkey hits upon a string of $2k$ alternating 0s and 1s?

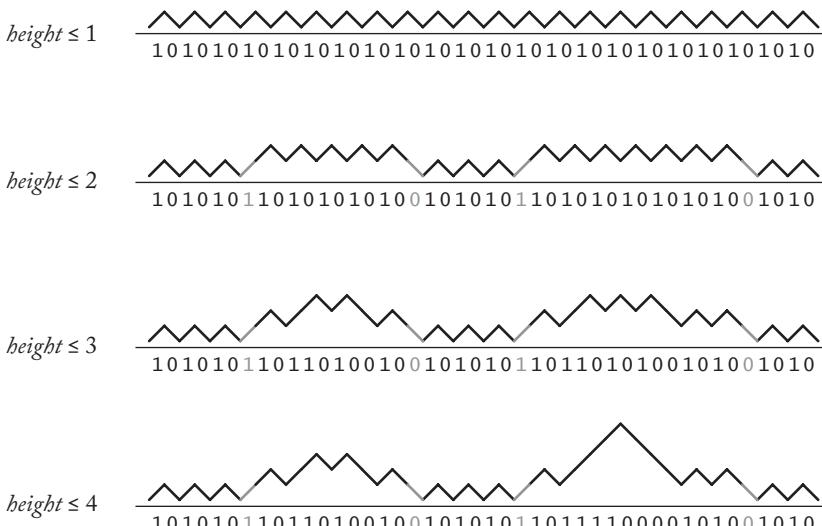


Figure 8.2 Gambler's ruin sequences

8.4 Finite-State Automata and the Knuth-Morris-Pratt Algorithm.

The brute-force algorithm for string matching is quite acceptable for many applications, but, as we saw earlier, it can run slowly for highly self-repetitive patterns. Eliminating this problem leads to an algorithm that not only is of practical interest, but also links string matching to basic principles of theoretical computer science and leads to more general algorithms.

For example, when searching for a string of P 0s, it is very easy to overcome the obvious inefficiency in Program 8.1: When a 1 is encountered at text position i , reset the “pattern” pointer j to the beginning and start looking again at position $i + 1$. This is taking advantage of specific properties of the all-0s pattern, but it turns out that the idea generalizes to give an optimal algorithm for all patterns, which was developed by Knuth, Morris, and Pratt in 1977 [25].

The idea is to build a pattern-specific finite-state automaton that begins at an initial state, examining the first character in the text; scans text characters; and makes state transitions based on the value scanned. Some of the states are designated as *final* states, and the automaton is to terminate in a final state if and only if the associated pattern is found in the text. The implementation of the string search is a simulation of the FSA, based on a table indexed by the state. This makes the implementation extremely simple, as shown in Program 8.2.

```
public static int search(char[] pattern, char[] text)
{
    int P = pattern.length;
    int N = text.length;
    int i, j;
    for (i = 0, j = 0; i < N && j < P; i++)
        j = dfa[text[i]][j];
    if (j == P) return i - P; // Found at offset i-P.
    return N; // Not found.
}
```

Program 8.2 String searching with an FSA (KMP algorithm)

<i>state</i>	0	1	2	3	4	5	6	7
0-transition	0	2	0	4	5	0	2	8
1-transition	1	1	3	1	3	6	7	1

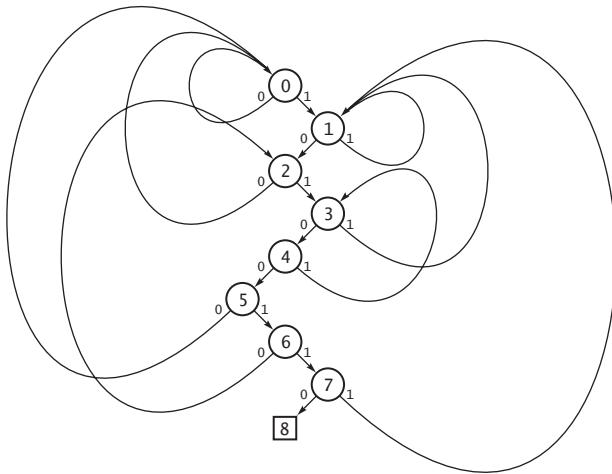


Figure 8.3 Knuth-Morris-Pratt FSA for 10100110

The key to the algorithm is the computation of the transition table, which depends on the pattern. For example, the proper table for the pattern 10100110 is shown, along with a graphic representation of the FSA, in Figure 8.3. When this automaton is run on the sample piece of text given below, it takes the state transitions as indicated—below each character is given the state the FSA is in when that character is examined.

0111010111000110010000011010011000001010111010001
0011123431120011120120000112345678

Exercise 8.23 Give the state transitions for the FSA in Figure 8.3 for searching in the text 010101010101010101010.

Exercise 8.24 Give the state transitions for the FSA in Figure 8.3 for searching in the text 1110010111010110100010100101010011110100110.

Exercise 8.25 Give a text string of length 25 that maximizes (among all strings of length 25) the number of times the KMP automaton from Figure 8.3 reaches step 2.

Once the state table has been constructed (see below), the KMP algorithm is a prime example of an algorithm that is sufficiently sophisticated that it is trivial to analyze, since it just examines each character in the text once. Remarkably, the algorithm *also* can build the transition table by examining each character in the pattern just once!

Theorem 8.5 (KMP string matching). The Knuth-Morris-Pratt algorithm does N bit comparisons when seeking a pattern of length P in a binary text string of length N .

Proof. See the previous discussion. ■

The construction of the state transition table depends on correlations of prefixes of the pattern, as shown in Table 8.7. We define state i to correspond to this situation where the first $i - 1$ characters in the pattern have been matched in the text, but the i th character does not match. That is, state i corresponds to a specific i -bit pattern in the text. For example, for the pattern in Table 8.7, the FSA is in state 4 if and only if the previous four bits in the text were 1010. If the next bit is 0, we would go on to state 5; if the next bit is 1, we know that this is not the prefix of a successful match *and* that the previous five bits in the text were 10101. What is required is to go to the state corresponding to the first point at which this pattern matches itself when shifted right—in this case, state 3. In general, this is precisely the position (measured in bits from the right) of the second 1 in the correlation of this bitstring (0 if the first 1 is the only one).

10100110		0 1	
0	1	0	0 1
11	11	1	2 1
100	100	0	0 3
1011	1001	1	4 1
10101	10101	3	5 3
101000	100000	0	0 6
1010010	1000010	2	2 7
10100111	10000001	1	8 1
mismatch	autocorrelation	first match	table

Table 8.7 Example of KMP state transition table

Exercise 8.26 Give the KMP state transition table for the pattern 110111011101.

Exercise 8.27 Give the state transitions made when using the KMP method to determine whether the text 01101110001110110111101100110111011101 contains the pattern in the previous exercise.

Exercise 8.28 Give the state transition table for a string of $2k$ alternating 0s and 1s.

THE STRING-SEARCHING PROBLEM THAT we have been considering is equivalent to determining whether the text string is in the language described by the (ambiguous) regular expression

$$(0+1)^* \text{ <pattern>} (0+1)^*.$$

This is the *recognition* problem for regular expressions: given a regular expression and a text string, determine if the string is in the language described by the regular expression. In general, regular expression recognition can be done by building an FSA, and properties of such automata can be analyzed using algebraic techniques as we have been doing. As is usually the case, however, more specialized problems can be solved effectively with more specialized techniques. The KMP finite-state automaton is a prime example of this principle.

Generalization to larger alphabets involves a transition table of size proportional to the size of the pattern times the size of the alphabet, though various improvements upon this have been studied. Details on such issues and on numerous text searching applications may be found in books by Gonnet and Baeza-Yates [15] and by Gusfield [20].

Exercise 8.29 Give the KMP state transition table for the pattern 313131, assuming a 4-character alphabet 0, 1, 2, and 3. Give the state transitions made when using the KMP method to determine whether the text 1232032313230313131 contains this pattern.

Exercise 8.30 Prove directly that the language recognized by a deterministic FSA has an OGF that is rational.

Exercise 8.31 Write a computer algebra program that computes the standard rational form of the OGF that enumerates the language recognized by a given deterministic FSA.

8.5 Context-Free Grammars. Regular expressions allow us to define languages in a formal way that turns out to be amenable to analysis. Next in the hierarchy of languages comes the *context-free* languages. For example, we might wish to know:

- How many bitstrings of length $2N$ have N 0s and N 1s?
- Given a random bitstring of length N , how many of its prefixes have equal numbers of 0s and 1s, on the average?
- At what point does the number of 0s in a random bitstring first exceed the number of 1s, on the average?

All these problems can be solved using *context-free grammars*, which are more expressive than regular expressions. Though the first question is trivial combinatorially, it is known from language theory that such a set cannot be described with regular expressions, and context-free languages are needed. As with REs, automatic mechanisms involving generating functions that correspond to the symbolic method are effective for enumerating unambiguous context-free languages and thus open the door to studying a host of interesting questions.

To begin, we briefly summarize some basic definitions from formal language theory. A context-free grammar is a collection of *productions* relating *nonterminal symbols* and letters (also called *terminal symbols*) by means of unions and concatenation products. The basic operations are similar to those for regular expressions (the “star” operation is not needed), but the introduction of nonterminal symbols creates a more powerful descriptive mechanism because of the possibility of nonlinear recursion. A language is context-free if it can be described by a context-free grammar. Indeed, we have actually been using mechanisms equivalent to context-free grammars to define some of the combinatorial structures that we have been analyzing.

For example, our definition of binary trees, in Chapter 6, can be recast formally as the following unambiguous grammar:

```
<bin tree> ::= <ext node> | <int node> <bin tree> <bin tree>
<int node> ::= 0
<ext node> ::= 1
```

Nonterminal symbols are enclosed in angle brackets. Each nonterminal can be considered as representing a context-free language, defined by direct assignment to a letter of the alphabet, or by the union or concatenation product

operations. Alternatively, we can consider each equation as a *rewriting rule* indicating how the nonterminal can be rewritten, with the vertical bar denoting alternate rewritings and juxtaposition denoting concatenation. This grammar generates bitstrings associated with binary trees according to the one-to-one correspondence introduced in Chapter 6: visit the nodes of the tree in preorder, writing 0 for internal nodes and 1 for external nodes.

Now, just as in the symbolic method, we view each nonterminal as representing the set of strings that can be derived from it using rewriting rules in the grammar. Then, we again have a general approach for translating unambiguous context-free grammars into functional equations on generating functions:

- Define an OGF corresponding to each nonterminal symbol.
- Translate occurrences of terminal symbols to variables.
- Translate concatenation in the grammar to multiplication of OGFs.
- Translate union in the grammar to addition of OGFs.

When this process is carried out, there results a system of polynomial equations on the OGFs. The following essential relationship between CGFs and OGFs was first observed by Chomsky and Schützenberger [3].

Theorem 8.6 (OGFs for context-free grammars). Let $\langle A \rangle$ and $\langle B \rangle$ be nonterminal symbols in an unambiguous context-free grammar and suppose that $\langle A \rangle \mid \langle B \rangle$ and $\langle A \rangle \langle B \rangle$ are also unambiguous. If $A(z)$ is the OGF that enumerates the strings that can be derived from $\langle A \rangle$ and $B(z)$ is the OGF that enumerates $\langle B \rangle$, then

$A(z) + B(z)$ is the OGF that enumerates $\langle A \rangle \mid \langle B \rangle$

$A(z)B(z)$ is the OGF that enumerates $\langle A \rangle \langle B \rangle$.

Moreover, any OGF that enumerates an unambiguous context-free language satisfies a polynomial equation whose terms are themselves polynomials with rational coefficients. (Such functions are said to be *algebraic*.)

Proof. The first part of the theorem follows immediately as for the symbolic method. Each production in the CFG corresponds to a OGF equation, so the result is a system of polynomial equations on the OGFs. Solving for the OGF that enumerates the language may be achieved by an *elimination* process that reduces a polynomial system to a unique equation relating the variable z and the OGF under consideration. For instance, *Gröbner basis* algorithms

that are implemented in some computer algebra systems are effective for this purpose (see Geddes, et al. [14]). If $L(z)$ is the OGF of an unambiguous context-free language, this process leads to a bivariate polynomial $P(z, y)$ such that $P(z, L(z)) = 0$, which proves that $L(z)$ is algebraic. ■

This theorem relates basic operations on languages to OGFs using the symbolic method in the same way as Theorem 8.4, but the expressive power of context-free grammars by comparison to regular expressions leads to differences in the result in two important respects. First, a more general type of recursive definition is allowed (it can be nonlinear) so that the resulting OGF has a more general form—the system of equations is in general nonlinear. Second, ambiguity plays a more essential role. Not every context-free language has an unambiguous grammar (the ambiguity problem is even undecidable), so we can claim the OGF to be algebraic only for languages that have an unambiguous grammar. By contrast, it is known that there exists an unambiguous regular expression for every regular language, so we can make the claim that OGFs for all regular languages are rational.

Theorem 8.6 spells out a method for solving a “context-free” counting problem, by these last two steps:

- Solve to get an algebraic equation for the OGF.
- Solve, expand, and/or develop asymptotic estimates for coefficients.

In some cases, the solution of that equation admits to explicit forms that can be expanded, as we see in a later example. In some other cases, this solution can be a significant challenge, even for computer algebra systems. But one of the hallmarks of analytic combinatorics (see [12]) is a *universal* transfer theorem of sweeping generality that tells us that the growth rate of the coefficients is of the form $\beta^n / \sqrt{n^3}$. Since we cannot do justice to this theorem without appealing to complex asymptotics, we restrict ourselves in this book to examples where explicit forms are easily derived.

2-ordered permutations. The discussion in §7.6 about enumerating 2-ordered permutations corresponds to developing the following unambiguous context-free grammar for strings with equal numbers of 0s and 1s:

$$\begin{aligned} <\mathbf{S}> &:= <\mathbf{U}>1<\mathbf{S}> \mid <\mathbf{D}>0<\mathbf{S}> \mid \epsilon \\ <\mathbf{U}> &:= <\mathbf{U}><\mathbf{U}>1 \mid 0 \\ <\mathbf{D}> &:= <\mathbf{D}><\mathbf{D}>0 \mid 1 \end{aligned}$$

The nonterminals in this grammar may be interpreted as follows: $\langle S \rangle$ corresponds to all bitstrings with equal numbers of 0s and 1s; $\langle U \rangle$ corresponds to all bitstrings with precisely one more 0 than 1, with the further constraint that no prefix has equal numbers of 0s and 1s; and $\langle D \rangle$ corresponds to all bitstrings with precisely one more 1 than 0, with the further constraint that no prefix has equal numbers of 0s and 1s.

Now, by Theorem 8.6, each production in the grammar translates to a functional equation on the generating functions:

$$\begin{aligned}S(z) &= zU(z)S(z) + zD(z)S(z) + 1 \\U(z) &= z + zU^2(z) \\D(z) &= z + zD^2(z).\end{aligned}$$

In this case, of course, $U(z)$ and $D(z)$ are familiar generating functions from tree enumeration, so we can solve explicitly to get

$$U(z) = D(z) = \frac{1}{2z}(1 - \sqrt{1 - 4z^2}),$$

then substitute to find that

$$S(z) = \frac{1}{\sqrt{1 - 4z^2}} \quad \text{so} \quad [z^{2N}]S(z) = \binom{2N}{N}$$

as expected.

Gröbner basis elimination. In general, explicit solutions might not be available, so we sketch for this problem how the Gröbner basis elimination process will systematically solve this system. First, we note that $D(z) = U(z)$ because both satisfy the same (irreducible) equation. Thus, what is required is to eliminate U from the system of equations

$$\begin{aligned}P_1 &\equiv S - 2zUS - 1 = 0 \\P_2 &\equiv U - zU^2 - z = 0.\end{aligned}$$

The general strategy consists of eliminating higher-degree monomials from the system by means of repeated combinations of the form $AP - BQ$, with

A, B monomials and P, Q polynomials subject to elimination. In this case, forming $UP_1 - 2SP_2$ cross-eliminates the U^2 to give

$$P_3 \equiv -US - U + 2zS = 0.$$

Next, the US term can be eliminated by forming $2zP_3 - P_1$, so we have

$$P_4 \equiv -2Uz + 4Sz^2 - S + 1 = 0.$$

Finally, the combination $P_1 - SP_4$ completely eliminates U , and we get

$$P_5 \equiv S^2 - 1 - 4S^2z^2 = 0$$

and therefore $S(z) = 1/\sqrt{1 - 4z^2}$ as before. We have included these details for this example to illustrate the fundamental point that Theorem 8.6 gives an “automatic” way to enumerate unambiguous context-free languages. This is of particular importance with the advent of computer algebra systems that can perform the routine calculations involved.

Ballot problems. The final result above is elementary, but context-free languages are of course very general, so the same techniques can be used to solve a diverse class of problems. For example, consider the classical *ballot problem*: Suppose that, in an election, candidate 0 receives $N + k$ votes and candidate 1 receives N votes. What is the probability that candidate 0 is always in the lead during the counting of the ballots? In the present context, this problem can be solved by enumerating the number of bitstrings with $N + k$ 0s and N 1s that have the property that no prefix has an equal number of 0s and 1s. This is also the number of paths through an $(N + k)$ -by- N lattice that do not touch the main diagonal. For $k = 0$ the answer is zero, because, if both candidates have N votes, they must be tied somewhere during the counting, if only at the end. For $k = 1$ the count is precisely $[z^{2N+1}]U(z)$ from our discussion of 2-ordered permutations. For $k = 3$, we have the grammar

$$\begin{aligned} <\mathbf{B}> &:= <\mathbf{U}><\mathbf{U}><\mathbf{U}> \\ <\mathbf{U}> &:= <\mathbf{U}><\mathbf{U}>1 \quad | \quad 0 \end{aligned}$$

and the answer is $[z^{2N+3}](U(z))^3$. This immediately generalizes to give the result for all k .

Theorem 8.7 (Ballot problem). The probability that a random bitstring with k more 0s than 1s has the property that no prefix has an equal number of 0s and 1s is $k/(2N + k)$.

Proof. By the previous discussion, this result is given by

$$\frac{[z^{2N+k}]U(z)^k}{\binom{2N+k}{N}} = \frac{k}{2N+k}.$$

Here, the coefficients are extracted by a direct application of Lagrange inversion (see §6.12). ■

The ballot problem has a rich history, dating back to 1887. For detailed discussions and numerous related problems, see the books by Feller [9] and Comtet [7].

Beyond the direct relationship to trees, the problems like those we have been considering arise frequently in the analysis of algorithms in connection with so-called *history* or *sequence of operations* analysis of dynamic algorithms and data structures. For example, the gambler's ruin problem is equivalent to determining the probability that a random sequence of “push” and “pop” operations on an initially empty pushdown stack is “legal” in the sense that it never tries to pop an empty stack and leaves an empty stack. The ballot problem generalizes to the situation where the sequence is legal but leaves k items on the stack. Other applications may involve more operations and different definitions of legal sequences—some examples are given in the exercises below. Such problems typically can be approached via context-free grammars. A number of applications of this type are discussed in an early paper by Pratt [29]; see also Knuth [23].

Exercise 8.32 Given a random bitstring of length N , how many of its prefixes have equal numbers of 0s and 1s, on the average?

Exercise 8.33 What is the probability that the number of 0s in a random bitstring never exceeds the number of 1s?

Exercise 8.34 Given a random bitstring of length N , how many of its prefixes have k more 0s than 1s, on the average? What is the probability that the number of 0s in a random bitstring never exceeds the number of 1s by k ?

Exercise 8.35 Suppose that a stack has a fixed capacity M . What is the probability that a random sequence of N push and pop operations on an initially empty push-down stack never tries to pop the stack when it is empty or push when it is full?

Exercise 8.36 [Pratt] Consider a data structure with one “insert” and two different types of “remove” operations. What is the probability that a random sequence of operations of length N is legal in the sense that the data structure is empty before and after the sequence, and “remove” is always applied to a nonempty data structure?

Exercise 8.37 Answer the previous exercise, but replace one of the “remove” operations with an “inspect” operation, which is applied to a nonempty data structure but does not remove any items.

Exercise 8.38 Suppose that a monkey types random parentheses, hitting left and right with equal probability. What is the expected number of characters typed before the monkey hits upon a legal balanced sequence? For example, $((()))$ and $((())())$ are legal but $((()$) and $(())()$ are not.

Exercise 8.39 Suppose that a monkey types randomly at a 26-key keyboard that has 26 letters A through z. What is the expected number of characters typed before the monkey types a palindrome of length at least 10? That is for some $k \geq 10$, what is the expected number of characters typed before the last k characters are the same when taken in reverse order? Example: KJASDLKUYMBUWKASDMBVJDMADAMIMADAM.

Exercise 8.40 Suppose that a monkey types randomly at a 32-key keyboard that has 26 letters A through z; the symbols +, *, (, and); a space key; and a period. What is the expected number of characters typed before the monkey hits upon a legal regular expression? Assume that spaces can appear anywhere in a regular expression and that a legal regular expression must be enclosed in parentheses and have exactly one period, at the end.

8.6 Tries. Any set of N distinct bitstrings (which may vary in length) corresponds to a *trie*, a binary tree structure where we associate links with bits. Since tries can be used to represent sets of bitstrings, they provide an alternative to binary trees for conventional symbol-table applications. In this section, we focus on the fundamental relationship between sets of bitstrings and binary tree structures that tries embody. In the next section, we describe many applications of tries in computer science. Following that, we look at the analysis of properties of tries, one of the classic problems in the analysis of algorithms. Then, we briefly discuss the algorithmic opportunities and analytic challenges presented by extensions to M -way tries and sets of bytestrings (strings drawn from an M -character alphabet).

We start with a few examples that illustrate how to associate sets of strings to binary trees. Given a binary tree, imagine that the left links are labelled 0 and the right links are labelled 1, and identify each external node with the labels of the links on the path from the root to that node. This gives a mapping from binary trees to sets of bitstrings. For example, the tree on the left in Figure 8.4 maps to the set of strings

000 001 01 10 11000 11001 11010 11011 1110 1111,

and the tree on the right to

0000 0001 0010 0011 010 011 100 101 110 111.

(Which set of bitstrings does the trie in the middle represent?) There is one and only one set of bit strings associated with any given trie in this way. All

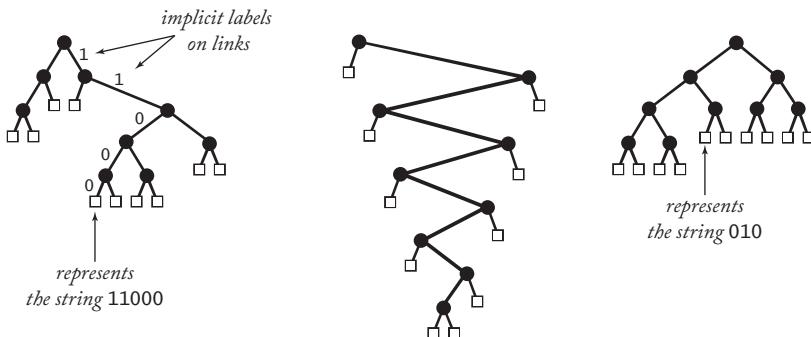


Figure 8.4 Three tries, each representing 10 bitstrings

the sets of bitstrings that are obtained in this way have by construction the *prefix-free* property: no string is the prefix of another.

Conversely (and more generally), given a set of bitstrings that satisfy the prefix-free property, we can uniquely construct an associated binary tree structure if we can associate bitstrings with external nodes, by recursively dividing the set according to the leading bit of the strings, as in the following formal definition. This is one of several possible ways to associate tries with sets of bit strings; we will soon consider alternatives.

Definition Given a set \mathcal{B} of bitstrings that is prefix-free, the associated trie is a binary tree defined recursively as follows: If \mathcal{B} is empty, the trie is null and represented by a void external node. If $|\mathcal{B}| = 1$, the trie consists of one external node corresponding to the bitstring. Otherwise, define \mathcal{B}_0 (respectively, \mathcal{B}_1) to be the set of bitstrings obtained by taking all the members of \mathcal{B} that begin with 0 (respectively, 1) and removing the initial bit from each. Then the trie for \mathcal{B} is an internal node connected to the trie for \mathcal{B}_0 on the left and the trie for \mathcal{B}_1 on the right.

A trie for N bitstrings has N nonvoid external nodes, one corresponding to each bitstring, and may have any number of void external nodes. As earlier, by considering 0 as “left” and 1 as “right,” we can reach the external node corresponding to any bitstring by starting at the root and proceeding down the trie, moving left or right according to the bits in the string read from left to right. This process ends at the external node corresponding to the bitstring as soon as it can be distinguished from all the other bitstrings in the trie.

Our definition is convenient and reasonable for studying properties of tries, but several practical questions naturally arise that are worth considering:

- How do we handle sets of strings that are not prefix-free?
- What role do void nodes play? Are they necessary?
- Adding more bits to the bitstrings does not change the structure. How do we handle the leftover bits?

Each of these has important implications, both for applications and analysis. We consider them in turn.

First, the prefix-free assumption is justified, for example, if the strings are infinitely long, which is also a convenient assumption to make when analyzing properties of tries. Indeed, some applications involve implicit bitstrings that are potentially infinitely long. It is possible to handle prefix strings by

associating extra information with the internal nodes; we leave this variant for exercises.

Second, the void external nodes correspond to situations where bitstrings have bits in common that do not distinguish them from other members of the set. For example, if all the bitstrings start with a 0-bit, then the right child of the root of the associated trie would be such a node, not corresponding to any bitstring in the set. Such nodes can appear throughout the trie and need to be marked to distinguish them from external nodes that represent bit strings. For example, Figure 8.5 shows three tries with 10 external nodes, of which 3, 8, and 0 are void, respectively. In the figure, the void nodes are represented by small black squares and the nonvoid nodes (which each correspond to a string) are represented by larger open squares. The trie on the left represents the set of bitstrings

000 001 11000 11001 11100 11101 1111

and the trie on the right represents the set

0000 0001 0010 0011 010 011 100 101 110 111.

(Which set of bitstrings does the trie in the middle represent?) It is possible to do a precise analysis of the number of void external nodes needed for random bitstrings. It is also possible to arrange matters so that unneeded bits do not directly correspond to unneeded nodes in the trie structure but are represented otherwise. We discuss these matters in some detail later.

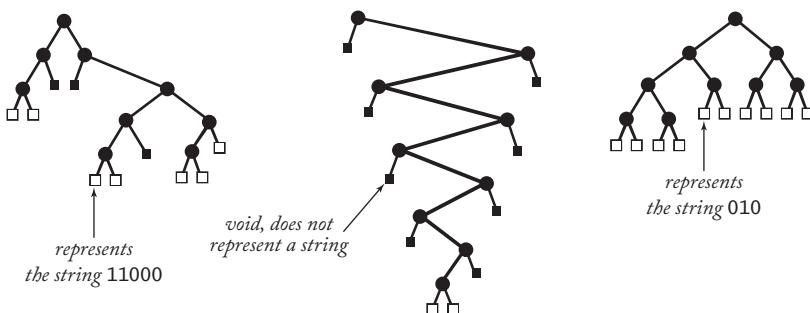


Figure 8.5 Three tries, representing 7, 2, and 10 bitstrings, respectively

Third, the fact that adding more bits to the bitstrings does not change the structure follows from the “if $|\mathcal{B}| = 1$ ” clause in the definition, which is there because in many applications it is convenient to stop the trie branching as soon as the bitstrings are distinguished. For finite bitstrings, this condition could be removed, and the branching could continue until the end of each bitstring is reached. We refer to such a trie as a *full* trie for the set of bitstrings.

Enumeration. The recursive definition that we have given gives rise to binary tree structures with the additional properties that (i) external nodes may be void and (ii) children of leaves must be nonvoid. That is, we never have two void nodes that are siblings, or a void and a nonvoid node as siblings. To enumerate all the different tries, we need to consider all the possible trie structures and all the different ways to mark the external nodes as void or nonvoid, consistent with these rules. Figure 8.6 shows all the different tries with four or fewer external nodes.

Minimal sets. We also refer to the *minimal set* of bitstrings for a trie. These sets are nothing more than encodings of the paths from the root to each nonvoid external node. For example, the bitstring sets that we have given for

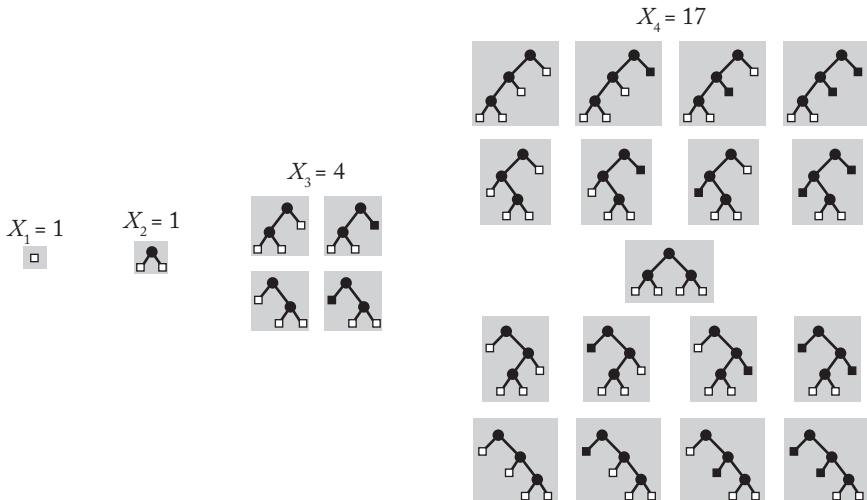


Figure 8.6 Tries with 1, 2, 3, and 4 external nodes

Figure 8.4 and for Figure 8.5 are both minimal. Figure 8.7 gives the minimal bitstring sets associated with each of the tree shapes with five external nodes (each of the tries with five external nodes, all of which are nonvoid). To find the minimal bitstring sets associated with *any* trie with five external nodes, simply delete the bitstrings corresponding to any void external nodes in the associated tree structure in the figure.

WE MIGHT ANALYZE PROPERTIES of trie structures using the symbolic method, in a manner analogous to Catalan trees in Chapter 5; we leave such questions for exercises. Instead, in the context of the analysis of algorithms, we focus on sets of bitstrings and bitstring algorithms—where tries are most often used—and concentrate on viewing tries as mechanisms to efficiently *distinguish* among a set of strings, and as structures to efficiently *represent* sets of strings. Moreover, we work with probability distributions induced when the strings are random, an appropriate model in many situations. Before doing so, we consider algorithmic applications of tries.

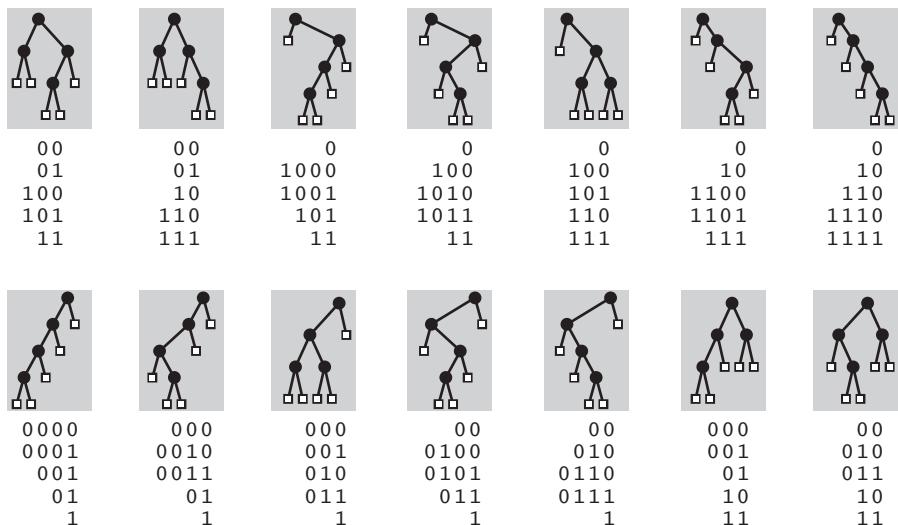


Figure 8.7 Bitstring sets for tries with five external nodes (none void)

Exercise 8.41 Give the three tries corresponding to the minimal sets of strings for Figure 8.5, but reading each string in right-to-left order.

Exercise 8.42 There are $\binom{8}{5} = 56$ different sets of five three-bit bitstrings. Which trie is associated with the most of these sets? The least?

Exercise 8.43 Give the number of different tries that have the same structure as each of the tries in Figures 8.4 and 8.5.

Exercise 8.44 How many different tries are there with N external nodes?

Exercise 8.45 What proportion of the external nodes are void in a “random” trie (assuming each different trie structure to be equally likely to occur)?

Exercise 8.46 Given a finite set of strings, devise a simple test to determine whether there are any void external nodes in the corresponding trie.

8.7 Trie Algorithms. Binary strings are ubiquitous in digital computing, and trie structures are naturally associated with sets of binary strings, so it should not be surprising that there are a number of important algorithmic applications of tries. In this section, we survey a few such algorithms, to motivate the detailed study of the properties of tries that we tackle in §8.8.

Tries and digital searching. Tries can be used as the basis for algorithms for searching through a collection of binary data in a manner similar to binary search trees, but with bit comparisons replacing key comparisons.

Search tries. Treating bitstrings as keys, we can use tries as the basis for a conventional symbol table implementation such as Program 6.2. Nonvoid external nodes hold references to keys that are in the symbol table. To search, set x to the root and b to 0, then proceed down the trie until an external node is encountered, incrementing b and setting x to $x.\text{left}$ if the b th bit of the key is 0, or setting x to $x.\text{right}$ if the b th bit of the key is 1. If the external node that terminates the search is void, then the bitstring is not in the trie; otherwise, we can compare the key to the bitstring referenced by the nonvoid external node. To insert, follow the same procedure, then store a reference to the key in the void external node that terminates the search, making it nonvoid. This can be a very efficient search algorithm under proper conditions on the set of keys involved; for details see Knuth [23] or Sedgewick [32]. The analysis that we will consider can be used to determine how the performance

of tries might compare with that of binary search trees for a given application. As discussed in Chapter 1, the first consideration in attempting to answer such a question is to consider properties of the implementation. This is especially important in this particular case, because accessing individual bits of keys can be very expensive on some computers if not done carefully.

Patricia tries. We will see below that about 44% of the external nodes in a random trie are void. This factor may be unacceptably high. The problem can be avoided by “collapsing” one-way internal nodes and keeping the index of the bit to be examined with each node. The external path length of this trie is somewhat smaller, though some extra information has to be associated with each node. The critical property that distinguishes Patricia tries is that there are no void external nodes, or, equivalently, there are $N - 1$ internal nodes. Various techniques are available for implementing search and insertion using Patricia tries. Again, details may be found in Knuth [23] or Sedgewick [32].

Radix-exchange sort. As mentioned in Chapter 1, a set of bitstrings of equal length can be sorted by partitioning them to put all those beginning with 0 before all those beginning with 1 (using a process similar to the partitioning process of quicksort) then sorting the two parts recursively. This method, called *radix-exchange sort*, bears the same relationship to tries as quicksort does to binary search trees. The time required by the sort is essentially proportional to the number of bits examined. For keys comprised of random bits, this turns out to be the same as the “nonvoid external path length” of a random trie—the sum of the distances from the root to each of the nonvoid external nodes.

Trie encoding. Any trie with labelled external nodes defines a *prefix code* for the labels of the nodes. For example, if the external nodes in the trie on the left in Figure 8.4 are labelled, left to right, with the letters

(space) D O E F C R I P X

then the bitstring

111011010101100011011111000110010100110

encodes the phrase

PREFIX CODE.

Decoding is simple: starting at the root of the trie and the beginning of the bitstring, travel through the trie as directed by the bitstring (left on 0, right on

1), and, each time an external node is encountered, output the label and restart at the root. If frequently used letters are assigned to nodes with short paths, then the number of bits used in such an encoding will be significantly fewer than for the standard encoding. The well-known *Huffman encoding* method finds an optimal trie structure for given letter frequencies (see Sedgewick and Wayne [33] for details).

Tries and pattern matching. Tries can also be used as a basic data structure for searching for multiple patterns in text files. For example, tries have been used successfully in the computerization of large dictionaries for natural languages and other similar applications. Depending upon the application, the trie can contain the patterns or the text, as described below.

String searching with suffix tries. In an application where the text string is fixed (as for a dictionary) and many pattern lookups are to be handled, the search time can be dramatically reduced by preprocessing the text string, as follows: Consider the text string to be a set of N strings, one starting at each position of the text string and running to the end of the string (stopping k characters from the end, where k is the length of the shortest pattern to be sought). Build a trie from this set of strings (such a trie is called the *suffix trie* for the text string). To find out whether a pattern occurs in the text, proceed down the trie from the root, going left on 0 and right on 1 as usual, according to the pattern bits. If a void external node is hit, the pattern is not in the text; if the pattern exhausts on an internal node, it is in the text; and if an external node is hit, compare the remainder of the pattern to the text bits represented in the external node as necessary to determine whether there is a match. This algorithm was used effectively in practice for many years before it was finally shown by Jacquet and Szpankowski that a suffix trie for a random bitstring is roughly equivalent to a search trie built from a set of random bitstrings [21][22]. The end result is that a string search requires a small constant times $\lg N$ bit inspections on the average—a very substantial improvement over the cost of the basic algorithm in situations where the initial cost of building the trie can be justified (for example, when a huge number of patterns are to be sought in the same text).

Searching for multiple patterns. Tries can also be used to find multiple patterns in one pass through a text string, as follows: First, build a full trie from the pattern strings. Then, for each position i in the text string, start at the top of the trie and match characters in the text while proceeding down the trie,

going left for 0s in the text and right for 1s. Such a search must terminate at an external node. If the external node is not void, then the search was successful: one of the strings represented by the trie was found starting at position i in the text. If the external node is void, then none of the strings represented by the trie starts at i , so that the text pointer can be incremented to $i + 1$ and the process restarted at the top of the trie. The analysis below implies that this requires $O(N \lg M)$ bit inspections, as opposed to the $O(NM)$ cost of applying the basic algorithm M times.

Trie-based finite-state automata. When the search process just described terminates in a void external node, we can do better than going to the top of the trie and backing up the text pointer, in precisely the same manner as with the Knuth-Morris-Pratt algorithm. A termination at a void node tells us not just that the sought string is not in the database, but also which characters in the text precede this mismatch. These characters show exactly where the next search will require us to examine a text character where we can totally avoid the comparisons wasted by the backup, just as in the KMP algorithm. Indeed, Program 8.2 can be used for this application with no modification; we need only build the FSA corresponding to a set of strings rather than to a single string. For example, the FSA depicted in Figure 8.8 corresponds to the set of strings 000, 011, and 1010. When this automaton is run on the sample piece of text shown on the next page, it takes the state transitions as indicated—

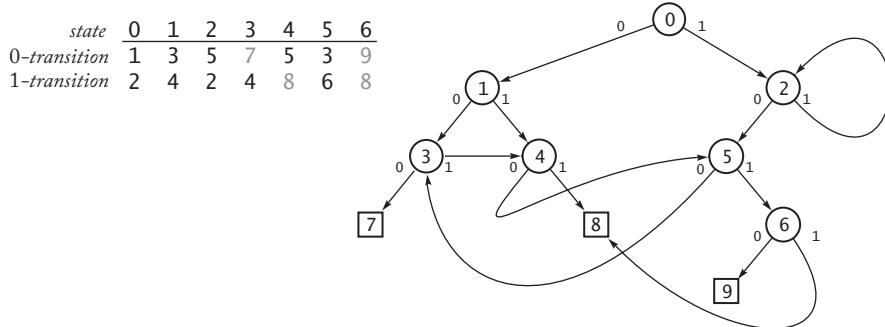


Figure 8.8 Aho-Corasick FSA for 000, 011, and 1010

below each character is given the state the FSA is in when that character is examined.

```
11110010010010111010000011010011000001010111010001
02222534534534568
```

In this case, the FSA stops in state 8, having found the pattern 011. The process of building such an automaton for a given set of patterns is described by Aho and Corasick [1].

Distributed leader election. The random trie model is very general. It corresponds to a general probabilistic process where “individuals” (keys, in the case of trie search) are recursively separated by coin flippings. That process can be taken as the basis of various resource allocation strategies, especially in a distributed context. As an example, we will consider the following distributed algorithm for electing a leader among N individuals sharing an access channel. The method proceeds by rounds; individuals are selected or eliminated according to coin flips. Given a set of individuals:

- If the set is empty, then report failure.
- If the set has one individual, then declare that individual the leader.
- If the set has more than one individual, flip independent 0–1 coins for all members of the set and invoke the procedure recursively for the subset of individuals who got 1.

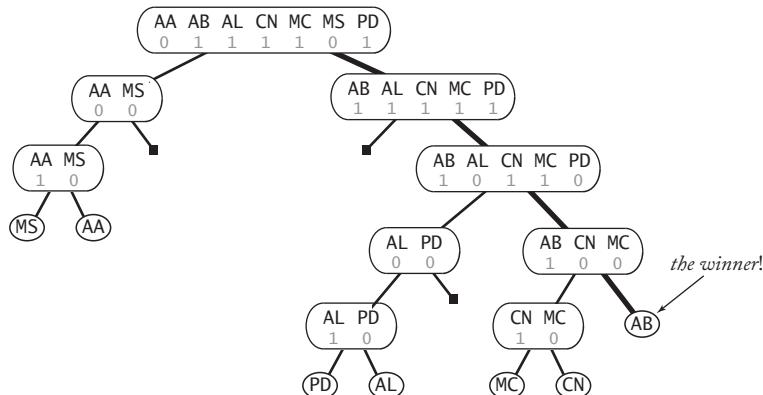


Figure 8.9 Distributed leader election

Figure 8.9 shows an example. We show the full trie where we imagine that the losers need to select a winner among that group, and so forth, extending the method to give a full ranking of the individuals. At the first stage, AB, AL, CN, MC, and PD all flip heads and survive. At the second stage, they all flip heads again, so no one is eliminated. This leads to a void node in the trie. Eventually, AB is declared the winner, the only individual to flip heads at every opportunity. If we start with N individuals, then we expect N to be reduced roughly to $N/2, N/4, \dots$ in the course of the execution of the algorithm. Thus, we expect the procedure to terminate in about $\lg N$ steps, and a more precise analysis may be desirable. Also, the algorithm may fail (if everyone flips 0 and is eliminated with no leader elected), and we are also interested in knowing the probability that the algorithm is successful.

THIS LIST OF ALGORITHMS and applications is representative, and demonstrates the fundamental importance of the trie data structure in computer applications. Not only are tries important as explicit data structures, but also they arise implicitly in algorithms based on bits, or in algorithms where truly “binary” decisions are made. Thus, analytic results describing properties of random tries have a variety of applications.

Exercise 8.47 How many bits are examined when using the trie in the middle in Figure 8.5 to search for one of the patterns 1010101010 or 1010101011 in the text string 1001010011110010101000101010100010010?

Exercise 8.48 Given a set of pattern strings, describe a method for counting the number of times one of the patterns occurs in a text string.

Exercise 8.49 Build the suffix trie for patterns of eight bits or longer from the text string 1001010011110010101000101010100010010.

Exercise 8.50 Give the suffix tries corresponding to all four-bit strings.

Exercise 8.51 Give the Aho-Corasick FSA for the set of strings 01 100 1011 010.

8.8 Combinatorial Properties of Tries. As combinatorial objects, tries have been studied only recently, especially by comparison with classical combinatorial objects such as permutations and trees. As we will see, full understanding of even the most basic properties of tries requires the full array of analytic tools that we consider in this book.

Certain properties of tries naturally present themselves for analysis. How many void external nodes might be expected? What is the average external path length or the average height? As with binary search trees, knowledge of these basic properties gives the information necessary to analyze string searching and other algorithms that use trees.

A related, more fundamental, point to consider is that the model of computation used in the analysis needs to differ in a fundamental way: for binary search trees, only the relative order of the keys is of interest; for tries, the binary representation of the keys as bitstrings must come into play. What exactly is a random trie? Though several models are possible, it is natural to consider a random trie to be one built from a set of N random infinite bitstrings. This model is appropriate for many of the important trie algorithms, such as symbol table implementations for bitstring keys. As mentioned earlier, the model is sufficiently robust that it well approximates other situations as well, such as suffix tries.

Thus, we will consider the analysis of properties of tries under the assumption that each bit in each bitstring is independently 0 or 1 with probability $1/2$.

Theorem 8.8 (Trie path length and size). The trie corresponding to N random bitstrings has external path length $\sim N \lg N$, on the average. The mean number of internal nodes is asymptotic to $(1/\ln 2 \pm 10^{-5})N$.

Proof. We start with a recurrence: for $N > 0$, the probability that exactly k of the N bitstrings begins with a 0 is the Bernoulli probability $\binom{N}{k}/2^N$, so if we define C_N to be the average external path length in a trie corresponding to N random bitstrings, we must have

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

This is precisely the recurrence describing the number of bit inspections used by radix-exchange sort that we examined for Theorem 4.9 in §4.9, where we

showed that

$$C_N = N![z^N]C(z) = N \sum_{j \geq 0} \left(1 - \left(1 - \frac{1}{2^j}\right)^{N-1}\right)$$

and then we used the exponential approximation to deduce that

$$C_N \sim N \sum_{j \geq 0} (1 - e^{-N/2^j}) \sim N \lg N.$$

A more precise estimate exposes a periodic fluctuation in the value of this quantity as N increases. As we saw in Chapter 4, the terms in the sum are exponentially close to 1 for small k and exponentially close to 0 for large k , with a transition when k is near $\lg N$ (see Figure 4.5). Accordingly, we split the sum:

$$\begin{aligned} C_N/N &\sim \sum_{0 \leq j < \lfloor \lg N \rfloor} (1 - e^{-N/2^j}) + \sum_{j \geq \lfloor \lg N \rfloor} (1 - e^{-N/2^j}) \\ &= \lfloor \lg N \rfloor - \sum_{0 \leq j < \lfloor \lg N \rfloor} e^{-N/2^j} + \sum_{j \geq \lfloor \lg N \rfloor} (1 - e^{-N/2^j}) \\ &= \lfloor \lg N \rfloor - \sum_{j < \lfloor \lg N \rfloor} e^{-N/2^j} + \sum_{j \geq \lfloor \lg N \rfloor} (1 - e^{-N/2^j}) + O(e^{-N}) \\ &= \lfloor \lg N \rfloor - \sum_{j < 0} e^{-N/2^{j+\lfloor \lg N \rfloor}} + \sum_{j \geq 0} (1 - e^{-N/2^{j+\lfloor \lg N \rfloor}}) + O(e^{-N}). \end{aligned}$$

Now, separating out the fractional part of $\lg N$ as we did earlier in Chapter 2 (see Figures 2.3 and 2.4), we have

$$\lfloor \lg N \rfloor = \lg N - \{\lg N\} \quad \text{and} \quad N/2^{\lfloor \lg N \rfloor} = 2^{\lg N - \lfloor \lg N \rfloor} = 2^{\{\lg N\}}.$$

This leads to the expression

$$C_N/N \sim \lg N - \epsilon(N)$$

where

$$\epsilon(N) \equiv \{\lg N\} + \sum_{j < 0} e^{-2^{\{\lg N\}-j}} - \sum_{j \geq 0} (1 - e^{-2^{\{\lg N\}-j}}).$$

This function is clearly periodic, with $\epsilon(2N) = \epsilon(N)$ because its dependence on N is solely in terms of $\{\lg N\}$. This observation does not immediately rule out the possibility that $\epsilon(N)$ might be constant, but it is easy to check that it is not: $(C_N - N \lg N)/N$ does not approach a limit, but it fluctuates as N grows. Figure 8.10 is a plot of the function after subtracting its mean value for $N < 256$, and Table 8.7 shows exact values of $\epsilon(N)$ for $4 \leq N \leq 16$ (also for any power of 2 times these values). The function $\epsilon(N)$ is very close to 1.332746 numerically, and the amplitude of the fluctuating part is $< 10^{-5}$.

A similar analysis can be used to analyze the number of internal nodes in a trie on N bitstrings, which is described by the recurrence

$$A_N = 1 + \frac{1}{2^N} \sum_k \binom{N}{k} (A_k + A_{N-k}) \quad \text{for } N > 1 \text{ with } A_0 = A_1 = 0.$$

In this case, the periodic fluctuation appears in the leading term:

$$A_N \sim \frac{N}{\ln 2} (1 + \hat{\epsilon}(N))$$

where the absolute value of $\hat{\epsilon}(N)$ is less than 10^{-5} . The details of this analysis are similar to the preceding analysis, and are left as an exercise. ■

N	$\{\lg N\}$	$\sum_{j<0} e^{-2\{\lg N\}-j}$	$\sum_{j\geq 0} (1 - e^{-2\{\lg N\}-j})$	$-\epsilon(N)$
4	0.000000000	0.153986497	1.486733879	1.332747382
5	0.321928095	0.088868348	1.743543002	1.332746559
6	0.584962501	0.052271965	1.969979089	1.332744624
7	0.807354922	0.031110097	2.171210673	1.332745654
8	0.000000000	0.153986497	1.486733879	1.332747382
9	0.169925001	0.116631646	1.619304291	1.332747643
10	0.321928095	0.088868348	1.743543002	1.332746559
11	0.459431619	0.068031335	1.860208218	1.332745265
12	0.584962501	0.052271965	1.969979089	1.332744624
13	0.700439718	0.040279907	2.073464469	1.332744844
14	0.807354922	0.031110097	2.171210673	1.332745654
15	0.906890596	0.024071136	2.263708354	1.332746622
16	0.000000000	0.153986497	1.486733879	1.332747382

$$\gamma/\ln 2 + 1/2 \approx 1.332746177$$

Table 8.8 Periodic term in trie path length

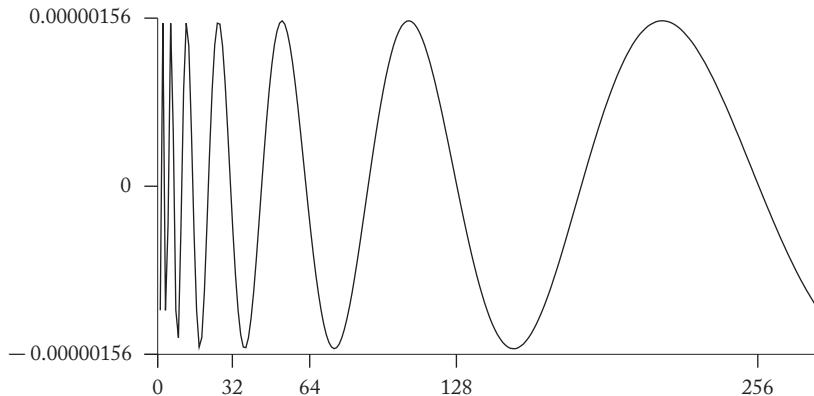


Figure 8.10 Periodic fluctuation in trie path length

This result is due to Knuth [23], who derived explicit expressions for the mean value and for $\epsilon(N)$. This and related problems have been studied in detail by Guibas, Ramshaw, and Sedgewick [19] and by Flajolet, Gourdon, and Dumas [10] (see also [12] and [37]). Knuth's analysis uses complex asymptotics and introduced to the analysis of algorithms a method he called the "Gamma-function method." It is a special case of the *Mellin transform*, which has emerged as one of the most important tools in analytic combinatorics and the analysis of algorithms. For example, this method is effective in explaining the periodic terms in the analysis of mergesort (see §2.6) and the similar terms that arise in the analysis of many other algorithms. We conclude this section with a brief discussion of some of the algorithms that we have already discussed.

Trie search. Theorem 8.8 directly implies that the average number of bits inspected during a search in a trie built from random bitstrings is $\sim \lg N$, which is optimal. Knuth also shows that this is true for both Patricia and digital tree searching (a variant where keys are also stored in internal nodes). It is possible to get accurate asymptotics and explicit expressions for the constants involved and for the oscillating term; see the references mentioned above for full details.

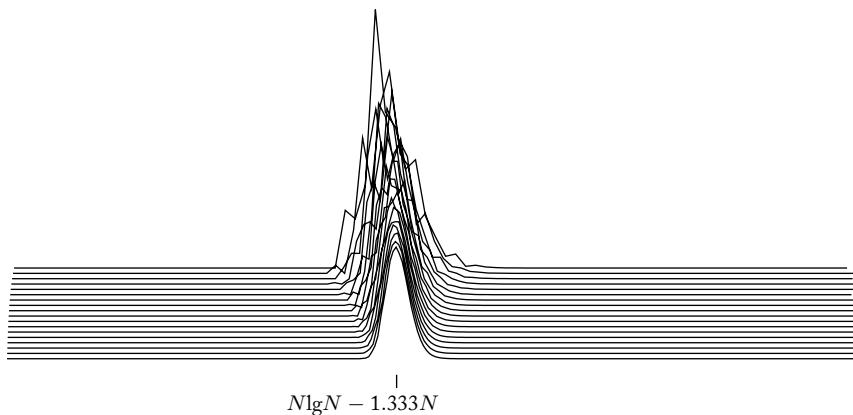


Figure 8.11 Distributions for path length in tries, even N from 10 to 50 (scaled and translated to center and separate curves)

Radix-exchange sort. As noted in the proof, Theorem 8.8 also directly implies that radix-exchange sort uses $\sim N \lg N$ bit comparisons to sort an array of N random bitstrings, since the average number of bit inspections for the sort is the same as the average path length of the trie. Again, our assumption is that the bitstrings are sufficiently long (say, much more than $\lg N$ bits) that we can consider them to be “infinite.” Figure 8.11 shows the distribution of costs, for comparison with Figure 1.3.

Exercise 8.52 Show that A_N/N is equal to $1/\ln 2$ plus a fluctuating term.

Exercise 8.53 Write a program to compute A_N to within 10^{-9} for $N < 10^6$ and explore the oscillatory nature of A_N/N .

Exercise 8.54 Multiply both sides of the functional equation for $C(z)$ by e^{-z} to transform it into a simpler equation on $\hat{C}(z) \equiv e^{-z}C(z)$. Use this equation to find $\hat{C}_N = [z^N]\hat{C}(z)$. Then apply the convolution implied by $C(z) = e^z\hat{C}(z)$ to show that

$$C_N = \sum_{2 \leq k \leq N} \binom{N}{k} \frac{k(-1)^k}{1 - 1/2^{k-1}}.$$

Exercise 8.55 Show directly that the sum given in the previous exercise is equivalent to the expression for C_N given in the proof of Theorem 8.8.

Distributed leader election. As an example of the analysis of an algorithm where we do not build an explicit trie, where the model of infinite random strings is certainly valid, but where analysis of properties of tries is still of practical interest, we turn to the distributed leader election problem that we introduced at the end of the previous section.

Theorem 8.9 (Leader election). The average number of rounds used by the randomized algorithm to seek a leader out of N contenders is $\lg N + O(1)$, with probability of success asymptotic to $1/(2\ln 2) \pm 10^{-5} \approx .72135$.

Proof. By representing an execution of the algorithm as a trie, it is clear that the average number of rounds is the expected length of the rightgoing branch in a trie built from N random bitstrings, and satisfies the recurrence

$$R_N = 1 + \frac{1}{2^N} \sum_k \binom{N}{k} R_k \quad \text{for } N > 1 \text{ with } R_0 = R_1 = 0.$$

Similarly, the probability of success satisfies the recurrence

$$p_N = \frac{1}{2^N} \sum_k \binom{N}{k} p_k \quad \text{for } N > 1 \text{ with } p_0 = 0 \text{ and } p_1 = 1.$$

These recurrences are quite similar to the path length recurrence that we considered in detail earlier. Solving them is left for exercises. As usual, the stated result for p_N is accurate to within an oscillating term of mean 0 and amplitude less than 10^{-5} . ■

If the method fails, it can be executed repeatedly to yield an algorithm that succeeds with probability 1. On average, this will require about $2\ln 2 \approx 1.3863$ iterations, for a grand total of $\sim 2\ln N$ rounds. This algorithm is a simplified version of an algorithm of Prodinger that succeeds in an average of $\lg N + O(1)$ rounds [30].

Exercise 8.56 Solve the recurrence for R_N given in the proof of Theorem 8.9, to within the oscillating term.

Exercise 8.57 Solve the recurrence for p_N given in the proof of Theorem 8.9, to within the oscillating term.

Exercise 8.58 Analyze the version of the leader election algorithm that repeats rounds until a success occurs.

8.9 Larger Alphabets. The results of this chapter generalize to strings comprising characters from alphabets of size M with $M > 2$. None of the combinatorial techniques used depends in an essential way on the strings being binary. In practice, it is reasonable to assume that the alphabet size M is a constant that is not large: values such as 26, 2⁸, or even 2¹⁶ are what might be expected in practice. We will not examine results for larger alphabets in detail, but we will conclude with a few general comments.

In §8.2 we noted that the first run of P 0s in a random M -bytestring ends at position $M(M^P - 1)/(M - 1)$. This makes it seem that the alphabet size can be a very significant factor: for example, we expect to find a string of ten 0s within the first few thousand bits of a random bitstring, but a string of ten identical characters would be extremely unlikely in a random string composed of eight-bit bytes, since the expected position of the first such string is about at the 2⁸⁰th byte. But this is just looking at the results of the analysis in two ways, since we can transform a bitstring into a string of bytes in the obvious way (consider eight bits at a time). That is, the alphabet size is less of a factor than it seems, because it intervenes only through its logarithm.

From both practical and theoretical standpoints, it generally suffices to consider bitstrings instead of bytestrings, since the straightforward encoding can transform any effective algorithm on bytestrings into an effective algorithm on bitstrings, and vice versa.

The primary advantage of considering bits as groups is that they can be used as indices into tables or treated as integers, in essence using the computer addressing or arithmetic hardware to match a number of bits in parallel. This can lead to substantial speedups for some applications.

Multiway tries. If M is not large (say, keys are decimal numbers), multiway tries are the method of choice for symbol table applications. An M -ary trie built from a random set of M -bytestrings with M nodes has external path length asymptotic to $N \lg_M N$, on the average. That is, using an alphabet of size M can save a factor of $\lg N / \lg M$ in search time. But this comes at a cost of about M times more storage, since each node must have M links, most of which will be null if M is large. Various modifications to the basic algorithm have been developed to address the excess space usage, and Knuth's analysis extends to help evaluate the schemes (see [23]).

Ternary tries. Bentley and Sedgewick [2] developed a way to represent tries with *ternary trees*, which essentially amounts to replacing each trie node with

a BST symbol table where keys are characters and links are values. This approach keeps the space cost low ($3N$ links) while still limiting the number of characters examined for a search miss. Clément, Flajolet, and Vallée [4][5] showed that $\sim \ln N$ *characters* are examined for a search miss in a ternary tree as opposed to $\sim 2\ln N$ *string compares* in a binary search tree. This analysis used a “dynamical systems” model that extends to give a true comparison of the costs of searching and sorting with string keys [6]. TSTs are the method of choice for string keys as the implementations are compact, easy to understand, and efficient (see Sedgewick and Wayne [33] for details).

Right-left string searching. Consider the problem of searching for a (relatively short) pattern in a (relatively long) text when both are bytestrings. To begin the search, we test the M th character in the text with the last character in the pattern. If it matches, we then test the $(M - 1)$ st text character against the next-to-last pattern character, and so on until a mismatch is found or a full match is verified. One approach to this process is to develop an “optimal shift” based on the mismatch, in much the same way as for the KMP algorithm. But a simpler idea is probably more effective in this case. Suppose that the text character does not appear in the pattern. Then this one character is evidence that we can skip M more characters, because any positioning of the right end of the pattern over any of those characters would necessitate matching the “current” character, which does not appear in the pattern. For many reasonable values of the relevant parameters, this event is likely, so this takes us close to the goal of examining N/M text characters when searching for a pattern of length M in a text string of length N . Even when the text character does appear in the pattern, we can precompute the appropriate shift for each character in the alphabet (distance from the right end of the pattern) to line up the pattern with its “next possible” match in the text. Details of this method, which originates from ideas of Boyer and Moore, are discussed by Gonnet and Baeza-Yates [15].

LZW data compression. Multiway tries or ternary trees are the method of choice for implementing one of the most widely used data compression algorithms: the Lempel-Ziv-Welch (LZW) method [39][38]. This algorithm requires a data structure that can support the “longest prefix match” and “insert a new key formed by appending a character to an existing key” operations, which are perfectly suited to a trie implementation. See [33] for details. This is one of a family of algorithms that dates to the 1978 seminal paper by Ziv

and Lempel. Despite their importance, the analysis of the effectiveness of these algorithms was open for nearly two decades until it was resolved by Jacquet, Szpankowski, and Louchard in the 1990s. This work is founded on basic research on the properties of tries that applies to many other problems. Details may be found in Szpankowski's book [35].

Exercise 8.59 How many bits are examined, on the average, by an algorithm that searches for runs of M 0s in a text string (M not small) by, for $k = 1, 2, 3$, and higher, checking the t bits ending at kM and, if they are all 0, checking the bits on each side to determine the length of the run? Assume that the text string is random except for a run of M 0s hidden somewhere within it.

Exercise 8.60 Give a way to adapt the method of the previous exercise to find the longest string of 0s in a random bitstring.

BITSTRINGS are arguably *the* fundamental combinatorial object for computer science, and we have touched on a number of basic paradigms in considering the combinatorics of strings and tries. Some of the results that we have examined relate to classical results from probability theory, but many others relate both to fundamental concepts of computer science and to important practical problems.

In this chapter, we discussed a number of basic string-processing algorithms. Such algorithms are quite interesting from the standpoint of algorithm design and analysis, and they are the focus of extensive research because of the importance of modern applications that range from processing genetic sequencing data to developing search engines for the Internet. Indeed, such applications are so important (and the topic so interesting) that the study of strings has emerged on its own as an important area of research, with several books devoted to the topic [20][27][28][35].

Considering sets of strings leads to the study of *formal languages*, and the useful result that there is a specific connection between fundamental properties of formal language systems and analytic properties of the generating functions. Not the least of the implications of this is that it is possible to analyze a wide class of problems in an "automatic" fashion, particularly with the aid of modern computer algebra systems.

The *trie* is a combinatorial structure and a data structure that is of importance in all of these contexts. It is a practical data structure for classical search problems and it models a basic sort procedure; it is a natural representation for fixed sets of strings and applicable to string-search problems; as

a type of tree, it directly relates to the basic divide-and-conquer paradigm; and association with bits relates it directly to low-level representations and probabilistic processes.

Finally, analysis of the basic properties of tries presents a significant challenge that cannot be fully met with elementary methods. Detailed analysis requires describing oscillatory functions. As we have seen, this kind of function arises in a variety of algorithms, because many algorithms process binary strings either explicitly (as in this chapter) or implicitly (as in algorithms based on divide-and-conquer recursion). Advanced mathematical techniques to deal with the kinds of functions that arise in such problems are a prime focus of modern research in analytic combinatorics (see [10][12][35]).

References

1. A. V. AHO AND M. J. CORASICK. "Efficient string matching: An aid to bibliographic search," *Communications of the ACM* **18**, 1975, 333–340.
2. J. BENTLEY AND R. SEDGEWICK. "Fast algorithms for sorting and searching strings," *8th Symposium on Discrete Algorithms*, 1997, 360–369.
3. N. CHOMSKY AND M. P. SCHÜTZENBERGER. "The algebraic theory of context-free languages," in *Computer Programming and Formal Languages*, P. Braffort and D. Hirschberg, eds., North Holland, 1963, 118–161.
4. J. CLÉMENT, J. A. FILL, P. FLAJOLET, AND B. VALLÉE. "The number of symbol comparisons in quicksort and quickselect," *36th International Colloquium on Automata, Languages, and Programming*, 2009, 750–763.
5. J. CLÉMENT, P. FLAJOLET, AND B. VALLÉE. "Analysis of hybrid tree structures," *9th Symposium on Discrete Algorithms*, 1998, 531–539.
6. J. CLÉMENT, P. FLAJOLET, AND B. VALLÉE. "Dynamical sources in information theory: A general analysis of trie structures," *Algorithmica* **29**, 2001, 307–369.
7. L. COMTET. *Advanced Combinatorics*, Reidel, Dordrecht, 1974.
8. S. EILENBERG. *Automata, Languages, and Machines*, Volume A, Academic Press, New York, 1974.
9. W. FELLER. *An Introduction to Probability Theory and Its Applications*, John Wiley, New York, 1957.
10. P. FLAJOLET, X. GOURDON, AND P. DUMAS. "Mellin transforms and asymptotics: Harmonic sums," *Theoretical Computer Science* **144**, 1995, 3–58.
11. P. FLAJOLET, M. REGNIER, AND D. SOTTEAU. "Algebraic methods for trie statistics," *Annals of Discrete Math.* **25**, 1985, 145–188.
12. P. FLAJOLET AND R. SEDGEWICK. *Analytic Combinatorics*, Cambridge University Press, 2009.
13. P. FLAJOLET AND R. SEDGEWICK. "Digital search trees revisited," *SIAM Journal on Computing* **15**, 1986, 748–767.
14. K. O. GEDDES, S. R. CZAPOR, AND G. LABAHN. *Algorithms for Computer Algebra*, Kluwer Academic Publishers, Boston, 1992.

15. G. H. GONNET AND R. BAEZA-YATES. *Handbook of Algorithms and Data Structures in Pascal and C*, 2nd edition, Addison-Wesley, Reading, MA, 1991.
16. I. GOULDEN AND D. JACKSON. *Combinatorial Enumeration*, John Wiley, New York, 1983.
17. L. GUIBAS AND A. ODLYZKO. “Periods in strings,” *Journal of Combinatorial Theory, Series A* **30**, 1981.
18. L. GUIBAS AND A. ODLYZKO. “String overlaps, pattern matching, and nontransitive games,” *Journal of Combinatorial Theory, Series A* **30**, 1981, 19–42.
19. L. GUIBAS, L. RAMSHAW, AND R. SEDGEWICK. Unpublished work, 1979.
20. D. GUSFIELD. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.
21. P. JACQUET AND W. SZPANKOWSKI. “Analytic approach to pattern matching,” in [28].
22. P. JACQUET AND W. SZPANKOWSKI. “Autocorrelation on words and its applications,” *Journal of Combinatorial Theory, Series A* **66**, 1994, 237–269.
23. D. E. KNUTH. *The Art of Computer Programming. Volume 3: Sorting and Searching*, 1st edition, Addison-Wesley, Reading, MA, 1973. Second edition, 1998.
24. D. E. KNUTH. “The average time for carry propagation,” *Indagationes Mathematicae* **40**, 1978, 238–242.
25. D. E. KNUTH, J. H. MORRIS, AND V. R. PRATT. “Fast pattern matching in strings,” *SIAM Journal on Computing*, 1977, 323–350.
26. M. LOTHAIRE. *Applied Combinatorics on Words*, Encyclopedia of Mathematics and its Applications **105**, Cambridge University Press, 2005.
27. M. LOTHAIRE. *Combinatorics on Words*, Addison-Wesley, Reading, MA, 1983.
28. G. LOUCHARD AND W. SZPANKOWSKI. “On the average redundancy rate of the Lempel-Ziv code,” *IEEE Transactions on Information Theory* **43**, 1997, 2–8.
29. V. PRATT. “Counting permutations with double-ended queues, parallel stacks and parallel queues,” in *Proceedings 5th Annual ACM Symposium on Theory of Computing*, 1973, 268–277.

30. H. PRODINGER. "How to select a loser," *Discrete Mathematics* **120**, 1993, 149–159.
31. A. SALOMAA AND M. SOITTO LA. *Automata-Theoretic Aspects of Formal Power Series*, Springer-Verlag, Berlin, 1978.
32. R. SEDGEWICK. *Algorithms (3rd edition) in Java: Parts 1-4: Fundamentals, Data Structures, Sorting, and Searching*, Addison-Wesley, Boston, 2003.
33. R. SEDGEWICK AND K. WAYNE. *Algorithms*, 4th edition, Addison-Wesley, Boston, 2011.
34. N. SLOANE AND S. PLLOUFFE. *The Encyclopedia of Integer Sequences*, Academic Press, San Diego, 1995. Also accessible as *On-Line Encyclopedia of Integer Sequences*, <http://oeis.org>.
35. W. SZPANKOWSKI. *Average-Case Analysis of Algorithms on Sequences*, John Wiley and Sons, New York, 2001.
36. L. TRABB-PARDO. Ph.D. thesis, Stanford University, 1977.
37. 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.
38. T. A. WELCH, "A Technique for high-performance data compression," *IEEE Computer*, June 1984, 8–19.
39. J. ZIV AND A. LEMPEL, "Compression of individual sequences via variable-rate encoding," *IEEE Transactions on Information Theory*, September 1978.

This page intentionally left blank

CHAPTER NINE

WORDS AND MAPPINGS

STRINGS of characters from a fixed alphabet, or *words*, are of interest in a broad variety of applications beyond the types of algorithms considered in the previous chapter. In this chapter, we consider the same family of combinatorial objects studied in Chapter 8 (where we called them “bytestrings”), but from a different point of view.

A word may be viewed as a function taking an integer i in the interval 1 to N (the character position) into another integer j in the interval 1 to M (the character value, from an M -character alphabet). In the previous chapter, we primarily considered “local” properties, involving relationships among values associated with successive indices (correlations and so forth); in this chapter, we consider “global” properties, including the frequency of occurrence of values in the range and more complex structural properties. In Chapter 8, we generally considered the alphabet size M to be a small constant and the string size N to be large (even infinite); in this chapter, we consider various other possibilities for the relative values of these parameters.

As basic combinatorial objects, words arise often in the analysis of algorithms. Of particular interest are *hashing* algorithms, a fundamental and widely used family of algorithms for information retrieval. We analyze a number of variations of hashing, using the basic generating function counting techniques of Chapter 3 and asymptotic results from Chapter 4. Hashing algorithms are very heavily used in practice, and they have a long history in which the analysis of algorithms has played a central role. Conversely, the ability to accurately predict performance of this important class of practical methods has been an impetus to the development of techniques for the analysis of algorithms. Indeed, Knuth mentions that the analysis of a hashing algorithm had a “strong influence” on the structure of his pioneering series of books [23][24][25][27].

The elementary combinatorial properties of words have been heavily studied, primarily because they model sequences of independent Bernoulli trials. The analysis involves properties of the binomial distribution, many of which we have already examined in detail in Chapters 3 and 4. Some of the

problems we consider are called *occupancy problems*, because they can be cast in a model where N balls are randomly distributed into M urns, and the “occupancy distribution” of the balls in the urns is to be studied. Many of these classical problems are elementary, though, as usual, simple algorithms can lead to variations that are quite difficult to analyze.

If the alphabet size M is small, the normal approximation to the binomial distribution is appropriate for studying occupancy problems; if M grows with N , then we use the Poisson approximation. Both situations arise in the analysis of hashing algorithms and in other applications in the analysis of algorithms.

Finally, we introduce the concept of a *mapping*: a function from a finite domain into itself. This is another fundamental combinatorial object that arises often in the analysis of algorithms. Mappings are related to words in that they might be viewed as N -letter words in an N -letter alphabet, but it turns out that they are also related to trees, forests, and permutations; analysis of their properties unveils a rich combinatorial structure that generalizes several of those that we have studied. The symbolic method is particularly effective at helping us study the basic properties of mappings. We conclude with an application of the properties of random mappings in an algorithm for factoring integers.

9.1 Hashing with Separate Chaining. Program 9.1 shows a standard method for information retrieval that cuts the search time through a table of N keys by a factor of M . By making M sufficiently large, it is often possible to make this basic algorithm outperform the symbol-table implementations based on arrays, trees, and tries that we examined in §2.6, §6.6, and §8.7.

We transform each key into an integer between 1 and M using a so-called *hash* function, assuming that, for any key, each value is equally likely to be produced by the hash function. When two keys hash to the same value (we refer to this as a *collision*), we keep them in a separate symbol table. To find out if a given key is in the table, we use the hash function to identify the secondary table containing the keys with the same hash value and do a secondary search there. This is simple to arrange in modern programming languages that support data abstraction: we maintain an array `st[]` of *symbol tables* where `st[i]` holds all the keys that hash to the value `i`. A typical choice for the secondary symbol table (which we expect to be small) is to use a linked

list, where a new element is added in constant time, but all the elements on the list need to be examined for a search (see [33]).

The performance of a hashing algorithm depends on the effectiveness of the hash function, which converts arbitrary keys into values in the range 1 to M with equal likelihood. A typical way to do this is to use a prime M , then convert keys to large numbers in some natural way and use that number modulo M for the hash value. More sophisticated schemes have also been devised. Hashing algorithms are used widely for a broad variety of applications, and experience has shown that, if some attention is paid to the process, it is not difficult to ensure that hash functions transform keys into hash values that appear to be random. Analysis of algorithms is thus particularly effective for predicting performance because a straightforward randomness assumption is justified, being built into the algorithm.

The performance of a hashing algorithm also depends on the data structures that are used for the secondary search. For example, we might consider keeping the elements in the secondary table in increasing order in an array. Or, we might consider using a binary search tree for keys that hash to the same value. Analysis of algorithms can help us choose among these variations, bearing in mind it is common practice to choose M sufficiently large that the secondary tables are so small that further improvements are marginal.

If N is large by comparison to M and the hash function produces random values, we expect that each list will have about N/M elements, cutting the search time by a factor of M . Though we cannot make M arbitrarily large because increasing M implies the use of extra space to maintain the secondary data structure, we perhaps could make N a constant multiple of M , which gives constant search time. Variations of hashing that can achieve this performance goal are in widespread use.

```
public void insert(int key)
{ return st[hash(key)].insert(); }
public int search(int key)
{ return st[hash(key)].search(key); }
```

Program 9.1 Hashing with separate chaining

We are interested in analyzing hashing algorithms not only to determine how they compare to one another, but also to determine how they compare to other symbol-table implementations and how best to set parameters (such as hash table size). To focus on the mathematics, we adopt the normal convention of measuring the performance of hashing algorithms by counting the number of *probes* used, or key comparisons made with elements in the data structure, for successful and unsuccessful searches. As usual, more detailed analysis involving the cost of computing the hash function, the cost of accessing and comparing keys, and so forth is necessary to make definitive statements about the relative performance of algorithms. For instance, even when hashing involves only a constant number of probes, it might be slower than a search based on tries (for example) if keys are long, because computing the hash function involves examining the whole key, while a trie-based method might distinguish among keys after examining relatively few bits.

A sequence of N hash values into a table of size M is nothing more than a word of length N comprising letters from an alphabet of size M . Thus, the analysis of hashing corresponds directly to the analysis of the combinatorial properties of words. We consider this analysis next, including the analysis of hashing with separate chaining, then we look at other hashing algorithms.

9.2 The Balls-and-Urns Model and Properties of Words. To set the stage for the rest of the chapter, we begin by defining words, considering the basic combinatorics surrounding them, and placing them into context both among the other combinatorial objects that we have been studying and among classical results in combinatorics.

Definition An M -word of length N is a function f mapping integers from the interval $[1 \dots N]$ into the interval $[1 \dots M]$.

As with permutations and trees, one way to specify a word is to write down its functional table:

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
word	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3	2	3	8	4

We drop the index and specify the function by writing $f(1)f(2)\dots f(N)$, which makes it plain that an M -word of length N is equivalent to a bytestring of length N (bytesize M). When studying words, we normally concentrate on properties of the *set* of values in the word. How many 1s are there? How

many of the possible values do not appear? Which value appears most often? These are the sorts of questions we address in this chapter.

In discrete probability, these combinatorial objects are often studied in the context of a *balls-and-urns* model. We imagine N balls to be randomly distributed among M urns, and we ask questions about the result of the distribution. This directly corresponds to words: N balls correspond to the N letters in a word, and M urns correspond to the M different letters in an alphabet, so we specify which urn each ball lands in. That is, a word of length N is a sequence of M urns containing N balls labelled from 1 to N , where the first urn gives the indices in the word having the first letter, the second urn gives the indices in the word having the second letter, and so forth. Figure 9.1 illustrates the balls-and-urns model for our example.

Combinatorially, the balls-and-urns model is equivalent to viewing an M -word as an M -sequence of sets of labelled objects. This view immediately leads to the combinatorial construction

$$\mathcal{W}_M = \text{SEQ}_M(\text{SET}(\mathcal{Z}))$$

which immediately transfers via Theorem 5.2 to the EGF equation

$$W_M(z) = (e^z)^M$$

and therefore $W_{MN} = N![z^N]e^{zM} = M^N$ as expected. This result is simple, but, as usual with the symbolic method, we will be making slight modifications to this construction to develop easy solutions to problems that otherwise would be much more difficult to solve.

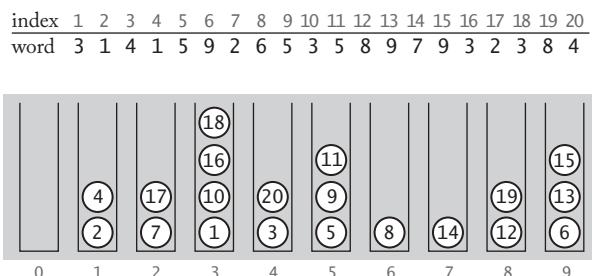


Figure 9.1 A 10-word of length 20
(20 balls thrown into 10 urns)

As detailed in Table 9.1, this view represents a completely different view of the same set of objects from the view taken in Chapter 8. Are they N -sequences drawn from M unlabelled objects or M urns filled with N labelled objects? The answer to this question is that *both* are valid views, each leading to useful information with all sorts of applications. To be consistent, we try in this book to consistently use the term “bytestring” when taking the view detailed in Chapter 8 and the terms “word,” “balls-and-urns,” or even “hash table” when taking the viewpoint just outlined. Figure 9.2 illustrates, in our standard style, all of the 2-words and all of the 3-words for some small values of N .

To add to the confusion (or, thinking positively, to reinforce the concept), we normally drop the labels when working with balls and urns, with the implicit understanding that we are analyzing algorithms that map random words to different unlabeled urn occupancy distributions with different probabilities. For example, from Figure 9.2 you can immediately see that when we throw three balls at random into two urns, there are four possibilities:

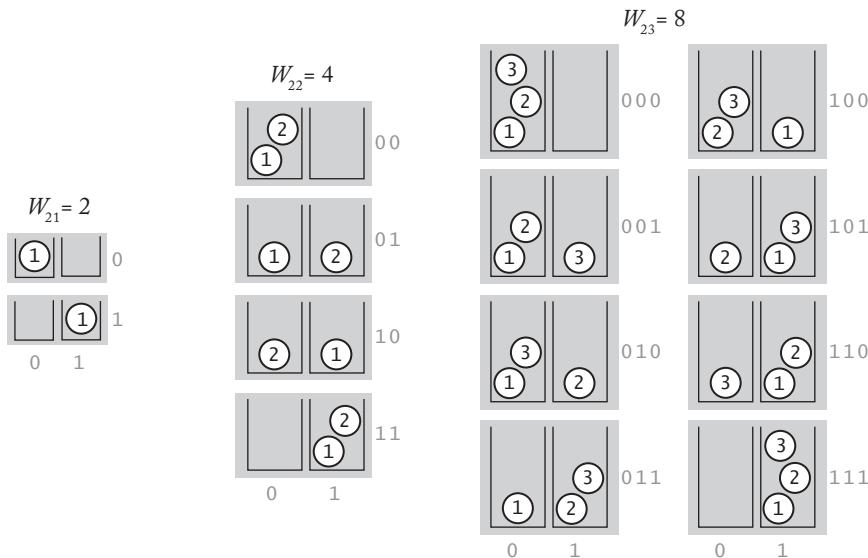
- 3 balls in the left urn (which occurs with probability 1/8),
- 2 balls in the left urn and 1 ball in the right urn (probability 3/8),
- 1 ball in the left urn and 2 balls in the right urn (probability 3/8),
- 3 balls in the right urn (probability 1/8).

This situation arises often in the analysis of algorithms, and we have encountered it on several occasions before. For example, BSTs map permutations to

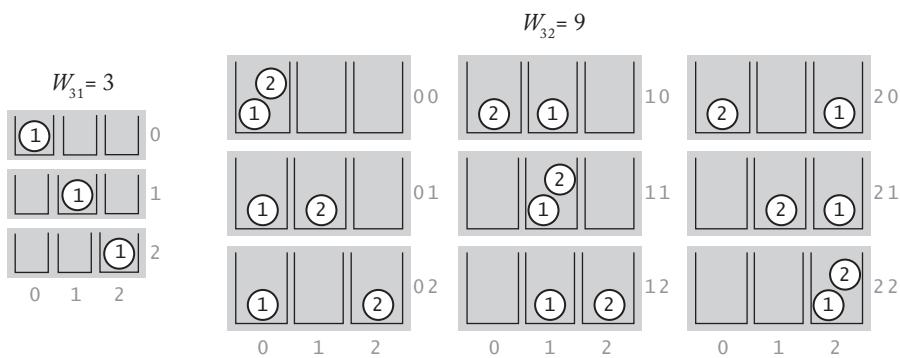
	construction	symbolic transfer	GF	analytic transfer	coefficient asymptotics
Unlabelled class					
bytestrings	$SEQ(\mathcal{Z}_0 + \dots + \mathcal{Z}_{M-1})$	5.1	$\frac{1}{1 - Mz}$	Taylor	M^N
Labelled class					
words	$SEQ_M(SET(\mathcal{Z}))$	5.2	e^{Mz}	Taylor	M^N

Table 9.1 Two ways to view the same set of objects

2-words on 1, 2, and 3 characters



3-words on 1 and 2 characters

**Figure 9.2** 2-words and 3-words with small numbers of characters

binary trees and binary tries map bitstrings to tries. In all of these cases, we have three interesting combinatorial problems to consider: the source model, the target model, and the target model with probabilities induced by the mapping. For permutations and trees, we considered all three (random permutations, random binary Catalan trees, and BSTs built from random permutations). In this chapter, the first is straightforward and we concentrate on the third, leaving the second for an exercise, as we did for tries in Chapter 8.

Exercise 9.1 How many different occupancy distributions are there for N unlabelled balls in a sequence of M urns? For example, if we denote the desired quantity by C_{MN} , we have $C_{2N} = N + 1$ because there is one configuration with k balls in the first urn and $N - k$ balls in the second urn for each k from 0 to N .

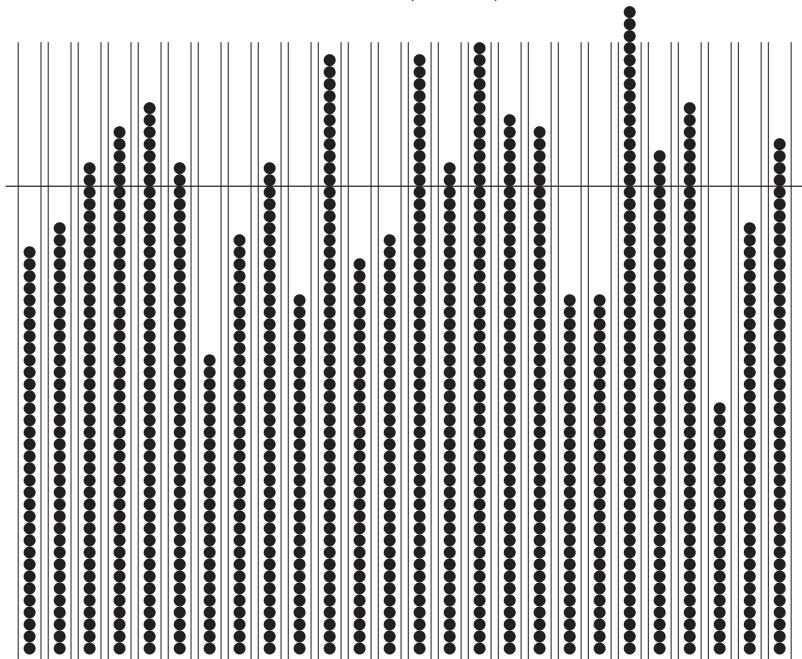
The relative growth rates of M and N are of central importance in the analysis, and a number of different situations arise in practice. For strings of text or other kinds of “words,” we normally think of M as being fixed and N as being the variable of interest. We make words of varying length from letters in a fixed alphabet, or we throw varying numbers of balls at a fixed number of urns. For other applications, particularly hashing algorithms, we think of M growing with N , typically $M = \alpha N$ with α between 0 and 1. The characteristics of these two different situations are illustrated in Figure 9.3. At the top is an example with a relatively large number of balls thrown into a relatively small number of urns; in this case we are interested in knowing that the urns have roughly the same number of balls and in quantifying the discrepancy. At the bottom are five trials with $\alpha = 1$: many of the urns tend to be empty, some have just one ball, and very few have several balls.

In both cases, as we will see, our analysis leads to a precise characterization of the occupancy distributions illustrated in Figure 9.3. As the reader may have surmised by now, the distribution in question is of course the binomial distribution, and the results that we consider are classical (nearly two centuries old); we already covered them in some detail in Chapter 4.

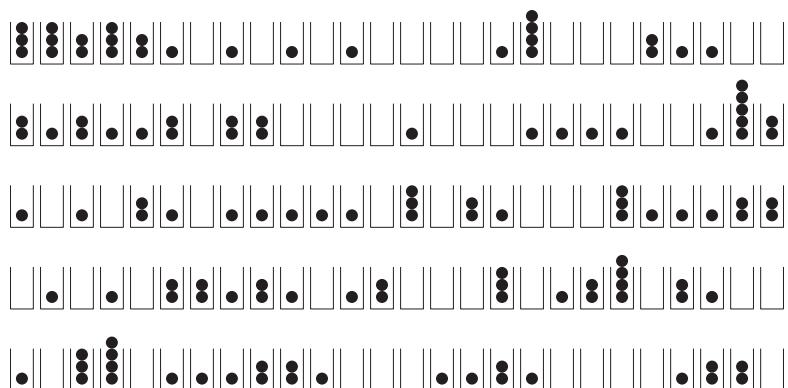
Table 9.2 gives all the 3-words of length 4 (or all the ways to distribute 4 balls into 3 urns), along with the occupancy counts showing the number of letters that are used 0, 1, 2, 3, and 4 times (or the number of urns that contain 0, 1, 2, 3, and 4 balls). A few other statistics of the type we are considering are also included.

Generally speaking, when N balls are randomly distributed among M urns, we are interested in the following sorts of questions:

Random distribution of 1024 balls into 26 urns (one trial)



Random distribution of 26 balls into 26 urns (five trials)

**Figure 9.3** Examples of balls-and-urns experiments

word	0	1	2	3	4	word	0	1	2	3	4	word	0	1	2	3	4
1111	2	0	0	0	1	2111	1	1	0	1	0	3111	1	1	0	1	0
1112	1	1	0	1	0	2112	1	0	2	0	0	3112	0	2	1	0	0
1113	1	1	0	1	0	2113	0	2	1	0	0	3113	1	0	2	0	0
1121	1	1	0	1	0	2121	1	0	2	0	0	3121	0	2	1	0	0
1122	1	0	2	0	0	2122	1	1	0	1	0	3122	0	2	1	0	0
1123	0	2	1	0	0	2123	0	2	1	0	0	3123	0	2	1	0	0
1131	1	1	0	1	0	2131	0	2	1	0	0	3131	1	0	2	0	0
1132	0	2	1	0	0	2132	0	2	1	0	0	3132	0	2	1	0	0
1133	1	0	2	0	0	2133	0	2	1	0	0	3133	1	1	0	1	0
1211	1	1	0	1	0	2211	1	0	2	0	0	3211	0	2	1	0	0
1212	1	0	2	0	0	2212	1	1	0	1	0	3212	0	2	1	0	0
1213	0	2	1	0	0	2213	0	2	1	0	0	3213	0	2	1	0	0
1221	1	0	2	0	0	2221	1	1	0	1	0	3221	0	2	1	0	0
1222	1	1	0	1	0	2222	2	0	0	0	1	3222	1	1	0	1	0
1223	0	2	1	0	0	2223	1	1	0	1	0	3223	1	0	2	0	0
1231	0	2	1	0	0	2231	0	2	1	0	0	3231	0	2	1	0	0
1232	0	2	1	0	0	2232	1	1	0	1	0	3232	1	0	2	0	0
1233	0	2	1	0	0	2233	1	0	2	0	0	3233	1	1	0	1	0
1311	1	1	0	1	0	2311	0	2	1	0	0	3311	1	0	2	0	0
1312	0	2	1	0	0	2312	0	2	1	0	0	3312	0	2	1	0	0
1313	1	0	2	0	0	2313	0	2	1	0	0	3313	1	1	0	1	0
1321	0	2	1	0	0	2321	0	2	1	0	0	3321	0	2	1	0	0
1322	0	2	1	0	0	2322	1	1	0	1	0	3322	1	0	2	0	0
1323	0	2	1	0	0	2323	1	0	2	0	0	3323	1	1	0	1	0
1331	1	0	2	0	0	2331	0	2	1	0	0	3331	1	1	0	1	0
1332	0	2	1	0	0	2332	1	0	2	0	0	3332	1	1	0	1	0
1333	1	1	0	1	0	2333	1	1	0	1	0	3333	2	0	0	0	1
												total	48	96	72	24	3

Occupancy distributions

$$3 \times 20001 + 24 \times 11010 + 18 \times 10200 + 36 \times 02100$$

$\Pr\{\text{no urn empty}\}$	$36/81 \approx 0.444$
$\Pr\{\text{urn occupancies all } < 3\}$	$(18 + 36)/81 \approx 0.667$
$\Pr\{\text{urn occupancies all } < 4\}$	$(24 + 18 + 36)/81 \approx 0.963$
Avg. number empty urns	$(1 \cdot 42 + 2 \cdot 3)/81 \approx 0.593$
Avg. maximum occupancy	$(2 \cdot 54 + 3 \cdot 24 + 4 \cdot 3)/81 \approx 2.370$
Avg. occupancy	$(1 \cdot 96 + 2 \cdot 72 + 3 \cdot 24 + 4 \cdot 3)/3 \cdot 81 \approx 1.333$
Avg. minimum occupancy	$(1 \cdot 36)/81 \approx 0.444$

Table 9.2 Occupancy distribution and properties of 3-words of length 4 or configurations of 4 balls in 3 urns or hash sequences of length 4 (table size 3)

- What is the probability that no urn will be empty?
- What is the probability that no urn will contain more than one ball?
- How many of the urns are empty?
- How many balls are in the urn containing the most balls?
- How many balls are in the urn containing the fewest balls?

These are immediately relevant to practical implementations of hashing and other algorithms: we want to know how long we may expect the lists to be when using hashing with separate chaining, how many empty lists we might expect, and so on. Some of the questions are enumeration problems akin to our enumeration of permutations with cycle length restrictions in Chapter 7; others require analysis of properties of words in more detail. These and related questions depend on the *occupancy distribution* of the balls in the urns, which we study in detail in this chapter.

Table 9.3 gives the values of some of these quantities for three urns with the number of balls ranging from 1 to 10, calculated using the results given in the next section. The fourth column corresponds to Table 9.2, which illustrates how these values are calculated. As the number of balls grows, we have a situation similar to that depicted at the top in Figure 9.3, with balls distributed about equally into urns, roughly N/M balls per urn. Other phenomena that we expect intuitively are exhibited in this table. For example, as

	<i>balls</i> →	1	2	3	4	5	6	7	8	9	10
Probability											
urn occ. all < 2	1	.667	.222	0	0	0	0	0	0	0	0
urn occ. all < 3	1	1	.889	.667	.370	.123	0	0	0	0	0
urn occ. all < 4	1	1	1	.963	.864	.700	.480	.256	.085	0	0
urn occ. all > 0	0	0	.222	.444	.617	.741	.826	.883	.922	.948	
urn occ. all > 1	0	0	0	0	0	.123	.288	.448	.585	.693	
urn occ. all > 2	0	0	0	0	0	0	0	0	.085	.213	
Average											
# empty urns	2	1.33	.889	.593	.395	.263	.176	.117	.0780	.0520	
max. occupancy	1	1.33	1.89	2.37	2.78	3.23	3.68	4.08	4.50	4.93	
min. occupancy	0	0	.222	.444	.617	.864	1.11	1.33	1.59	1.85	

Table 9.3 Occupancy parameters for balls in three urns

the number of balls increases, the number of empty urns becomes small, and the probability that no urn is empty becomes large.

The relative values of M and N dictate the extent to which the answers to the various questions posed earlier are of interest. If there are many more balls than urns ($N \gg M$), then it is clear that the number of empty urns will be very low; indeed, we expect there to be about N/M balls per urn. This is the case illustrated at the top in Figure 9.3. If there are many fewer balls than urns ($N \ll M$), most urns are empty. Some of the most interesting and important results describe the situation when N and M are within a constant factor of each other. Even when $M = N$, urn occupancy is relatively low, as illustrated at the bottom in Figure 9.3.

Table 9.4 gives the values corresponding to Table 9.3, but for the larger value $M = 8$, again with N ranging from 1 to 10. When the number of balls is small, we have a situation similar to that depicted at the bottom in Figure 9.3, with many empty urns and few balls per urn generally. Again, we are able to calculate exact values from the analytic results given §9.3.

Exercise 9.2 Give a table like Table 9.2 for three balls in four urns.

Exercise 9.3 Give tables like Tables 9.2 and 9.4 for two urns.

Exercise 9.4 Give necessary and sufficient conditions on N and M for the average number of empty urns to equal the average minimum urn occupancy.

	<i>balls</i> →	1	2	3	4	5	6	7	8	9	10
Probability											
occupancies all < 2	1	.875	.656	.410	.205	.077	.019	.002	0	0	
occupancies all < 3	1	1	.984	.943	.872	.769	.642	.501	.361	.237	
occupancies all < 4	1	1	1	.998	.991	.976	.950	.910	.855	.784	
occupancies all > 0	0	0	0	0	0	0	0	.002	.011	.028	
occupancies all > 1	0	0	0	0	0	0	0	.000	.000	.000	
Average											
# empty urns	7	6.13	5.36	4.69	4.10	3.59	3.14	2.75	2.41	2.10	
max. occupancy	1	1.13	1.36	1.65	1.93	2.18	2.39	2.60	2.81	3.02	
min. occupancy	0	0	0	0	0	0	0	.002	.011	.028	

Table 9.4 Occupancy parameters for balls in eight urns

AS WITH PERMUTATIONS, WE can develop combinatorial constructions among words for use in deriving functional relationships among CGFs that yield analytic results of interest. Choosing among the many possible correspondences for a particular application is part of the art of analysis. Our constructions for M -words of length N build on words with smaller values of N and M .

“First” or “last” construction. Given an M -word of length $N - 1$, we can construct M different M -words of length N simply by, for each k from 1 to M , prepending k . This defines the “first” construction. For example, the 4-word 3 2 4 gives the following 4-words:

$$1 \ 3 \ 2 \ 4 \quad 2 \ 2 \ 3 \ 4 \quad 3 \ 2 \ 3 \ 4 \quad 4 \ 2 \ 3 \ 4$$

One can clearly do the same thing with any other position, not just the first. This construction implies that the number of M -words of length N is M times the number of M -words of length $N - 1$, a restatement of the obvious fact that the count is M^N .

“Largest” construction. Given an M -word of length N , consider the $(M - 1)$ -word formed by simply removing all occurrences of M . If there were k such occurrences (k could range from 0 to N), this word is of length $N - k$, and corresponds to exactly $\binom{N}{k}$ different words of length N , one for every possible way to add k elements. Conversely, for example, we can build the following 3-words of length 4 from the 3-word 2 1:

$$3 \ 3 \ 2 \ 1 \quad 3 \ 2 \ 3 \ 1 \quad 3 \ 2 \ 1 \ 3 \quad 2 \ 3 \ 3 \ 1 \quad 2 \ 3 \ 1 \ 3 \quad 2 \ 1 \ 3 \ 3$$

This construction leads to the recurrence

$$M^N = \sum_{0 \leq k \leq N} \binom{N}{k} (M - 1)^{N-k},$$

a restatement of the binomial theorem.

9.3 Birthday Paradox and Coupon Collector Problem. We know that the distribution of balls in urns is the binomial distribution, and we discuss properties of that distribution in detail later in this chapter. Before doing so, however, we consider two classical problems about ball-and-urn occupancy distributions that have to do with the dynamics of the process of the urns

filling up with balls. As N balls are randomly distributed, one after another, among M urns, we are interested in knowing how many balls are thrown, on the average, before

- A ball falls into a nonempty urn for the first time; and
- No urns are empty for the first time.

These are called the *birthday problem* and the *coupon collector problem*, respectively. They are immediately relevant to the study of hashing and other algorithms. The solution to the birthday problem will tell us how many keys we should expect to insert before we find the first collision; the solution to the coupon collector problem will tell us how many keys we should expect to insert before finding that there are no empty lists.

Birthday problem. Perhaps the most famous problem in this realm is often stated as follows: How many people should one gather in a group for it to be more likely than not that two of them have the same birthday? Taking people one at a time, the probability that the second has a different birthday than the first is $(1 - 1/M)$; the probability that the third has a different birthday than the first two is (independently) $(1 - 2/M)$, and so on, so (if $M = 365$) the probability that N people have different birthdays is

$$\left(1 - \frac{1}{M}\right)\left(1 - \frac{2}{M}\right) \dots \left(1 - \frac{N-1}{M}\right) = \frac{N!}{M^N} \binom{M}{N}.$$

Subtracting this quantity from 1, we get the answer to our question, plotted in Figure 9.4.

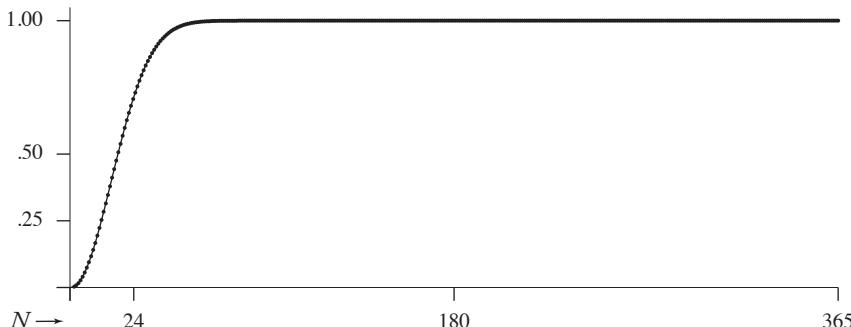


Figure 9.4 Probability that N people do not all have different birthdays

Theorem 9.1 (Birthday problem). The probability that there are no collisions when N balls are thrown into M urns is given by

$$\left(1 - \frac{1}{M}\right)\left(1 - \frac{2}{M}\right) \dots \left(1 - \frac{N-1}{M}\right).$$

The expected number of balls thrown until the first collision occurs is

$$1 + Q(M) = \sum_k \binom{M}{k} \frac{k!}{M^k} \sim \sqrt{\frac{\pi M}{2}} + \frac{2}{3}$$

where $Q(M)$ is the Ramanujan Q -function.

Proof. See the previous discussion for the probability distribution. To find the expected value, let X denote the random variable for the number of balls until the first collision occurs. Then the given probability is precisely $\Pr\{X > N\}$. Summing these, we get an expression for the expectation:

$$\sum_{N \geq 0} \left(1 - \frac{1}{M}\right)\left(1 - \frac{2}{M}\right) \dots \left(1 - \frac{N-1}{M}\right),$$

which is precisely $1 + Q(M)$ by the definition of $Q(M)$ (see §4.7). The asymptotic form follows from Theorem 4.8. ■

The value of $1 + Q(365)$ is between 24 and 25, so that it is more likely than not that at least two people in a group of 25 or larger have the same birthday. This is often referred to as the “birthday paradox” because one might expect the number to be much higher. For generations, teachers have surprised skeptical students by finding two having the same birthday, knowing that the experiment is more likely than not to succeed in a class of 25 or larger, with the chance of success much improved for larger classes.

It is also of interest to find the median value: the value of N for which the probability given above is closest to $1/2$. This could be done with a quick computer calculation, or an asymptotic calculation can also be used. Asymp-

totically, the cutoff point is determined by

$$\begin{aligned} \left(1 - \frac{1}{M}\right) \left(1 - \frac{2}{M}\right) \cdots \left(1 - \frac{N-1}{M}\right) &\sim \frac{1}{2} \\ \sum_{1 \leq k < N} \ln\left(1 - \frac{k}{M}\right) &\sim \ln(1/2) \\ \sum_{1 \leq k < N} \frac{k}{M} &\sim \ln 2 \\ \frac{N(N-1)}{2M} &\sim \ln 2 \\ N &\sim \sqrt{2M \ln 2}, \end{aligned}$$

which slightly underestimates the answer $N = 23$ for $M = 365$.

The coefficient of \sqrt{M} for the mean number of people to be gathered before a birthday collision occurs is $\sqrt{\pi/2} \approx 1.2533$, and the coefficient of \sqrt{M} for the median (the number of people to be gathered to be 50% sure that a birthday collision will occur) is $\sqrt{2\ln 2} \approx 1.1774$. These give rise to the approximate values 24.6112 and 22.4944, respectively, for $M = 365$, and it is interesting to note that the mean and median are *not* asymptotically equivalent in this case.

Exercise 9.5 For $M = 365$, how many people are needed to be 99% sure that two have the same birthday?

Exercise 9.6 Estimate the variance associated with the birthday distribution that is given in Theorem 9.1, and explain the apparent discrepancy concerning the asymptotic values of the mean and the median.

Coupon collector problem. Another famous and classical problem in this realm is often posed as follows: if each box of a product contains one of a set of M coupons, how many boxes must one buy, on the average, before getting all the coupons? This value is equivalent to the expected number of balls thrown until all urns have at least one ball or the expected number of keys added until all the chains in a hash table built by Program 9.1 have at least one key.

To solve this problem, we define a k -collection to be a word that consists of k different letters, with the last letter in the word being the only time that

letter occurs, and \mathcal{P}_{Mk} to be the combinatorial class of words that are k -collections of M possible coupons. The number of k -collections of length N divided by M^N is the probability that N coupons need to be collected to get k different ones. In the balls-and-urns model, this is the probability that the last ball falls into an empty urn, making the number of nonempty urns equal to k .

For this problem, it is convenient to view words as bytestrings and use unlabelled objects, OGFs, and PGFs. In any k -collection w , the first coupon is either in the collection of $k - 1$ coupons (not including the last one) in the rest of w , in which case the rest of w is a k -collection, or it is one of the $M - (k - 1)$ coupons that is not in the rest of w , in which case the rest of w is a $(k - 1)$ -collection. Therefore, we have the combinatorial construction

$$\mathcal{P}_{Mk} = ((k - 1)\mathcal{Z}) \times \mathcal{P}_{Mk} + ((M - (k - 1))\mathcal{Z}) \times \mathcal{P}_{M(k-1)},$$

which immediately translates, via Theorem 5.1, to the OGF equation

$$\begin{aligned} P_{Mk}(z) &= (k - 1)zP_{Mk}(z) + (M - (k - 1))zP_{M(k-1)}(z) \\ &= \frac{(M - (k - 1))z}{1 - (k - 1)z}P_{M(k-1)}(z), \end{aligned}$$

with $P_0(z) = 1$. Note that $P_{Mk}(z/M)$ is the PGF for the length of k -collections, so we have $P_{Mk}(1/M) = 1$ and the average length of a k -collection is $P'_{Mk}(z/M)|_{z=1}$. Differentiating both sides of the equation

$$P_{Mk}(z/M) = \frac{(M - (k - 1))z}{M - (k - 1)z}P_{M(k-1)}(z/M),$$

evaluating at $z = 1$, and simplifying gives the recurrence

$$\frac{d}{dz}P_{Mk}(z/M)|_{z=1} = 1 + \frac{(k - 1)}{M - (k - 1)} + \frac{d}{dz}P_{M(k-1)}(z/M)|_{z=1},$$

which telescopes to the solution

$$\frac{d}{dz}P_{Mk}(z/M)|_{z=1} = \sum_{0 \leq j < k} \frac{M}{M - j} = M(H_M - H_{M-k}).$$

Theorem 9.2 (Coupon collector problem). The number of balls thrown until all M urns are filled is on average

$$MH_M = M \ln M + M\gamma + O(1)$$

with variance

$$M^2 H_M^{(2)} - MH_M \sim M^2 \pi^2 / 6.$$

The probability that the N th ball fills the last empty urn is

$$\frac{M!}{M^N} \binom{N-1}{M-1}.$$

Proof. The mean is derived in the earlier discussion. Alternatively, telescoping the recurrence on the OGF for k -collections gives the explicit form

$$P_{Mk}(z) = \frac{M(M-1)\dots(M-k+1)}{(1-z)(1-2z)\dots(1-(k-1)z)} z^k,$$

which leads immediately to the PGF

$$P_{MM}(z/M) = \frac{M! z^M}{M(M-z)(M-2z)\dots(M-(M-1)z)}.$$

It is easily verified that $P_{MM}(1/M) = 1$ and that differentiating with respect to z and evaluating at $z = 1$ gives MH_M , as above. Differentiating a second time and applying Theorem 3.10 leads to the stated expression for the variance. The distribution follows immediately by extracting coefficients from the PGF using the identity

$$\sum_{N \geq M} \binom{N}{M} z^N = \frac{z^M}{(1-z)(1-2z)\dots(1-Mz)},$$

from Table 3.7 in §3.11. ■

Exercise 9.7 Find all the 2-collections and 3-collections in Table 9.2, then compute $P_2(z)$ and $P_3(z)$ and check the coefficients of z^4 .

Stirling numbers of the second kind. As pointed out already in §3.11, the Stirling “subset” numbers $\{N\}_M$ also represent the number of ways to partition an N -element set into M nonempty subsets. We will see a derivation of this fact later in this section. Starting with this definition leads to an alternate derivation of the coupon collector distribution, as follows. The quantity

$$M \left\{ \begin{matrix} N-1 \\ M-1 \end{matrix} \right\} (M-1)!$$

gives the number of ways the last of N balls fills the last of M urns, because the Stirling number is the number of ways for the $N-1$ balls to fall into $M-1$ different urns, the factor of $(M-1)!$ accounts for all possible orders of those urns, and the factor of M accounts for the fact that any of the urns could be the last urn to be filled. Dividing by M^N gives the same result as given in Theorem 9.2.

The classical derivation of the mean for the coupon collector problem (see, for example, Feller [9]) is elementary and proceeds as follows. Once k coupons have been collected, the probability that j or more additional boxes are needed to get the next coupon is $(k/M)^j$, so the average number of such boxes is

$$\sum_{j \geq 0} \left(\frac{k}{M} \right)^j = \frac{1}{1 - k/M} = \frac{M}{M - k},$$

and summing on k gives the result $M H_M$, as above. This requires less computation than the derivation above, but the generating functions capture the full structure of the problem, making it possible to calculate the variance without explicitly worrying about dependencies among the random variables involved, and also giving the complete probability distribution.

Exercise 9.8 Expand the PGF by partial fractions to show that the probability that the N th ball fills the last empty urn can also be expressed as the alternating sum

$$\sum_{0 \leq j < M} \binom{M}{j} (-1)^j \left(1 - \frac{j}{M}\right)^{N-1}.$$

Exercise 9.9 Give an expression for the probability that collecting *at least* N boxes gives a full collection of M coupons.

Enumeration of M -surjections. Balls-and-urns sequences with M urns, none of which are empty (M -words with at least one occurrence of each letter), are called M -surjections. These combinatorial objects arise in many contexts, since they correspond to dividing N items into exactly M distinguished nonempty groups. Determining the EGF for M -surjections is easy via the symbolic method, as it is a slight modification of our derivation for words at the beginning of §9.2. An M -surjection is an M -sequence of nonempty urns, so we have the combinatorial construction

$$\mathcal{F}_M = \text{SEQ}_M(\text{SET}_{>0}(\mathcal{Z}))$$

which immediately transfers via Theorem 5.2 to the EGF equation

$$F_M(z) = (e^z - 1)^M.$$

This is an exponential generating function for the Stirling numbers of the second kind (see Table 3.7), so we have proved that the number of M -surjections of length N is

$$N![z^N](e^z - 1)^M = M! \left\{ \begin{matrix} N \\ M \end{matrix} \right\}.$$

Again, starting with the combinatorial definition that Stirling numbers of the second kind enumerate partitions of N elements into M subsets gives a direct proof of this same result, since each of the $M!$ orderings of the sets yields a surjection.

Enumeration of surjections. We digress slightly to consider words that are made up of at least one occurrence of M characters for *some* M , which are known as *surjections*, because they are well studied in combinatorics. The analysis follows the same argument as just considered for M -surjections and is a good example of the power of analytic combinatorics. A surjection is a sequence of nonempty urns, so we have the combinatorial construction

$$\mathcal{F} = \text{SEQ}(\text{SET}_{>0}(\mathcal{Z})),$$

which immediately transfers via Theorem 5.2 to the EGF equation

$$F(z) = \frac{1}{1 - (e^z - 1)} = \frac{1}{2 - e^z}.$$

Elementary complex asymptotics (see [14]) provides the coefficient asymptotics, with the result that the number of surjections of length N is

$$N![z^N]F(z) \sim \frac{N!}{2\ln 2} \left(\frac{1}{\ln 2}\right)^N.$$

Exercise 9.10 Consider the “largest” construction among M -surjections: given an M -surjection of length N , consider the $(M - 1)$ -surjection formed by removing all occurrences of M . Find the EGF for surjections using this construction.

Exercise 9.11 Write a program that takes N and M as parameters and prints out all M -surjections of length N whenever the number of such objects is less than 1000.

Exercise 9.12 Expand $(e^z - 1)^M$ by the binomial theorem to show that

$$N![z^N]F_M(z) = \sum_j \binom{M}{j} (-1)^{M-j} j^N = M! \begin{Bmatrix} N \\ M \end{Bmatrix}.$$

(See Exercise 9.8.)

Exercise 9.13 Show that the number of partitions of N elements into nonempty subsets is

$$N![z^N]e^{e^z-1}.$$

(This defines the so-called *Bell numbers*.)

Exercise 9.14 Show that

$$N![z^N]e^{e^z-1} = \frac{1}{e} \sum_{k \geq 0} \frac{k^N}{k!}.$$

Exercise 9.15 Prove that the bivariate EGF for the Stirling numbers of the second kind is $\exp(u(e^z - 1))$.

Exercise 9.16 Applying the “largest” construction to find the number of M -words of length N might lead to the recurrence

$$F_{NM} = \sum_{0 \leq k \leq M} \binom{N}{k} F_{(N-k)(M-1)}.$$

Show how to solve this recurrence using BGFs.

Caching algorithms. Coupon collector results are classical, and they are of direct interest in the analysis of a variety of useful algorithms. For example, consider a “demand paging” system where a k -page cache is used to increase the performance of an M -page memory by keeping the k most recently referenced pages in the cache. If the “page references” are random, then the coupon collector analysis gives the number of references until the cache fills up. From the computation of $P_k(z)$ given in the proof of Theorem 9.2, it is immediately obvious that the average number of page references until a cache of size k fills up in an M -page memory system, assuming page references are independent and uniformly distributed, is $M(H_M - H_{M-k})$. Specifically, note that, although it takes $\sim M \ln M$ references before all the pages are hit, a cache of size αM fills up after about $M \ln(1/(1-\alpha))$ references. For instance, a cache of size $M/2$ will fill after about $M \ln 2 \approx .69M$ page references, resulting in a 19% savings. In practice, references are not random, but correlated and nonuniform. For example, recently referenced pages are likely to be referenced again, resulting in much higher savings from caching. The analysis thus should provide lower bounds on cache efficiency in practical situations. In addition, the analysis provides a starting point for realistic analyses under nonuniform probabilistic models [10].

THE BIRTHDAY PROBLEM AND the coupon collector problem appear at opposite ends of the process of filling cells with balls. In the birthday problem, we add balls and look at the first time some cell gets more than one ball, which takes about about $\sqrt{\pi M/2}$ steps, on the average. Continuing to add the balls, we eventually fill each cell with at least one ball, after about $M \ln M$ steps on the average, by the coupon collector result. In between, when $M = N$, we will see in the next section that about $1 - 1/e \approx 36\%$ of the cells are empty, and, furthermore, one of the cells should have about $\ln N / \ln \ln N$ balls.

These results have various practical implications for hashing algorithms. First, the birthday problem implies that collisions tend to occur early, so a collision resolution strategy must be designed. Second, the filling tends to be rather uneven, with a fair number of empty lists and a few long lists (almost logarithmic in length). Third, empty lists do not completely disappear until after a rather large number of insertions.

Many more details on the birthday problem, coupon collector problems, and applications are given in Feller’s classic text [9] and in the 1992 paper by Flajolet, Gardy, and Thimonier [10].

9.4 Occupancy Restrictions and Extremal Parameters. Solving the birthday problem involves enumerating arrangements (the number of words with no letter occurring twice) and solving the coupon collector problem involves enumerating surjections (the number of words with at least one occurrence of each letter). In this section, we describe generalizations of these two enumeration problems on words.

In Chapter 7 we discussed enumeration of permutations with restrictions on cycle length; in Chapter 8 we discussed enumeration of bitstrings with restrictions on patterns of consecutive bits. Here we use similar techniques to discuss enumeration of words with restrictions on frequency of occurrence of letters, or, equivalently, ball-and-urn occupancy distributions with restrictions, or hash sequences with collision frequency restrictions, or functions with range frequency restrictions.

Table 9.5 shows the situation for 3-words. The top four rows in Table 9.5 give the numbers of 3-words with no more than 1, 2, 3, and 4 occurrences of any letter, and the bottom four rows give the number of 3-words with at least 1, 2, 3, and 4 occurrences of every letter. (The fifth row corresponds to 3-surjections and is OEIS A001117 [35].) These are frequency counts, so dividing each entry by M^N yields Table 9.3. The fourth column corresponds to Table 9.2. The many relationships among these numbers are most easily uncovered via the symbolic method.

frequency	balls →	1	2	3	4	5	6	7	8	9	10	11	12
< 2		3	6	6									
< 3		3	9	24	54	90	90						
< 4		3	9	27	78	210	510	1050	1680	1680			
< 5		3	9	27	81	240	690	1890	4830	11130	22050	34650	34650
> 0			6	36	150	540	1806	5796	18150	55980	171006	519156	
> 1					90	630	2940	11508	40950	125100	445896		
> 2							1680	12600	62370	256410			
> 3											34650		

Table 9.5 Enumeration of 3-words with letter frequency restrictions or occupancy distributions of balls in 3 urns with restrictions or hash sequences (table size 3) with collision restrictions

Maximal occupancy. The EGF for a word comprising at most k occurrences of a given letter is $1 + z + z^2/2! + \dots + z^k/k!$. Therefore,

$$(1 + z + z^2/2! + \dots + z^k/k!)^M$$

is the EGF for words comprising at most k occurrences of each of M different letters. This is a straightforward application of the symbolic method for labelled objects (see Theorem 3.8). As we have seen, removing the restriction gives the EGF

$$(1 + z + z^2/2! + \dots + z^k/k! + \dots)^M = e^{zM}$$

and the total number of M -words of length N is $N![z^N]e^{zM} = M^N$ as expected. Taking $k = 1$ gives the EGF

$$(1 + z)^M,$$

which says that the number of words with at most one occurrence of each letter (no duplicates) is

$$N![z^N](1 + z)^M = M(M - 1)(M - 2)\dots(M - N + 1) = N! \binom{M}{N}.$$

This quantity is also known as the number of *arrangements*, or ordered combinations, of N elements chosen among M possibilities. Dividing the number of arrangements by M^N gives the probability distribution in Theorem 9.1 for the birthday problem, and the use of the symbolic method provides a straightforward generalization.

Theorem 9.3 (Maximal occupancy). The number of words of length N with at most k occurrences of each letter is

$$N![z^N] \left(1 + \frac{z}{1!} + \frac{z^2}{2!} + \dots + \frac{z^k}{k!} \right)^M.$$

In particular, the number of arrangements ($k = 1$) is $N! \binom{M}{N}$.

Proof. See the above discussion. ■

The numbers in the top half of Table 9.5 correspond to computing coefficients of these EGFs for $1 \leq k \leq 4$. For example, the second line corresponds to the expansion

$$\begin{aligned}\left(1 + z + \frac{z^2}{2!}\right)^3 &= 1 + 3z + \frac{9}{2}z^2 + 4z^3 + \frac{9}{4}z^4 + \frac{3}{4}z^5 + \frac{1}{8}z^6 \\ &= 1 + 3z + 9\frac{z^2}{2!} + 24\frac{z^3}{3!} + 54\frac{z^4}{4!} + 90\frac{z^5}{5!} + 90\frac{z^6}{6!}.\end{aligned}$$

Exercise 9.17 Find the EGF for M -words with all letter frequencies even.

Exercise 9.18 Prove that the number of ways to distribute $M(k+1)$ balls among M urns with all urns having $> k$ balls is equal to the number of ways to distribute $M(k+1)-1$ balls among M urns with all urns having $< (k+2)$ balls, for all $k \geq 0$. (See Table 9.5.) Give an explicit formula for this quantity as a quotient of factorials.

Exercise 9.19 What is the expected number of balls thrown in N urns before the second collision occurs? Assume that “collision” here means the event “ball falling into nonempty urn.”

Exercise 9.20 What is the expected number of balls thrown in N urns before the second collision occurs, when we assume that “collision” means the event “ball falling into urn with exactly one ball in it?”

Exercise 9.21 Give an explicit expression for the number of M -words with no three occurrences of the same letter.

Exercise 9.22 Give a plot like Figure 9.3 for the probability that *three* people have the same birthday.

Exercise 9.23 For $M = 365$, how many people are needed to be 50% sure that three have the same birthday? Four?

Minimal occupancy. By arguments similar to those used earlier for surjections and for maximal occupancy, the EGF for words comprising more than k occurrences of each letter is

$$(e^z - 1 - z - z^2/2! - \dots - z^k/k!)^M.$$

In particular, taking $k = 1$ gives the EGF for M -surjections, or the number of words with at least one occurrence of each of M characters, or the number of ball-and-urn sequences with M urns, none of which are empty:

$$(e^z - 1)^M.$$

Thus, we are considering a generalization of the coupon collector problem of the previous section.

Theorem 9.4 (Minimal occupancy). The number of words of length N with at least k occurrences of each letter is

$$N![z^N] \left(e^z - 1 - \frac{z}{1!} - \frac{z^2}{2!} - \cdots - \frac{z^{k-1}}{(k-1)!} \right)^M.$$

In particular, the number of M -surjections of length N is

$$N![z^N](e^z - 1)^M = M! \begin{Bmatrix} N \\ M \end{Bmatrix}.$$

Proof. See the earlier discussion. ■

The numbers in the bottom half of Table 9.5 correspond to computing coefficients of these EGFs. For example, the third line from the bottom corresponds to the expansion

$$\begin{aligned} (e^z - 1 - z)^3 &= \left(\frac{z^2}{2} + \frac{z^3}{6} + \frac{z^4}{24} + \frac{z^5}{120} + \dots \right)^3 \\ &= \frac{1}{8}z^6 + \frac{1}{8}z^7 + \frac{7}{96}z^8 + \frac{137}{4320}z^9 + \frac{13}{1152}z^{10} + \dots \\ &= 90\frac{z^6}{6!} + 630\frac{z^7}{7!} + 2940\frac{z^8}{8!} + 11508\frac{z^9}{9!} + 40950\frac{z^{10}}{10!} + \dots \end{aligned}$$

As for maximal occupancy, generating functions succinctly describe the computation of these values.

Table 9.6 gives a summary of the generating functions in Theorems 9.3 and 9.4, for enumerating words with letter frequency restrictions. These theorems are the counterparts to Theorems 7.2 and 7.3 for permutations with cycle length restrictions. These four theorems, containing the analysis of arrangements, surjections, involutions, derangements, and their generalizations, are worthy of review, as they account for a number of basic combinatorial structures and classical enumeration problems in a uniform manner.

Characterizing the asymptotic values of the functions in Theorems 9.3 and 9.4 involves addressing multivariate asymptotic problems, each with different asymptotic regimes according to the ranges considered. This may be viewed as a generalization of our treatment of the binomial distribution (see Chapter 4 and the following discussion), where different approximations are

used for different ranges of the parameter values. For instance, for fixed M , the coefficients $[z^N](1+z+z^2/2)^M$ are eventually 0, as $N \rightarrow \infty$. At the same time, for fixed M , the coefficients sum to $(5/2)^M$, with a peak near $5M/4$, and a normal approximation near the peak can be developed for the coefficients. Thus, there are interesting regions when M and N are proportional. Similarly, consider $[z^N](e^z - 1 - z)^M$, and fixed M , as $N \rightarrow \infty$. Here, we are counting functions that assume each value at least twice. But all but a very small fraction of the functions will assume all values at least twice (actually about N/M times). Thus, this coefficient is asymptotic to $[z^N](e^z)^M$. Again, there will be an interesting transition when M grows and becomes $O(N)$. Such asymptotic results are best quantified by saddle point methods, as discussed in detail by Kolchin [28] (see also [14]).

Exercise 9.24 What is the average number of balls thrown into M urns before each urn is filled at least twice?

Exercise 9.25 Derive an expression for the exponential CGF for the expected minimal occupancy when N balls are distributed into M urns. Tabulate the values for M and N less than 20.

Expected maximum occupancy. What is the average of the *maximum* number of balls in an urn, when N balls are distributed randomly among M urns? This is an extremal parameter similar to several others that we have encountered. As we have done for tree height and maximum cycle length in permutations, we can use generating functions to compute the maximum occupancy.

one urn, occupancy k	$z^k/k!$
all words	e^{z^M}
all occupancies > 1 (surjections)	$(e^z - 1)^M$
all occupancies $> k$	$(e^z - 1 - z - z^2/2! - \dots - z^k/k!)^M$
no occupancies > 1 (arrangements)	$(1+z)^M$
no occupancies $> k$	$(1+z+z^2/2! + \dots + z^k/k!)^M$

Table 9.6 EGFs for words with letter frequency restrictions or ball-and-urn occupancy distributions with restrictions or hash sequences with collision frequency restrictions

From Theorem 9.4, we can write down the generating functions for the ball-and-urn configurations with at least one urn with occupancy $> k$, or, equivalently, those for which the maximum occupancy is $> k$:

$$\begin{aligned}
 e^{3z} - (1)^3 &= 3z + 9\frac{z^2}{2!} + 27\frac{z^3}{3!} + 81\frac{z^4}{4!} + 243\frac{z^5}{5!} + \dots \\
 e^{3z} - (1+z)^3 &= 3\frac{z^2}{2!} + 21\frac{z^3}{3!} + 81\frac{z^4}{4!} + 243\frac{z^5}{5!} + \dots \\
 e^{3z} - \left(1+z+\frac{z^2}{2!}\right)^3 &= 6\frac{z^3}{3!} + 27\frac{z^4}{4!} + 153\frac{z^5}{5!} + \dots \\
 e^{3z} - \left(1+z+\frac{z^2}{2!}+\frac{z^3}{3!}\right)^3 &= 3\frac{z^4}{4!} + 33\frac{z^5}{5!} + \dots \\
 e^{3z} - \left(1+z+\frac{z^2}{2!}+\frac{z^3}{3!}+\frac{z^4}{4!}\right)^3 &= 3\frac{z^5}{5!} + \dots \\
 &\vdots
 \end{aligned}$$

and so on, and we can sum these to get the exponential CGF

$$= 3z + 12\frac{z^2}{2!} + 54\frac{z^3}{3!} + 192\frac{z^4}{4!} + 675\frac{z^5}{5!} + \dots$$

for the (cumulative) maximum occupancy when balls are distributed in three urns. Dividing by 3^N yields the average values given in Table 9.3. In general, the average maximum occupancy is given by

$$\frac{N!}{M^N} [z^N] \sum_{k \geq 0} \left(e^{Mz} - \left(\sum_{0 \leq j \leq k} \frac{z^j}{j!} \right)^M \right).$$

This quantity was shown by Gonnet [16] to be $\sim \ln N / \ln \ln N$ as $N, M \rightarrow \infty$ in such a way that $N/M = \alpha$ with α constant (the leading term is independent of α). Thus, for example, the length of the longest list when Program 9.1 is used will be $\sim \ln N / \ln \ln N$, on the average.

Exercise 9.26 What is the average number of blocks of contiguous equal elements in a random word?

Exercise 9.27 Analyze “rises” and “runs” in words (cf. §7.1).

9.5 Occupancy Distributions. The probability that an M -word of length N contains exactly k instances of a given character is

$$\binom{N}{k} \left(\frac{1}{M}\right)^k \left(1 - \frac{1}{M}\right)^{N-k}.$$

This is established by a straightforward calculation: the binomial coefficient counts the ways to pick the positions, the second factor is the probability that those letters have the value, and the third factor is the probability that the other letters do not have the value. We studied this distribution, the familiar *binomial* distribution, in detail in Chapter 4. and have already encountered it on several different occasions throughout this book. For example, Table 4.6 gives values for $M = 2$. Another example is given in Table 9.7, which gives corresponding values for $M = 3$; the fourth line in this table corresponds to Table 9.2.

Involvement of two distinct variables (number of balls and number of urns) and interest in different segments of the distribution mean that care needs to be taken to characterize it accurately for particular applications. The intuition of the balls-and-urns model is often quite useful for this purpose.

In this section, we will be examining precise formulae and asymptotic estimates of the distributions for many values of the parameters. A few sample values are given in Table 9.8. For example, when 100 balls are distributed in 100 urns, we expect that about 18 of the urns will have 2 balls, but the chance

N		0	1	2	3	4	5	6
$\downarrow k \rightarrow$								
1	0.666667	0.333333						
2	0.444444	0.444444	0.111111					
3	0.296296	0.444444	0.222222	0.037037				
4	0.197531	0.395062	0.296296	0.098765	0.012346			
5	0.131687	0.329218	0.329218	0.164609	0.041152	0.004115		
6	0.087791	0.263375	0.329218	0.219479	0.082305	0.016461	0.001372	

Table 9.7 Occupancy distribution for $M = 3$: $\binom{N}{k} (1/3)^k (2/3)^{N-k}$
 $\Pr\{\text{an urn has } k \text{ balls after } N \text{ balls are distributed in 3 urns}\}$

that any urn has as many as 10 balls is negligible. On the other hand, when 100 balls are distributed in 10 urns, 1 or 2 of the urns are likely to have 10 balls (the others are likely to have between 7 and 13), but very few are likely to have 2 balls. As we saw in Chapter 4, these results can be described with the normal and Poisson approximations, which are accurate and useful for characterizing this distribution for a broad range of values of interest.

The total number of M -words of length N having a character appear k times is given by

$$M \binom{N}{k} (M-1)^{N-k},$$

by the following argument: There are M characters; the binomial coefficient counts the indices where a given value may occur, and the third factor is the

urns M	balls N	occupancy k	average # urns with k balls $M \binom{N}{k} \left(\frac{1}{M}\right)^k \left(1 - \frac{1}{M}\right)^{N-k}$
2	2	2	0.500000000
2	10	2	0.087890625
2	10	10	0.001953125
10	2	2	0.100000000
10	10	2	1.937102445
10	10	10	0.000000001
10	100	2	0.016231966
10	100	10	1.318653468
100	2	2	0.010000000
100	10	2	0.415235112
100	10	10	0.000000000
100	100	2	18.486481882
100	100	10	0.000007006

Table 9.8 Occupancy distribution examples

number of ways the other indices can be filled with the other characters. This leads to the BGF, which we can use to compute moments, as usual. Dividing by M^N leads us to the classical formulation of the distribution, which we restate here in the language of balls and urns, along with asymptotic results summarized from Chapter 4.

Theorem 9.5 (Occupancy distribution). The average number of urns with k balls, when N balls are randomly distributed in M urns, is

$$M \binom{N}{k} \left(\frac{1}{M}\right)^k \left(1 - \frac{1}{M}\right)^{N-k}.$$

For $k = N/M + x\sqrt{N/M}$ with $x = O(1)$, this is

$$M \frac{e^{-x^2}}{\sqrt{2\pi}} + O\left(\frac{1}{\sqrt{N}}\right) \quad (\text{normal approximation}),$$

and for $N/M = \alpha > 0$ fixed and $k = O(1)$, this is

$$M \frac{\alpha^k e^{-\alpha}}{k!} + o(M) \quad (\text{Poisson approximation}).$$

Proof. See earlier discussion. The stated approximations to the binomial distribution are from Chapter 4 (Exercise 4.66 and Theorem 4.7). ■

Corollary When $N/M = \alpha$ (constant), the average number of empty urns is asymptotic to $Me^{-\alpha}$.

Corollary The average number of balls per urn is N/M , with standard deviation $\sqrt{N/M - N/M^2}$.

Proof. Multiplying the cumulative cost given above by u^k and z^N , we get the BGF

$$\begin{aligned} C^{[M]}(z, u) &= \sum_{N \geq 0} \sum_{k \geq 0} C_{Nk}^{[M]} z^N u^k = \sum_{N \geq 0} \sum_{k \geq 0} \binom{N}{k} (M-1)^{N-k} u^k z^N \\ &= \sum_{N \geq 0} (M-1+u)^N z^N \\ &= \frac{1}{1 - (M-1+u)z}. \end{aligned}$$

Dividing by M^N or, equivalently, replacing z by z/M converts this cumulative BGF into a PGF that is slightly more convenient to manipulate. The cumulated cost is given by differentiating this with respect to u and evaluating at $u = 1$, as in Table 3.6.

$$[z^N] \frac{\partial C^{[M]}(z/M, u)}{\partial u} \Big|_{u=1} = [z^N] \frac{1}{M} \frac{z}{(1-z)^2} = \frac{N}{M}$$

and

$$[z^N] \frac{\partial^2 C^{[M]}(z/M, u)}{\partial u^2} \Big|_{u=1} = [z^N] \frac{1}{M^2} \frac{z^2}{(1-z)^3} = \frac{N(N-1)}{M^2}$$

so the average is N/M and the variance is $N(N-1)/M^2 + N/M - (N/M)^2$, which simplifies to the stated result. ■

Alternative derivations. We have presented these calculations along familiar classical lines, but the symbolic method of course provides a quick derivation. For a particular urn, the BGF for a ball that misses the urn is $(M-1)z$ and the BGF for a ball that hits the urn is uz ; therefore the ordinary BGF for a sequence of balls is

$$\sum_{N \geq 0} ((M-1+u)z)^N = \frac{1}{1-(M-1+u)z},$$

as before.

Alternatively, the exponential BGF

$$F(z, u) = \left(e^z + (u-1) \frac{z^k}{k!} \right)^M$$

gives the cumulated number of urns with k balls:

$$N! [z^N] \frac{\partial F(z, u)}{\partial u} \Big|_{u=1} = N! [z^N] M e^{(M-1)z} \frac{z^k}{k!} = M \binom{N}{k} (M-1)^{N-k}$$

as before.

The occupancy and binomial distributions have a broad variety of applications and have been very widely studied, so many other ways to derive

these results are available. Indeed, it is important to note that the average number of balls per urn, which would seem to be the most important quantity to analyze, is completely independent of the distribution. *No matter how* the balls are distributed in the urns, the cumulative cost is N : counting the balls in each urn, then adding the counts, is equivalent to just counting the balls. The average number of balls per urn is N/M , whether or not they were distributed randomly. The variance is what tells us whether the number of balls in a given urn can be expected to be near N/M .

Figures 9.4 and 9.5 show the occupancy distribution for various values of M . The bottom series of curves in Figure 9.4 corresponds precisely to Figure 4.4, the binomial distribution centered at $1/5$. For large M , illustrated in Figure 9.5, the Poisson approximation is appropriate. The limiting curves for the bottom two families in Figure 9.5 are the same as the limiting curves for the top two families in Figure 4.5, the Poisson distribution for $N = 60$ with $\lambda = 1$ and $\lambda = 2$. (The other limiting curves in Figure 4.5, for $N = 60$, are the occupancy distributions for $M = 20$ and $M = 15$.) As M gets smaller with respect to N , we move into the domain illustrated by Figure 9.4, where the normal approximation is appropriate.

Exercise 9.28 What is the probability that one urn will get all the balls when 100 balls are randomly distributed among 100 urns?

Exercise 9.29 What is the probability that each urn will get one ball when 100 balls are randomly distributed among 100 urns?

Exercise 9.30 What is the standard deviation for the average number of empty urns?

Exercise 9.31 What is the probability that each urn will contain an even number of balls when N balls are distributed among M urns?

Exercise 9.32 Prove that

$$C_{Nk}^{[M]} = (M - 1)C_{(N-1)k}^{[M]} + C_{(N-1)(k-1)}^{[M]}$$

for $N > 1$ and use this fact to write a program that will print out the occupancy distribution for any given M .

Analysis of hashing with separate chaining. Properties of occupancy distributions are the basis for the analysis of hashing algorithms. For example, an unsuccessful search in a hash table using separate chaining involves accessing a random list, then following it to the end. The cost of such a search thus satisfies an occupancy distribution.

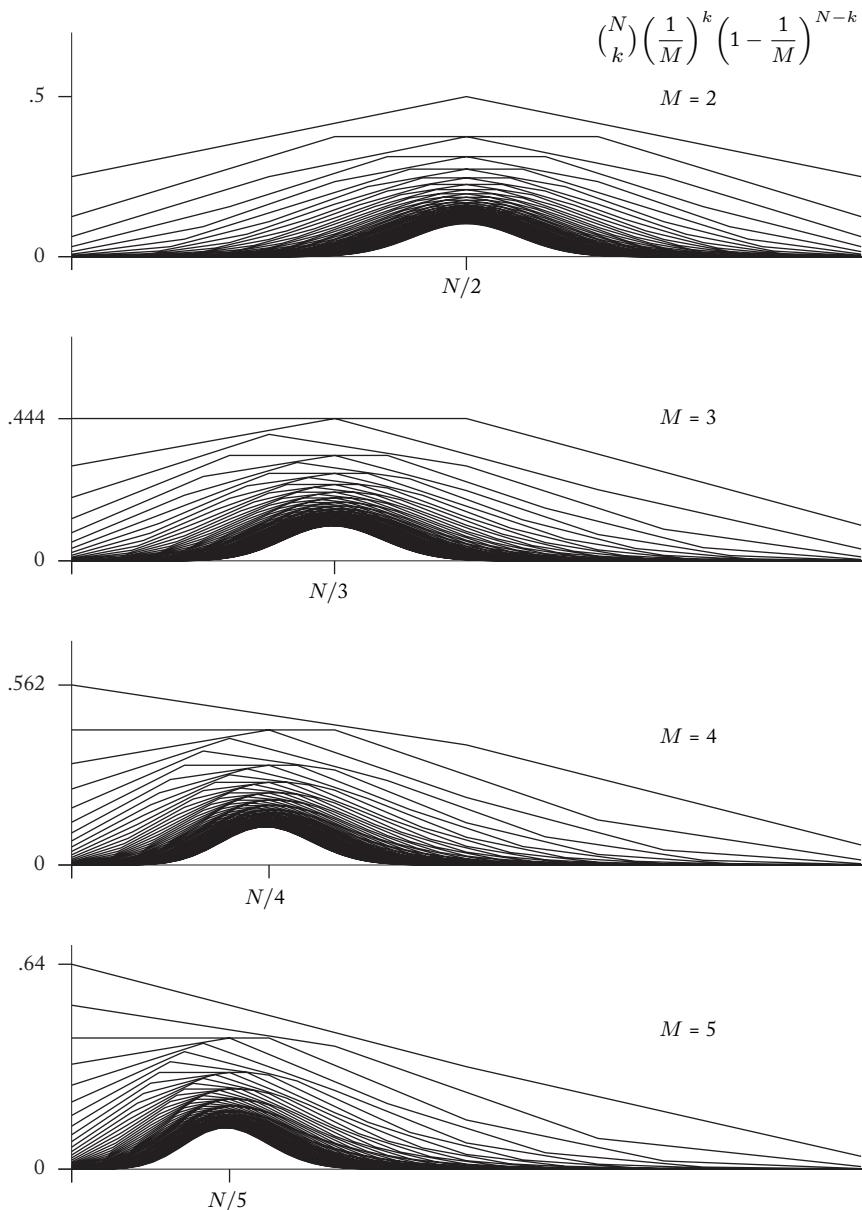


Figure 9.5 Occupancy distributions for small M and $2 \leq N \leq 60$
(k -axes scaled to N)

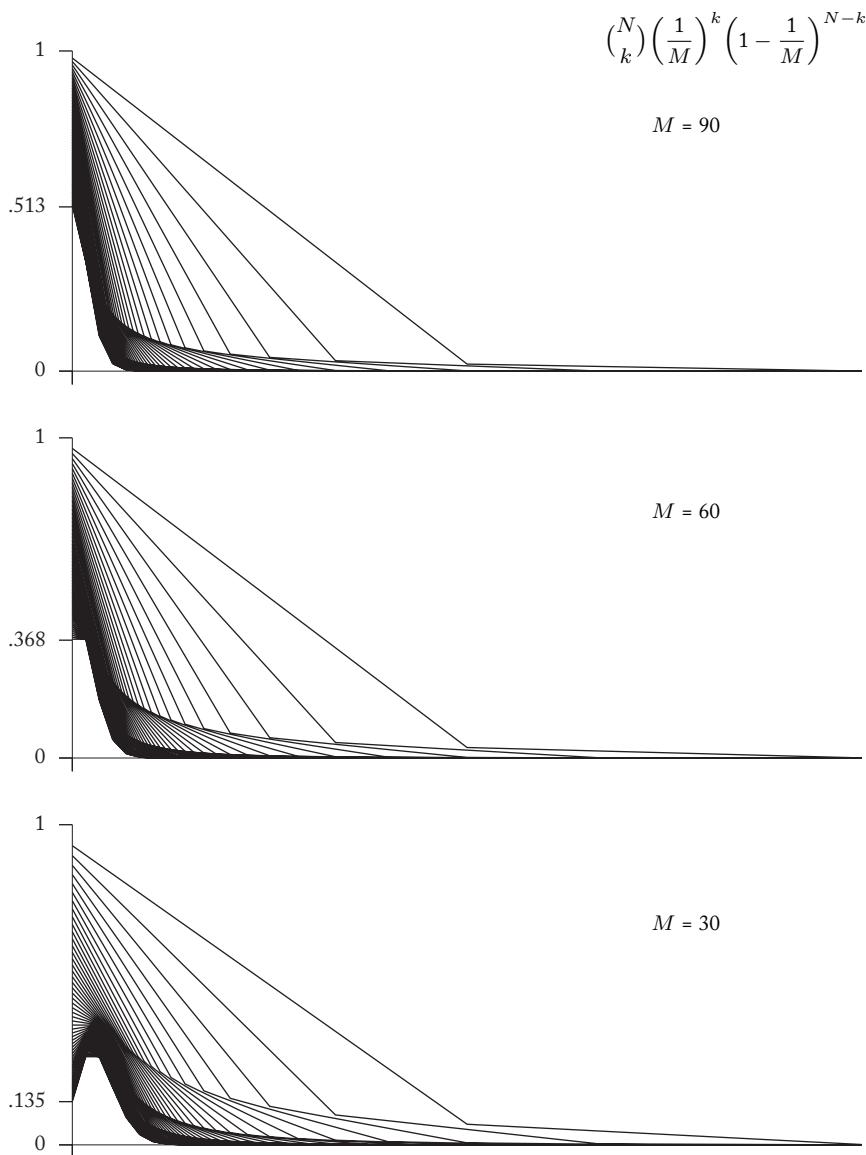


Figure 9.6 Occupancy distributions for large M and $2 \leq N \leq 60$
(k -axes scaled to N)

Theorem 9.6 (Hashing with separate chaining). Using a table of size M for N keys, hashing with separate chaining requires N/M probes for an unsuccessful search and $(N + 1)/(2M)$ probes for a successful search, on the average.

Proof. The result for an unsuccessful search follows directly from the earlier discussion. The cost of accessing a key that is in the table is the same as the cost of putting it into the table, so the average cost of a successful search is the average cost of all the unsuccessful searches used to build the table—in this case

$$\frac{1}{N} \sum_{1 \leq k \leq N} \frac{k}{M} = \frac{N + 1}{2M}.$$

This relationship between unsuccessful and successful search costs holds for many searching algorithms, including binary search trees. ■

The Chebyshev inequality says that with 1000 keys, we could use 100 lists and expect about 10 items per list, with at least 90% confidence that a search will examine no more than 20 items. With 1 million keys, one might use a 1000 lists, and the Chebyshev inequality says that there is at least 99.9% confidence that a search will examine no more than 2000 items. Though generally applicable, the Chebyshev bounds are actually quite crude in this case, and we can show through direct numerical computation or through use of the Poisson approximation that for 1 million keys and a table size of 1000, the probability that more than 1300 probes are needed is on the order of 10^{-20} , and the probability that more than 2000 probes are needed is around 10^{-170} .

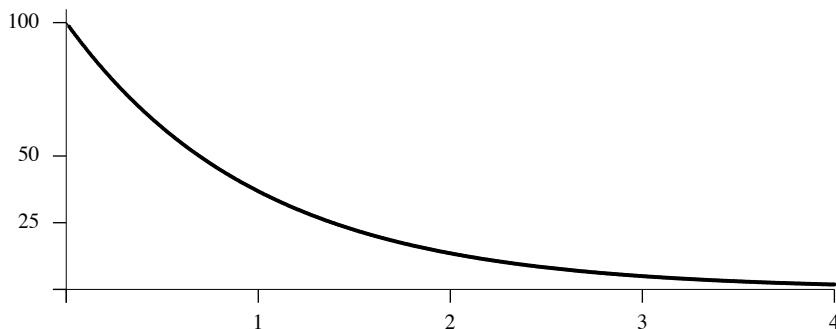


Figure 9.7 Percentage of empty urns as a function of load factor N/M

Furthermore, we can know many other properties of the hash structure that may be of interest. For example, Figure 9.6 is a plot of the function $e^{-\alpha}$, which tells us the percentage of empty lists as a function of the ratio of the number of keys to the number of lists. Such information can be instrumental in tuning an algorithm to best performance.

A number of variants to the basic separate chaining scheme have been devised to economize on space in light of these two observations. The most notable of these is *coalesced hashing*, which has been analyzed in detail by Vitter and Chen [37] (see also Knuth [25]). This is an excellent example of the use of analysis to set values of performance parameters in a practical situation.

Exercise 9.33 For 1000 keys, which value of M will make hashing with separate chaining access fewer keys than a binary tree search? For 1 million keys?

Exercise 9.34 Find the standard deviation of the number of comparisons required for a successful search in hashing with separate chaining.

Exercise 9.35 Determine the average and standard deviation of the number of comparisons used for a search when the lists in the table are kept in sorted order (so that a search can be cut short when a key larger than the search key is found).

Exercise 9.36 [Broder and Karlin] Analyze the following variant of Program 9.1: compute two hash functions and put the key on the shorter of the two lists.

9.6 Open Addressing Hashing. If we take M to be a constant multiple of N in hashing with separate chaining, then our search time is constant, but we use a significant amount of space, in the form of pointers, to maintain the data structure. So-called *open addressing* methods do not use pointers and directly address a set of N keys within a table of size M with $M \geq N$.

The birthday paradox tells us that the table does not need to be very large for some keys to have the same hash values, so collision resolution strategy is immediately needed to decide how to deal with such conflicts.

Linear probing. Perhaps the simplest such strategy is *linear probing*: keep all the keys in an array of size M and use the hash value of the key as an index into the array. If, when inserting a key into the table, the addressed position (given by the hash value) is occupied, then simply examine the previous position. If that is also occupied, examine the one before that, continuing until an empty position is found. (If the beginning of the table is reached, simply cycle back

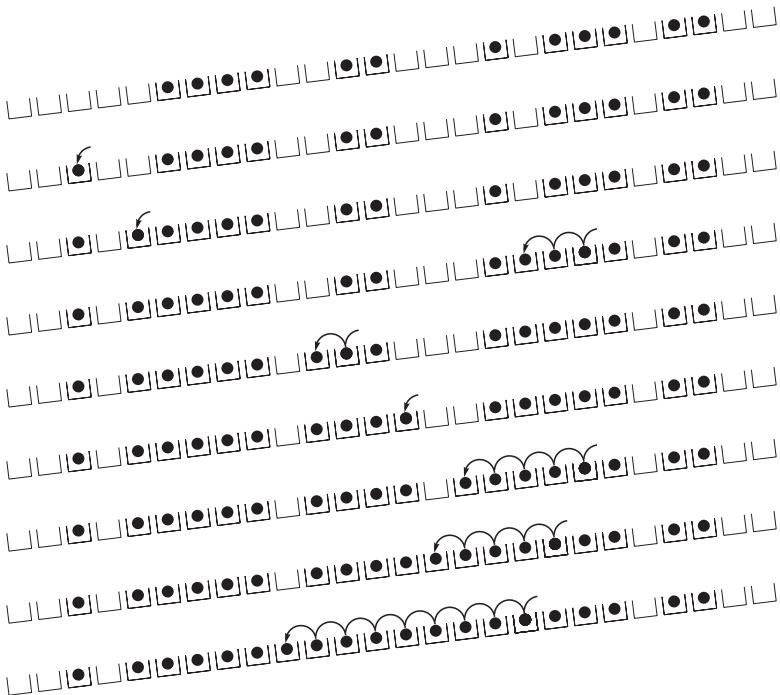


Figure 9.8 Hashing with linear probing

to the end.) In the balls-and-urns model, we might imagine linear probing to be a sort of pachinko machine, where one ball fills up an urn and new balls bounce to the left until an empty urn is found. An implementation of search and insertion for linear probing is given in Program 9.2. The program keeps the keys in an array $a[]$ and assumes that the hash function does not return zero, so that zero can be used to mark empty positions in the hash table.

We will see later that linear probing performs badly for a nearly full table but reasonably well for a table with enough empty space. As the table fills up, the keys tend to “cluster” together, producing long chains that must be searched to find an empty space. Figure 9.7 shows an example of a table filling up with linear probing, with a cluster developing in the last two insertions.

An easy way to avoid clustering is to look not at the previous position but at the t th previous position each time a full table entry is found, where t is computed by a second hash function. This method is called *double hashing*.

Uniform hashing. Linear probing and double hashing are difficult to analyze because of interdependencies among the lists. A simple approximate model is to assume that each occupancy configuration of N keys in a table of size M is equally likely to occur. This is equivalent to the assumption that a hash function produces a random *permutation* and the positions in the hash table are examined in random order (different for different keys) until an empty position is found.

Theorem 9.7 (Uniform hashing). Using a table of size M for N keys, the number of probes used for successful and unsuccessful searches with uniform hashing is

$$\frac{M+1}{M-N+1} \quad \text{and} \quad \frac{M+1}{N}(H_{M+1} - H_{M-N+1}),$$

```
public void insert(int key)
{
    for (i = hash(key); a[i] != 0; i = (i - 1) % M)
        if (a[i] == key) return;
    a[i] = key;
}
public boolean search(int key)
{
    int i;
    for (i = hash(key); a[i] != 0; i = (i - 1) % M)
        if (a[i] == key)
            return true;
    return false;
}
```

Program 9.2 Hashing with linear probing

(respectively), on the average.

Proof. An unsuccessful search will require k probes if $k - 1$ table locations starting at the hashed location are full and the k th empty. With k locations and $k - 1$ keys accounted for, the number of configurations for which this holds is the number of ways to distribute the other $N - k + 1$ keys among the other $M - k$ locations. The total unsuccessful search cost in all the occupancy configurations is therefore

$$\sum_{1 \leq k \leq M} k \binom{M - k}{N - k + 1} = \sum_{1 \leq k \leq M} k \binom{M - k}{M - N - 1} = \binom{M + 1}{M - N + 1}$$

(see Exercise 3.34) and the average cost for unsuccessful search is obtained by dividing this by the total number of configurations $\binom{M}{N}$.

The average cost for a successful search is obtained by averaging the unsuccessful search cost, as in the proof of Theorem 9.6. ■

Thus, for $\alpha = N/M$, the average cost for a successful search is asymptotic to $1/(1 - \alpha)$. Intuitively, for small α , we expect that the probability that the first cell examined is full to be α , the probability that the first two cells examined are full to be α^2 , and so on, so the average cost should be asymptotic to

$$1 + \alpha + \alpha^2 + \alpha^3 + \dots$$

This analysis validates that intuition under the uniformity assumption. The cost for successful searches can be calculated by averaging the average costs for unsuccessful searches, as in the proof of Theorem 9.6.

The uniform hashing algorithm is impractical because of the cost of generating a permutation for each key, but the corresponding model does provide a performance goal for other collision resolution strategies. Double hashing is an attempt at approximating such a “random” collision resolution strategy, and it turns out that its performance approximates the results for uniform hashing, though this is a difficult result that was some years in the making (see Guibas and Szemerédi [20] and Lueker and Molodowitch [30]).

Analysis of linear probing. Linear probing is a fundamental searching method, and an analytic explanation of the clustering phenomenon is clearly of interest. The algorithm was first analyzed by Knuth in [25], where he states in a footnote that this derivation had a strong influence on the structure of

his books. Since Knuth's books certainly have had a strong influence on the structure of research in the mathematical analysis of algorithms, we begin by presenting Kunth's classic derivation as a prototype example showing how a simple algorithm can lead to nontrivial and interesting mathematical problems.

Following Knuth, we define three quantities that we will use to develop an exact expression for the cumulative cost for unsuccessful searches:

$$f_{NM} = \{\# \text{ words where 0 is left empty}\}$$

$$g_{NMk} = \{\# \text{ words where 0 and } k+1 \text{ are left empty, 1 through } k \text{ full}\}$$

$$p_{NMj} = \{\# \text{ words where inserting the } (N+1)\text{st key takes } j+1 \text{ steps}\}.$$

As usual, by *words* in this context, we mean " M -words of length N ," the sequences of hashed values that we assume to be equally likely, each occurring with probability $1/M^N$.

First, we get an explicit expression for f_{NM} by noting that position 0 is equally likely to be empty as any other table position. The M^N hash sequences each leave $M - N$ empty table positions, for a grand total of $(M - N)M^N$, and dividing by M gives

$$f_{NM} = (M - N)M^{N-1}.$$

Second, we can use this to get an explicit expression for g_{NMk} . The empty positions divide each hash sequence to be included in the count into two independent parts, one containing k elements hashing into positions 0 through k and leaving 0 empty, and the other containing $N - k$ elements hashing into positions $k+1$ through $M-1$ and leaving $k+1$ empty. Therefore,

$$\begin{aligned} g_{NMk} &= \binom{N}{k} f_{k(k+1)} f_{(N-k)(M-k-1)} \\ &= \binom{N}{k} (k+1)^{k-1} (M-N-1) (M-k-1)^{N-k-1}. \end{aligned}$$

Third, a word will involve $j+1$ steps for the insertion of the $(N+1)$ st key whenever the hashed position is in the j th position of a block of k consecutive

occupied cells (with $k \geq j$) delimited at both ends by unoccupied cells. Again by circular symmetry, the number of such words is g_{NMk} , so

$$p_{NMj} = \sum_{j \leq k \leq N} g_{NMk}.$$

Now, we can use the cumulated counts p_{NMj} to calculate the average search costs, just as we did earlier. The cumulated cost for an unsuccessful search is

$$\begin{aligned} \sum_{j \geq 0} (j+1)p_{NMj} &= \sum_{j \geq 0} (j+1) \sum_{j \leq k \leq N} g_{NMk} = \sum_{k \geq 0} g_{NMk} \sum_{0 \leq j \leq k} (j+1) \\ &= \frac{1}{2} \sum_{k \geq 0} (k+1)(k+2)g_{NMk} \\ &= \frac{1}{2} \sum_{k \geq 0} ((k+1) + (k+1)^2)g_{NMk}. \end{aligned}$$

Substituting the expression for g_{MNk} just derived, dividing by M^N , and simplifying, we find that the average cost for an unsuccessful search in linear probing is

$$\frac{1}{2}(S_{NM1}^{[1]} + S_{NM1}^{[2]})$$

where

$$S_{NMt}^{[i]} \equiv \frac{M-t-N}{M^N} \sum_k \binom{N}{k} (k+t)^{k-1+i} (M-k-t)^{N-k-i}.$$

This rather daunting function is actually rather easily evaluated with Abel's identity (Exercise 3.66 in §3.11). This immediately gives the result

$$tS_{NMt}^{[0]} = 1 - \frac{N}{M}.$$

For larger i it is easy to prove (by taking out one factor of $(k+t)$) that

$$S_{NMt}^{[i]} = \frac{N}{M} S_{(N-1)M(t+1)}^{[i]} + tS_{NMt}^{[i-1]}.$$

Therefore,

$$S_{NMt}^{[1]} = \frac{N}{M} S_{(N-1)M(t+1)}^{[1]} + 1 - \frac{N}{M},$$

which has the solution

$$S_{NMt}^{[1]} = 1.$$

This is to be expected, since, for example, $S_{NM1}^{[1]} = \sum_k (p_{NMk})/M^N$, a sum of probabilities. Finally, for $i = 2$, we have

$$S_{NMt}^{[2]} = \frac{N}{M} S_{(N-1)M(t+1)}^{[2]} + t,$$

which has the solution

$$S_{NM1}^{[2]} = \sum_{0 \leq i \leq N} i \frac{N!}{M^i (N-i)!}.$$

Theorem 9.8 (Hashing with linear probing). Using a table of size M for N keys, linear probing requires

$$\frac{1}{2} + \frac{1}{2} \sum_{0 \leq i < N} \frac{(N-1)!}{M^i (N-i-1)!} = \frac{1}{2} \left(1 + \frac{1}{1-\alpha} \right) + O\left(\frac{1}{N}\right)$$

probes for a successful search and

$$\frac{1}{2} + \frac{1}{2} \sum_{0 \leq i \leq N} i \frac{N!}{M^i (N-i)!} = \frac{1}{2} \left(1 + \frac{1}{(1-\alpha)^2} \right) + O\left(\frac{1}{N}\right)$$

for an unsuccessful search, on the average. The asymptotic forms hold for $\alpha = N/M$ with $\alpha < 1$.

Proof. See the earlier discussion. The expression for a successful search is obtained by averaging the result for an unsuccessful search, as usual.

If α is strictly less than 1, the sum is similar to the Ramanujan Q -function of Theorem 4.8, and it is not difficult to estimate using the Laplace method. We have

$$\sum_{0 \leq i \leq N} \frac{N!}{M^i (N-i)!} = \sum_{0 \leq i \leq N} \left(\frac{N}{M} \right)^i \frac{N!}{N^i (N-i)!}.$$

Splitting the sum into two parts, we can use the fact that terms in this sum begin to get negligibly small after $i > \sqrt{N}$ to prove that this sum is

$$\sum_{i \geq 0} \left(\frac{N}{M} \right)^i \left(1 + O\left(\frac{i^2}{N}\right) \right) = \frac{1}{1 - \alpha} + O\left(\frac{1}{N}\right).$$

Adding 1 and dividing by 2 gives the stated result for a successful search. A similar calculation gives the stated estimate for an unsuccessful search. ■

Corollary The average number of table entries examined by linear probing during a successful search in a full table is $\sim \sqrt{\pi N}/2$.

Proof. Taking $M = N$ gives precisely the Ramanujan Q -function, whose approximate value is proved in Theorem 4.8. ■

Despite the relatively simple form of the solution, a derivation for the average cost of linear probing via analytic combinatorics challenged researchers for many years. There are many interesting relationships among the quantities that arise in the analysis. For example, if we multiply the expression for a successful search in Theorem 9.8 by $z^{\bar{N}-1}$, divide by $(N-1)!$, and sum for all $N > 0$, we get the rather compact explicit result

$$\frac{1}{2} \left(e^z + \frac{e^M}{1 - z/M} \right).$$

This is not directly meaningful for linear probing because the quantities are defined only for $N \leq M$ but it would seem a fine candidate for a combinatorial interpretation.

In 1998, motivated by a challenge in a footnote in the first edition of this book [13], Flajolet, Poblete, Viola, and Knuth, in a pair of companion papers [12][26], developed independent analytic-combinatorial analyses of linear probing that uncover the rich combinatorial structure of this problem. The end results provide in themselves a convincing example of the utility of analytic combinatorics in the analysis of algorithms. They include moments and even full distributions in sparse and full tables and relate the problem to graph connectivity, inversions in Cayley trees, path length in trees, and other problems. Analysis of hashing algorithms remains an area of active research.

Exact costs for N keys in a table of size M

	successful search	unsuccessful search
separate chaining	$1 + \frac{N}{2M}$	$1 + \frac{N}{M}$
uniform hashing	$\frac{M+1}{M-N+1}$	$\frac{M+1}{N}(H_{M-1} - H_{M-N+1})$
linear probing	$\frac{1}{2} \left(1 + \sum_k \frac{k!}{M^k} \binom{N-1}{k} \right)$	$\frac{1}{2} \left(1 + \sum_k k \frac{k!}{M^k} \binom{N}{k} \right)$

Asymptotic costs as $N, M \rightarrow \infty$ with $\alpha \equiv N/M$

	average	.5	.9	.95	small α
<i>unsuccessful search</i>					
separate chaining	$1 + \alpha$	2	2	2	$1 + \alpha$
uniform hashing	$\frac{1}{1 - \alpha}$	2	10	20	$1 + \alpha + \alpha^2 + \dots$
double hashing	$\frac{1}{1 - \alpha}$	2	10	20	$1 + \alpha + \alpha^2 + \dots$
linear probing	$\frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2} \right)$	3	51	201	$1 + \alpha + \frac{3\alpha^2}{2} + \dots$
<i>successful search</i>					
separate chaining	$1 + \frac{\alpha}{2}$	1	1	1	$1 + \frac{\alpha}{2}$
uniform hashing	$\frac{1}{\alpha} \ln(1 + \alpha)$	1	3	4	$1 + \frac{\alpha}{2} + \frac{\alpha^2}{3} + \dots$
double hashing	$\frac{1}{\alpha} \ln(1 + \alpha)$	1	3	4	$1 + \frac{\alpha}{2} + \frac{\alpha^2}{3} + \dots$
linear probing	$\frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right)$	2	6	11	$1 + \frac{\alpha}{2} + \frac{\alpha^2}{2} + \dots$

Table 9.9 Analytic results for hashing methods

THE ASYMPTOTIC PERFORMANCE OF the hashing methods we have discussed is summarized in Table 9.9. This table includes the asymptotic cost as a function of the load factor $\alpha \equiv N/M$ as the table size M and the number of keys N grow; an expansion of the cost function that estimates the cost for small α ; and approximate values of the functions for typical values of α . The table shows that all the methods perform roughly the same for small α ; that linear probing begins to degrade to an unacceptable level when the table gets 80–90% full; and that the performance of double hashing is quite close to “optimal” (same as separate chaining) unless the table is very full. These and related results can be quite useful in the application of hashing in practice.

Exercise 9.37 Find $[z^n]e^{\alpha C(z)}$ where $C(z)$ is the Cayley function (see the discussion at the end of §6.14 and in §9.7 in this chapter).

Exercise 9.38 (“Abel’s binomial theorem.”) Use the result of the previous exercise and the identity $e^{(\alpha+\beta)C(z)} = e^{\alpha C(z)}e^{\beta C(z)}$ to prove that

$$(\alpha + \beta)(n + \alpha + \beta)^{n-1} = \alpha\beta \sum_k \binom{n}{k} (k + \alpha)^{k-1} (n - k + \beta)^{n-k-1}.$$

Exercise 9.39 How many keys can be inserted into a linear probing table of size M before the average search cost gets to be greater than $\ln N$?

Exercise 9.40 Compute the exact cost of an unsuccessful search using linear probing for a full table.

Exercise 9.41 Give an explicit representation for the EGF for the cost of an unsuccessful search.

Exercise 9.42 Use the symbolic method to derive the EGF of the number of probes required by linear probing in a successful search, for fixed M .*

* The temptation to include a footnote at this point cannot be resisted: while we *still* do not quite know the answer to this exercise (see the comment at the end of [26]), it is perhaps irrelevant because we do have full information on the performance of a large class of hashing algorithms that includes linear probing (see [21] and [36]).

9.7 Mappings. The study of hashing into a full table leads naturally to looking at properties of mappings from the set of integers between 1 and N onto itself. The study of these leads to a remarkable combinatorial structure that is simply defined, but it encompasses much of what we have studied in this book.

Definition An N -mapping is a function f mapping integers from the interval $[1 \dots N]$ into the interval $[1 \dots N]$.

As with words, permutations, and trees, we specify a mapping by writing down its functional table:

index	1	2	3	4	5	6	7	8	9
mapping	9	6	4	2	4	3	7	8	6

As usual, we drop the index and specify a mapping simply as a sequence of N integers in the interval 1 to N (the *image* of the mapping). Clearly, there are N^N different N -mappings. We have used similar representations for permutations (Chapter 7) and trees (Chapter 6)—mappings encompass both of these as special cases. For example, a permutation is a mapping where the integers in the image are distinct.

Naturally, we define a *random mapping* to be a sequence of N random integers in the range 1 to N . We are interested in studying properties of random mappings. For example, the probability that a random mapping is a permutation is $N!/N^N \sim \sqrt{2\pi N}/e^N$.

Image cardinality. Some properties of mappings may be deduced from properties of words derived in the previous section. For example, by Theorem 9.5, we know that the average number of integers that appear k times in the mapping is $\sim Ne^{-1}/k!$, the Poisson distribution with $\alpha = 1$. A related question of interest is the distribution of the number of different integers that appear, the cardinality of the image. This is N minus the number of integers that do not appear, or the number of “empty urns” in the occupancy model, so the average is $\sim (1 - 1/e)N$ by the corollary to Theorem 9.5. A simple counting argument says that the number of mappings with k different integers in the image is given by $\binom{N}{k}$ (choose the integers) times $k! \left\{ \begin{smallmatrix} N \\ k \end{smallmatrix} \right\}$ (count all the surjections with image of cardinality k). Thus,

$$C_{Nk} = k! \binom{N}{k} \left\{ \begin{smallmatrix} N \\ k \end{smallmatrix} \right\}.$$

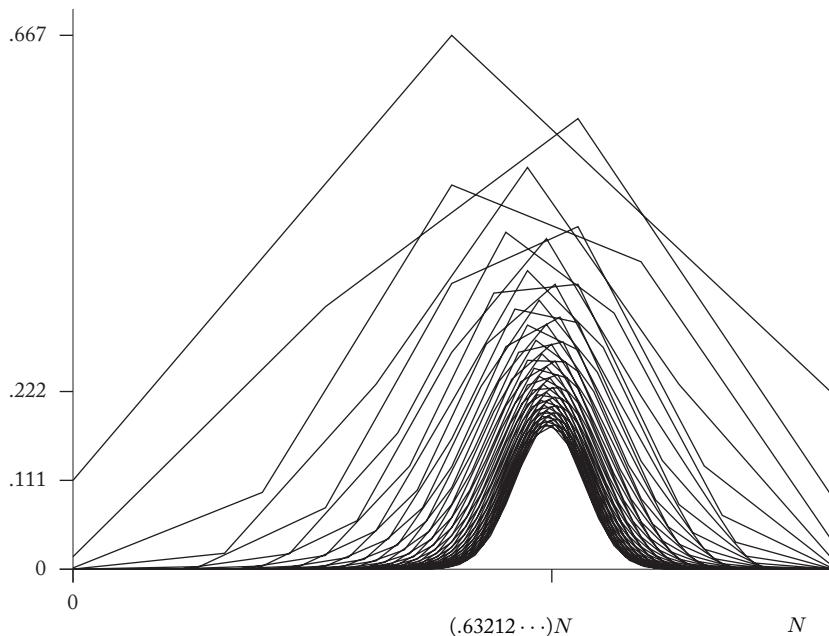


Figure 9.9 Image cardinality of random mappings for $3 \leq N \leq 50$
(k -axes scaled to N)

This distribution is plotted in Figure 9.8.

Exercise 9.43 Find the exponential BGF for the image cardinality distribution.

Exercise 9.44 Use a combinatorial argument to find the exponential BGF for the image cardinality distribution.

Exercise 9.45 Give a recurrence relationship for the number of mappings of size N with k different integers in the image, and use that to obtain a table of values for $N < 20$.

Exercise 9.46 Give an explicit expression for the number of M -words of length N having k different letters.

Random number generators. A random N -mapping is any function f with the integers 1 to N as both domain and range, where all N^N such functions

are taken with equal likelihood. For example, the following mapping is defined by the function $f(i) \equiv 1 + i^2 \bmod 9$:

index	1	2	3	4	5	6	7	8	9
mapping	2	5	1	8	8	1	5	2	1

One application of such functions is to model *random number generators*: subroutines that return sequences of numbers with properties as similar as possible to those of random sequences. The idea is to choose a function that is an N -mapping, then produce a (pseudo) random sequence by iterating $f(x)$ starting with an initial value called the *seed*. Given a seed u_0 , we get the sequence

$$\begin{aligned} & u_0 \\ & u_1 = f(u_0) \\ & u_2 = f(u_1) = f(f(u_0)) \\ & u_3 = f(u_2) = f(f(f(u_0))) \\ & \vdots \end{aligned}$$

For example, *linear congruential* random number generators are based on

$$f(x) = (ax + b) \bmod N,$$

and *quadratic* random number generators are based on

$$f(x) = (ax^2 + bx + c) \bmod N.$$

Quadratic random number generators are closely related to the *middle square* method, an old idea that dates back to von Neumann's time: Starting with a seed u_0 , repeatedly square the previously generated value and extract the middle digits. For example, using four-digit decimal numbers, the sequence generated from the seed $u_0 = 1234$ is $u_1 = 5227$ (since $1234^2 = 01522756$), $u_2 = 3215$ (since $5227^2 = 27321529$), $u_3 = 3362$ (since $3215^2 = 10336225$), and so forth.

It is easy to design a linear congruential generator so that it produces a permutation (that is, it goes through N different values before it repeats). A complete algebraic theory is available, for which we refer the reader to Knuth [24].

Quadratic random number generators are harder to analyze mathematically. However, Bach [2] has shown that, on average, quadratic functions have characteristics under iteration that are essentially equivalent to those of random mappings. Bach uses deep results from algebraic geometry; as we are going to see, properties of random mappings are somewhat easier to analyze given all the techniques developed so far in this book. There are N^3 quadratic trinomials modulo N , and we are just asserting that these are representative of the N^N random mappings, in the sense that the average values of certain quantities of interest are asymptotically the same. The situation is somewhat analogous to the situation for double hashing described earlier: in both cases, the practical method (quadratic generators, double hashing) is studied through asymptotic equivalence to the random model (random mappings, uniform hashing).

In other words, quadratic generators provide one motivation for the study of what might be called *random random number generators*, where a randomly chosen function is iterated to produce a source of random numbers. In this case, the result of the analysis is negative, since it shows that linear congruential generators might be preferred to quadratic generators (because they have longer cycles), but an interesting outcome of these ideas is the design and analysis of the *Pollard rho method* for integer factoring, which we discuss at the end of this section.

Exercise 9.47 Prove that every random mapping must have at least one cycle.

Exercise 9.48 Explore properties of the random mappings defined by $f(i) \equiv 1 + (i^2 + 1) \bmod N$ for $N = 100, 1000, 10,000$, and primes near these values.

Path length and connected components. Since the operation of applying a mapping to itself is well defined, we are naturally led to consider what happens if we do so successively. The sequence $f(k), f(f(k)), f(f(f(k))), \dots$ is well defined for every k in a mapping: what are its properties? It is easy to see that, since only N distinct values are possible, the sequence must ultimately repeat a value, at which point it becomes cyclic. For example, as shown in Figure 9.10, if we start at $x_0 = 3$ in the mapping defined by $f(x) = x^2 + 1 \bmod 99$, we have the ultimately cyclic sequence $3, 10, 2, 5, 26, 83, 59, 17, 92, 50, 26, \dots$. The sequence always has a cycle preceded by a “tail” of values leading to the cycle. In this case, the cycle is of length 6 and the tail of length 4. We are interested in knowing the statistical properties of both cycle and tail lengths for random mappings.

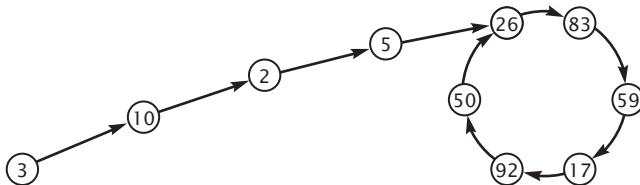


Figure 9.10 Tail and cycle iterating $f(x) = x^2 + 1 \bmod 99$ from $x_0 = 3$

Cycle and tail length depend on the starting point. Figure 9.11 is a graphical representation showing i connected to $f(i)$ for each i for three sample mappings. For example, in the top mapping, if we start at 7 we immediately get stuck in the cycle $7 \ 8 \ 7 \ \dots$, but if we start at 1 we encounter a two-element tail followed by a four-element cycle. This representation more clearly exposes the structure: every mapping decomposes into a set of connected components, also called *connected mappings*. Each component consists of the set of all points that wind up on the same cycle, with each point on the cycle attached to a tree of all points that enter the cycle at that point. From the point of view of each individual point, we have a tail-cycle as in Figure 9.10, but the structure as a whole is certainly more informative about the mapping.

Mappings generalize permutations: if we have the restriction that each element in the range must appear once, we have a set of cycles. All tail lengths are 0 for mappings that correspond to permutations. If all the *cycle* lengths in a mapping are 1, it corresponds to a forest. In general, we are naturally led to consider the idea of path length:

Definition The path length or *rho length* for an index k in a mapping f is the number of distinct integers obtained by iterating

$$f(k), f(f(k)), f(f(f(k))), f(f(f(f(k)))), \dots$$

The *cycle length* for an index k in a mapping f is the length of the cycle reached in the iteration, and the *tail length* for an index k in a mapping f is the rho length minus the cycle length or, equivalently, the number of steps taken to connect to the cycle.

The path length of an index is called the “rho length” because the shape of the tail plus the cycle is reminiscent of the Greek letter ρ (see Figure 9.10). Beyond these properties of the mapping as seen from a single point, we are also interested in global measures that involve all the points in the mapping.

Definition The rho length of a mapping f is the sum, over all k , of the rho length for k in f . The tree path length of a mapping f is the sum, over all k , of the tail length for k in f .

Thus, from Figure 9.11, it is easy to verify that $9 \ 6 \ 4 \ 2 \ 4 \ 3 \ 8 \ 7 \ 6$ has rho length 36 and tree path length 4; $3 \ 2 \ 3 \ 9 \ 4 \ 9 \ 9 \ 4 \ 4$ has rho length 20 and tree path length 5; and $1 \ 3 \ 1 \ 3 \ 3 \ 6 \ 4 \ 7 \ 7$ has rho length 27 and tree path length 18. In these definitions, tree path length does not include costs for any nodes on cycles, while rho length includes the whole length of

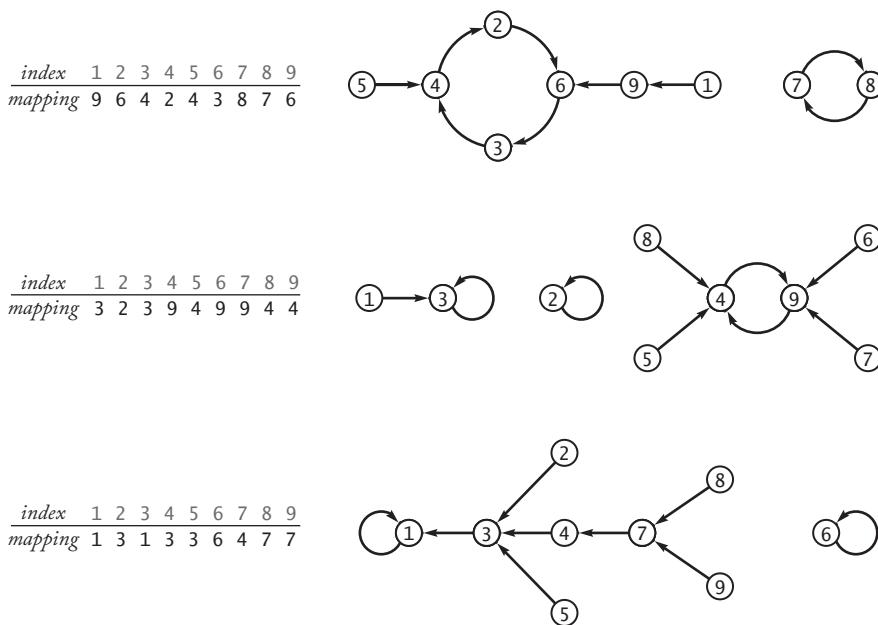


Figure 9.11 Tree-cycle representation of three random mappings

mapping	cycles	trees	rho length	longest cycle	longest path
123	3	0	3	1	1
113	2	1	4	1	2
121	2	1	4	1	2
122	2	1	4	1	2
133	2	1	4	1	2
223	2	1	4	1	2
323	2	1	4	1	2
112	1	1	6	1	3
131	1	1	6	1	3
221	1	1	6	1	3
322	1	1	6	1	3
233	1	1	6	1	3
313	1	1	6	1	3
111	1	2	5	1	2
222	1	2	5	1	2
333	1	2	5	1	2
213	2	0	5	2	2
321	2	0	5	2	2
132	2	0	5	2	2
211	1	1	7	2	3
212	1	1	7	2	3
232	1	1	7	2	3
311	1	1	7	2	3
331	1	1	7	2	3
332	1	1	7	2	3
231	1	0	9	3	3
312	1	0	9	3	3

Figure 9.12 Basic properties of all mappings of three elements

the cycle for each node in the structure. Both definitions give the standard notion of path length for mappings that are trees.

We are interested in knowing basic properties of the kinds of structures shown in Figure 9.11:

- How many cycles are there?
- How many points are on cycles, and how many on trees?
- What is the average cycle size?
- What is the average rho length?
- What is the average length of the longest cycle?
- What is the average length of the longest path to a cycle?
- What is the average length of the longest rho-path?

Figure 9.12 gives an exhaustive list of the basic measures for all 3-mappings, and Table 9.11 gives six random 9-mappings. On the right in Figure 9.12 are the seven different tree-cycle structures that arise in 3-mappings, reminding us that our tree-cycle representations of mappings are labelled and ordered combinatorial objects.

As with several other problems that we have seen in this chapter, some properties of random mappings can be analyzed with a straightforward probabilistic argument. For example, the average rho length of a random mapping is easily derived.

Theorem 9.9 (Rho length). The rho length of a random point in a random mapping is $\sim \sqrt{\pi N/2}$, on the average. The rho length of a random mapping is $\sim N\sqrt{\pi N/2}$, on the average.

mapping	occupancy	distribution	cycles	trees	rho length	longest cycle	longest path
323949944	012300003	5112	3	3	20	2	3
131336477	203101200	4221	2	1	27	1	5
517595744	100230201	4221	2	4	29	3	5
215681472	220111110	2520	1	2	42	2	8
213693481	212101011	2520	3	2	20	2	4
964243876	011212111	1620	2	2	36	4	6

Table 9.10 Basic properties of some random 9-mappings

Proof. Suppose that we start at x_0 . The probability that $f(x_0) \neq x_0$ is clearly $(N - 1)/N$. This is the same as the probability that the rho length is greater than or equal to 1. Similarly, the probability that the rho length is greater than or equal to 2 is the probability that the first two elements are different ($f(x_0) \neq x_0$) and the third is different from both of the first two ($f(f(x_0)) \neq x_0$ and $f(f(x_0)) \neq f(x_0)$), or $(N - 1)/N$ times $(N - 2)/N$. Continuing, we have

$$\Pr\{\text{rho length} \geq k\} = \frac{N-1}{N} \frac{N-2}{N} \cdots \frac{N-k}{N}.$$

Thus, the average rho length of a random point in a random mapping is the sum of these cumulative probabilities, which is precisely the Ramanujan Q -function, so the approximation of Theorem 4.8 provides our answer. The same argument holds for each of the N points in the mapping, so the expected rho length of the mapping is obtained by multiplying this by N . ■

This problem is equivalent to the birthday problem of §9.3, though the models of randomness are not formally identical.

Exercise 9.49 Show that the analysis of the rho length of a random point in a random mapping is equivalent to that for the birthday problem.

Generating functions. Many other properties of mappings depend more upon global structural interactions. They are best analyzed with generating functions. Mappings are sets of cycles of trees, so their generating functions are easily derived with the symbolic method. We proceed exactly as for counting “sets of cycles” when we introduced the symbolic method in Chapter 5, but with trees as the basic object.

We begin, from §6.14, with the EGF for Cayley trees:

$$C(z) = ze^{C(z)}.$$

As in Chapter 5, the EGF that enumerates cycles of trees (connected mappings) is

$$\sum_{k \geq 1} \frac{C(z)^k}{k} = \ln \frac{1}{1 - C(z)}$$

and the EGF that enumerates sets of connected mappings is

$$\exp\left(\ln \frac{1}{1 - C(z)}\right) = \frac{1}{1 - C(z)}.$$

The functional equations involve the implicitly defined Cayley function $C(z)$, and the Lagrange inversion theorem applies directly. For example, applying the theorem to the EGF just derived leads to the following computation:

$$\begin{aligned}[z^N] \frac{1}{1 - C(z)} &= \frac{1}{N} [u^{N-1}] \frac{1}{(1-u)^2} e^{Nu} \\ &= \sum_{0 \leq k \leq N} (N-k) \frac{N^{k-1}}{k!} = \sum_{0 \leq k \leq N} \frac{N^k}{k!} - \sum_{1 \leq k \leq N} \frac{N^{k-1}}{(k-1)!} \\ &= \frac{N^N}{N!}. \end{aligned}$$

This is a check on the fact that there are N^N mappings of size N .

Table 9.11 gives the EGFs for mappings, all derived using the following Lemma, which summarizes the application of the Lagrange inversion theorem to functions of the Cayley function.

Lemma For the Cayley function $C(z)$, we have

$$[z^N]g(C(z)) = \sum_{0 \leq k < N} (N-k)g_{N-k} \frac{N^{k-1}}{k!} \quad \text{when } g(z) \equiv \sum_{k \geq 0} g_k z^k.$$

Proof. Immediate from Theorem 6.11. ■

class	EGF	coefficient
trees	$C(z) = ze^{C(z)}$	$(N-1)![u^{N-1}]e^{Nu} = N^{N-1}$
cycles of trees	$\ln \frac{1}{1 - C(z)}$	$(N-1)![u^{N-1}] \frac{1}{1-u} e^{Nu} \sim N^N / \sqrt{\pi N}$
mappings	$\exp\left(\ln \frac{1}{1 - C(z)}\right)$	$(N-1)![u^{N-1}] \frac{1}{(1-u)^2} e^{Nu} = N^N$

Table 9.11 Exponential generating functions for mappings

Thus, the number of connected mappings of N nodes is given by

$$N![z^N] \ln \frac{1}{1 - C(z)} = N! \sum_{0 \leq k < N} (N - k) \frac{1}{N - k} \frac{N^{k-1}}{k!} = N^{N-1} Q(N).$$

The Ramanujan Q -function again makes an appearance.

The above results imply that the probability that a random mapping is a tree is exactly $1/N$ and the probability that a random mapping is a single connected component is asymptotically $\sqrt{\pi/(2N)}$. We can use BGFs in a similar manner to analyze other properties of random mappings.

Theorem 9.10 (Components and cycles). A random N -mapping has $\sim \frac{1}{2} \ln N$ components and $\sim \sqrt{\pi N}$ nodes on cycles, on the average.

Proof. By the earlier discussion, the BGF for the distribution of the number of components is given by

$$\exp\left(u \ln \frac{1}{1 - C(z)}\right) = \frac{1}{(1 - C(z))^u}.$$

The average number of components is then given by

$$\begin{aligned} \frac{1}{N^N} [z^N] \frac{\partial}{\partial u} \frac{1}{(1 - C(z))^u} \Big|_{u=1} &= \frac{1}{N^N} [z^N] \frac{1}{1 - C(z)} \ln \frac{1}{1 - C(z)} \\ &= \sum_{0 \leq k \leq N} (N - k) H_{N-k} \frac{N^{k-1}}{k!} \end{aligned}$$

by the lemma given previously. The stated asymptotic result is a straightforward calculation in the manner of the proof of Theorem 4.8.

The BGF for the number of nodes on cycles is

$$\exp\left(\ln \frac{1}{1 - uC(z)}\right) = \frac{1}{1 - uC(z)},$$

from which coefficients can be extracted exactly as shown earlier. ■

Average as seen from a random point

rho length	$\sqrt{\pi N/2}$
tail length	$\sqrt{\pi N/8}$
cycle length	$\sqrt{\pi N/8}$
tree size	$N/3$
component size	$2N/3$

Average number of

k -nodes	$\frac{Ne^{-1}}{k!}$
k -cycles	$\frac{1}{k}$
k -components	$\frac{e^{-k}}{k!} \{ \# k\text{-node connected mappings} \}$
k -trees	$(\sqrt{\pi N/2}) \frac{e^{-k}}{k!} \{ \# k\text{-node trees} \}$

Extremal parameters (expected number of nodes in)

longest tail	$\sqrt{2\pi N} \ln 2 \approx 1.74\sqrt{N}$
longest cycle	$\approx 0.78\sqrt{N}$
longest rho-path	$\approx 2.41\sqrt{N}$
largest tree	$\approx 0.48N$
largest component	$\approx 0.76N$

Table 9.12 Asymptotic properties of random mappings

Other properties. Various other properties can be handled similarly. Also, proceeding just as for permutations in Chapter 7 and for words earlier in this chapter, we find that the number of mappings having exactly k components has EGF $C(z)^k/k!$; the number of mappings having at most k components has EGF $1 + C(z) + C(z)^2/2! + \dots + C(z)^k/k!$; and so forth. These can be used to give explicit expressions for extremal parameters, as we have done several times before. Asymptotic methods for estimating these quantities are discussed in detail by Flajolet and Odlyzko [11] and by Kolchin [28] (see also [14]). The results given in [11] are summarized in Table 9.13. The various constants in the extremal parameters can be expressed explicitly, though with rather complicated definite integrals. It is interesting to note that cycle length plus tail length equals rho length for the averages, but not for the extremal parameters. This means that the tallest tree is not attached to the longest cycle for a significant number of mappings.

Exercise 9.50 Which N -mappings have maximal and minimal rho length? Tree path length?

Exercise 9.51 Write a program to find the rho length and tree path length of a random mapping. Generate 1000 random mappings for N as large as you can and compute the average number of cycles, rho length, and tree path length.

Exercise 9.52 Write a program to find the rho length and tree path length of a random mapping *without* using any extra storage.

Exercise 9.53 A mapping with no repeated integers is a permutation. Give an efficient algorithm for determining whether a mapping is a tree.

Exercise 9.54 Compute the average size of the largest component in a random N -mapping, for all $N < 10$.

Exercise 9.55 Prove Theorem 9.9 using BGFs.

Exercise 9.56 What is the average number of different integers in the image when a random mapping is iterated *twice*?

Exercise 9.57 Consider the N^N tree-cycle structures that correspond to all the N -mappings. How many of these are different when considered as unlabelled, unordered objects, for $N \leq 7$? (These are called *random mapping patterns*.)

Exercise 9.58 Describe the graph structure of *partial* mappings, where the image of a point may be undefined. Set up the corresponding EGF equations and check that the number of partial mappings of size N is $(N + 1)^N$.

Exercise 9.59 Analyze “path length” in sequences of $2N$ random integers in the range 1 to N .

Exercise 9.60 Generate 100 random mappings of size 10, 100, and 1000 and empirically verify the statistics given in Table 9.13.

9.8 Integer Factorization and Mappings. In this section we will examine *Pollard's rho method*, an efficient algorithm for factoring integers (in the “intermediate” range of 10 to 30 decimal digits) that relies on structural and probabilistic properties of mappings. The method is based on exploiting the following two facts:

- A point on some cycle can be found quickly (in time proportional to the rho length of a starting point), using $O(1)$ memory.
- A random point on a random mapping has rho length $O(\sqrt{N})$ on the average.

First, we will consider the cycle detection problem, and then we look at the Pollard rho method itself.

Cycle detection. The naive method for finding a point on a cycle of a mapping is to iterate the mapping, storing all the function values in a search structure and looking up each new value to see if some value already in the structure has been reached again. This algorithm is impractical for large mappings because one cannot afford the space to save all the function values.

Program 9.3 gives a method due to Floyd (cf. Knuth [24]) for finding cycle points in an arbitrary mapping using only constant space without sacrificing time. In the program, we can view the point **a** as moving along the rho-graph (see Figure 9.9) at speed 1 and the point **b** as moving along the rho-graph at speed 2. The algorithm depends on the fact that the two points must collide at some point once they are on the cycle. For example, suppose

```
int a = x, b = f(x), t = 0;
while (a != b)
{ a = f(a); b = f(f(b)); t++; }
```

Program 9.3 Floyd method for cycle detection

that the method is used for the mapping of Figure 9.9 with the starting point 10. Then the cycle is detected in 7 steps, as shown by the following trace of the values taken on by a and b :

a	10	2	5	26	83	59	17
b	10	5	83	17	50	83	17

Theorem 9.11 (Cycle detection). Given a mapping and a starting point x , the Floyd method finds a cycle point on the mapping using constant space and time proportional to the rho length of x .

Proof. Let λ be the tail length of x , μ the cycle length, and $\rho = \lambda + \mu$ the rho length. After λ steps of the loop, point a reaches the cycle, while point b is already on the cycle. We now have a race on the cycle, with a speed differential of 1. After at most μ steps, b will catch up with a .

Let t be the value of the variable t when the algorithm terminates. We also must have

$$t \leq \rho \leq 2t.$$

The inequality on the left holds because t is the position of point a , and the algorithm terminates before a starts on a second trip around the cycle. The inequality on the right holds because $2t$ is the position of point b , and the algorithm terminates after b has been around the cycle at least once. Thus, the algorithm gives not only a point on the cycle (the value of the variables a and b on termination), but also an estimate of the rho length of the starting point to within a factor of 2. ■

By saving some more function values, it is possible to virtually eliminate the extra factor of 2 in the time taken by the algorithm (and the uncertainty in the estimate of the rho length), while still using a reasonable amount of space. This issue is studied in detail by Sedgewick, Szymanksi, and Yao [34].

Exercise 9.61 Use Floyd's method to test the random number generators on your machine for short cycles.

Exercise 9.62 Use Floyd's algorithm to test the middle square random number generator.

Exercise 9.63 Use Floyd's method to estimate the rho length associated with various starting values, c , and N for the function $f(x) = (x^2 + c) \bmod N$.

Pollard's rho method. The rho method is a randomized algorithm that factors integers with high probability. An implementation is given in Program 9.4, with the caveat that it assumes that arithmetic operations on very large integers are available. The method is based on choosing a value c at random, then iterating the quadratic function $f(x) = (x^2 + c) \bmod N$ from a randomly chosen starting point until a cyclic point is found.

For simplicity, assume that $N = pq$ where p and q are primes to be found by the algorithm. By the Chinese remainder theorem, any integer y modulo N is determined by its values mod p and mod q . In particular, the function f is determined by the pair

$$f_p(x) = x^2 + c \bmod p \quad \text{and} \quad f_q(x) = x^2 + c \bmod q.$$

If the cycle detection algorithm were applied to f_p starting at an initial value x , then a cycle would be detected after t_p steps where t_p is at most twice the rho length of x (modulo p). Similarly, a cycle (modulo q) for t_q would be detected after t_q steps. Thus, if $t_p \neq t_q$ (which should occur with a very high probability for large integers), we find after $\min(t_p, t_q)$ steps that the values a and b of the variables a and b in the algorithm satisfy

$$\begin{aligned} a &\equiv b \pmod{p} \quad \text{and} \quad a \not\equiv b \pmod{q}, & \text{if } t_p < t_q \\ a &\not\equiv b \pmod{p} \quad \text{and} \quad a \equiv b \pmod{q}, & \text{if } t_p > t_q. \end{aligned}$$

In either case, the greatest common divisor of $a - b$ and N is a nontrivial divisor of N .

```

int a = (int) (Math.random()*N), b = a;
int c = (int) (Math.random()*N), d = 1;
while (d == 1)
{
    a = (a*a + c) % N;
    b = (b*b + c)*(b*b + c) + c % N;
    d = gcd((a - b) % N, N);
}
// d is a factor of N

```

Program 9.4 Pollard's rho method for factoring

Connection with random mappings. Assume that quadratic functions of the form $x^2 + c \bmod N$ have path length properties that are asymptotically equivalent to path lengths in random mappings. This heuristic assumption asserts that properties of the N quadratic mappings (there are N possible choices for c) are similar to properties of the N^N random mappings. In other words, quadratic functions are assumed to be a “representative sample” of random mappings. This assumption is quite plausible and has been extensively validated by simulations, but it has only been partially proven [2]. Nevertheless, it leads to a useful approximate analysis for Pollard’s method.

The earlier discussion revealed that the number of steps taken by the algorithm is $\min(t_p, t_q)$, where t_p and t_q are the rho lengths of f_p and f_q , respectively. By Theorem 9.9, the rho length of a random point on a random N -mapping is $O(\sqrt{N})$, so under the assumption discussed in the previous paragraph we should expect the algorithm to terminate in $O(\min(\sqrt{p}, \sqrt{q}))$ steps, which is $O(N^{1/4})$. This argument obviously generalizes to the situation where N has more than two factors.

Theorem 9.12 (Pollard’s rho method). Under the heuristic assumption that path length in quadratic functions is asymptotic to path length in random mappings on average, Pollard’s rho method factors a composite integer number N in $O(\sqrt{p})$ steps, on the average, where p is the smallest prime factor of N . In particular, the method factors N in $O(N^{1/4})$ steps, on the average.

N	number of steps
13·23	4
127·331	10
1237·4327	21
12347·54323	132
123457·654323	243
1234577·7654337	1478
12345701·87654337	3939
123456791·987654323	11225
1234567907·10987654367	23932

Table 9.13 Sample applications of Pollard’s algorithm ($c = 1$)

Proof. See the earlier discussion. The global bound follows from the fact that $p \leq \sqrt{N}$. ■

In 1980, Brent [31] used Pollard's method to factor the eighth Fermat number for the first time. Brent discovered that

$$F_8 = 2^{2^8} + 1 \approx 1.11579 \cdot 10^{77}$$

has the prime factor

$$1238926361552897$$

(also see Knuth [24]). The fact that the approximate analysis currently rests on a partly unproven assumption does not detract from the utility of the algorithm. Indeed, knowledge of properties of random mappings gives confidence that the method should factor efficiently, and it does.

Table 9.13 shows the number of steps used by Pollard's method to factor numbers of the form $N = pq$ where p and q are chosen to be primes near numbers of the form $1234 \cdots$ and $\cdots 4321$, respectively. Though c and the starting points are supposed to be chosen at random, the value $c = 1$ and starting points $a = b = 1$ work sufficiently well for this application (and make it easy to reproduce the results). From the table, we can see that the cost rises roughly by a factor of 3 when N rises by a factor of about 100, in excellent agreement with Theorem 9.12. The method factors the last number on the list, which is $\approx 1.35 \cdot 10^{19}$, in fewer than 24,000 steps, while exhaustive trials would have required about 10^9 operations.

WORDS and mappings relate directly to classical problems in combinatorics and “classical” problems in the analysis of algorithms. Many of the methods and results that we have discussed are well known in mathematics (Bernoulli trials, occupancy problems) and are widely applicable outside the analysis of algorithms. They are directly relevant to modern applications such as predicting the performance of hashing algorithms, and detailed study of problems in this new domain leads to new problems of independent interest.

Hashing algorithms were among the first to be analyzed mathematically, and they are still of paramount practical importance. New types of applications and changes of fundamental characteristics in hardware and software contribute to the continued relevance of the techniques and results about hashing algorithms presented here and in the literature.

The analysis of random mappings succinctly summarizes our general approach to the analysis of algorithms. We develop functional equations on generating functions corresponding to the underlying combinatorial structure, then use analytic tools to extract coefficients. The symbolic method is particularly effective for the former in this case, and the Lagrange inversion theorem is an important tool for the latter.

Mappings are characterized by the fact that each element maps to precisely one other element. In the graphical representation, this means that there are precisely N edges and that each element has exactly one edge pointing “from” it, though many elements might point “to” a particular element. The next generalization is to *graphs*, where this restriction is removed and each element can point “to” any number of other elements. Graphs are more complicated than mappings or any of the other combinatorial structures that we have examined in this book because they are more difficult to decompose into simpler substructures—normally our basis for analysis by solving recurrences or exploiting structural decomposition to develop relationships among generating functions.

Random graphs have a wealth of interesting properties. A number of books have been written on the subject, and it is an active research area. (See, for example, Bollobás [3] for a survey of the field.) Analysis of random graphs centers on the “probabilistic method,” where the focus is not on exactly enumerating properties of *all* graphs, but rather on developing suitable inequalities that relate complex parameters to tractable ones. There are many important fundamental algorithms for processing graphs, and there are many examples in the literature of the analysis of such algorithms. Different models of randomness are appropriate for different applications, making analysis along the lines we have been studying appropriate in many cases. Learning properties of random graphs is a fruitful area of study in the analysis of algorithms.

Random mappings are an appropriate topic on which to close for many reasons. They generalize basic and widely-used structures (permutations and trees) that have occupied so much of our attention in this book; they are of direct practical interest in the use of random number generators and random sequences; their analysis illustrates the power, simplicity, and utility of the symbolic enumeration method and other tools that we have been using; and they represent the first step toward studying random graphs (for example, see Janson, Knuth, Luczak, and Pittel [22]), which are fundamental and widely

applicable structures. It is our hope that the basic tools and techniques that we have covered in this book will provide readers with the interest and expertise to attack these and other problems in the analysis of algorithms that arise in the future.

References

1. M. ABRAMOWITZ AND I. STEGUN. *Handbook of Mathematical Functions*, Dover, New York, 1970.
2. E. BACH. “Toward a theory of Pollard’s rho method,” *Information and Computation* **30**, 1989, 139–155.
3. B. BOLLOBÁS. *Random Graphs*, Academic Press, London, 1985.
4. R. P. BRENT AND J. M. POLLARD. “Factorization of the eighth Fermat number,” *Mathematics of Computation* **36**, 1981, 627–630.
5. B. CHAR, K. GEDDES, G. GONNET, B. LEONG, M. MONAGAN, AND S. WATT. *Maple V Library Reference Manual*, Springer-Verlag, New York, 1991.
6. L. COMTET. *Advanced Combinatorics*, Reidel, Dordrecht, 1974.
7. T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, AND C. STEIN. *Introduction to Algorithms*, MIT Press, New York, 3rd edition, 2009.
8. F. N. DAVID AND D. E. BARTON. *Combinatorial Chance*, Charles Griffin, London, 1962.
9. W. FELLER. *An Introduction to Probability Theory and Its Applications*, John Wiley, New York, 1957.
10. P. FLAJOLET, D. GARDY, AND L. THIMONIER. “Birthday paradox, coupon collectors, caching algorithms and self-organizing search,” *Discrete Applied Mathematics* **39**, 1992, 207–229.
11. P. FLAJOLET AND A. M. ODLYZKO. “Random mapping statistics,” in *Advances in Cryptology*, J.-J. Quisquater and J. Vandewalle, eds., Lecture Notes in Computer Science No. 434, Springer-Verlag, New York, 1990, 329–354.
12. P. FLAJOLET, P. POBLETE, AND A. VIOLA. “On the analysis of linear probing hashing,” *Algorithmica* **22**, 1988, 490–515.
13. P. FLAJOLET AND R. SEDGEWICK. *Analytic Combinatorics*, Cambridge University Press, 2009.
14. D. FOATA AND J. RIORDAN. “Mappings of acyclic and parking functions,” *Aequationes Mathematicae* **10**, 1974, 10–22.
15. G. H. GONNET. “Expected length of the longest probe sequence in hash code searching,” *Journal of the ACM* **28**, 1981, 289–309.

16. G. H. GONNET AND R. BAEZA-YATES. *Handbook of Algorithms and Data Structures in Pascal and C*, 2nd edition, Addison-Wesley, Reading, MA, 1991.
17. I. GOULDEN AND D. JACKSON. *Combinatorial Enumeration*, John Wiley, New York, 1983.
18. R. L. GRAHAM, D. E. KNUTH, AND O. PATASHNIK. *Concrete Mathematics*, 1st edition, Addison-Wesley, Reading, MA, 1989. Second edition, 1994.
19. L. GUIBAS AND E. SZEMEREDI. “The analysis of double hashing,” *Journal of Computer and Systems Sciences* **16**, 1978, 226–274.
20. S. JANSON. “Individual displacements for linear probing hashing with different insertion policies,” *ACM Transactions on Algorithms* **1**, 2005, 177–213.
21. S. JANSON, D. E. KNUTH, T. LUCZAK, AND B. PITTEL. “The birth of the giant component,” *Random Structures and Algorithms* **4**, 1993, 233–358.
22. D. E. KNUTH. *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, 1st edition, Addison-Wesley, Reading, MA, 1968. Third edition, 1997.
23. D. E. KNUTH. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*, 1st edition, Addison-Wesley, Reading, MA, 1969. Third edition, 1997.
24. D. E. KNUTH. *The Art of Computer Programming. Volume 3: Sorting and Searching*, 1st edition, Addison-Wesley, Reading, MA, 1973. Second edition, 1998.
25. D. E. KNUTH. *The Art of Computer Programming. Volume 4: Combinatorial Algorithms, Part I*, Addison-Wesley, Boston, 2011.
26. D. E. KNUTH. “Linear probing and graphs,” *Algorithmica* **22**, 1988, 561–568.
27. V. F. KOLCHIN. *Random Mappings*, Optimization Software, New York, 1986.
28. V. F. KOLCHIN, B. A. SEVASTYANOV, AND V. P. CHISTYAKOV. *Random Allocations*, John Wiley, New York, 1978.
29. G. LUEKER AND M. MOLODOWITCH. “More analysis of double hashing,” in *Proceedings 20th Annual ACM Symposium on Theory of Computing*, 1988, 354–359.

30. J. M. POLLARD. “A Monte Carlo method for factorization,” *BIT* **15**, 1975, 331–334.
31. R. SEDGEWICK. *Algorithms (3rd edition) in Java: Parts 1–4: Fundamentals, Data Structures, Sorting, and Searching*, Addison-Wesley, Boston, 2002.
32. R. SEDGEWICK AND P. FLAJOLET. *An Introduction to the Analysis of Algorithms*, Addison-Wesley, Reading, MA, 1996.
33. R. SEDGEWICK, T. SZYMANSKI, AND A. YAO. “The complexity of finding cycles in periodic functions,” *SIAM Journal on Computing* **11**, 1982, 376–390.
34. R. SEDGEWICK AND K. WAYNE. *Algorithms*, 4th edition, Addison-Wesley, Boston, 2011.
35. 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>.
36. A. VIOLA. “Exact distribution of individual displacements in linear probing hashing,” *ACM Transactions on Algorithms* **1**, 2005, 214–242.
37. J. S. VITTER AND W. CHEN. *Design and Analysis of Coalesced Hashing*, Oxford University Press, New York, 1987.
38. 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.

This page intentionally left blank

LIST OF THEOREMS

- 1.1 Mergesort compares, 9
- 1.2 Complexity of sorting, 11
- 1.3 Quicksort analysis, 21

- 2.1 First-order linear recurrences, 49
- 2.2 Linear recurrences with constant coefficients, 54
- 2.3 Binary search, 71
- 2.4 Mergesort, 73
- 2.5 Divide-and-conquer functions, 80
- 2.6 Divide-and-conquer sequences, 83

- 3.1 OGF operations, 93
- 3.2 EGF operations, 97
- 3.3 OGFs for linear recurrences, 102
- 3.4 Quicksort OGF, 108
- 3.5 Median-of-three Quicksort, 119
- 3.6 OGF for binary trees, 123
- 3.10 Mean and variance from PGFs, 127
- 3.11 BGFs and average costs, 133
- 3.12 Quicksort variance, 136

- 4.1 Asymptotics for linear recurrences, 155
- 4.2 Euler-Maclaurin summation formula, first form, 178
- 4.3 Euler-Maclaurin summation formula, second form, 182
- 4.4 Ramanujan Q -distribution, 186
- 4.5 Ramanujan R -distribution, 189
- 4.6 Normal approximation, 194
- 4.7 Poisson approximation, 198
- 4.8 Ramanujan Q -function, 202
- 4.9 Catalan sums, 207
- 4.10 Trie sum, 210

- 5.1 Symbolic method for unlabelled class OGFs, 223
- 5.2 Symbolic method for labelled class EGFs, 231
- 5.3 Symbolic method for unlabelled class OBGFs, 239
- 5.4 Symbolic method for labelled class EBGFs, 240
- 5.5 Radius-of-convergence transfer theorem, 247

- 6.1 Enumeration of binary trees, 258
- 6.2 Enumeration of forests and trees, 261
- 6.3 Path length in binary trees, 286
- 6.4 Path length in general trees, 290

- 6.5 Construction cost of BSTs, 292
- 6.6 Search costs in BSTs, 294
- 6.7 Additive parameters in random trees, 296
- 6.8 Binary tree height, 301
- 6.9 Catalan tree height GF, 302
- 6.10 Binary search tree height, 306
- 6.11 Lagrange inversion theorem, 310
- 6.12 Enumeration of rooted unordered trees, 323
- 6.13 Enumeration of unordered labelled trees, 328
- 6.14 Enumeration of t -ary trees, 331
- 6.15 Enumeration of t -restricted trees, 333

- 7.1 Frequency of HOTs and BSTs, 362
- 7.2 Maximal cycle lengths, 365
- 7.3 Minimal cycle lengths, 366
- 7.4 Involutions, 367
- 7.5 Eulerian numbers, 374
- 7.6 Increasing subsequences, 378
- 7.7 Local properties of permutations and nodes in HOTs/BSTs, 380
- 7.8 Inversion distribution, 384
- 7.9 Inversions in 2-ordered permutations, 389
- 7.10 Left-to-right minima distribution, 392
- 7.11 Selection sort, 396
- 7.12 Cycle distribution, 400
- 7.13 Singleton cycle distribution, 402

- 7.14 Maximum inversion table entry, 405

- 8.1 Pattern occurrence enumeration, 415
- 8.2 Runs of 0s, 419
- 8.3 Pattern autocorrelation, 425
- 8.4 OGFs for regular expressions, 429
- 8.5 KMP string matching, 435
- 8.6 OGFs for context-free grammars, 438
- 8.7 Ballot problem, 442
- 8.8 Trie path length and size, 455
- 8.9 Leader election, 460

- 9.1 Birthday problem, 483
- 9.2 Coupon collector problem, 486
- 9.3 Maximal occupancy, 492
- 9.4 Minimal occupancy, 494
- 9.5 Occupancy distribution, 499
- 9.6 Hashing with separate chaining, 504
- 9.7 Uniform hashing, 507
- 9.8 Hashing with linear probing, 511
- 9.9 Rho length, 522
- 9.10 Components and cycles, 525
- 9.11 Cycle detection, 529
- 9.12 Pollard rho method, 531

LIST OF TABLES

- | | |
|---|---|
| 1.1 Average number of compares used by quicksort, 29

2.1 Classification of recurrences, 44
2.2 Elementary discrete sums, 48
2.3 Ruler and population count functions, 76
2.4 Binary divide-and-conquer recurrences and solutions, 78 | 4.6 Asymptotic expansions for special numbers, 168

4.7 Ramanujan Q -distribution, 188
4.8 Ramanujan R -distribution, 191
4.9 Binomial distribution, 194
4.10 Poisson distribution, 202
4.11 Ramanujan P -, Q -, and R -functions, 206
4.12 Catalan sums, 210 |
| 3.1 Elementary OGFs, 93
3.2 Operations on OGFs, 94
3.3 Elementary EGFs, 98
3.4 Operations on EGFs, 100
3.5 Calculating moments from a BGF, 137
3.6 Classic “special” generating functions, 141 | 5.1 Counting sequences and OGFs for classes in Figure 5.1, 223
5.2 Familiar labelled combinatorial classes, 231
5.3 Analytic combinatorics examples in this chapter, 254 |
| 4.1 Asymptotic estimates for quicksort comparison counts, 161

4.2 Asymptotic expansions derived from Taylor series, 162
4.3 Accuracy of Stirling’s formula for $N!$, 164
4.4 Asymptotic estimates of the harmonic numbers, 165
4.5 Absolute error in Stirling’s formula for $\ln N!$, 166 | 6.1 Expected path length of trees, 310
6.2 Expected height of trees, 311
6.3 Enumeration of unlabelled trees, 322
6.4 Enumeration of labelled trees, 328
6.5 Tree enumeration generating functions, 331
6.6 Generating functions for other types of trees, 338
6.7 Analytic combinatorics examples in this chapter, 341 |

- 7.1 Basic properties of some random permutations of nine elements, 352
- 7.2 Basic properties of all permutations of four elements, 353
- 7.3 Cumulative counts and averages for properties of permutations, 354
- 7.4 Enumeration of permutations with cycle length restrictions, 366
- 7.5 EGFs for permutations with cycle length restrictions, 370
- 7.6 Distribution of runs in permutations (Eulerian numbers), 377
- 7.7 Analytic results for properties of permutations (average case), 383
- 7.8 Three 2-ordered permutations, with inversion tables, 390
- 7.9 Distribution of left-to-right minima and cycles (Stirling numbers of the first kind), 398
- 7.10 Distribution of maximum inversion table entry, 408
- 8.1 Cost of searching for 4-bit patterns (basic method), 418
- 8.2 Position of first runs of 0s on sample bitstrings, 422
- 8.3 Enumerating bitstrings with no runs of P 0s, 424
- 8.4 Autocorrelation of 101001010, 428
- 8.5 Generating functions and wait times for 4-bit patterns, 430
- 8.6 REs and OGFs for gambler's ruin sequences, 435
- 8.7 Example of KMP state transition table, 439
- 8.8 Periodic term in trie path length, 461
- 9.1 Two ways to view the same set of objects, 478
- 9.2 Occupancy distribution and properties of 3-words of length 4, 482
- 9.3 Occupancy parameters for balls in three urns, 483
- 9.4 Occupancy parameters for balls in eight urns, 484
- 9.5 Enumeration of 3-words with letter frequency restrictions, 495
- 9.6 EGFs for words with letter frequency restrictions, 499
- 9.7 Occupancy distribution for $M = 3$, 501
- 9.8 Occupancy distribution examples, 502
- 9.9 Analytic results for hashing methods, 517
- 9.10 Basic properties of some random 9-mappings, 526
- 9.11 Exponential generating functions for mappings, 528
- 9.12 Asymptotic properties of random mappings, 530
- 9.13 Sample applications of Pollard's algorithm ($c = 1$), 535

LIST OF FIGURES

- 1.1 Quicksort compare counts: empirical and analytic, 23
- 1.2 Distributions for compares in quicksort, 30
- 1.3 Distributions for compares in quicksort (scaled and centered), 31
- 1.4 Empirical histogram for quicksort compare counts, 32

- 2.1 Solutions to binary divide-and-conquer recurrences, 70
- 2.2 Periodic terms in binary divide-and-conquer recurrences, 71
- 2.3 $\lg N$ (top); $\lfloor \lg N \rfloor$ (middle); $\{\lg N\}$ (bottom), 74
- 2.4 Composition of the periodic function $\theta(1 - \{\lg N\})$, 75
- 2.5 Periodic and fractal terms in bit counting, 77
- 2.6 Divide-and-conquer for $\beta = 3$ and $\alpha = 2, 3, 4$, 83

- 3.1 All binary trees with 1, 2, 3, 4, and 5 external nodes, 124

- 4.1 Ramanujan Q -distribution, 189
- 4.2 Ramanujan R -distribution, 193
- 4.3 Binomial distribution, 195

- 4.4 Binomial distribution ($p = 1/5$), 199
- 4.5 Binomial distributions tending to Poisson distributions, 201
- 4.6 Laplace method, 203
- 4.7 Asymptotic behavior of the terms in $\sum_{j \geq 0} (1 - e^{-N/2^j})$, 212

- 5.1 An overview of analytic combinatorics, 219
- 5.2 Basic unlabelled classes, 222
- 5.3 A simple combinatorial construction (unlabelled), 224
- 5.4 Bitstrings with no 00, 227
- 5.5 Basic labelled classes, 230
- 5.6 Star product example, 232
- 5.7 $\mathcal{Z} \star \mathcal{P}_3 = \mathcal{P}_4$, 235
- 5.8 Sets of cycles, 236
- 5.9 A permutation is a set of cycles., 237
- 5.10 Derangements, 238
- 5.11 Enumerating generalized derangements via analytic combinatorics, 251

- 6.1 Three binary trees, 259
- 6.2 Enumerating binary trees via analytic combinatorics, 260
- 6.3 A three-tree forest, 261
- 6.4 Forests and trees, 262

- 6.5 Rotation correspondence between trees and binary trees, 264
- 6.6 Lattice-path representations of binary trees in, 269
- 6.7 Binary tree corresponding to a triangulated N -gon, 270
- 6.8 Binary trees, trees, parentheses, triangulations, and ruin sequences, 271
- 6.9 Path length and height in a forest and in a binary tree, 273
- 6.10 A random forest and a random binary tree, 276
- 6.11 Binary tree representation of an arithmetic expression, 279
- 6.12 Three binary search trees, 281
- 6.13 Permutations and binary search trees, 284
- 6.14 Permutations associated with BSTs, 285
- 6.15 Catalan distribution (subtree sizes in random binary trees), 288
- 6.16 A binary search tree built from 237 randomly ordered keys, 295
- 6.17 Rooted unordered trees , 315
- 6.18 Representations of a rooted (unordered) forest, 317
- 6.19 Unrooted unordered (free) trees, 318
- 6.20 A large graph and two of its spanning trees, 320
- 6.21 Summary of tree nomenclature, 321
- 6.22 Trees with five nodes (ordered, unordered, and unrooted), 323
- 6.23 Labelled trees with three nodes (ordered, unordered, and unrooted), 328
- 6.24 Cayley (labelled rooted unordered) trees, 330
- 6.25 Examples of various other types of trees, 332
- 6.26 AVL distribution (subtree sizes in random AVL trees), 339

- 7.1 Two-line, cycle, and Foata representations of a permutation, 349
- 7.2 Anatomy of a permutation, 351
- 7.3 Lattice representation of a permutation and its inverse, 360
- 7.4 Binary search tree corresponding to a permutation, 361
- 7.5 Heap-ordered tree corresponding to a permutation, 362
- 7.6 Two combinatorial constructions for permutations, 374
- 7.7 Distribution of runs in permutations (Eulerian numbers), 377
- 7.8 Insertion sort and inversions, 385
- 7.9 Distribution of inversions, 387
- 7.10 Lattice paths for 2-ordered permutations, 391
- 7.11 Selection sort and left-to-right minima, 394
- 7.12 Distribution of left-to-right minima and cycles (Stirling numbers of the first kind), 398
- 7.13 Bubble sort (permutation and associated inversion table), 407

- 7.14 Distribution of maximum inversion table entry, 408
- 8.1 Distribution of longest run of 0s in a random bitstring, 427
- 8.2 Gambler's ruin sequences, 436
- 8.3 Knuth-Morris-Pratt FSA for 10100110, 438
- 8.4 Three tries, each representing 10 bitstrings, 448
- 8.5 Three tries, representing 7, 2, and 10 bitstrings, respectively, 450
- 8.6 Tries with 1, 2, 3, and 4 external nodes, 451
- 8.7 Bitstring sets for tries with five external nodes (none void), 452
- 8.8 Aho-Corasick FSA for 000, 011, and 1010, 456
- 8.9 Distributed leader election, 457
- 8.10 Periodic fluctuation in trie path length, 462
- 8.11 Distributions for path length in tries, 463

- 9.1 A 10-word of length 20 (20 balls thrown into 10 urns), 477
- 9.2 2-words and 3-words with small numbers of characters, 479
- 9.3 Examples of balls-and-urns experiments, 481
- 9.4 Probability that N people do not all have different birthdays, 486
- 9.5 Occupancy distributions for small M , 506
- 9.6 Occupancy distributions for large M , 507
- 9.7 Percentage of empty urns as a function of load factor N/M , 508
- 9.8 Hashing with linear probing, 510
- 9.9 Image cardinality of random mappings, 520
- 9.10 Tail and cycle iterating $f(x) = x^2 + 1 \bmod 99$, 523
- 9.11 Tree-cycle representation of three random mappings, 524
- 9.12 Basic properties of all mappings of three elements , 525

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 algo-
 rithm
 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
 $([n]_k)$, 140
 asymptotics, 168
 counting cycles, 402–403
 counting minima/maxima, 396–398
 Stirling numbers of the second kind
 $\{\{n\}_k\}$, 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 also* 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



Addison
Wesley

Essential Information about Algorithms and Data Structures



ISBN-13: 978-0-321-57351-3

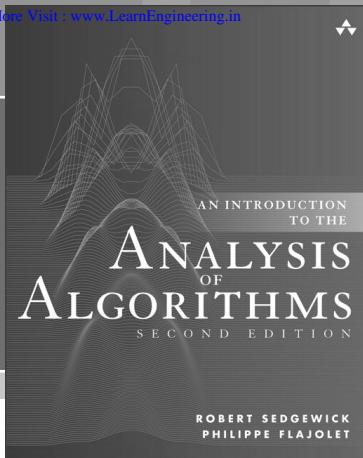
Robert Sedgewick's *Algorithms* has long been the definitive practical guide to using algorithms: the book that professionals and students in programming, science, and engineering rely on to solve real-world problems. Now, with the fourth edition, Sedgewick and Wayne have thoroughly updated this classic to reflect today's latest, most powerful algorithms. The authors bring together an indispensable body of knowledge for improving computer performance and solving larger problems.



Available in print and eBook formats



For more information and sample content visit
informit.com/title/9780321573513



FREE Online Edition

Your purchase of ***An Introduction to the Analysis of Algorithms, Second Edition***, includes access to a free online edition for 45 days through the Safari Books Online subscription service. Nearly every Addison-Wesley Professional book is available online through Safari Books Online, along with thousands of books and videos from publishers such as Cisco Press, Exam Cram, IBM Press, O'Reilly Media, Prentice Hall, Que, Sams, and VMware Press.

Safari Books Online is a digital library providing searchable, on-demand access to thousands of technology, digital media, and professional development books and videos from leading publishers. With one monthly or yearly subscription price, you get unlimited access to learning tools and information on topics including mobile app and software development, tips and tricks on using your favorite gadgets, networking, project management, graphic design, and much more.

**Activate your FREE Online Edition at
informat.com/safarifree**

STEP 1: Enter the coupon code: DLTNVH.

STEP 2: New Safari users, complete the brief registration form.
Safari subscribers, just log in.

If you have difficulty registering on Safari or accessing the online edition,
please e-mail customer-service@safaribooksonline.com



Addison
Wesley



Adobe Press



Cisco Press



Microsoft
Press



O'REILLY



SAMS
Redbooks
www.it-ebooks.info

vmware PRESS

