International Lecture Series in Computer Science

The International Chair in Computer Science was created by IBM Belgium in cooperation with the National Foundation for Scientific Research

Probability Theory and Computer Science

Edited by
G. Louchard
& G. Latouche

Academic Press

Probability Theory and Computer Science

Edited by

G. Louchard and G. Latouche

Laboratoire d'Informatique Théorique
Université Libre de Bruxelles, Belgium

1983

ACADEMIC PRESS

A Subsidiary of Harcourt Brace Jovanovich, Publishers

London · New York
Paris · San Diego · San Francisco · Sao Paũlo · Sydney
Tokyo · Toronto

ACADEMIC PRESS INC. (LONDON) LTD.
24–28 Oval Road
London NW1 7DX

U.S. Edition published by
ACADEMIC PRESS INC
111 Fifth Avenue
New York, New York 10003

British Library Cataloguing in Publication Data

Probability theory and computer science.—
(International lecture series in computer science)
1. Electronic data processing—Probabilities
I. Louchard, G. II. Latouche, G.
III. Series
519.2′028′54 QA273

ISBN 0-12-455820-8

Typeset by Preface Ltd, Salisbury, Wilts.
Printed in Great Britain by Thomson Litho Ltd., East Kilbride, Scotland

Contributors

Donald P. Gaver Naval Postgraduate School, Operations Research Department, Monterey, California 93940, U.S.A.

Hisashi Kobayashi IBM Japan Science Institute, 5–1 Azabudai 3-chome, Monato-ku, Tokyo 106, Japan.

Robert Sedgewick Brown University, Department of Computer Science, Providence, Rhode Island 02912, U.S.A.

Foreword

The present volume gathers together the writings on which Professors Donald P. Gaver, Hisashi Kobayashi and Robert Sedgewick based the series of lectures they presented at the "Université Libre de Bruxelles" in the period from the Fall of 1980 to the Summer of 1981. These lectures took place within the framework of an International Professorship in Computer Science funded by IBM Belgium, and organized by the Belgian National Scientific Research Foundation (FNRS).

The lectures were intended as variations on the central theme "Probability Theory and Computer Science": a theme which had been chosen by Professor Guy Louchard, Head of the Computer Science department of Brussels University, as being in unison with an important part of the research work conducted in this department.

The calculus of probabilities provides a major mathematical tool in the analysis of computer systems and computer programs. But bringing it into play in these types of problems can be a difficult task. The preliminary stages of model building are often tricky owing to the high degree of complexity of the real world of computers and programs. The empirical simplifications of the real problem intricacies, the inference of pertinent analytical characteristics, and the definition of a conceptual model which lends itself to mathematical analysis may require much familiarity with this real world.

Then, when the model has been formulated, the conventional tools and methods of analysis are seldom readily applicable without refinements, generalizations, or more efficient computation techniques to cope with the huge amount of calculations which may be involved. Sometimes such new developments broach fundamental issues in Probability or Stochastic Process Theory; in this respect, the last decade has witnessed the emergence of a closer interaction between computer scientists and applied probabilists.

In the final validation and verification of the analysis, when the practical values of the model must be put to the test for prediction, planning or control purposes, or for a more complete understanding of the real problem under investigation, again much expertise in computers and programming is required.

This classical three-step cyclic process of model inference, analysis and validation is competently and fruitfully laboured in a great variety of applications in each of the three parts of the book.

Besides, very appropriately, each author lays stress on a different step of the process. In the first part, Gaver gives special attention to the construction of stochastic models in these early stages where both imagination and care are required; several problems and techniques are used to show how the basic assumptions of a model can be checked against the observed reality.

Queueing networks and discrete-time queueing processes are fundamental models for the analysis of data traffic in computer and communication systems. Kobayashi, in the second part, surveys and proposes several approaches, exact and approximate, to evaluate efficiently performance measures for these models.

Finally, Sedgewick's material is a very good illustration of how a detailed mathematical analysis can lead to a better understanding of an algorithm, to a simpler and more elegant version of it, and to a more efficient computer implementation.

Each part is self-contained and keeps a fair balance of introductive material. The book is thus accessible to both applied mathematicians and computer scientists, challenging, to paraphrase the last author, the former by its models, and the latter by its mathematical developments. Thanks are due to Professors G. Latouche and G. Louchard who took in charge the practical organization of the lectures and the editing of this book.

Philips Research Laboratory P. J. Courtois
Brussels, February 1982

Preface

Probability theory and computer science: the former is an ancient science, where names such as Fermat and Pascal play a pre-eminent role—the analysis of games of chance has progressively added to the modern concepts of the theory of probability—the latter led us in thirty years from the 30 tons and 18 000 tubes of ENIAC to the silicon chip and its tens of thousands of transistors per square millimeter.

The connections between the two sciences are ancient and multiple. Pascal himself, in 1642, invented one of the first mechanical adding machines: fifty were built, of which a few may be found in museums, alongside the prototypes of Babbage, and ENIAC itself. These connections formed the theme of the International Professorship in Computer Science, 1980–1981, hosted by the Université Libre de Bruxelles. This professorship has been created by IBM Belgium. The Fonds National de la Recherche Scientifique is entrusted with the organization. Three topics were covered during the courses and seminars: stochastic modeling, queueing system models and the mathematical analysis of combinatorial algorithms.

The first topic is developed by D. P. Gaver in Part I of this book. Gaver emphasizes a modeling attitude, illustrating his subject with numerous examples. Attention is given to problems of, and models for, redundant system reliability and availability, queueing with priorities, first-passage times and areas under path functions of stochastic processes (total waiting time), and various other examples. Also included is a brief account of aspects of modern data analysis, with the implication that its usefulness is significant at the pre-modeling and model-assessment stages of an investigation. Special attention is given in Chapter 3 to distributional sculpturing, an elegant alteration of conventional distributions in order to represent empirical reality more closely.

Queueing system models are examined by H. Kobayashi in Part II. Chapter 4 is devoted to queueing systems that operate on a discrete-time basis: all events are allowed to occur only at regularly spaced points in time. Aside from such discrete structures of an intrinsic nature, it is often computationally convenient to deal with discrete-time systems when one must obtain a numerical solution of a given problem. Some fundamental issues of discrete-time point processes and related queueing systems are treated in detail. In Chapter 5, Kobayashi presents a discussion of diffusion approximations and shows applications to the performance analysis of some simple computer system and communication system models. In Chapter 6 computational algorithms are presented which mark some of the recent progress in the performance analysis of Markovian queueing networks.

Combinatorial algorithms and their mathematical analysis form the subject of Part III by R. Sedgewick. The intersection of fundamental old techniques from mathematics with fundamental new techniques from computer science form an interesting field of study around which a substantial body of knowledge has been built over the past fifteen years. Sedgewick presents algorithms covering a broad range of application areas: algorithms for search in binary trees, permutations, sort, merge, hashing, etc. The mathematical tools, from probability generating functions to asymptotics in the complex plane, are described and applied to these algorithms. The various types of difficulties encountered are clearly identified and illustrated.

It now remains to fulfill the pleasant task of acknowledging the various individuals and institutions whose help and support made possible the organization of this series of lectures. Thanks are due to IBM Belgium for its financial contribution. Mr. t'Kint de Roodenbeke, Directeur des relations extérieures, was particularly helpful in solving organizational problems.

We also wish to thank the FNRS for its permanent effort towards the promotion of research, fundamental and applied, at the highest level. In particular, the effective cooperation of Mr. Levaux, Secrétaire général du FNRS, and Mr. Delers, Secrétaire adjoint, was greatly appreciated.

The three authors have played a crucial role in the success of the International Professorship in Computer Science, 1980–1981. They have shared without restraint their knowledge and expertise with Belgian specialists, from industrial and public organizations as well as from the academic world, giving them a unique opportunity of being brought into contact with new and challenging thoughts.

Brussels Guy Louchard
June 1982 Guy Latouche

Contents

PREFACE

Computer programs as objects of study for mathematical analysis can be complicated and unsatisfying, or simple and elegant. The detailed study of the dynamic properties of computer programs, an intersection of fundamental old techniques from mathematical analysis with fundamental new techniques from computer science, is an interesting field of study around which a substantial body of knowledge has been built over the past fifteen years. This is a survey (partly) and tutorial (mostly) treatment of some of this work, presented at a level appropriate for both computer scientists and mathematicians.

The major reference work for the material described here is D. E. Knuth's *The Art of Computer Programming*, the first three volumes of which have been published (Knuth 1973a, 1973b, 1980). Knuth pioneered the art of detailed mathematical analysis of algorithms at the level considered here, and many of the derivations that we will consider in detail are taken from his work. In a sense, this could be viewed as an introduction to the serious mathematical material in Knuth's books, for the reader interested in learning what the books have to offer and interested in doing research in the area.

In some places, material contained here summarizes results which have appeared in the research literature. For example, most of the material in Chapter 12 comes from Yao (1976) and Knuth and Schonhage (1978), and some of the material in Chapters 10 and 11 comes from Sedgewick (1978b). Where possible, these notes are intended to supplement such material: the treatment here may seem sketchy because full details are presented in the papers.

Another important source for the preparation of these chapters was the notes for a graduate course in the mathematical analysis of algorithms introduced by Knuth at Stanford University in 1974, taught there by Knuth in 1976 and 1980 and by A. Yao in 1978, and taught by me at Brown University in 1977 and 1981. A significant part of these courses comprised detailed notes prepared by the graduate teaching assistants; these are quite well written and are invaluable as reference material.

A third influence on the material presented here has been the work of P. Flajolet at INRIA in France. Flajolet reintroduced me to the idea of avoiding laborious calculations with recurrences and sums for many problems by doing direct derivations with combinatorial generating functions, then using classical analysis to do asymptotics on the generating functions. I believe that this approach simplifies the analysis significantly for many problems and may have the potential tó allow us to extend the range of algorithms that we can analyze; Flajolet and others are doing active research in this area.

The material is intended to be largely self-contained, but assumes that the reader has some familiarity with computer programming and with discrete mathematics. A broad range of material is presented in both domains, so every reader is likely to find things that are too elementary or too advanced. On the one hand, enough elementary mathematical material is included so that these notes may serve as an introduction to applicable mathematical analysis for a trained computer scientist: on the other hand, enough elementary algorithmic material is included so that they may serve as an introduction to combinatorial algorithms for the trained mathematician. My experience in teaching this material to a mix of computer scientists and mathematicians has been that the mathematicians are as challenged by the algorithms as the computer scientists are by the analysis.

Chapter 7 introduces the subject, with an outline of general techniques and a detailed example. Chapter 8 describes algorithms on trees and techniques for solving simple recurrences. Chapter 9 describes algorithms on permutations and introduces the use of generating functions. Chapter 10 introduces asymptotic methods through a detailed treatment of two particular algorithms. Chapter 11 extends the treatment of asymptotics in Chapter 10 by showing how methods from complex analysis must be used for many problems. Chapter 12 deals with analyses for a problem where several different simple algorithms and several different input models have been suggested: its purpose is to review much of the previous material and to illustrate the difficulty (and importance) of using the proper input model for many problems.

Acknowledgments

The generous support of IBM Belgium and the Université Libre de Bruxelles, which made possible the preparation of these notes and the lectures upon which they are based, is gratefully acknowledged. Also, some of this work was done under support from the National Science Foundation, Grant MCS80-17579 while the author was at Brown University.

Many people looked at an early draft of these notes and provided useful comments, including Trina Avery and Tom Freeman. In particular, Janet Incerpi carefully read and corrected several drafts.

7. Introduction

In this chapter we examine on a general level the basic approach espoused by Knuth (1971) for the detailed mathematical analysis of algorithms. First we consider the general motivations for algorithmic analysis, then we look at the major components of a full analysis, then we analyze an algorithm of fundamental practical importance, Quicksort, and then we discuss the material to appear in later chapters.

7.1 WHY ANALYZE AN ALGORITHM?

There are several answers to this basic question, depending on context: the intended use of the algorithm, the importance of the algorithm in relationship to others (from both practical and theoretical standpoints), and the difficulty of analysis and accuracy of the answer required.

First, the most straightforward reason for analyzing an algorithm is to discover its vital statistics in order to evaluate its suitability for various applications or compare it with other algorithms. Generally, the vital statistics of interest are the primary resources of time and space, most often time. Put simply, we are interested in determining how long an implementation of a particular algorithm will run on a particular computer, and how much space it will require. The analysis generally is kept relatively independent of particular implementations, concentrating instead on deriving results for essential characteristics of the algorithm which can be used to estimate precisely true resource requirements on actual machines.

Occasionally, some expensive resource other than time or space is of interest, and the focus of the analysis changed accordingly. For example, an algorithm to drive a plotting device might be studied to determine the total distance moved by the pen. Also, it is sometimes appropriate to combine resources in the analysis. For example, an algorithm which uses a large amount of memory may use much less time than an algorithm which gets by with very little memory. One way to compare algorithms in such situations is to analyze the product of their time and space requirements: this corresponds to using a "memory rental fee" as the resource to be studied.

The analysis of an algorithm can help one to understand it better, and can suggest informed improvements. The more complicated the algorithm, the more difficult the analysis, and algorithms tend to become shorter, simpler, and more elegant during the analysis process. More important, the careful scrutiny required for proper analysis often leads to more efficient and more correct implementations of algorithms. Analysis requires a far more complete understanding of an algorithm than merely producing a working implementation. Indeed, when the results of analytic and empirical studies agree, one becomes 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 analysis can add to the body of mathematical tools available for mathematical analysis. Such algorithms may be of no practical interest, but may have properties similar to algorithms of practical interest which indicate that understanding them may help to someday understand more important methods. Unfortunately, results of this type are most often negative: many algorithms which seem to be very simple require extremely sophisticated mathematical machinery.

Many algorithms (some of intense practical interest, some of little or none) 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 which are worth studying in their own right.

7.2 GENERAL METHOD

The following general methodology is commonly used for the precise study of the performance of particular algorithms. This approach is a natural one and is very old, but it is generally attributed to Knuth (1971), whose books serve as witness to the utility of the method for fully understanding important algorithms.

The first step is to carefully implement the algorithm on a particular computer. We shall use the term *program* to describe such an implementation, so that one algorithm corresponds to many programs. This implementation not only provides a concrete object to study, but also can give useful empirical data to aid in or to check the analysis.

The implementation presumably is designed to make efficient use of expensive resources. The resources of primary interest must be identified

so that the detailed analysis may be properly focused. The resource most often analyzed is the running time, so the steps below are outlined in terms of studying the running time.

The next step is to estimate the time required by each component instruction of the program. This can usually be done very precisely, depending on the characteristics of the computer system being used.

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 the running time of the entire program can be derived simply by multiplying the frequency and time requirements of each component instruction and adding these products.

The next step is to model the input to the program, to form a basis for the mathematical analysis. Often several different models are used for the same algorithm: initially a model is chosen to make the analysis as simple as possible; finally a model is chosen to 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, the quantities can be analyzed individually, then the averages can be multiplied by instruction times and added to give the running time of the whole program. For worst-case analysis, it is usually difficult to get an exact result for the whole program, and so an upper bound is often derived by multiplying worst-case values of the individual quantities by instruction times, then adding.

The average-case results can be compared with the empirical data to verify the implementation, the model, and the analysis.

7.3 AN EXAMPLE

To illustrate the methodology outlined above, results are sketched here for a particular algorithm of importance, the Quicksort sorting method. This analysis is covered in great detail elsewhere (Sedgewick, 1980 and 1977b), so only a very brief treatment will be given here.

First, an implementation of Quicksort in the PASCAL programming language follows:

```
var a: array [0 . .N] of integer;
procedure quicksort (l, r: integer);
  var v, t, i, j: integer;
  begin
```

```
if $r > l$ then
  begin
  $v := a[r]$; $i := l - 1$; $j := r$;
  repeat
    repeat $i := i + 1$ until $a[i] >= v$;
    repeat $j := j - 1$ until $a[j] <= v$;
    $t := a[i]$; $a[i] := a[j]$; $a[j] := t$;
  until $j < i$;
  $a[j] := a[i]$; $a[i] := a[r]$; $a[r] := t$;
  quicksort (l, j);
  quicksort (i + 1, r)
  end
end;
```

This is a recursive program which sorts the numbers in an **array** $a[l{:}r]$ by partitioning it into two independent parts, then sorting those parts. 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. This is accomplished by maintaining two pointers, one scanning from the left, one from the right. The left pointer is incremented until an element larger than the partitioning 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. The call *quicksort(1,N)* will sort the array, provided that $a[0]$ is set to a value smaller than any other element in the array (in case, for example, the first partitioning element happened to be the smallest element).

The first step in the analysis is to estimate the resource requirements of individual instructions for this program. This is a straightforward step for any particular computer, and we will omit the details. For example, the "inner loop" instruction **repeat** $i: = j \div 1$ **until** $a[i] > = v$ translates, on most computers, to assembly language instructions like

```
LOOP  INC  I, 1
      CMP  V, A(I)
      BL   LOOP
```

This might require four time units (one for each memory reference).

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, there are three natural quantities involved:

A—the number of stages;
B—the number of exchanges; and
C—the number of comparisons.

On a typical computer, the total running time might be about $4C + 11B + 35A$. (The exact values of these coefficients depend on the assembly language program and properties of the machine being used; the values given above are typical.)

The input model for the analysis is to assume that the **array** a contains randomly ordered, distinct numbers. This is the most convenient to analyze; however, it is also possible to study this program under perhaps more realistic models allowing equal numbers (see Sedgewick 1977a).

The average-case analysis for this program involves defining and solving recurrence relations which mirror directly the recursive nature of the program. For example, if C_N is the average number of comparisons to sort N elements, we have $C_0 = C_1 = 0$ and

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

To get the total average number of comparisons, we add the number of comparisons for the first partitioning stage $(N + 1)$ to the number of comparisons used for the subfiles after partitioning. When the partitioning element is the jth largest (which occurs with probability $1/N$ for each $1 \le j \le N$), the subfiles after partitioning are of size $j - 1$ and $N - j$. For this to be valid, it is necessary to prove that the subfiles left after partitioning a random file are still random. Now the analysis is reduced to a mathematical problem which does not depend on properties of the program or the algorithm. This particular problem is not difficult to solve: first change j to $N - j + 1$ in the second part of the sum to get

$$C_N = (N + 1) + \frac{2}{N} \sum_{1 \le j \le N} C_{j-1}$$

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

$$NC_N - (N - 1)C_{N-1} = 2N + 2C_{N-1}$$

Rearrange terms and divide by $N(N + 1)$ to get a simple recurrence

$$\frac{C_N}{N + 1} = \frac{C_{N-1}}{N} + \frac{2}{N + 1}$$

which holds for $N > 2$. This telescopes to a simple sum, giving the result

$$C_N = 2(N+1)(H_{N+1} - \tfrac{4}{3})$$

where H_{N+1} is the $(N+1)$st harmonic number $(\sum_{1\le k\le N+1} 1/k)$. The average values of the other quantities can be derived in a similar manner.

This program can be improved in several ways to make it the sorting method of choice in many computing environments. A complete analysis can be carried out even for much more complicated improved versions, and expressions for the average running time can be derived which match closely observed empirical times (Sedgewick 1980, 1978a). Furthermore, the combinatorial mathematics involved in these analyses becomes quite interesting (see Sedgewick, 1977b).

7.4 PERSPECTIVE

The analysis above is in many ways an "ideal" methodology: not all algorithms can be as smoothly dealt with as this.

First, a full analysis like that above requires a fair amount of effort which should be reserved only for our most important algorithms. Most often, the parts of the methodology which are program-specific (dependent on the implementation) are skipped, 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. However, it is an unfortunate fact that as full an analysis as possible seems to be required to compare algorithms properly. Many researchers have been led astray by prematurely applying incomplete analyses.

In succeeding chapters, we will concentrate on techniques of mathematical analysis which are applicable to the study of the performance of algorithms. At the same time, we will survey some fundamental combinatorial algorithms, including several of practical importance. We will see that algorithms which seem to be quite simple can lead to quite intricate mathematical analyses, but that the analyses can uncover significant differences between algorithms which have direct bearing on the way they are used in practice.

The most serious problem in the analysis of most algorithms in common use on computers today is the formulation of proper models which realistically represent the input and which lead to manageable analysis problems.

Serious research in this seems to be required in several areas of application. However, there is a large class of *combinatorial* algorithms for which the model is very straightforward, as in sorting.

For the most part, algorithms of this type will be considered here. Many of these algorithms are of fundamental importance in a wide variety of computer applications, and so are deserving of the effort involved for detailed analysis. Furthermore, the input model leads immediately to mathematical problems of a combinatorial nature similar to those which arise in probability, so that classical methods of analysis provide a firm basis for their solution. The number of algorithms which have been studied in this way is steadily growing, and the analytic tools available are becoming increasingly better understood by computer scientists.

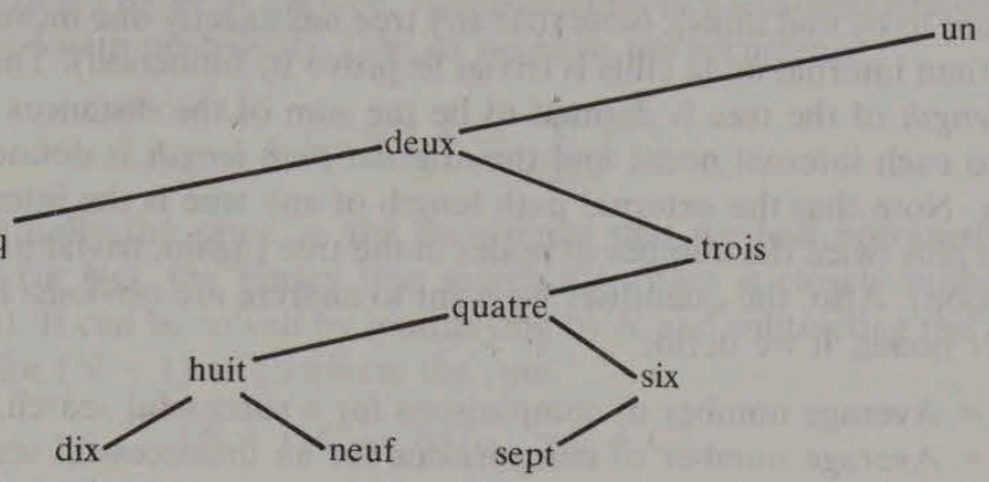
8. Trees

Trees are fundamental structures used in many practical algorithms, and it is important to understand their properties in order to be able to analyze these algorithms. In this section, we examine in detail several problems in analysis which relate to trees and to a fundamental tree search algorithm. These analyses not only provide results of practical interest, but also exhibit several fundamental techniques: for solving linear recurrences, for using generating functions and probability generating functions, and for using double and combinatorial generating functions.

The properties of trees have been studied for quite some time by combinatorial mathematicians, and many results are known. However, as we will see, there is a difference between viewing trees as static (combinatorial) objects and viewing them as dynamic objects, built by algorithms.

8.1 BINARY TREE SEARCH

The so-called "dictionary", "symbol table" or simply "search" problem is a fundamental one in computer science: a set of keys (perhaps with associated information) is to be organized so that efficient searches can be made for particular keys (or associated information). A *binary search tree* is an elementary structure commonly used for this problem: it consists of a root node containing a key and links to left and right subtrees which are defined in the same way. The left subtree contains all keys less than the key at the root, and the right subtree contains all keys greater. It is convenient to include an artificial "header" node with a key smaller than all other keys ($-\infty$) as the root of every binary tree. For example, the following binary tree contains the French words for the numbers 1 to 10. Such a tree can be



searched for a node with the value v using the following program:

```
x: = head;
repeat
  if v < x↑.key then x: = x↑.left else x: = x↑.right
until v = x↑.key
```

If the key sought is not in the tree, this program will not work properly, since x will eventually be assigned to one of the null pointers at the bottom of the tree. The program is easily modified to check for this case and to insert a new node with the key sought if desired (see Knuth, 1973b; Sedgewick 1983). For example, if the key *onze* were to be inserted, a new node would be created as the right son of *neuf*. Note that the nodes of the tree can be printed out in order with the simple program:

```
procedure print (x: link);
  begin
  if x <> nil then
    begin
    print (x↑.left);
    write (x↑.key);
    print (x↑.right)
    end
end;
```

Many other useful operations are easily defined on binary search trees; see Knuth (1973b).

There seem to be two quantities of interest in the analysis of the binary tree search program: the number of nodes visited in a successful search and the number of nodes visited in an unsuccessful search. These turn out to be closely related. First, call the nodes of the tree which have keys *internal nodes*, and define imaginary nodes at the bottom called *external nodes* (pointed to by null links). Note that any tree has exactly one more external node than internal node (this is trivial to prove by induction). The *internal path length* of the tree is defined to be the sum of the distances from the root to each internal node, and the *external path length* is defined analogously. Note that the external path length of any tree is the internal path length plus twice the number of nodes in the tree (again, trivial to prove by induction). Also, the quantities we want to analyze are obvious. For a tree with N nodes, if we define

C_N = Average number of comparisons for a successful search, and
C'_N = Average number of comparisons for an unsuccessful search then we have

$$C_N = \frac{\{\text{Internal path length}\}}{N} + 1$$

and

$$C'_N = \frac{\{\text{External path length}\}}{(N+1)}$$

Then the relationship described above between internal and external path length implies that

$$(N+1)C'_N = NC_N + N$$

Now, our notion of "average" must be more carefully defined. The above relationships hold for any given tree, if each internal (or external) node is equally likely to be sought: they are static properties of the tree. Our analysis must take into account the dynamic properties of the trees, since the way in which the keys are inserted can drastically affect the shape of the tree.

The tree above was created by inserting the keys in the order *un*, *deux*, *trois*, *quatre*, etc. If they are inserted in alphabetic order (*cinq*, *deux*, *dix*, *huit*, etc.) then a degenerate tree with a high path length results. If N nodes are inserted into an initially empty tree in order, then the resulting tree is a single string of nodes, connected by their right links, all with null left links. In other words, there is one internal node at distance i from the root for each $0 \le i < N$, so the internal path length is $\sum_{0 \le i < N} i = N(N-1)/2$. Thus trees with quite different shapes can be built by the algorithm, and path lengths can vary widely. But, as we will see, it is *not* true that every possible tree is equally likely to result.

Knuth (1973b) gives a simple derivation that gives the average values of C_N and C'_N; we will do a more direct derivation later. The simple argument is to observe that 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. Any particular key searched was the kth one inserted with probability $1/N$, so we have the recurrence

$$C_N = 1 + \frac{1}{N} \sum_{1 \le k \le N} C'_{k-1}$$

This is virtually the same as the recurrence that we had previously for Quicksort (in fact, the binary tree search algorithm is closely related to Quicksort). It can be solved by multiplying by N and subtracting the same equation for $(N-1)$ to eliminate the sum.

$$(N+1)C'_N - NC'_{N-1} = 2 + C'_{N-1}$$

Rearranging terms, we have

$$C'_N = C'_{N-1} + \frac{2}{N+1}$$

which telescopes to the answer

$$C'_N = 2H_{N+1} - 2$$

which means that

$$C_N = 2\left(1 + \frac{1}{N}\right)H_{N+1} - \frac{2}{N} - 3$$

Now, H_N is about $\ln N$ (we will see how to say so more precisely in Chapter 10), so that, with N nodes in the tree, only about $2\ln N$ nodes need be examined for a typical search.

A fundamental point in this derivation is that we are "averaging" not over all trees, but over all possible orders of the keys inserted into the tree. (The reader should check that our assumptions are equivalent to this statement.) This "permutation" model is natural and realistic for this problem, but it is quite different from the assumption that all trees are equally likely to occur.

8.2 DIGRESSION: SOLVING FIRST-ORDER LINEAR RECURRENCES

Above, we took the recurrence

$$(N+1)C'_N = (N+1)C'_{N-1} + 2$$

and divided by $(N+1)$ to get a recurrence which telescoped. For Quicksort, we had a more complicated recurrence

$$NC_N = (N+1)C_{N-1} + 2N$$

which telescopes when divided by $N(N+1)$. It turns out that it is always possible to transform recurrences of this nature into sums by telescoping. For example,

$$NC_N = (N-2)C_{N-1} + 2N$$

telescopes when multiplied by $(N-1)$, and

$$C_N = 2C_{N-1} + N$$

telescopes when divided by 2^N.

In general, the recurrence

$$C_N = X_N C_{N-1} + Y_N$$

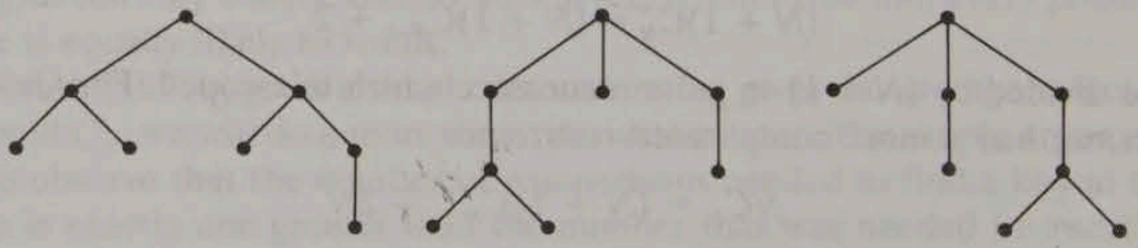
telescopes when divided by $X_N X_{N-1} \ldots X_1$ (written $\prod_{N \ge k \ge 1} X_c$). (The reader may wish to check this formula on the examples above.) For some problems, the recurrence can be made to telescope by multiplying by $\prod_{K>N} X_K$, if it converges.

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". Of course, for many problems, we may be left with a sum which is difficult to evaluate: we will study this in more detail later.

8.3 ELEMENTARY COMBINATORICS OF TREES

Trees as (static) objects have been intensely studied by combinatorial mathematicians, because they arise as natural models in many actual problems, and they have many interesting properties.

The most general kind of tree is a *free tree*: a set of N nodes and $N-1$ edges connecting them together (this implies that there can be no cycles). If one of the nodes is designated as the *root*, then a free tree becomes an *oriented tree*, and if the order of the subtrees at every node is specified, we have an *ordered tree*. The first two trees below are equivalent as free trees but not as oriented trees; the second and third are equivalent as oriented



trees but not as ordered trees. A *binary tree* built by our algorithm is a special kind of ordered tree in which not only does each node have 0, 1 or 2 sons but also each son of a 1-son node is specified to be "left" or "right".

The most natural combinatorial question that arises for any defined tree structure is the enumeration problem: how many different trees are there? For some tree structures, this can be a difficult problem indeed; for binary trees it not only will shed some light on our algorithm, but also will illustrate another fundamental analytic tool.

If we let b_N be the number of different binary trees with N nodes, then the following recurrence relation holds:

$$b_N = \sum_{0 \le k < N} b_k b_{N-1-k}, \qquad N > 0$$

It is convenient to define $b_0 = 1$, $b_N = 0$ for all N negative, and make the recurrence hold for all N:

$$b_N = \sum_{0 \le k < N} b_k b_{N-1-k} + \delta_{N0}$$

(Here δ_{N0} is 1 for $N = 0$, 0 otherwise.) This can be checked against the small values in the table below:

N	0	1	2	3	4
b_N	1	1	2	5	14

This recurrence is much more complicated than those we have seen before, and requires more powerful tools. Define the *generating function*

$$B(z) = \sum_{N \ge 0} b_N z^N$$

where z is some artificial variable. This function describes the entire sequence succinctly. Multiplying both sides of the recurrence by z^N and summing on N, we have

$$\begin{aligned} B(z) &= \sum_{N \ge 0} \sum_{0 \le k < N} b_k b_{N-1-k} z^N + 1 \\ &= \sum_{k \ge 0} \sum_{N > k} b_k b_{N-1-k} z^N + 1 \text{ (interchange order)} \\ &= \sum_{k \ge 0} \sum_{N \ge 0} b_k b_N z^{N+k+1} + 1 \text{ (change } N \text{ to } N+k+1) \\ &= z \sum_{k \ge 0} b_k z^k \sum_{N \ge 0} b_N z^N + 1 \\ B(z) &= zB(z)^2 + 1 \end{aligned}$$

(A double sum of this type is called a *convolution*; it arises whenever two power series are multiplied.) This formula for $B(z)$ can be solved with the quadratic equation

$$B(z) = \frac{1}{2z}(1 \pm \sqrt{1-4z})$$

To solve for b_N we need to expand back to power series. This is easily done with the binomial theorem:

$$(1-4z)^{1/2} = \sum_{N\geq 0}\binom{\frac{1}{2}}{N}(-4z)^N$$

Thus, $B(z) = 1/2z(\pm(1 - 2z + \ldots))$ and we must choose the root with the negative sign in order for $B(0)$ to be defined. This equation can be used to derive b_N in familiar terms: the binomial coefficient $\binom{x}{N}$ is defined to be

$$\prod_{1\leq j\leq N}\frac{x-j+1}{N-j+1}$$

This is the familiar $x!/N!(x-N)!$ for integer x, but the more general definition allows the binomial theorem to be used for noninteger exponents. This leads to

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

so we can set coefficients equal to get

$$b_N = \binom{\frac{1}{2}}{N+1}(-1)^N 2^{2N+1} = \frac{1}{N+1}\binom{2N}{N}$$

These numbers are called the Catalan numbers, which appear very frequently in combinatorics. In Chapter 10 we will see how to show that the approximate value is $b_N \approx 4^N/N\sqrt{\pi N}$.

There are many more permutations on N objects than binary trees of N items; therefore, many different permutations give rise to the same tree in the binary tree search algorithm. But not all trees are equally likely: as we shall see, some trees (fortunately the more balanced ones) appear much more frequently than others.

8.4 GENERATING FUNCTION SOLUTION OF RECURRENCES

Generating functions provide a mechanical method for solving many recurrence relations, although some facility for manipulating power series is required. For example, a direct solution for the binary tree search

recurrence

$$(N+1)\,C'_N = 2N + \sum_{0\leq k<N} C'_k$$

can be derived as follows. Define $C(z) = \sum_{N\geq 0} C'_N z^{N+1}$, multiply both sides of the equation by z^N and sum on N to get

$$C'(z) = 2\sum_{N\geq 0} Nz^N + \sum_{N\geq 0}\sum_{0\leq k\leq N} C'_k z^N$$

The double sum is a convolution, as before, so

$$C'(z) = 2\sum_{N\geq 0} Nz^N + \sum_{k\geq 0} C'_k z^{k+1}\sum_{N\geq 0} z^N$$

But

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

is elementary, and differentiating both sides gives

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

so we have a differential equation on the generating function

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

The solution to this differential equation is

$$\rho C(z) = \int \rho\frac{2z}{(1-z)^2}\,dz \quad \text{where} \quad \rho = \exp\left(-\int\frac{1}{1-z}\,dz\right)$$

Carrying out the calculation gives $\rho = 1 - z$ and

$$C(z) = \frac{2}{1-z}\ln\frac{1}{1-z} - \frac{2z}{1-z}$$

We know from the above that

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

Integrating this gives

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

and multiplying both sides by $1/(1-z)$ gives

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

Setting coefficients of z^N equal gives the solution

$$C'_N = 2H_{N+1} - 2$$

as before.

For some problems, an explicit formula for the generating function can be difficult to derive. For others, the expansion back to power series can present the main obstacle. However, this general method can be relied on to produce a solution for many recurrences, if a solution is available.

8.5 UNSUCCESSFUL SEARCH IN BINARY SEARCH TREES

The above derivation shows how to find the average cost of an unsuccessful search in a binary tree, but it cannot be extended to find more information about the distribution of this quantity (for example, the variance). In this section, we examine a detailed direct analysis for this problem.

As above, we assume that a random permutation of N elements is used to build a binary search tree. Let $P_{Nk} \equiv$ {Probability that the last insertion takes k steps}. Then the average unsuccessful search time is given by

$$\sum_k kP_{Nk}$$

and the variance by

$$\sum_k (k - C'_N)^2 P_{Nk} = \sum_k k^2 P_{Nk} - C'^2_N$$

To calculate P_{Nk} directly, the first step is to use a combinatorial argument based on permutations to set up a recurrence relation. This will be a general topic of the next section; we will postpone discussion of this particular argument until then. The recurrence which results is

$$NP_{Nk} = 2P_{(N-1)(k-1)} + (N-2)P_{(N-1)k}$$

If we use the generating function

$$P_N(z) = \sum_{k\geq 0} P_{Nk} z^k$$

we get, after multiplying by z^k and summing,

$$NP_N(z) = (2z + N - 2)P_{N-1}(z)$$

which telescopes immediately to

$$P_N(z) = \prod_{2\leq j\leq N} \frac{2z + j - 2}{j}$$

This complicated explicit formula for our generating function is actually well known: it turns out that

$$P_{Nk} = \frac{1}{N!}\begin{bmatrix} N-1 \\ k \end{bmatrix} 2^k$$

where the brackets indicate Stirling numbers of the first kind. The mean and variance can then be calculated directly (although some facility with sums involving Stirling numbers is necessary). Fortunately, there is a much easier way.

8.6 PROBABILITY GENERATING FUNCTIONS

When generating functions are used to manipulate probabilities, they have several simple properties which make the calculation of the average, variance, and other moments easy. If

$$P(z) = \sum_{k\geq 0} p_k z^k$$

where $\{p_k\}$ is a sequence of probabilities (positive values which sum to 1), then we can exploit the following facts:

(i) $P(1) = \Sigma_{k\geq 0} p_k = 1$;
(ii) the average, defined to be $\Sigma_{k\geq 0} kp_k$, is simply $P'(1)$;
(iii) the variance is $P''(1) + P'1) - P'(1)^2$; and
(iv) 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, because if $R(z) = P(z)Q(z)$ and $P(1) = Q(1) = 1$ then $R'(1) = P'(1) + Q'(1)$ and $R''(1) = P''(1) + Q''(1) + 2P'(1)Q'(1)$.

These four properties apply directly to our problem, because the generating function $P_N(z)$ is the product of a number of very simple probability generating functions

$$f_j(z) = \frac{j-2}{j} + \frac{2}{j}z$$

(note that $f_j(1) = 1$). Therefore, we need only compute moments for these simple functions and then sum. We have

$$f_j'(1) = \frac{2}{j} \quad\text{and}\quad f_j''(1) = 0$$

so

$$P_N'(1) = \sum_{2\le j\le N} \frac{2}{j} = 2H_N - 2$$

as before, and

$$P_N''(1) + P_N'(1) - P_N'(1)^2 = \sum_{2\le j\le N} \frac{2}{j} - \frac{4}{j^2} = 2H_N - 4H_N^{(2)} + 2$$

Probability generating functions provide a very convenient way to calculate the mean and variance of the number of steps required for an unsuccessful search.

Two more useful properties of probability generating functions can help to directly formulate recurrence relations involving them:

(v) if $X(z)$ and $Y(z)$ are probability generating functions for independent random variables X and Y, then $X(z)Y(z)$ is the probability generating function for $X + Y$; and

(vi) if $X_1(z)$ and $X_2(z)$ are the probability generating functions for a random variable X, conditional on whether or not some independent event happens (with probability p), then the unconditional probability generating function for X is $pX_1(z) + (1 - p)X_2(z)$. For example, property (v) can be used to give a direct argument for the equation

$$P_N(z) = \left(\frac{N-2}{N} + \frac{2z}{N}\right) P_{N-1}(z)$$

for unsuccessful search: with probability $2/N$, the last two elements are on the same search path, contributing 1 to the total cost of an unsuccessful search; otherwise (with probability $(N - 2)/N$) the contribution is 0.

8.7 SUCCESSFUL SEARCH IN BINARY TREES

More detailed analyses of successful search can be carried out using the methods of the previous section in conjunction with the fundamental relationship between internal and external path length.

Another quantity of interest is the *total* internal path length: the total cost of building the tree. If we let $q_{Nk} = Pr\,\{k$ is the total internal path length of a tree built from a random permutation of N elements$\}$ then

properties (v) and (vi) above lead directly to the recurrence

$$q_N(z) = \frac{z^{N-1}}{N} \sum_{1\le j\le N} q_{j-1}(z)q_{N-j}(z)$$

on the generating function

$$q_N(z) = \sum_{k\ge 0} q_{Nk}z^k$$

This recurrence is difficult to solve explicitly for q_{Nk}, but differentiating and evaluating at $z = 1$ gives the familiar recurrence

$$q_N'(1) = N - 1 + \frac{1}{N} \sum_{1\le j\le N} (q_{j-1}'(1) + q_{N-j}'(1))$$

for the average. The variance can be derived in a similar way.

8.8 INTERNAL PATH LENGTH IN STATIC TREES

We have seen that there are many fewer binary trees on N elements than permutations, and that the mapping from permutations to trees effected by the binary tree search algorithm is such that the average internal path length is $2(N + 1)H_{N+1} - 4N - 2$. In this section we analyze the internal path length of binary trees in the static model, where each tree is considered equally likely.

If we define q_{Nk} to be the probability that k is the total internal path length, then the same argument as before gives a recurrence relation on the generating function $q_N(z) = \sum_{k\ge 0} q_{Nk}z^k$:

$$q_N(z) = z^{N-1} \sum_{1\le j\le N} \frac{\frac{1}{j}\binom{2j-2}{j-1} \frac{1}{N-j+1}\binom{2N-2j}{N-j}}{\frac{1}{N+1}\binom{2N}{N}} q_{j-1}(z)q_{N-j}(z)$$

(Before, the probability that the subtrees were of size $j - 1$ and $N - j$ was always $1/N$; here it is a complicated-looking function derived from the Catalan numbers in a simple way.) As before, this recurrence is hard to solve for q_{Nk}. Here, it is even difficult to find $q_N'(1)$: the resulting recurrence requires generating functions. It will be slightly more convenient to work with the number of trees of size N with internal path length k,

$$Q_{Nk} \equiv \frac{1}{N+1}\binom{2N}{N} q_{Nk}$$

which satisfies (from the above recurrence)

$$\sum_{k\geq 0} Q_{Nk} z^k = z^{N-1} \sum_{1\leq j\leq N} \sum_{r\geq 0} Q_{(j-1)r} z^r \sum_{s\geq 0} Q_{(N-j)s} z^s$$

Since we will need a generating function of N later, it is convenient to include N right away in a *double generating function*:

$$Q(w,z) = \sum_{N\geq 0} \sum_{k\geq 0} Q_{Nk} w^N z^k$$

Multiplying the above by w^N and summing on N gives

$$\begin{aligned} Q(w,z) &= \sum_{N\geq 1} \sum_{1\leq j\leq N} \sum_{r\geq 0} Q_{(j-1)r} z^r \sum_{s\geq 0} Q_{(N-j)s} z^s w^N z^{N-1} + 1 \\ &= w \sum_{j\geq 0} \sum_{r\geq 0} Q_{jr} z^r (wz)^j \sum_{N\geq j} \sum_{s\geq 0} Q_{(N-j)s} z^s (wz)^{N-j} + 1 \\ &= w \sum_{j\geq 0} \sum_{r\geq 0} Q_{jr} z^r (wz)^j \sum_{N\geq 0} \sum_{s\geq 0} Q_{Ns} z^s (wz)^N + 1 \\ &= wQ(wz,z)^2 + 1 \end{aligned}$$

Now we can differentiate this with respect to z and evaluate it at 1 to get the average. Note that

$$\begin{aligned} Q(w,1) &= \sum_{N\geq 0} \sum_{k\geq 0} Q_{Nk} w^N \\ &= \sum_{N\geq 0} \frac{1}{N+1} \binom{2N}{N} w^N = B(w), \end{aligned}$$

the generating function for the Catalan numbers, and

$$\begin{aligned} q(w) &\equiv \frac{\partial}{\partial z} Q(w,z)|_{z=1} \\ &= \sum_{N\geq 0} \sum_{k\geq 0} k Q_{Nk} w^N \\ &= \sum_{N\geq 0} \frac{1}{N+1} \binom{2N}{N} q'_N(1) w^N \end{aligned}$$

is the generating function for the total internal path length. Differentiating both sides of our equation for $Q(w,z)$ with respect to z and evaluating at $z = 1$ gives

$$q(w) = 2wB(w)(q(w) + wB'(w))$$

which has the solution

$$q(w) = \frac{1}{1-4w} - \frac{1}{w}\left(\frac{1-w}{\sqrt{1-4w}} - 1\right)$$

and gives the eventual result

$$q'_N(1) = \frac{4^N(N+1)}{\binom{2N}{N}} - 3N - 1 \approx N\sqrt{\pi N} - 3N$$

Notice that this is substantially larger, for large N, than for the dynamic case: the average internal path length for binary search trees is proportional to $N\log N$, not $N\sqrt{N}$. These results show that there are many degenerate trees which are very unbalanced, but these trees are built only rarely by the binary tree search algorithm.

8.9 COMBINATORIAL GENERATING FUNCTIONS

The derivation above includes some quite complicated manipulations with triple sums to prove the simple formula

$$Q(w,z) = wQ(wz,z)^2 + 1$$

It is reasonable to ask whether there might be a simpler proof of this formula. Fortunately there is, using a direct argument based upon the generating function itself.

Our approach to this point has been to treat probabilistic analyses as counting problems; then derive recurrence relations for the counting problems; then discover an implied relationship between associated generating functions, and then use analytic techniques to learn about the generating function from this relationship. However, for many, if not most, problems amenable to solution by this approach, it turns out to be easy to derive the relationship on the generating function directly. The key to this new approach is to use the combinatorial object being analyzed as the index of summation for the generating function rather than the parameters of the analysis as we have been using. For example, in the derivation above we were working with

$$Q(w,z) = \sum_{N\geq 0} \sum_{k\geq 0} Q_{Nk} w^N z^k$$

where Q_{Nk} is the number of trees with N nodes and internal path length k. This may be expressed equivalently as

$$Q(w,z) = \sum_{\text{all trees } T} w^{|T|} z^{ipl(T)}$$

where $|T|$ is the number of nodes in T and $ipl(T)$ is the internal path length of T. Now, we can change this into a double sum, because any nonempty tree T consists of left and right subtrees, trees T_L and T_R, joined together by a root node. This leads immediately to

$$Q(w,z) = \sum_{\text{all trees } T_L} \sum_{\text{all trees } T_R} w^{|T_L+T_R+1|} z^{ipl(T_L)+ipl(T_R)+|T_L|+|T_R|} + 1$$

(The "+1" takes into account the case when T is empty). Note that the number of nodes in a tree is one plus the number of nodes in its subtrees, and the internal path length of a tree is the sum of the internal path lengths of its subtrees plus one for each node in the subtrees. Now, this double sum is easily rearranged to make two independent sums:

$$\begin{aligned} Q(w,z) &= w \sum_{\text{all trees } T_L} (wz)^{|T_L|} z^{ipl(T_L)} \sum_{\text{all trees } T_R} (wz)^{|T_R|} z^{ipl(T_R)} + 1 \\ &= wQ(wz,z)^2 + 1 \end{aligned}$$

as before. The reader may wish to study this example carefully to appreciate both its simplicity and its subtleties.

For another example, consider the application of combinatorial generating functions to find the total internal path length of binary search trees. Here, the combinatorial objects of interest are permutations, so we start with the generating function

$$Q(w,z) = \sum_{\text{all perms } P} w^{|P|} \frac{z^{ipl(P)}}{|P|!}$$

Here $ipl(P)$ 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. Note that, in this generating function, we need to divide by $|P|!$. One reason for this is that there are many more permutations than trees and the function would not converge otherwise. Another reason that dividing by $|P|!$ is convenient is that it makes $Q(w,z)$ a probability generating function: we have the equivalent expression

$$Q(w,z) = \sum_{N\geq 0} \sum_{k\geq 0} P_{Nk} w^N z^k$$

where P_{Nk} is the probability that k is the internal path length of a tree built from a random permutation of N elements. As above, we can directly derive a functional equation by splitting the combinatorial definition into a double sum. Given two permutations P_L, P_R, we can create

$$\binom{|P_L| + |P_R|}{|P_L|}$$

permutations of size $|P_L| + |P_R| + 1$ all of which led to the same binary search tree 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 $|P_L| + 1$. This leads to the following double sum representation for $Q(w,z)$:

$$\sum_{\text{all perms } P_L} \sum_{\text{all perms } P_R} \binom{|P_L| + |P_R|}{|P_L|} \frac{w^{|P_L|+|P_R|+1} z^{ipl(P_L)+ipl(P_R)+|P_L|+|P_R|}}{(|P_L| + |P_R| + 1)!} + 1$$

(The permutation of size 0 can not be split in this way: this accounts for the "+1".) This equation is somewhat more complicated than the one above, but it can be simplified by differentiating with respect to w first, then rearranging terms as above. If $Q_w(w,z)$ denotes the derivative of $Q(w,z)$ with respect to w, we have

$$\begin{aligned} Q_w(w,z) &= \sum_{\text{all perms } P_L} \frac{(wz)^{|P_L|} z^{ipl(P_L)}}{|P_L|!} \sum_{\text{all perms } P_R} \frac{(wz)^{|P_R|} z^{ipl(P_R)}}{|P_R|!} \\ &= Q(wz,z)^2 \end{aligned}$$

Now if we differentiate this with respect to z and evaluate at $z = 1$, then we get the differential equation for the average internal path length that we solved in the previous chapter.

8.10 ADVANCED TREE ALGORITHMS

The elementary binary tree search algorithm has the undesirable property of having a very bad worst case. Several more advanced structures have been developed to deal with this and other problems: the analysis of these structures leads to a wealth of interesting mathematical problems.

One type of solution uses *radix search trees*. These structures are built using individual bits of the keys, rather than comparing keys as entities. The worst-case performance is proportional to the number of bits in the keys. The analysis of the average-case performance for this method requires advanced techniques that we will study in a later section.

A second type of solution uses *balanced binary trees*. These structures are built with algorithms that do local transformations on binary trees during an insertion, to prevent them from getting too far out of balance. The worst case for any search is proportional to $\log N$. Properties and implementations of these algorithms are discussed in some detail in Guibas and Sedgewick (1978). The mathematical questions involved in studying these trees are interesting, but difficult: very few results have been derived. For example, for some of the algorithms, the difference between the static and the dynamic structures is even more pronounced than above. The algorithms achieve balance by allowing more keys per node, so that, for example, 3-nodes (with two keys and three sons) and 4-nodes (with three keys and four sons) are allowed. Enumerating all such trees is difficult enough (Odlyzko, 1979), but for some algorithms, there are trees which cannot even be constructed: the enumeration problem for the dynamic case is unsolved, let alone the path length problem.

The unsolved mathematical problems on trees are not restricted to advanced structures. For example, the average height (the length of the longest path from the root to an external node) has only recently been derived for the static model (Flajolet and Odlyzko, 1980, and the full answer for the dynamic model is still not known (Robson, 1979).

9 Permutations

Combinatorial algorithms deal only with the relative order of a linear array of N elements and so can be thought of as operating on the numbers 1 to N in some order. Such an ordering is called a *permutation*, and is a well-defined combinatorial object with a wealth of interesting properties. In the previous chapter, we analyzed an algorithm which transforms permutations into trees; in this chapter, we will look at the analysis of more algorithms on permutations, the properties of permutations that arise in these analyses, and some more tools (evaluation of finite sums involving harmonic numbers and binomial coefficients) for use in such analyses. The algorithms that we will study are much simpler than those in the previous section, but we will see that even simple properties of permutations can be difficult to analyze.

9.1 FINDING THE MINIMUM

The trivial algorithm for finding the minimum element in an array may be implemented as follows:

```
$v$: = $\infty$;
**for** $i$: = $1$ **to** $N$ **do if** $A[i] < v$ **then** $v$: = $A[i]$;
```

The running time of this algorithm, is proportional to $c_1N + c_2A + c_3$, where c_1, c_2, c_3 are appropriate constants and A (the only "variable" involved) is the number of times $v := A[i]$ is executed. This is the number of *left-to-right minima* of the permutation: the number of new minimum values encountered when scanning from left to right.

For example, the permutation

3 8 4 **2** 5 9 6 **1** 7

has three left-to-right minima (in bold face). As we have before, we will assume all $N!$ permutations to be equally likely as input and define the probabilities

$$P_{Nk} \equiv Pr\{A = k\} = \frac{\{\text{number of perms for which } A = k\}}{N!}$$

The method of solution is to set up a recurrence relation on P_{Nk} by writing the permutations in an appropriate order, in this case sorted by their last element, as in the example below for $N = 4$.

```
1 2 3 4   1 2 4 3   1 3 4 2   2 3 4 1
2 1 3 4   2 1 4 3   3 1 4 2   3 2 4 1
1 3 2 4   1 4 2 3   1 4 3 2   2 4 3 1
3 1 2 4   4 1 2 3   4 1 3 2   4 2 3 1
2 3 1 4   2 4 1 3   3 4 1 2   3 4 2 1
3 2 1 4   4 2 1 3   4 3 1 2   4 3 2 1
```

Writing the permutations down in this way makes it plain that the last element does not affect the number of left-to-right minima unless it is the smallest. More precisely, every permutation of $N - 1$ elements with k left-to-right minima corresponds to $(N - 1)$ permutations of N elements with k left-to-right minima and 1 permutation of N elements with $(k + 1)$ left-to-right minima. This leads directly to the recurrence

$$N!P_{Nk} = (N - 1)(N - 1)!P_{(N-1)k} + (N - 1)!P_{(N-1)(k-1)}$$

$$P_{Nk} = \left(1 - \frac{1}{N}\right)P_{(N-1)k} + \frac{1}{N}P_{(N-1)(k-1)}$$

In terms of the probability generating function $P_N(z) = \sum_{k\geq 0}P_{Nk}z^k$, this is

$$P_N(z) = \frac{z + N - 1}{N}P_{N-1}(z)$$

This formula could be derived directly, as in Chapter 8, with the argument that the last element independently contributes 1 to the number of left-to-right minima with probability $1/N$. Also, as in Chapter 8, we can 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 $(z + k - 1)/k$, with the eventual result that the mean is $H_N - 1$ with variance $H_N - H_N^{(2)}$.

This problem can also be dealt with using the "combinatorial" double generating function technique introduced in the previous chapter. While the derivation is certainly not simpler than the one given above, it is very instructive and will prepare us well for more difficult problems that cannot be handled easily with the elementary techniques used above. As before,

the combinatorial objects of interest are permutations, so we start with the generating function

$$B(w, z) = \sum_{\text{all perms } P} w^{|P|}\frac{z^{lrm(P)}}{|P|!}$$

Here $lrm(P)$ denotes the number of left-to-right minima in the permutation P and an alternate representation for the generating function is

$$B(w, z) = \sum_{N\geq 0}\sum_{k\geq 0}P_{Nk}w^Nz^k$$

where P_{Nk} is the probability that a random permutation of N elements has k left-to-right minima. As before, we can directly derive a functional equation from the combinatorial definition. Given a permutation P_1, we can create $|P_1| + 1$ permutations of size $|P_1| + 1$, one of which ends in 1 (and so has one more left-to-right minimum than P_1), and $|P_1|$ of which do not end in 1 (and so have the same number of left-to-right minima as P_1). This leads to the formulation

$$B(w, z) = \sum_{\text{all perms } P_1}\frac{w^{|P_1|+1}z^{lrm(P_1)+1}}{(|P_1| + 1)!} + \sum_{\text{all perms } P_1}\frac{|P_1|w^{|P_1|+1}z^{lrm(P_1)}}{(|P_1| + 1)!}$$

Differentiating with respect to w, we have

$$B_w(w, z) = \sum_{\text{all perms } P_1}\frac{w^{|P_1|}z^{lrm(P_1)+1}}{|P_1|!} + \sum_{\text{all perms } P_1}\frac{w^{|P_1|}z^{lrm(P_1)}}{(|P_1| - 1)!}$$
$$= zB(w, z) + wB_w(w, z)$$

Solving for $B_w(w, z)$, we get a simple first-order differential equation

$$B_w(w, z) = \frac{z}{1 - w}B(w, z)$$

which has the solution

$$B(w, z) = \frac{1}{(1 - w)^z}$$

(since $B(0, 0) = 1$). Differentiating with respect to z, we have

$$B_z(w, z) = \ln\frac{1}{1 - w}\,e^{z\ln(1/1-w)}$$

Now, evaluating at $z = 1$ gives the result

$$B_z(w, 1) = \frac{1}{1 - w}\ln\frac{1}{1 - w}$$

the generating function for the harmonic numbers, as expected.

9.2 *IN-SITU* PERMUTATION

A direct use for permutations is to specify how to rearrange elements in an array. For example, the permutation

3 8 4 2 5 9 6 7

could direct that the array

E A S R M L D C B

should be rearranged by putting the third element in position 1, the eighth element in position 2, etc. to leave

S C R A M B L E D

If the permutation is in an array $P[1:N]$, the array in $A[1:N]$ and an output array $B[1:N]$ is available, the program is trivial:

```
for i := 1 to N do B[i] := A[P[i]];
```

In some situations, the B array might not be available, and the A array needs to be permuted "in place". A straightforward way to do this is to start by saying $A[1]$ 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, so it requires a bit array to mark those elements which have been moved, as in the following implementation:

```
for j := 1 to N do
  if not m[j] then
    begin
    s := j; z := A[j]; t := P[j];
    while t <> j do
      begin m[s] := true; A[s] := A[t]; s := t; t := P[s] end;
    A[s] := z; m[s] := true;
    end;
```

In our example, first $A[3] = S$ is moved to position 1, then $A[4] = R$ is moved to position 3, etc., to leave

S C R A M L D E B

Next, the M is marked (and not moved) and then B, D and L are marked and moved.

This is not truly an "in-place" algorithm because N extra bits are needed for the marks: we will see how to eliminate the "mark" bits later. The only "variable" in the running time of this program is the number of times the **if** statement succeeds, the number of *cycles* in the permutation. The analysis of this quantity turns out to be simple because of a combinatorial correspondence between permutations. A permutation can be defined by writing out its cycles: the example permutation above can be written

(1 3 4 2 8) (5) (6 9 7)

which means that $A[3]$ is to be moved to position 1, then $A[4]$ is to be moved to position 3, etc. Since there are several ways to write the same permutation in this "cycle" notation (for example, (976)(5)(28134) is another for the permutation above), it is convenient to define a canonical representation: for each cycle, write the smallest element in the cycle first (call this the *leader*), then write the cycles in decreasing order of their leaders:

(6 9 7) (5) (1 3 4 2 8)

But now the parentheses are no longer needed, and we have a one-to-one correspondence with another permutation:

6 9 7 5 1 3 4 2 8

In fact, it is the left-to-right minima that determine where cycles begin and end. This means that we have already completed the analysis, and the average and variance for the number of cycles is the same as for the number of left-to-right minima, H_N and $H_N - H_N^{(2)}$.

To eliminate the "mark" bits, we can decide to permute each cycle only when its leader is encountered, as follows:

```
for j := 1 to N do
    begin
    k := P[j]; while k > j do k := P[k]
    if k = j then permute cycle;
    end;
```

Now the analysis must also include a quantity (call it B) that counts the iterations of this **while** loop, the "distance" from each element to a smaller one in the same cycle. This is most easily counted in the permutation describing the cycle structure: in our example, the following table gives the contribution to this quantity due to each element:

6 9 7 5 1 3 4 2 8
2 0 0 0 4 1 0 1 0

That is, when the 3 is encountered, one more element must be examined (the 4) before a smaller one in the same cycle (the 2) is encountered (which bears witness to the fact that 3 is not the cycle leader).

An easy way to analyze this quantity is to expand the table above to a two-dimensional array in which the ith row has 1s in positions corresponding to "right-to-left" minima in the permutation defined by considering only the first i positions. In our example, this table is

6	9	7	5	1	3	4	2	8
0	0	0	0	1	0	0	1	1
0	0	0	0	1	0	0	1	
0	0	0	0	1	1	1		
0	0	0	0	1	1			
0	0	0	0	1				
0	0	0	1					
1	0	1						
1	1							
1								

If we ignore the rightmost 1s, the columns in this table add up to the numbers that we had before. On the other hand, the rows (by definition) add up to right-to-left minima statistics: the ith row is the number of right-to-left minima in the permutation occupying the first $N - i + 1$ positions of the array. If we use the same correspondence as before and generate the permutations of N elements by adding a new element at the end of permutations of $N - 1$ elements, then it is plain that the average increase in B is just one less than the number of right-to-left minima in a random permutation of N elements. (The increase simply comes from the 1s in the first row of the table.) This gives a direct solution for the average:

$$B_N = B_{N-1} + H_N - 1$$
$$= \sum_{1\le k\le N} H_k - N$$
$$B_N = (N+1)H_N - 2N$$

(The sum of the harmonic numbers can be evaluated by substituting in the definition $H_k = \Sigma_{1\le j\le k} 1/j$, then switching the order of summation: we will see another method at the end of this chapter.) Note carefully that the amount contributed by the Nth element is *not* independent of the arrangement of the previous elements, so we cannot get a "simple" direct recurrence of the generating function by using this simple way to compute the average. Or, put another way, there does not seem to be a simple way to organize the permutations to derive a recurrence in the probabilities that $B = k$. However, Knuth (1971) found enough structure in this problem to derive the variance. Knuth's derivation is rather complicated, but we can develop a simple argument using combinatorial generating functions based on his way of splitting the problem.

Again, we write down the generating function in terms of the combinatorial structure being analyzed, in this case permutations. We have

$$B(w,z) = \sum_{\text{all perms } P} w^{|P|} \frac{z^{\nu(P)}}{|P|!}$$

where $\nu(P)$ is the value of B when the algorithm is run on P. As above, our goal is to derive a functional equation using this combinatorial definition, because that can be used to find the first two derivatives with respect to z of $B(w,z)$, evaluated at 1, from which the mean and variance can be computed. Given two permutations P_L, P_R, we can create

$$\binom{|P_L| + |P_R|}{|P_L|}$$

permutations of size $|P_L| + |P_R| + 1$ all of which have the same value of B by (i) concatenating them with a 1 in between; and (ii) making the values distinct by choosing $|P_L|$ values from $2, 3, \ldots, |P_L| + |P_R| + 1$ in all possible ways for assignment to P_L. All of the permutations formed in this way have the same value of B when the *in-situ* permutation algorithm is run on them: $\nu(P_L) + \nu(P_R) + |P_R|$. This follows from examining the table above: the 1 has all 1s below it; the elements to the left of these 1s are all 0; and this block divides the table into two independent parts, one for the left permutation, one for the right permutation. Since every permutation of size $|P_L| + |P_R| + 1$ is formed exactly once by this construction, we have the following double sum representation for $B(w,z)$:

$$\sum_{\text{all perms } P_L} \sum_{\text{all perms } P_R} \binom{|P_L| + |P_R|}{|P_L|} \frac{w^{|P_L|+|P_R|+1} z^{\nu(P_L)+\nu(P_R)+|P_R|}}{(|P_L| + |P_R| + 1)!} + 1$$

As in the analysis of the internal path length of binary search trees, this reduces, after differentiating by w and rearranging terms, to the simple functional equation:

$$B_w(w,z) = B(w,z)B(wz,z)$$

Then, differentiating by z and evaluating at 1 gives differential equations in the generating functions for the mean and variance that can be solved as in the analysis of binary search trees. (This part of the calculation, involving tricky manipulations with partial derivatives, is by no means simple, but it is so automatic as to be suitable for a computerized symbolic manipulation system such as MACSYMA (Mathlab Group, 1977).

Many interesting combinatorial problems derive from the "cycle" structure of permutations, some of which have direct relevance to various algorithms. For example, the above algorithm, might be slightly improved by a special test for singleton cycles. This could be easily analyzed, because the average number of cycles of size m for any particular m can be derived. On the other hand, questions such as "what is the length of the largest cycle, on the average?" lead to very difficult combinatorial problems (Shepp and Lloyd, 1966).

9.3 SELECTION SORT

A similar quantity arises in selection sort, a method of sorting by successively "finding the minimum":

```
for $j := 1$ to $N$ do
  begin
  $v := \infty$; $k := j$;
  for $i := j$ to $N$ do
    if $A[i] < v$ then begin $v := A[i]$; $k := i$ end;
  $t := A[j]$; $A[j] := A[k]$; $A[k] := t$;
  end;
```

The operation of this method on our sample file is diagrammed below: the elements in bold face are the left-to-right minima encountered.

3 8 4 **2** 5 9 6 1 7
1 **8** **4** **2** 5 9 6 3 7
1 2 **4** 8 5 9 6 **3** 7
1 2 3 **8** **5** 9 6 4 7
1 2 3 4 **5** 9 6 8 7
1 2 3 4 5 **9** **6** 8 7
1 2 3 4 5 6 **9** **8** **7**
1 2 3 4 5 6 7 **8** 9
1 2 3 4 5 6 7 8 **9**

The only "variable" in the running time of this program is B, the total number of left-to-right minima encountered during the life of the sort. (This is not the "leading" term in the running time, since the **if** statement is executed $(N + 1/2)$ times.) As above, we can define a correspondence between permutations as follows: given a permutation of $N - 1$ elements, create N permuations of N elements by incrementing each element and then exchanging each element with a prepended 1. For example, 3 1 2 corresponds to:

1 4 2 3
4 1 2 3
2 4 1 3
3 4 2 1

Each of these permutations will result is 1 4 3 2 when the algorithm is iterated once, which is equivalent to 3 1 2 for subsequent iterations. This correspondence immediately implies that

$$B_N = B_{N-1} + H_N = (N + 1)H_N - N$$

But again, we do *not* have independence: for example, if we have a low number of left-to-right minima on one iteration, we can expect a low number on the next. What is worse, each iteration modifies the permutation not yet seen (by exchanging a new element into the position occupied by the current minimum). This "dynamic" aspect seems to make this a much more difficult problem than the *in-situ* permutation problem: the variance of B is not yet known.

9.4 INSERTION SORT

A simple sorting program which can be more successfully analyzed is insertion sort. This method works by "inserting" each element into proper position among those previously considered moving larger elements over one position to make room:

```
$A[0] := -\infty$;
for $i := 2$ to $N$ do
  begin
  $v := A[i]$; $j := i-1$;
  while $A[j] > v$ do begin $A[j+1] := A[j]$; $j := j - 1$ end;
  $A[j+1] := v$;
  end
```

The table below shows the operation of this program on our sample file; elements in bold face are those that are moved.

3 8 4 2 5 9 6 1 7
3 8
3 4 **8**
2 **3** **4** **8**
2 3 4 5 **8**
2 3 4 5 8 9
2 3 4 5 6 **8** **9**
1 **2** **3** **4** **5** **6** **8** **9**
1 2 3 4 5 5 7 **8** **9**

The only "variable" in the running time of the program (call it B) is the total number of elements moved: this is the number of elements to the left of each element which are greater than that element, shown for our example in the following table:

3	8	4	2	5	9	6	1	7
0	0	1	3	1	0	2	7	2

(Note that this table can be built by considering the permutation as a static object, even though the elements involved may move around before the move which is counted.) This table is a well-known combinatorial object called an *inversion table*, and the total of the numbers in the table is called the number of *inversions* in the permutation. The important properties of inversion tables are that a table uniquely determines the corresponding permutation and that, for a random permutation, the ith entry can take on each value between 0 and $i - 1$ with probability $1/i$, independently of the other entries. This gives a direct generating function derivation for the average number of inversions: the generating function for the number of inversions involving the Nth element is $(1 + z + z^2 + \ldots + z^{N-1})/N$, independent of the arrangement of the previous elements, so that the generating function for the total number of inversions in a random permutation of N elements is given by

$$B_N(z) = \frac{1 + z + z^2 \ldots + z^{N-1}}{N} B_{N-1}(z)$$

Again, the mean and variance can be calculated by summing individual means and variances: the mean turns out to be $N(N - 1)/4$, and the variance $N(N - 1)(2N + 5)/72$.

It is an interesting exercise to derive these results using "combinatorial" double generating functions as we have done for several other problems.

9.5 DIGRESSION: DISCRETE SUMS

The calculation of the variance of the number of inversions involves evaluating sums of the form

$$\sum_{0 \le k \le n} k \quad \text{and} \quad \sum_{0 \le k \le n} k^2$$

Since we will be encountering more complicated sums of a similar nature, it is appropriate to consider methods of evaluating such sums at this point. It turns out that a few identities and techniques are sufficient for the evaluation of many sums involving binomial coefficients, harmonic numbers and polynomials.

For polynomials in the index of summation, the fundamental formula on

binomial coefficients

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

should be used. For example

$$\sum_{0 \le k < n} k^2 = \sum_{0 \le k < n} 2\binom{k}{2} + \sum_{0 \le k < n} \binom{k}{1} = 2\binom{n}{3} + \binom{n}{2} = \frac{n(n - \frac{1}{2})(n - 1)}{3}$$

In general, a polynomial in k can be expressed as a sum of binomial coefficients, the fundamental formula applied, and the binomial coefficients converted back to polynomials if desired. This identity is akin to a simple integration identity: binomial coefficients are sometimes written in terms of "falling factorial" powers $k^{\underline{m}} = k(k - 1) \ldots (k - m + 1)$, so $m!\binom{k}{m} = k^{\underline{m}}$ and our identity is

$$\sum_{0 \le k < n} k^{\underline{m}} = \frac{n^{\underline{m+1}}}{m + 1}$$

which is exactly analogous (for $m \ge 0$) to

$$\int_0^n x^m \, dx = \frac{n^{m+1}}{m + 1}$$

This analogy to integration holds elsewhere. For example, there is a "summation-by-parts" formula which can be used to simplify some complicated sums:

$$\sum_{0 \le k < n} (a_{k+1} - a_k)b_k = a_n b_n - a_0 b_0 - \sum_{0 \le k < n} a_{k+1}(b_{k+1} - b_k)$$

This corresponds to the familiar

$$\int_0^n u \, dv = uv \Big|_0^n - \int_0^n v \, du$$

The 'difference" $a_{k+1} - a_k$ in the discrete case corresponds to the "differential" operator dv in the continuous case. For example, summation by parts can be used to sum the harmonic numbers: taking $a_k = k$, $b_k = H_k$, we get

$$\sum_{1 \le k < n} H_k = nH_n - \sum_{1 \le k < n} (k + 1)(H_{k+1} - H_k) = nH_n - n$$

This is the analog to

$$\int_1^n \ln x \, dx = n \ln n - n + 1$$

(In fact, the definition of H_n itself is the analog to $\int_1^n 1/x \, dx = \ln n$.)

Similarly, we can evaluate

$$\sum_{1\le k<n}\binom{k}{m}H_k$$

in a manner analogous to integrating

$$\int_1^n x^m \ln x \, dx$$

using summation by parts.

The analogy is not a method for evaluating sums (it sometimes breaks down), but it is useful in allowing one to apply intuition about integrals towards sums. One further feature: the analog to e^x is 2^k, since

$$\sum_{0\le k<n} 2^k = 2^n - 1$$

and thus we can do sums such as $\Sigma k^2 2^k$, etc.

Later we will see more details on the important class of sums involving two binomial coefficients. Sums with the index of summation appearing in the lower index of a single binomial coefficient are more difficult: although we know that

$$\sum_k \binom{n}{k} = 2^n$$

by the binomial theorem, the partial sum

$$\sum_{0\le k<m}\binom{n}{k}$$

is very difficult to evaluate. We will see techniques for such an evaluation later.

9.6 TWO-ORDERED PERMUTATIONS

A practical improvement to insertion sort, called Shellsort, 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. This is easily implemented as follows:

```
for $h := h_t, h_{t-1}, \ldots h_1$ do
  for $i := h + 1$ to $N$ do
    begin
    $v := A[i]; j := i - h;$
    while $j > 0$ and $A[j] > v$ do begin $A[j + h] := A[j]; j := j - h$ end;
    $A[j + h] := v;$
    end;
```

The "increments" $h_t, h_{t-1}, \ldots h_1$ which control the sort must form a decreasing sequence which ends in 1. Considerable effort has gone into finding the best sequence of increments, with few analytic results: although it is a simple extension to insertion sort, Shellsort has proven to be extremely difficult to analyze. This may be appreciated by attempting to analyze the simplest version in which h takes on only two values, 2 and 1.

For example, suppose that the input array is initially

3 5 10 6 1 9 14 2 15 12 11 8 16 7 4 13

Then, after the first pass, with $h = 2$, the file becomes

1 2 3 5 4 6 10 7 11 8 14 9 15 12 16 13

Such a permutation, which consists of two interleaved sorted permutations, is called *2-ordered*. Since the next pass of Shellsort, with $h = 1$, is just insertion sort, its running time will vary with the number of inversions. We need to find the average number of inversions in a 2-ordered permutation. This will not only involve some interesting manipulations with finite sums but will also yield some results that will be of use later, since 2-ordered permutations arise in the analysis of several interesting algorithms (for example, they naturally model the input to merging algorithms).

To compute the average number of inversions in a 2-ordered file, we count the total number of inversions appearing in all 2-ordered files, by counting, for each i, the total number of inversions from all 2-ordered files involving the ith element from the odd part of the array. Now, this element can have value $i + j$ for $0 \le j \le N$. Since exactly i of the elements less than $i + j$ appear in the odd part of the array, all the elements in the first j positions of the even part of the array must have values less than $i + j$. From this it is easy to see that the number of inversions to be counted is $i - j - 1$ for $i > j$ and $j - i + 1$ for $j \ge i$, or simply $|i - j - 1|$. Since there are exactly

$$\binom{i+j-1}{j}\binom{2N-i-j}{N-j}$$

2-ordered files where the ith element in the odd part has the value $i + j$, we are led directly to the formula

$$\binom{2N}{N}A_N = \sum_{1\le i\le N}\sum_{0\le j\le N}|i-j-1|\binom{i+j-1}{j}\binom{2N-i-j}{N-j}$$
$$= \sum_{0\le i<N}\sum_{0\le j\le N}|i-j|\binom{i+j}{i}\binom{2N-i-j-1}{N-j}$$

for the average number of inversions in a 2-ordered permutation of length $2N$. (Knuth, 1973b, gives a graphical proof of this formula based on a correspondence between 2-ordered permutations and paths in an N by N lattice.)

The general strategy in evaluating this sum is to use symmetries to eliminate the absolute value and to get an inner sum involving only the two binomial coefficients, then evaluate that sum and simplify. In this case, the resulting inner sum is rather difficult to evaluate although it can be done with elementary techniques.

First, to "use symmetries" means to split the inner sum into two parts:

$$\sum_{0\le i<N}\sum_{0\le j\le N}(\ ) = \sum_{0\le i<N}\sum_{0\le j\le i}(\ ) + \sum_{0\le i<N}\sum_{i<j\le N}(\ )$$

then change i to $N-1-i$ and j to $N-j$ in the second sum and recombine to get

$$\binom{2N}{N}A_N = \sum_{0\le i<N}\sum_{0\le j\le i}(2(i-j)+1)\binom{i+j}{i}\binom{2N-i-j-1}{N-j}$$

Now, change j to $i-j$ in the inner sum, interchange the order of summation, then change i to $i+j$ to get

$$\binom{2N}{N}A_N = \sum_{0\le j<N}(2j+1)\sum_i\binom{2i+j}{i}\binom{2N-2i-j-1}{N-i-j-1}$$

Below it is shown that

$$\sum_i\binom{2i+j}{i}\binom{2N-2i-j-1}{N-i-j-1} = \sum_{0\le k<N-j}\binom{2N}{k} = \sum_{j<k\le N}\binom{2N}{N-k}$$

which implies, after interchanging the order of summation, that

$$\binom{2N}{N}A_N = \sum_{k\ge1}\binom{2N}{N-k}\sum_{0\le j<k}(2j+1)$$

Note that the only property of $|i-j|$ that we have made use of in this derivation is that it is constant along diagonals; in particular, the derivation works for any function $f(i,j)$ satisfying

$$f(i,i-j) = f(j,0)$$

and

$$f(i,i+j) = f(0,j) \quad \text{for } j\ge 0$$

For any weight function with these properties, we have proved the following combinatorial identity which (as we will see later) is useful in the

study of 2-ordered permutations:

$$\sum_{0\le i<N}\sum_{0\le j\le N} f(i,j)\binom{i+j}{i}\binom{2N-i-j-1}{N-j} = \sum_{k\ge1}\binom{2N}{N-k}\sum_{0\le j<k}(f(j,0)+f(0,j+1))$$

Returning to the number of inversions in a 2-ordered permutation, we find that the inner sum evaluates to k^2, so that

$$\binom{2N}{N}A_N = \sum_{k\ge1}\binom{2N}{N-k}k^2$$

The easiest way to evaluate this sum is to "absorb" the k^2 into the binomial coefficient by writing it as $N^2-(N-k)(N+k)$ so that

$$\binom{2N}{N-k}k^2 = N^2\binom{2N}{N-k} - 2N(2N-1)\binom{2N-2}{N-k-1}$$

Now we are left with sums on the bottom index of a binomial coefficient which can be evaluated since they are nearly over the whole range:

$$\sum_k\binom{2N}{k} = 2^{2N} = 2\sum_{k\ge1}\binom{2N}{N-k} + \binom{2N}{N}$$

The final result, which comes after some calculations from the previous three equations, is

$$\binom{2N}{N}A_N = N\,4^{N-1}$$

so A_N is approximately equal to $\sqrt{\pi N^3}/4$.

It is somewhat surprising that such a simple result requires such a long and complicated derivation: this result deserves a one-line proof! A shorter proof is available through a combinatorial generating function argument: using Knuth's correspondence to paths in a lattice diagram it is possible to show that the generating function

$$B(w,z) = \sum_{\substack{\text{all 2-ordered}\\ \text{perms } P}} w^{|P|}z^{inv(P)}$$

(where $inv(P)$ is the number of inversions in P) satisfies

$$B_w(w,z)\big|_{z=1} = \frac{w}{(1-4w)^2}$$

However, this derivation not only involves an indirect argument using the generating function for particular types of paths in the lattice but also some complicated manipulations with derivatives of these generating functions (see Knuth, 1973b, Ex. 5.2.1-15). A direct, simple proof of this result seems to be an elusive goal.

9.7 SUMS INVOLVING TWO BINOMIAL COEFFICIENTS

Many sums involving two binomial coefficients reduce in some way to the fundamental Vandermonde convolution:

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

This is trivially proved with generating functions from the identity

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

since $(1+z)^r$ is the generating function for $\binom{r}{k}$, etc. Many other sums involving two binomial coefficients can be derived in this way: for example, Vandermonde's convolution on the upper index

$$\sum_{0\le k\le r} \binom{r-k}{m}\binom{s+k}{n} = \binom{r+s+1}{m+n+1}$$

comes from the simple identity

$$\frac{1}{(1+z)^r}\,\frac{1}{(1+z)^s} = \frac{1}{(1+z)^{r+s}}$$

A similar (though much more complicated) argument can be used to remove one binomial coefficient from the inner sum which arose above:

$$\sum_i \binom{2i+j}{i}\binom{2N-2i-j-1}{N-i-j-1} = \sum_{i\ge0}\binom{2N-1-i}{N-i-j-1}2^i$$

Now Vandermonde's convolution can be used in an unexpected way: since

$$2^i = \sum_k \binom{i}{k},$$

the sum is

$$\sum_k\sum_{i\ge0}\binom{2N-1-i}{N-i-j-1}\binom{i}{k} = \sum_k\sum_{i\ge0}\binom{2N-1-i}{N+j}\binom{i}{k}$$

$$= \sum_{k\ge0}\binom{2N}{N+j+k+1} = \sum_{0\le k<N-j}\binom{2N}{k}$$

This derivation is typical: many more examples and many more techniques are given in Knuth 1973a).

9.8 SHELLSORT

Some aspects of the above analysis can be extended to analyzing the general Shellsort algorithm, but the full treatment of this program remains an unsolved problem. Yao has recently done an analysis of $(h, k, 1)$ Shellsort using techniques similar to those above, but the results and methods become much more complicated. For general Shellsort, the functional form is not even known: some conjecture it to be N^α, for some small constant α; others conjecture $N(\log N)^2$. This problem has an interesting combinatorial quality and direct practical relevance (since Shellsort is the method of choice for medium-size files, especially if space for the program is limited). For many more details on this problem and an example of the extensive use of the techniques of this section see Yao (1980).

10. Elementary asymptotic approximations

The methods and examples that we have studied to this point have been oriented towards deriving *exact* average-case results. Unfortunately, such exact solutions may not be always available, or if available they may be too unwieldy to be of much use. In this section, we examine some methods of deriving approximate solutions to problems or of approximating exact solutions.

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, and one reason to do asymptotic analysis is to "translate" between them. It is sometimes necessary to do such translations in order to invoke more powerful analytic tools.

Generally, the "size" of a problem to be solved is expressed in terms of one (perhaps a few) parameters, and we are interested in approximations that become more accurate as the parameters become 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 convergent power series in N and $\log N$. Therefore, our general approach will be to convert quantities to truncated versions of such power series, then manipulate them in well-defined ways. When results cannot be so expressed, then we will need much more powerful tools, as we will see in the next chapter.

10.1 *O*-NOTATION

The standard way to write down precise asymptotic approximations is the so-called O-notation. For example, it is shown below that

$$H_N = \ln N + \gamma + \frac{1}{2N} + O\left(\frac{1}{N^2}\right)$$

(Here γ is a constant with approximate value 0.57721. . .) The quantity represented by the $O(1/N^2)$ term (the error in the approximation) is less in absolute value than some constant divided by N^2 for large enough N. A precise definition of the O-notation may be found in Knuth (1973b, 1976). It is easy to check many elementary properties of the O-notation which facilitate simple calculations involving *asymptotic formulas* such as the one above. For example, consider the formula derived in Chapter 8 for the average internal path length of binary search trees:

$$(N+1)C'_N = 2(N+1)H_N - 2N$$

Since $N\ O(1/N^2) = O(1/N)$ and $1/N + O(1/N)$ can be replaced by $O(1/N)$, etc., we have

$$(N+1)C'_N = 2N\ln N + N(2\gamma - 2) + 1 + O\left(\frac{1}{N}\right)$$

No matter what the value of the constant involved in the O-notation (in principle it could be large, in practice it is small), the error in this approximation becomes small as N grows.

The O-notation is useful because it can allow suppression of unimportant details without loss of mathematical rigor or precise results. Note that we could determine C_N to within $O(1/N^2)$ with only a few more calculations. If a more accurate answer is desired, one can be obtained, but most of the detailed calculations are suppressed otherwise. We shall be most interested in methods that allow us to keep this "potential accuracy", producing answers which could be calculated to arbitrarily fine precision if desired.

10.2 BATCHER'S ODD–EVEN MERGE

An example of a problem for which the exact answer is surely very complicated is determining the average number of exchanges required by Batcher's "odd–even" merging method. In this section, we examine the derivation of an approximate answer for this problem that illustrates many basic techniques used for estimating combinatorial sums. In the next section we will study the more powerful methods that are needed to make the answer precise. Details on this derivation may be found in Sedgewick (1978b).

Batcher's method is a way to sort a 2-ordered permutation on $2N$ elements (which is equivalent to doing an N by N merge) as follows:

```
**for** $j := 1$ **to** $N$ **do** $compexch(A[2j-1], A[2j])$;
**for** $k := lg(N)$ **downto** $1$ **do**
        **for** $j := 1$ **to** $N - 2^{k-1}$ **do** $compexch(A[2j], A[2j+2^k-1])$;
```

Here *compexch* is a procedure which compares its two arguments and exchanges them, if necessary, to make the first smaller. The operation of this method on a sample file is diagrammed below:

1	2	3	5	4	6	10	7	11	8	14	9	15	12	16	13
1	2	3	5	4	6	7	10	8	11	9	14	12	15	13	16
1	2	3	5	4	6	7	10	8	11	9	14	12	15	13	16
1	2	3	5	4	6	7	9	8	11	10	13	12	15	14	16
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

This algorithm seems mysterious at first glance, but actually its operation can be quite easily understood in several ways (see Knuth 1973b; Sedgewick 1978b, 1982a for details). It can be shown that the first stage ensures that the ith element in the odd part of the array has value $i+j$ with $j \le i$, then the subsequent stages ensure that $(i-j) < 2^k$, until reaching the state where the ith element in the odd part of the array contains $i + j$ with $j = i - 1$ for each i, at which point the array is sorted.

Note that the sequence of compare–exchange operations is predetermined (independent of the data) and that all the operations in each stage can be done in parallel, so the method is appropriate for implementation in hardware.

The only "unknown" quantity in the above program is the number of times exchanges are actually done: we will denote the average value of this quantity for a file of size $2N$ by B_N. In Sedgewick (1978b) it is shown that our combinatorial identity for 2-ordered permutations from the previous section holds, with a "weight function" $f(k)$ that turns out to be equal to the number of 1s in the Gray code representation of k (see Ramshaw and Flajolet, 1980). Thus, by that derivation, the average number of exchanges is given by

$$B_N = \sum_{k\ge1} \frac{\binom{2N}{N-k}}{\binom{2N}{N}} F(k) \quad \text{where} \quad F(k) = 2\sum_{0\le j<k} f(j) + k$$

This sum is essentially a sum over the lower index of a binomial coefficient which, as we saw when calculating the number of inversions in a 2-ordered permuation, can be easy to evaluate if $F(k)$ is a simple polynomial in k. However, for other $F(k)$, such sums are difficult to evaluate exactly, and we must resort to asymptotic techniques. To approximate B_N, our method will be to estimate the summand in terms of classical functions, then approximate the sum with an integral, and integrate. We will describe the method in general terms, for it is applicable to a variety of similar sums involving factorials, powers, and other functions, and reasonably well behaved $F(k)$.

A precise way to estimate sums with integrals was given by Euler:

$$\sum_{1\le k<n} f(k) = \int_1^n f(x)\,dx + \sum_{1\le k\le m} \frac{B_k}{k!} f^{(k-1)}(x)\Big|_1^n + R_m$$

Here B_k are the Bernoulli numbers ($B_0 = 1, B_1 = -1/2, B_2 = 1/6, B_3 = 0$), m is the number of terms desired in the asymptotic estimate, and R_m is a term smaller in absolute value than the last term estimated. (The Bernoulli numbers grow to be quite large, so this formula is typically useful only for small m.) Of course, the function f must be differentiable enough times to make the formula valid.

For example, taking $f(k) = 1/k$ leads to an asymptotic expansion for the harmonic numbers:

$$H_N = \ln N + \gamma + \frac{1}{2N} - \frac{1}{12N^2} + O\left(\frac{1}{N^4}\right)$$

and taking $f(k) = \ln k$ leads to an expansion for $N!$:

$$\ln N! = (N+\tfrac12)\ln N - N + \ln\sqrt{2\pi} + \frac{1}{12N} + O\left(\frac{1}{N^3}\right)$$

These expansions require calculation of "Euler's constant" $\gamma = 0.57721\ldots$ and "Stirling's constant" $\ln\sqrt{2\pi}$: see Knuth (1973a) for many more details about these expansions and Euler's formula.

To use Euler's formula to approximate B_N, we need to approximate the summand with functions that we could hope to integrate. The first step, which is unnecessary for this problem but very useful for many others, is to "normalize" the sum by centering it so that the largest term occurs for $k = 0$, then dividing by that term. This is useful because not all the terms of the sum contribute significantly to the result. (In fact, as we will see, for this example most do not.)

Next, we need to approximate the summand. For this, we need only Stirling's approximation

$$\ln N! = (N+\tfrac12)\ln N - N + \ln\sqrt{2\pi} + O\left(\frac{1}{N}\right)$$

Applying this to the binomial coefficients in our summand, we find that

$$\frac{N!N!}{(N+k)!(N-k)!} = \exp\{2\ln N! - \ln(N+k)! - \ln(N-k)!\}$$

$$= \exp\left\{(2N+1)\ln N - (N+\tfrac{1}{2})(\ln(N+k) + \ln(N-k)) - \ln\sqrt{2\pi} - k(\ln(N+k) - \ln(N-k)) + O\left(\frac{1}{N}\right) + O\left(\frac{1}{N+k}\right) + O\left(\frac{1}{N-k}\right)\right\}$$

Note carefully that this approximation is meaningless for $|k|$ close to N; fortunately, we will not need it for k in that range.

The ln function is easily approximated by turning its Taylor series expansions into an asymptotic formula:

$$\ln(1-x) = -x - \frac{x^2}{2} - \frac{x^3}{3} + O(x^4), \quad \text{for} \quad |x| < 1$$

Approximation by Taylor series works for many other functions, for example

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + O(x^4)$$

Since $\ln(N+k) = \ln N + \ln\left(1 + \frac{k}{N}\right)$ and similarly for $\ln(N-k)$, we have

$$\ln(N+k) + \ln(N-k) = 2\ln N - \frac{k^2}{N^2} + O\left(\frac{k^4}{N^4}\right)$$

$$\ln(N+k) - \ln(N-k) = \frac{2k}{N} + O\left(\frac{k^3}{N^3}\right)$$

Substituting,

$$\frac{N!N!}{(N+k)!(N-k)!} = \exp\left\{-(N+\tfrac{1}{2})\left(-\frac{k^2}{N^2} + O\left(\frac{k^4}{N^4}\right)\right) - k\left(\frac{2k}{N} + O\left(\frac{k^3}{N^3}\right)\right) - \ln\sqrt{2\pi} + O\left(\frac{1}{N}\right) + O\left(\frac{1}{N+k}\right) + O\left(\frac{1}{N-k}\right)\right\}$$

Now we have even more O terms to worry about: we need to restrict k to be less than some value to make all the approximations useful. On the other hand, we cannot restrict k too much, since we will need to deal with the terms for large k in some other way. We will partially postpone this

problem as follows. Clearly, since k is our index of summation, we want as few occurrences of k as possible (preferably just one) in our final approximation. The "largest" term involving k is $-k^2/N$; next come $O(k^2/N^2)$ and $O(k^4/N^3)$. These become $O(1/N)$ for $|k| < \sqrt{N}$, so $\sqrt{N}$ is clearly a "critical value" beyond which more values of k may be needed. Recognizing that we may later need to allow k to be larger, we will postpone the exact decision by restricting $|k|$ to be $< \sqrt{Nt(N)}$ for some small function $t(N)$, in which case we have

$$\frac{N!\,N!}{(N+k)!\,(N-k)!} = \exp\left\{\frac{-k^2}{N} + O\left(\frac{t(N)}{N}\right)\right\} = e^{-k^2/N}\left(1 + O\left(\frac{t(N)}{N}\right)\right)$$

(This last equation follows from $e^{O(x)} = 1 + O(x)$, a fact the reader might wish to check.)

Applying the approximation to our sum, we have

$$B_N = \sum_{1 \le |k| \le \sqrt{Nt(N)}} e^{-k^2/N}\left(1 + O\left(\frac{t(N)}{N}\right)\right)F(k) + \sum_{|k| > \sqrt{Nt(N)}} \frac{N!\,N!}{(N+k)!\,(N-k)!}F(k)$$

But we can use the approximation calculated above to bound the second sum, since the terms are decreasing. For $|k| > \sqrt{Nt(N)}$, we know that

$$\frac{N!\,N!}{(N+k)!\,(N-k)!} < \frac{N!\,N!}{(N+\sqrt{Nt(N)})!\,(N-\sqrt{Nt(N)})!} = \exp\left\{-t(N) + O\left(\frac{t(N)}{N}\right)\right\}$$

In fact, exactly the same bound holds for $\exp(-k^2/N)$, for $|k| > \sqrt{Nt(N)}$, so we can write

$$B_N = \sum_{k \ge 1} e^{-k^2/N}\left(1 + O\left(\frac{t(N)}{N}\right)\right)F(k) + O\left(\sum_{|k| > \sqrt{Nt(N)}} e^{-t(N)}F(k)\right)$$

Now, if $F(k)$ does not get too large, then $t(N)$ can be chosen large enough to make the second sum small. For example, if $F(k)$ is $O(N^m)$ for some constant m, then $t(N)$ can be chosen to be $(m+2)\ln N$ to give

$$B_N = \sum_{k \ge 1} e^{-k^2/N}F(k)\left(1 + O\left(\frac{\log N}{N}\right)\right) + O\left(\frac{1}{N}\right)$$

As mentioned above, these calculations could, in principle, be carried out to better asymptotic accuracy, but there are complications involved. One problem, as the reader certainly must have noticed, is that it is difficult to predict *a priori* how far to carry out the asymptotic series at various steps in the process: the penalty for taking too few terms is an answer with less accuracy than might be expected; the penalty for taking too many terms is excessive needless calculation.

Now that we have estimated our summand in terms of the exponential function, we can apply Euler's summation formula to estimate it with an integral. In fact, the exponential is so well behaved that this introduces no further error, and we have

$$B_N = \int_1^\infty e^{-x^2/N} F(x)\, dx \left(1 + O\left(\frac{\log N}{N}\right)\right) + O\left(\frac{1}{N}\right)$$

as long as $F(x)$ is well behaved. For example, if $F(x) = 1$, then the integral is the well known normal distribution function (Abramouitz and Stegun 1972), with value $\sqrt{\pi N}$, confirming our earlier estimate for the Catalan numbers. Similarly, for $F(x) = x^2$, the integral is easily evaluated to get an asymptotic formula for the average number of inversions in a 2-ordered file.

For Batcher's merge, it is easy to prove by induction that $F(x) = x \log x + O(x)$. Combined with the previous result, this implies that

$$B_N = \int_1^\infty e^{-x^2/N} x \lg x \, dx + O(N)$$

The substitution $t = x^2/N$ transforms the integral to another well known integral, the "exponential integral function", with the result

$$B_N = \tfrac{1}{4} N \lg N + O(N)$$

Unfortunately, it is not easy to get a better approximation to $F(x)$, and more powerful tools are needed to get a more accurate estimate for B_N, as described in the next section.

However, the general method of approximating a sum by first approximating the summand in terms of simpler functions (using only the large terms), then using Euler's formula to approximate with an integral, is quite often used in asymptotic analysis. The particular example that we have used is a familiar one (corresponding to the normal distribution being the limiting case of the binomial distribution), but the same techniques apply to a wide range of sums. For example, Knuth (1973b) describes how to use the method to evaluate

$$\sum_{k>0} \frac{N!}{(N-2k)!\, 2^k k!}$$

which counts the number of permutations consisting of cycles all of length 1 or 2.

10.3 HASHING WITH LINEAR PROBING

For another concrete example of the use of asymptotic methods, we will consider a fundamental strategy for searching: an alternative method to trees called *hashing*.

The idea behind hashing is to try to address directly a set of N keys within a table of size M by using a *hash function h* which maps keys (which have a very large number of possible values) to table addresses (0 to $M - 1$). (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 been devised.) For example, our sample set of ten keys might have the following hash values:

un	deux	trois	quatre	cinq	six	sept	huit	neuf	dix
4	5	4	6	8	8	5	6	5	8

No matter how good the hash function, some keys will have the same hash values, and a *collision resolution* strategy is needed to decide how to deal with such conflicts. Perhaps the simplest such strategy is *linear probing*: if, when inserting a key into the table, the addressed position (given by the has 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" to the end). For the keys above, the table is filled as follows:

0	1	2	3	4	5	6	7	8	9
				un					
				un	deux				
			trois	**un**	deux				
			trois	un	deux	quatre			
			trois	un	deux	quatre		cinq	
			trois	un	deux	quatre	six	**cinq**	
		sept	**trois**	**un**	**deux**	quatre	six	cinq	
	huit	**sept**	**trois**	**un**	**deux**	**quatre**	six	cinq	
neuf	**huit**	**sept**	**trois**	**un**	**deux**	quatre	six	cinq	
neuf	**huit**	**sept**	**trois**	**un**	**deux**	**quatre**	**six**	**cinq**	dix

Collisions (occupied table positions examined) are printed in bold face in this table. This example (especially the last insertion) shows the algorithm at its worst: it performs badly for a nearly full table, but reasonably for a table with plenty of empty space. As the table fills up, the keys tend to "cluster" together, producing long chains which must be searched to find empty space.

An easy way to avoid clustering is to look not at the previous but at the tth 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. Other hashing methods use dynamic storage allocation, for example keeping all the keys with the same hash value on a simple linked list.

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, who states that this derivation had a strong influence on the structure of his books. His books certainly have had a strong influence on the structure of research in the mathematical analysis of algorithms, and this derivation is a prototype example showing how a simple algorithm can lead to nontrivial and interesting mathematical problems.

Knuth's derivation divides into two parts, a combinatorial argument leading to an exact answer and an asymptotic estimation of that answer. Since the combinatorial argument is long, largely specific to this problem, and similar to things we have done in previous chapters, it will be only summarized here. Some of the asymptotic estimation is similar to the above, so it also will be summarized here. Full details are found in Knuth (1973a, 1973b).

We are interested in knowing the average number of table entries examined for a successful search, after N keys have been inserted into an initially empty table of size M. Knuth shows that this quantity is given by

$$C_{NN} = \frac{1}{N} \sum_{0\le i<N} \sum_{0\le k\le i} \sum_{k\le j\le i} \frac{k+1}{M^i} \binom{i}{j} \left(\frac{1}{j+1}\right) \times (j+1)^j \binom{M-i-1}{M-j-1} (M-j-1)^{i-j}$$

This formula looks formidable, but the inner sum is actually quite similar to the sum in *Abel's binomial theorem*, which says that

$$(x+y)^i = \sum_j \binom{i}{j} x(x-jz)^{j-1}(y+jz)^{i-j}$$

or, with appropriate substitutions,

$$m^i = \sum_j \binom{i}{j} (j+1)^{j-1}(m-j-1)^{i-j}$$

Abel's theorem turns out to be a powerful enough tool to evaluate this

sum, although a substantial amount of computation is involved: the details may be found in Knuth (1973b). The eventual result is

$$2C_{MN} - 1 = \sum_{0\le i\le N} \frac{N!}{M^i(N-i)!}$$

(This formula seems a perfect candidate for derivation using triple "combinatorial" generating functions, but such a derivation has not yet been worked out.) This is an exact answer, but it is not in a particularly useful form, so that we would like to do an asymptotic estimation.

Practically speaking, it seems clear that the table should not be allowed to get nearly full: if we define $N/M = \alpha$, we certainly want $\alpha << 1$. If α is not too close to 1, the sum is not difficult to estimate using the same techniques as above:

$$2C_{MN} - 1 = \sum_{0\le i\le N} \frac{N!}{M^i(N-i)!} = \sum_{0\le i\le N} \left(\frac{N}{M}\right)^i \frac{N!}{M^i(N-i)!}$$

Splitting the sum into two parts, etc., exactly as above, we can use the fact that terms in this sum begin to get negligibly small after $i > \sqrt{N}$ to prove that

$$2C_{MN} - 1 = \sum_{i\ge 0} \left(\frac{N}{M}\right)^i \left(1 + O\left(\frac{i^2}{N}\right)\right) = \frac{1}{1-\alpha} + O\left(\frac{1}{N(1-\alpha)^3}\right)$$

Thus, for fixed α, as N (and M) get large, the average successful search cost for linear probing is about $1/(1-\alpha)$.

10.4 LINEAR PROBING IN A FULL TABLE

The constants implicit in the O-notation in the equation above are independent of N and α, but the equation becomes meaningless as α gets close to 1. For simplicity, consider the case $N = M$, or

$$2C_{NN} - 1 = \sum_{0\le i\le N} \frac{N!}{N^i(N-i)!}$$

In this case, it will be convenient to rewrite the sum as

$$2C_{NN} - 1 = \sum_i \binom{N}{i} \frac{i!}{N^i}$$

The methods above could be used, even in this case, to eventually estimate the sum with an integral, but it will be instructive to consider a somewhat more direct method which makes use of the Γ-function, defined by

$$\Gamma(i+1) = \int_0^\infty e^{-t} t^i \, dt$$

This is a generalization of the factorial: it is not difficult to prove that $\Gamma(i + 1) = i\Gamma(i)$ and $\Gamma(i + 1) = i!$, using integration by parts. It is a generalization because it is defined even for noninteger i. This formula becomes directly useful in our problem if we take $t = uN$ so that

$$\frac{\Gamma(i+1)}{N^{i+1}} = \int_0^\infty e^{-Nu} u^i \, du$$

This is equal to $i!/N^{i+1}$ for integer i, so it can be substituted directly into our sum to yield

$$2C_{NN} - 1 = \sum_i \binom{N}{i} N \int_0^\infty e^{-Nu} u^i \, du$$

Now we can interchange the order of integration and summation to get

$$2C_{NN} - 1 = N \int_0^\infty e^{-Nu} (1+u)^N \, du$$

and we need only evaluate the integral. Knuth (1973a, Section 1.2.11.3) shows how to get an asymptotic value for the integral by using reversion of power series: if we change variables to $v = u - \ln(1 + u)$, so that $(1 + u)^N = e^{N(u-v)}$ and $du = [1 + (1/u)]dv$, then we can expand the logarithm to get

$$v = \tfrac{1}{2}u^2 - \tfrac{1}{3}u^3 + O(u^4)$$

and solve for u, "bootstrapping" one term at a time, to get

$$u = \sqrt{2v} + \tfrac{2}{3}v + O(v^{3/2})$$

and

$$1 + \frac{1}{u} = \frac{1}{\sqrt{2v}} + \frac{2}{3} + O(\sqrt{v})$$

Knuth gives details and further terms of these expansions, as well as a proof that it is valid to use them within the range of integration under consideration. Thus the integral is

$$2C_{NN} - 1 = N \int_0^\infty e^{-Nv} \left(\frac{1}{\sqrt{2v}} + \frac{2}{3} + O(\sqrt{v}) \right) dv$$

But now we can use the definition of the Γ function for $i = -\tfrac{1}{2}, 0, \tfrac{1}{2}$, to

get the result

$$2C_{NN} - 1 = N \left(\frac{\Gamma(\frac{1}{2})}{\sqrt{2N}} + \frac{2}{3}\frac{\Gamma(1)}{N} + O(N^{-3/2}) \right)$$

$$C_{NN} = \sqrt{\pi N/8} + \frac{5}{6} + O(N^{-1/2})$$

($\Gamma(1/2)$ is easily calculated from the normal distribution function.) This result can be extended to show that $C_{MN} = O(\sqrt{N})$ whenever $N > M - \beta$ for fixed β.

The method used here of converting a sum to an integral may seem quite mysterious at first (especially since we used the same formula to evaluate the integral); in the next chapter we see that it is a special case of a powerful general technique which is applicable to many problems.

11. Asymptotics in the complex plane

The analysis of many algorithms cannot be understood without resorting to complex numbers. In the simplest cases, roots of polynomials can be involved; in advanced cases, complex analysis is necessary to derive answers which could not otherwise be stated, much less derived. A substantial body of mathematics has been built up, mostly in other contexts, which applies directly to analytic problems associated with simple algorithms. A survey treatment of the "classical analysis" necessary to solve such problems would be far beyond the scope of these notes; nonmathematicians who are convinced by now of the potential utility of such analyses should first read Knopp (1945). Advanced material on some topics relevant to the analysis of algorithms may be found in Bender (1974) and DeBruijn (1958).

11.1 POLYPHASE MERGING

For our first example, we will consider the problem of sorting a file which is too large to fit in our computer's memory but which does fit on a magnetic tape. To sort the contents of a magnetic tape, we will need some auxiliary tapes; a good algorithm will minimize the amount of movement among tapes.

Several methods have been developed for this problem: they all begin by reading the input tape into memory and writing sorted "runs" of information out onto the auxiliary tapes. (The naive method for doing so would result in runs about the size of the internal memory; the actual method usually used manages to create runs about twice the size of the internal memory.) Then the runs are merged together in successive phases to make longer and longer runs, ultimately producing the sorted file.

Many different algorithms have been developed which work this way. A

fundamental method is the *polyphase merge* which works (after the runs have been distributed onto the auxiliary tapes in some "perfect" distribution) by taking one run from each tape, and merging them together to make a longer run on an output tape until one tape becomes empty: at this point the process is repeated, using the emptied tape as the new output tape. For example, suppose that three tapes are used and the first step distributes 34 sorted runs onto Tape 1 and 21 onto Tape 2. Then the polyphase "merge until empty" strategy produces a fully sorted file on Tape 2 as follows:

Tape 1	Tape 2	Tape 3	Total runs
34(1)	21(1)	0	55
13(1)	0	21(2)	34
0	13(3)	8(2)	21
8(5)	5(3)	0	13
3(5)	0	5(8)	8
0	3(13)	2(8)	5
2(21)	1(13)	0	3
1(21)	0	1(34)	2
0	1(55)	0	1

The numbers in the table are the number of runs on the tape; the numbers in parentheses are the sizes of the runs (relative to the size of the initial runs). For example, in the second phase, 13 (initial) runs from Tape 1 are merged with 13 of the 21 runs on Tape 3 (of relative size 2) to make 13 runs (or relative size 3) on Tape 2, leaving Tape 1 empty and 8 runs (of size 2) on Tape 3.

The question immediately arises: how many phases are required to merge together N initial runs? For the three-tape case, this is not a particularly difficult question (many readers may already have recognized the Fibonacci numbers), but if more than three tapes are used, the situation becomes more complicated and it is best to use complex analysis.

The merge pattern is easy to derive by working backwards. For example, the following table shows the pattern for four tapes (the table ignores tape labels and is skewed to make the pattern obvious).

1	1	0	0	0					
4		1	1	1	1				
7			2	2	2	1			
13				4	4	3	2		
25					8	7	6	4	
49						15	14	12	8

Each line in the table is obtained from the previous line by removing the leftmost entry, adding it to the others, then appending it to the right. It is easy to verify that "merge until empty", starting with these distribution patterns, will implement a polyphase merge.

Polyphase merging and similar methods lead to very interesting and intricate patterns which have been subjected to intensive study and analysis. For example, the problem of how to add "dummy" runs if the number of runs to be merged is not a perfect total appearing in the table has been studied in detail, with the surprising answer that many more dummy runs should be added than seems necessary (see Knuth, 1973b). Our purpose in discussing polyphase merging is to motivate a particular method of analysis, so we will concentrate exclusively on the simplest problem of determining how many phases are required to merge N runs together.

The analysis starts from the observation that each number in the table (including the totals) is the sum of the previous four along the same diagonal: in general, for m tapes, the total number of runs in an n-phase merge is given by the recurrence

$$t_n = t_{n-1} + t_{n-2} + \dots + t_{n-m+1} \quad \text{for} \quad n > 0$$

with

$$t_0 = t_{-1} = \dots = t_{2-m} = 1$$

This recurrence directly yields a closed-form expression for the generating function:

$$T_m(z) = \sum_{n>0} t_n z^n = \frac{(m-1)z + (m-2)z^2 + \dots + z^{m+1}}{1 - z - z^2 - \dots - z^{m-1}}$$

For example,

$$T_5(z) = \frac{4z + 3z^2 + 2z^3 + z^4}{1 - z - z^2 - z^3 - z^4}$$

and

$$T_3(z) = \frac{2z + z^2}{1 - z - z^2}$$

The standard technique for finding t_n is to expand these expressions for $T_m(z)$ in power series, then equate coefficients of t_n. This can be done by factoring the denominator and then using partial fractions, as illustrated below for $M = 3$.

$$T_3(z) = \frac{2z + z^2}{(1 - \phi z)(1 - \hat{\phi} z)}$$

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

$$= \frac{2 + z}{\sqrt{5}} \left(\sum_{n \geq 0} \phi^n z^n - \sum_{n \geq 0} \hat{\phi}^n z^n \right)$$

where

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

Therefore,

$$t_n = \frac{1}{\sqrt{5}} (2\phi^n + \phi^{n-1} - 2\hat{\phi}^n - \hat{\phi}^{n-1}) = \frac{\phi^{n+2}}{\sqrt{5}} + O(\hat{\phi}^n)$$

(Note that $\phi + 1 = \phi^2$, so $2\phi + 1 = \phi^3$, and that $|\hat{\phi}| < 1$, so that the contribution of $\hat{\phi}^n$ is negligible compared to that of ϕ^n. Now, if $t_n = N$, we can take logs to find that

$$\log_\phi N = (n + 1) + \log_\phi \left(\frac{1}{\sqrt{5}} \right) + \log_\phi \left(1 + O\left(\frac{\hat{\phi}}{\phi} \right)^n \right)$$

so that the number of phases required to merge together N runs is

$$\log_\phi N - \tfrac{1}{2} \log_\phi 5 - 1 + O(N^{-\varepsilon})$$

for a positive ε depending on ϕ and $\hat{\phi}$.

This technique generalizes. If the roots of $1 - z - z^2 - \dots - z^{m-1}$ are $\alpha_1, \alpha_2, \dots, \alpha_{m-1}$, then we can apply partial fractions to give

$$T_m(z) = h_1(z) \sum_{n \geq 0} \alpha_1^n z^n + h_2(z) \sum_{n \geq 0} \alpha_2^n z^n + \dots + h_{m-1}(z) \sum_{n \geq 0} \alpha_{m-1}^n z^n$$

where $h_i(z)$ are polynomials coming from the partial fractions calculation. It turns out that the inverse of one of the roots say α_1, always dominates the others in the same way that φ dominates $\hat{\varphi}$, even though some of the roots are complex (see Knuth, 1973b, for a proof). Therefore, as above, we eventually find that the number of phases required to merge N runs using M tapes is

$$\log_{\alpha_1} N - \log_{\alpha_1} h_1 \left(\frac{1}{\alpha_1} \right) + O(N^{-\varepsilon})$$

as above. Now, Knuth shows that

$$\alpha_1 = 2 - \frac{1}{2^M} + O\left(\frac{M}{4^M} \right)$$

as M grows, and that all the other roots are smaller. This gives enough information to calculate the leading term. Unfortunately, the next term is available with this method only through a laborious partial fractions calculation involving all the other roots. However, complex analysis provides a convenient method for performing this calculation.

The idea is to note that, as a function in the complex plane, the generating function $T_m(z)$ has a simple pole at $1/\alpha_1$: put another way, $(1-\alpha_1 z)T_m(z)$ converges for $|z| < R$ with $R > 1/|\alpha_1|$. Therefore,

$$T_m(z) = \frac{H}{1-\alpha_1 z} + \sum_{n>0} r_n z^n \quad \text{with} \quad |r_n| < R^{-n}$$

As above, this leads to the expression

$$\log_{\alpha_1} N + \log_{\alpha_1} H + O(N^{-\epsilon})$$

for the number of phases required to merge N runs. The constant H is the "residue" of $T_m(z)$ at $1/\alpha_1$; it can be calculated using l'Hospital's rule. If we denote the numerator of $T_m(z)$ by $p_m(z)$ and the denominator by $q_m(z)$, we have

$$H = \lim_{z\to 1/\alpha_1} (1-\alpha_1 z)T_m(z) = \lim_{z\to 1/\alpha_1} \frac{(1-\alpha_1 z)p_m(z)}{q_m(z)} = \frac{\alpha_1 p_m(1/\alpha_1)}{q'_m(1/\alpha_1)}.$$

This calculation is a simple example of a general technique that we will examine more closely below: a function in the complex plane is approximated by another that performs similarly near its singularities. Our next example shows a more complicated application of this method.

11.2 COUNTING ORDERED TREES

Suppose that we wish to count the number of ordered trees with N internal nodes. We know from the direct correspondence between binary trees and ordered trees that the answer is given by the Catalan numbers. This result can also be derived with a direct generating function argument. If $G(z) = \sum_{N\geq 0} g_N z^N$ is the generating function for the number of ordered trees with N internal nodes, then $G(z)^2$ is the generating function for internal nodes in two ordered trees, $G(z)^3$ for three ordered trees, etc. Since every ordered tree consists of a root with one son (which is an ordered tree) or with two sons (which are both ordered trees) or with three sons, etc., we are immediately led to the recurrence

$$G(z) = z(1 + G(z) + G(z)^2 + G(z)^3 + \ldots +) = \frac{z}{1-G(z)}.$$

This gives a quadratic equation with the solution

$$G(z) = \tfrac{1}{2}(1 \pm \sqrt{1-4z})$$

This is very similar to the generating function for binary trees: as before, we choose the root which yields the right small values; then expand using the binomial theorem; then approximate using Stirling's formula to get the result

$$g_{N+1} = \frac{4^N}{N\sqrt{\pi N}}\left(1 + O\left(\frac{1}{N}\right)\right)$$

But suppose we want to count the number of ordered trees with N *external* nodes. This is not a well-posed problem because, for example, there are an infinite number of ordered trees with only one external node (any number of unary nodes strung together). This anomaly is fixed by dissallowing one-way branching: how many ordered trees are there with no unary nodes and N external nodes? Arguing as above, we immediately get the generating function recurrence:

$$G(z) = z + G(z)^2 + G(z)^3 + \ldots = z + \frac{G(z)^2}{1-G(z)}$$

This leads to a different quadratic equation with the solution of interest

$$G(z) = \tfrac{1}{4}(1 + z - \sqrt{1-6z+z^2})$$

But now we cannot get anywhere by applying the binomial theorem, since it would lead back to a convolution sum to find g_N. To simplify notation, we will consider the asymptotics of $\sqrt{(1-w)(1-\alpha w)}$ for $\alpha < 1$: to find g_N we will need to take $w = (3+2\sqrt{2})z$ and $\alpha = (3-2\sqrt{2})/(3+2\sqrt{2})$.

Now, $\sqrt{(1-w)(1-\alpha w)}$ has an *algebraic* singularity at $w = 1$: it is not a pole, so we will not be able to approximate the function accurately with negative powers of $(1-w)$, as above. Nevertheless, we do expect the function to behave something like $\sqrt{(1-w)(1-\alpha)}$ at $w = 1$, and this suggests that we compute

$$\sqrt{(1-w)(1-\alpha w)} = \sqrt{(1-w)(1-\alpha)} + \sqrt{1-w}\,(\sqrt{(1-\alpha w)} - \sqrt{1-\alpha})$$
$$= \sqrt{(1-w)(1-\alpha)} + \frac{\alpha(1-w)^{3/2}}{\sqrt{1-\alpha w} + \sqrt{1-\alpha}}$$

The function $(\sqrt{1-\alpha w} + \sqrt{1-\alpha})^{-1}$ converges for $w > 1$; therefore, as we have argued before, it is the generating function for a sequence which is $O(r^{-N})$ for some $r > 1$. By the binomial theorem $(1-w)^{3/2}$ is the generating function for

$$(-1)^N \binom{3/2}{N} = O(N^{-5/2})$$

It remains only to convolve these two sequences: the coefficient of w^N in this convolution is

$$\begin{aligned}\sum_{0\le k\le N} O\left(\frac{k^{-5/2}}{r^{N-k}}\right) &= \sum_{0\le k\le N/2} O\left(\frac{k^{-5/2}}{r^{N-k}}\right) + \sum_{N/2\le k\le N} O\left(\frac{k^{-5/2}}{r^{N-k}}\right)\\ &= O(r^{-N/2}) + O(N^{-5/2}) \sum_{N/2\le k\le N} r^{k-N}\\ &= O(N^{-5/2})\end{aligned}$$

This completes the approximation:

$$\sqrt{(1-w)(1-\alpha w)} = \sum_{N\ge 0}\left(\sqrt{1-\alpha}\binom{\frac{1}{2}}{N}(-1)^N + O(N^{-5/2})\right)w^N$$

which eventually leads us to conclude that the number of ordered trees with N external nodes and no unary nodes is approximately $((3\sqrt{2}-4)/16\pi N^3)^{1/2}(3+2\sqrt{2})^n$.

11.3 METHOD OF DARBOUX

Again we were able to estimate a function by subtracting a function which behaves similarly near a singularity, then estimating the error. In this case, the error was not exponentially small (as it was in the case of poles) but merely slightly smaller than the leading term. It turns out that this general method applies to many generating functions. The technique is well known in analysis as the *method of Darboux*: if $G(z) = \sum_{N\ge 0} g_N z^N$ is analytic near α and has only an algebraic singularity of the form $(1 - z/\alpha)^{-w}h(z)$, then

$$g_N = \frac{h(\alpha)N^{w-1}}{\Gamma(w)\alpha^N} + O\left(\frac{N^{w-1}}{r^N}\right).$$

This formula yields directly the three asymptotic results derived above. A more general statement (when more singularities are involved) is given in Bender (1974). This method can give a quick solution to many problems. On the other hand, functions do arise in the analysis of simple algorithms which are too ill-behaved for this theorem to be useful. The general idea of studying the behavior of the function around its singularities is preserved, but much deeper arguments are required; for examples of this see Odlyzko (1979) and Flajolet and Odlyzko (1980).

11.4 MELLIN TRANSFORMS

In the previous section, we were unable to find the coefficient of the linear term in the analysis of Batcher's sort. This problem can be solved using the Mellin integral transform, a technique which has wide applicability in the analysis of algorithms. As we shall see, the answer thus derived has characteristics that make it plain that simpler methods will not work for this problem. Furthermore, several problems from a variety of applications exhibit similar characteristics. This method is outlined in detail in Knuth *et al*. (1969), Knuth (1973b), and Sedgewick (1978b), so we will only sketch it here.

We will pick up the derivation at the point at which we need to evaluate

$$B_N = \sum_{k\ge 1} \mathrm{e}^{-k^2/N}F(k)$$

The asymptotic derivation for linear probing in the previous section involved "transforming" a sum involving a factorial (the Γ-function) into an integral involving an exponential. For this problem, we do the opposite: change the sum involving the exponential above into an integral involving a Γ-function.

The specific tool we need to start is the Mellin inversion theorem: If f is piecewise continuous and

$$F(s) = \int_0^\infty x^{s-1}f(x)\,\mathrm{d}x$$

is absolutely convergent for $c_1 < \mathrm{Re}(s) < c_2$ then

$$f(x) = \frac{1}{2\pi i}\int_{\sigma-i\infty}^{\sigma+i\infty} x^{-s}F(s)\,\mathrm{d}s$$

for $c_1 < \sigma < c_2$. This is a special case of Fourier inversion (see Titschmarsh, 1951) which has applicability in analytic number theory and many other fields.

Taking $f(x) = \mathrm{e}^{-x}$ gives $F(s) = \Gamma(s)$ by the definition of the Γ-function (see the previous section), so the inversion theorem says that

$$\mathrm{e}^{-x} = \frac{1}{2\pi i}\int_{\sigma-i\infty}^{\sigma+i\infty} x^{-s}\Gamma(s)\,\mathrm{d}s \quad \text{for} \quad \sigma > 0.$$

This formula may be proved independently by noting that the value of the integral is very closely approximated by the sum of the residues to the left of the line of integration because the Γ function is very small for arguments at the fringes of the left half plane. (See Knuth, 1973b for a detailed proof of this: the line integral is approximated by a contour integral which encloses the left half plane in the limit, and the contributions of the other parts of the contour are shown to vanish.) The Γ function has a pole at each nonpositive integer (see Whittaker and Watson, 1927, for properties of the Γ function); the residue at $-j$ is $(-1)^j/j!$. Therefore, the sum of the residues to the left of the line of integration in the integral above is

$$\sum_{j\geq 0} \frac{x^j(-1)^j}{j!} = e^{-x},$$

verifying the relationship between $\Gamma(s)$ and e^x implied by Mellin inversion.

To use Mellin inversion for our sum, we simply "transform" the exponential in our sum,

$$B_N = \sum_{k\geq 1} \frac{1}{2\pi i}\int_{\sigma-i\infty}^{\sigma+i\infty} \left(\frac{k^2}{N}\right)^{-s}\Gamma(s)\,ds\,F(k)$$

then interchange the order of summation and integration to get

$$B_N = \frac{1}{2\pi i}\int_{\sigma-i\infty}^{\sigma+i\infty}\left(\sum_{k\geq 1}\frac{F(k)}{k^{2s}}\right)N^s\Gamma(s)\,ds$$

then "transform" back to get the functional form of B_N. Note that the interchange of integration and summation is allowed only if

$$\sum_{k\geq 1}\frac{F(k)}{k^{2s}}$$

is defined on the path of integration. This depends both on $F(k)$ and σ, and plays an important role in the derivation, as we will see in the examples below. Moreover, we need to be able to study properties of this function in the complex plane. Fortunately, for many examples which arise in the analysis of algorithms, this function can be expressed in terms of classical special functions.

To begin, we will consider an example for which we already know the answer: if we take $F(k) = k^2$, then, as we saw in Chapter 9, B_N is the number of inversions in a 2-ordered permutation. We have

$$\sum_{k\geq 1}\frac{F(k)}{k^{2s}} = \sum_{k\geq 1}\frac{1}{k^{2s-2}} \equiv \zeta(2s-2)$$

the well known Riemann ζ-function (see, for example, Abramowitz and Stegun, 1972). The sum converges for $s > 3/2$, so we have from above

$$B_N = \frac{1}{2\pi i}\int_{\sigma-i\infty}^{\sigma+i\infty}\zeta(2s-2)N^s\Gamma(s)\,dx \quad \text{for} \quad \sigma > 3/2$$

Now we can evaluate this integral using a contour integration argument

similar to the proof of the Mellin inversion formula described above. Consider

$$\int_R \zeta(2s-2)N^s\Gamma(s)\,ds$$

where R is a rectangle comprised of two long vertical lines at $\mathrm{Re}(s) = 2$ and $\mathrm{Re}(s) = 1$ and two short horizontal lines connecting them, say at $\mathrm{Im}(s) = \pm M$. The Γ-function becomes exponentially small in the complex part of its argument (see Whittaker and Watson, 1927, for specific bounds), so it is possible to prove, by taking limits as M goes to infinity, that the leading aysmptotic term in the value of the integral is the sum of the residues in the area between $\mathrm{Re}(s) = 1$ and $\mathrm{Re}(s) = 2$. There is only one pole in this strip, at $s = 3/2$, so this leads to a short proof that

$$B_N \approx N^{3/2}\Gamma(3/2) = \sqrt{\pi N^3}/4$$

This might be too powerful a tool for this simple problem, but there are many problems for which Mellin transforms are applicable although more traditional methods fail. Solutions for these problems follow the same general structure as the argument above; significant differences show up in the type of convex integration which must be performed.

For example, in the analysis of radix-exchange sorting, Knuth (1973b) uses the general method above to derive the following integral:

$$B_N = \frac{1}{2\pi i}\int_{-3/2-i\infty}^{-3/2+i\infty}\frac{1}{2^{-1-s}-1}N^{-s}\Gamma(s)\,ds$$

This integral can be evaluated with residues as above, but a double pole at $s = 1$ is involved, as well as a series of new poles along the line at $\mathrm{Re}(s) = -1$. (For details see Knuth 1973b.) Derivations involving similar functions, with similar complications, may be found in the study of merging networks (Sedgewick, 1978b; Flajolet *et al.*, 1977) and in many other applications. The solutions of these problems take on a complicated form depending on the residue structure of the functions: they are unlikely to be accessible with more elementary techniques. Moreover, as we have seen, Mellin transforms do work properly for some simple problems, so there is some possibility that a general method of solution of a wide class of recurrences based on the Mellin transform could be developed.

12. Probabilistic models

Many algorithms readily admit natural input models. It is quite reasonable to analyze the cost of sorting a randomly ordered file, or searching for a random element, or finding the greatest common divisor of two randomly selected integers. Even for such algorithms, however, it is difficult to be confident that the model used for the analysis reflects reality sufficiently well to allow use of the analysis to predict the performance of algorithms in actual applications. Moreover, sometimes several different models suggest themselves. And for many problems, no "natural" input model is apparent or even likely to exist. In this chapter, we consider some of the difficulties inherent in analyzing algorithms in the face of such uncertainties.

Such difficulties can arise even in more "classical" analyses like those that we have been considering. For example, in our examination of the Quicksort algorithm, we were careful to note that the partitioning process preserves randomness in the sense that it produces random subfiles if used on a random file. Some variants of Quicksort do not have this property, and there are many other algorithms which operate on random inputs in a controlled way that (incidental to the algorithm) destroys the randomness. Some examples of algorithms in this class are balanced tree algorithms for searching (see Guibas and Sedgewick, 1978), and Heapsort (Sedgewick, 1982b). The analysis of such algorithms cannot take advantage of the natural input model, and is thus similar to the analysis of algorithms in the absence of a good input model.

When several different input models are available, how is one to choose among them? Different models may be appropriate for different situations, but the analyst is basically forced to trade off between the ease of analysis and the degree to which the model reflects reality. A description which purports to model the actual expected inputs will be of little us if it is too complicated to admit any analysis, and analytic results on simple artificial models are of limited utility. As we have seen, even very simple algorithms

can require quite sophisticated analysis, so it is reasonable to first find a model in which the analysis is tractable, then extend the analysis to more complicated (and more realistic) models.

One approach that is useful when average-case analysis is difficult or impossible is to concentrate on best-case and worst-case performance. This analysis might be easier, less sensitive to assumptions about the input, and still indicative of the performance of the algorithm. For example, it is not difficult to estimate the worst-case behavior of both Heapsort and balanced tree algorithms, and to prove that the average-case performance is within a constant factor of the worst-case. On the other hand, the worst-case performance may be misleading and indicative only of how the algorithm performs for a very few artificially constructed inputs that would never occur in practice. In fact, it is sometimes possible to prove that this situation exists, and take advantage of it to make precise statements about the average behavior of algorithms whose performance characteristics are about the same for virtually all inputs.

Below we will examine some of these issues by studying some algorithms for simple set maintenance. Several variants on a trivial algorithm have been proposed for this problem, and it is not clear which is best. Also, several models have been proposed for the input, and it is not clear which best reflects actual situations in which the algorithms are used.

12.1 UNION-FIND ALGORITHMS

The *union-find* problem involves processing a set of equivalence relations while maintaining an internal data structure which allows testing whether two given items are equivalent. The algorithm is to process two kinds of requests, *union* (x, y) *for* $x \equiv y$, and *find*(x), which is to return the "name" of the equivalence class containing x so that one can test whether x_1 and x_2 are in the same equivalence class by checking whether *find*(x_1) = *find*(x_2). For the purposes of analysis we will assume that the elements being processed are the integers 1 to N; in a practical implementation, hashing would be used to convert arbitrary names to this form.

One simple solution to this problem is to maintain an **array** $a[1\,..\,N]$ containing the name of the equivalence class of each element: to process *union*(x, y), simply give all elements in x's class the name of y's class. Equivalence class "names" are simply element indices, so we start with $a[i] = i$ to indicate that each element is in its own class. The table below shows how the array changes as the equivalence relations at the left hand side are processed.

	1	2	3	4	5	6	7	8	9	10	11	12	13
4 ≡ 9	1	2	3	9	5	6	7	8	9	10	11	12	13
2 ≡ 11	1	11	3	9	5	6	7	8	9	10	11	12	13
7 ≡ 3	1	11	3	9	5	6	3	8	9	10	11	12	13
6 ≡ 3	1	11	3	9	5	3	3	8	9	10	11	12	13
1 ≡ 5	5	11	3	9	5	3	3	8	9	10	11	12	13
11 ≡ 12	5	12	3	9	5	3	3	8	9	10	12	12	13
3 ≡ 13	5	12	13	9	5	13	13	8	9	10	12	12	13
9 ≡ 10	5	12	13	10	5	13	13	8	10	10	12	12	13
2 ≡ 3	5	13	13	10	5	13	13	8	10	10	13	13	13
4 ≡ 8	5	13	13	8	5	13	13	8	8	8	13	13	13
3 ≡ 5	5	5	5	8	5	5	5	8	8	8	5	5	5
9 ≡ 3	5	5	5	5	5	5	5	5	5	5	5	5	5

This algorithm was called the *quick-find* algorithm by Yao (1976), because the cost of a *find* operation is constant (the time required to access the array). The "variable" in the running time of this algorithm is the number of elements processed during a *union*. In the worst case, this could be large, since it is proportional to the size of one of the sets being merged.

One way to reduce the cost of *union* operations, called the *weighted quick-find* algorithm by Yao, is to change names in the smaller of the two classes to be merged. This requires maintaining an array of weights containing the number of elements in each equivalence class. The table below shows the operation of this version of the algorithm on our example.

	1	2	3	4	5	6	7	8	9	10	11	12	13
4 ≡ 9	1	2	3	9	5	6	7	8	9	10	11	12	13
2 ≡ 11	1	11	3	9	5	6	7	8	9	10	11	12	13
7 ≡ 3	1	11	3	9	5	6	3	8	9	10	11	12	13
6 ≡ 3	1	11	3	9	5	3	3	8	9	10	11	12	13
1 ≡ 5	5	11	3	9	5	3	3	8	9	10	11	12	13
11 ≡ 12	5	11	3	9	5	3	3	8	9	10	11	11	13
3 ≡ 13	5	11	3	9	5	3	3	8	9	10	11	11	3
9 ≡ 10	5	11	3	9	5	3	3	8	9	9	11	11	3
2 ≡ 3	5	3	3	9	5	3	3	8	9	9	3	3	3
4 ≡ 8	5	3	3	9	5	3	3	9	9	9	3	3	3
3 ≡ 5	3	3	3	9	3	3	3	9	9	9	3	3	3
9 ≡ 3	3	3	3	3	3	3	3	3	3	3	3	3	3

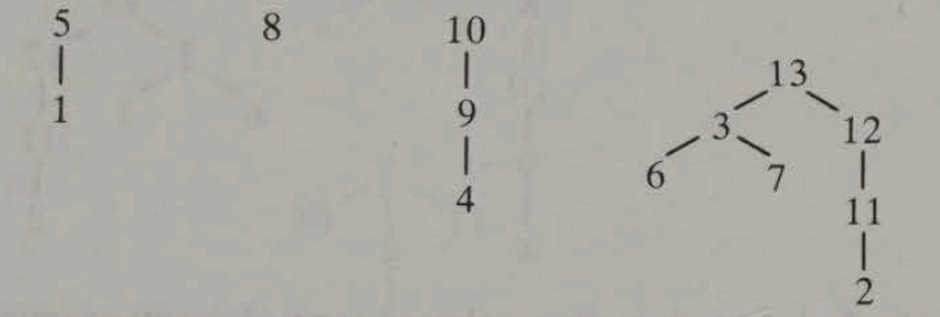
In this example only 18 names are changed during all *unions*, while the unweighted algorithm does 27 name changes.

An alternate approach is the *quick-union* algorithm (again using Yao's terminology) in which the *union* operation is done efficiently, but some

finds could be expensive. The method is to weaken the requirement that $a[i]$ contain the "name" of the set containing element i by having it contain the index of some other element in the same set, with no cycles allowed. Thus, each set is represented by a tree, which is stored in the *a* array. For example, the configuration

i	1	2	3	4	5	6	7	8	9	10	11	12	13
$\alpha[i]$	5	11	13	9	5	3	3	8	10	10	12	13	13

is an array representation of the forest



which means that the equivalence classes are {1, 5}, {2, 3, 6, 7, 11, 12, 13}, {8} and {4, 9, 10}. Note that an element i is at the root of a tree if and only if $a[i] = i$: these elements are the obvious choices for the representatives of the equivalence classes.

Now *find* is easily implemented by tracing back through the *a* array until a root element is found:

```
j := x; while α[j] <> j do j := α[j];
```

For *union* (x, y) the same method is used to find the roots of the trees containing x and y, and then one is set to point to the other:

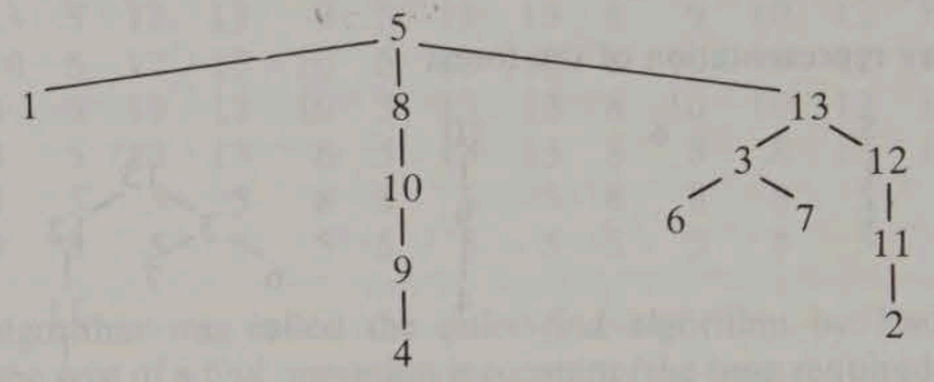
```
j := x, while α[j] < > j do j : = α[j];
i := y; while α[i] <> i do i := α[i];
if i <> j then α[j] := i;
```

The following table shows the execution of this quick-union algorithm on our example.

	1	2	3	4	5	6	7	8	9	10	11	12	13
4 ≡ 9	1	2	3	9	5	6	7	8	9	10	11	12	13
2 ≡ 11	1	11	3	9	5	6	7	8	9	10	11	12	13
7 ≡ 3	1	11	3	9	5	6	3	8	9	10	11	12	13
6 ≡ 3	1	11	3	9	5	3	3	8	9	10	11	12	13
1 ≡ 5	5	11	3	9	5	3	3	8	9	10	11	12	13
11 ≡ 12	5	11	3	9	5	3	3.	8	9	10	12	12	13
3 ≡ 13	5	11	13	9	5	3	3	8	9	10	12	12	13
9 ≡ 10	5	11	13	9	5	3	3	8	10	10	12	12	13

$2\equiv3$	5	11	13	9	5	3	3	8	10	10	12	13	13
$4\equiv8$	5	11	13	9	5	3	3	8	10	8	12	13	13
$3\equiv5$	5	11	13	9	5	3	3	8	10	8	12	13	5
$9\equiv3$	5	11	13	9	5	3	3	5	10	8	12	13	5

This produces the following tree which shows the equivalence class containing all the elements.



The cost of a *find* operation, the number of array elements examined, is the rank of x in the tree, and the cost of *union(x, y)* is the sum of the ranks of x and y in their trees.

The quick-union algorithm can build unbalanced trees, so there is a corresponding weighted quick-union algorithm that balances the trees by always making the root of the "light" tree point to the root of the "heavy" tree when the union is performed

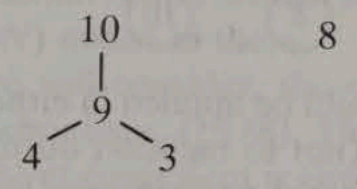
	1	2	3	4	5	6	7	8	9	10	11	12	13
$4\equiv9$	1	2	3	9	5	6	7	8	9	10	11	12	13
$2\equiv11$	1	11	3	9	5	6	7	8	9	10	11	12	13
$7\equiv3$	1	11	3	9	5	6	3	8	9	10	11	12	13
$6\equiv3$	1	11	3	9	5	3	3	8	9	10	11	12	13
$1\equiv5$	5	11	3	9	5	3	3	8	9	10	11	12	13
$11\equiv12$	5	11	3	9	5	3	3	8	9	10	11	11	13
$3\equiv13$	5	11	3	9	5	3	3	8	9	10	11	11	3
$9\equiv10$	5	11	3	9	5	3	3	8	9	9	11	11	3
$2\equiv3$	5	11	3	9	5	3	3	8	9	9	3	11	3
$4\equiv8$	5	11	3	9	5	3	3	9	9	9	3	11	3
$3\equiv5$	5	11	3	9	3	3	3	9	9	9	3	11	3
$9\equiv3$	5	11	3	9	3	3	3	9	3	9	3	11	3

This is implemented by replacing the last line of the *union* code above by

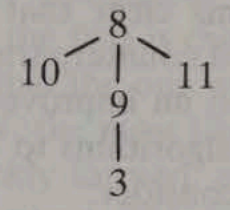
```
if i<>j then
if w[j] > w[i] then α[i] := j else α[j] := i;
```

This method clearly produces "flatter" trees than the unweighted method.

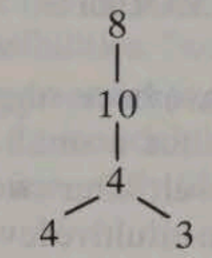
Still, none of the above methods are entirely satisfactory: the quick-union and the quick-find methods can be slow, and the weighted methods require extra storage. *Path compression* is a technique which improves the speed without requiring extra storage: after doing a quick-union, make all the nodes on the paths just traversed point to the new root. For example, the equivalence $4 \equiv 8$ transforms



to



Not to

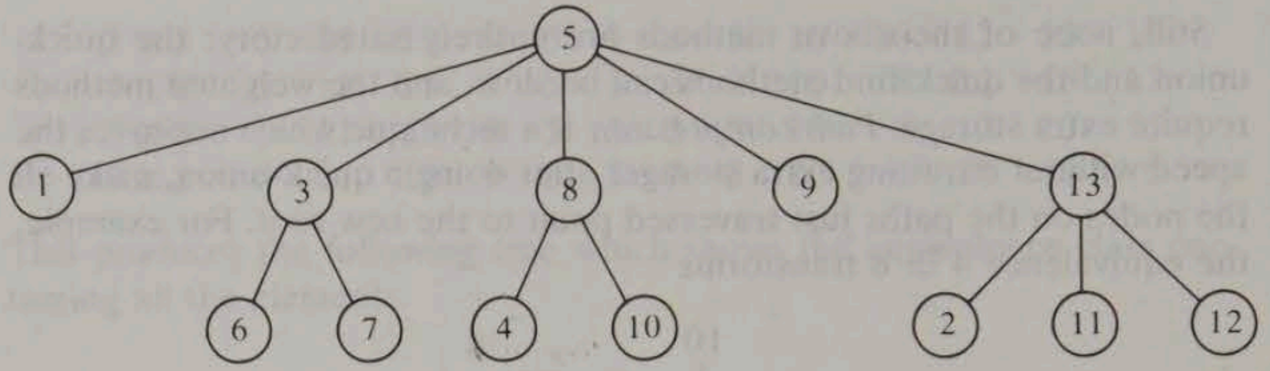


In our example, there is no difference from *quick-union* until the last four equivalences:

	1	2	3	4	5	6	7	8	9	10	11	12	13
		⋮											
$2\equiv3$	5	13	13	9	5	3	3	8	10	10	13	13	13
$4\equiv8$	5	13	13	8	5	3	3	8	8	8	13	13	13
$3\equiv5$	5	13	5	8	5	3	3	8	8	8	13	13	5
$9\equiv3$	5	13	5	8	5	3	3	5	5	8	13	13	5

The final tree is better even than that produced by the weighted algorithm, but still not perfecly balanced:
The cost of the *union* operation is multiplied by a constant factor, but the requirement for extra storage is eliminated.

The path compression rule can also be applied to the weighted quick-



union algorithm, and it could be applied in either algorithm before or after the actual union is done (not to mention during *finds*). These options all lead to methods with different performance characteristics.

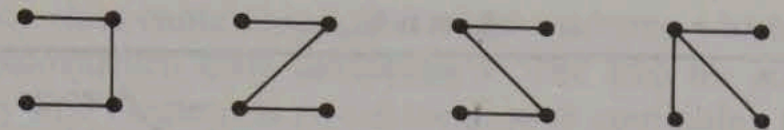
The algorithms above are all quite simple and easily coded, but they perform quite differently. It seems clear that the weighted algorithms improve upon their straightforward counterparts, but by how much? And to what extent is path compression an improvement? Unfortunately, not only do we have several possible algorithms to analyze, but also we have several different input models to consider.

12.2 MODELS

To contrast the three models that have been suggested for this problem, we will calculate a precise probability for a small case (the probability, for $N = 4$, that two sets of size two result after two relations have been processed). Also, we will try to see, on an intuitive level, what happens within the models for large N, just before the last few relations which bring together all the elements into one equivalence class are processed. In fact, a goal of the analysis will be to quantify these intuitive statements.

The simplest model is the *random sets* model, used by Doyle and Rivest (1976). Here we consider each pair of classes so far formed to be equally likely to be merged (regardless of size). For N = 4, we have three classes after the first relation, one of size two and two of size one. The probability that any two of these will be merged, in particular the singleton classes, is 1/3. For large N, we have sets of all different sizes, from $O(1)$ to $O(N)$, all equally likely to be merged. This model is generally viewed as the most unrealistic for most applications.

The next model, due to Yao (1976), is called the *random tree* model. Here we consider each way of merging the N items together with $N - 1$ equivalence operations to be equally likely. If we think of the elements as nodes in a graph and the equivalence relations as edges, then a set of edges which merges all the nodes together is a *spanning tree* for the graph. In this model, we first take a spanning tree at random, then use the edges from



that spanning tree in random order. In the fourth tree, the probability that we get two sets of size two after using two edges i 0; in the other three it is 1/3, so the unconditional probability is $(3/4)(1/3) + (1/4)0 = 1/4$. For large N, the most likely situation is that the last few edges will join together large components with $O(N)$ nodes in them.

The third model that we will consider, the *random graph* model, was analyzed by Knuth and Schonhage (1978). Here we suppose that each edge between elements not yet equivalent is equally likely to occur. This is the same as adding random edges to the graph but ignoring those connecting parts that are already connected. For $N = 4$, there are six different edges, with five left after the first is chosen. These occur with equal likelihood, so the probability that the one which connects the two singletons is chosen is 1/5. For large N, the most likely situation is that a large component of size $O(N)$ is quickly formed, and the last few edges just connect single nodes to this component. This model is generally thought to be the most realistic, although, as mentioned above, it is arguable how well actual applications match any of the models.

There are still other possibilities. For example, it is reasonable to do path compression even for calls to *union(x,y)* for which x and y are found to be equivalent. What is needed to model this situation is a *random edge* model, in which each pair $x \equiv y$ is equally likely, independent of the past.

12.3 SUMMARY OF RESULTS

Table 12.1 gives the leading term for the total running time (number of times the a array is accessed) for the various equivalence algorithms in the worst case and in the average case for the three models described above. We will look below at the analyses that produced most of these. With a few very significant exceptions, the constant implied by the O-notation is simply too complicated to fit in the table, but could be worked out. The function $\alpha(N)$ on the last line is an extremely slowly growing function which is constant for practical purposes (see below).

12.4 RANDOM SETS MODEL

The analysis for the quick-find algorithm under the random sets model is the easiest. Let C_N be the total average cost of doing $N - 1$ *union* operations-

TABLE 12.1

		Average		
	Worst	*Sets*	*Tree*	*Graph*
Quick find	$\frac{N^2}{2}$	$N \ln N$	$N\sqrt{\pi N/8}$	$\frac{N^2}{8}$
Weighted	$N \lg N$	$\frac{1}{2}N \ln N$	$\frac{1}{\pi} N \ln N$	$O(N)$
Quick union	$\frac{N^2}{2}$	$\left(\frac{\pi^2}{3} - 2\right)$N	$\frac{1}{4}N \ln N$	$O(N^2)$
Weighted	$N \lg N$	$O(N)$	$O(N)$	$O(N)$
Compression	$N \lg N$	$O(N)$	?	?
Both	$O(N\alpha(N))$	$O(N)$	$O(N)$	$O(N)$

tions (to form a set of size N). If the last *union* causes sets of size k and $N - k$ to be merged then the total cost is $k + C_k + C_{N-k}$. But the essence of the random sets model is that each value of k is equally likely to occur. This leads to the recurrence

$$C_N = \sum_{1 \le k < N} \frac{1}{N-1}(k + C_k + C_{N-k}),$$

which simplifies to a familiar recurrence (see Chapter 7)

$$C_{N+1} = \frac{N+1}{2} + \frac{2}{N}\sum_{1 \le k \le N} C_k.$$

This is only slightly different from the recurrence arising in the study of Quicksort and binary search trees. The solution is

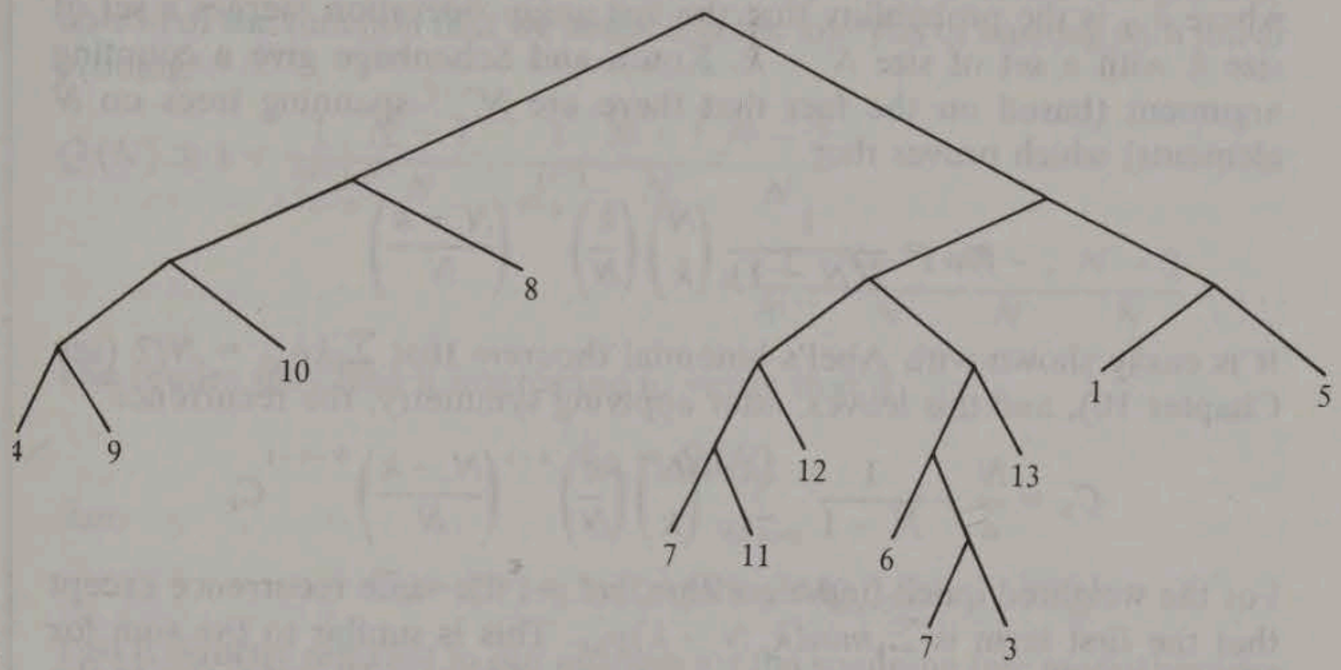
$$C_N = N(H_N - 1)$$

Analysis of the weighted version involves solving a similar recurrence,

$$C_N = \sum_{1} \le k < N \frac{1}{N-1}(min(k, N-k) + C_k + C_{N-k})$$

using precisely the same techniques.

Knuth and Schonage (1978) show how to make the correspondence with binary search trees even more explicit by defining a structure called the *union tree*, which is a full description of a sequence of *union* operations. Defined recursively, the union tree for a sequence of equivalence instructions ending with $x \equiv y$ consists of a left subtree which is the union tree for the equivalence class containing x and a right subtree which is the union tree for the equivalence class containing y. The tree for an equivalence class with only one element is an external node containing that element. The tree for the sequence of instructions used in all the examples above is drawn below.



The correspondence with binary search trees is easy to see from this construction. The dynamic process is the same, but run backwards. After the first *union*, we have a forest consisting of $N - 2$ singleton external nodes and one internal node with two external sons. A full tree is eventually built upon these $N - 1$ nodes. If we look at the process backwards we can see that a tree of N nodes comes from replacing an external node by an internal node with two external sons in a tree of $N - 1$ nodes. This is exactly the way that binary search trees are built.

In the union tree, the cost of the last *union* for the quick-find algorithm is the number of nodes in the left subtree (or in the smaller of the two subtrees for the weighted version). For the quick-union algorithm, and its weighted version, the costs are also available as simple properties of the tree, leading to recurrences similar to those above, but which relate the costs of quick-union and quick-find algorithms (see Knuth and Schonhage, 1978). Thus, we will consider only the analyses for quick-find methods under the random spanning tree and random graph models below.

12.5 RANDOM SPANNING TREE MODEL

Much of the machinery above can be applied to the analysis for the random spanning tree model, though the resulting recurrences are much more complicated. The same argument as above shows that the average cost of the quick-find algorithm under this model satisfies the recurrence

$$C_N = \sum_{0<k<N} p_{Nk}(k + C_k + C_{N-k})$$

where p_{Nk} is the probability that the last *union* operation merges a set of size k with a set of size $N - k$. Knuth and Schonhage give a counting argument (based on the fact that there are N^{N-2} spanning trees on N elements) which proves that

$$p_{Nk} = \frac{1}{2(N-1)}\binom{N}{k}\left(\frac{k}{N}\right)^{k-1}\left(\frac{N-k}{N}\right)^{N-k-1}$$

It is easily shown with Abel's binomial theorem that $\Sigma_k k p_{Nk} = N/2$ (see Chapter 10), and this leaves, after applying symmetry, the recurrence

$$C_N = \frac{N}{2} + \frac{1}{N-1}\sum_{0<k<N}\binom{N}{k}\left(\frac{k}{N}\right)^{k-1}\left(\frac{N-k}{N}\right)^{N-k-1} C_k$$

For the weighted quick-find algorithm, we get the same recurrence except that the first term is $\Sigma_k min(k, N-k)p_{Nk}$. This is similar to the sum for hashing with linear probing that we solved in Chapter 10 and yields to the same techniques: the value is $\sqrt{2N/\pi} + O(1)$.

To finish, then, we need to solve recurrences of the form

$$C_N = X_N + \frac{1}{N-1}\sum_{0<k<N}\binom{N}{k}\left(\frac{k}{N}\right)^{k-1}\left(\frac{N-k}{N}\right)^{N-k-1} C_k$$

for $X_N = N, \sqrt{N}$. The recurrence becomes somewhat less formidable when both sides are multiplied by $(N-1)N^{N-1}$ and divided by $N!$:

$$\frac{(N-1)N^{N-1}}{N!}C_N = \frac{(N-1)N^{N-1}}{N!}X_N + N\sum_{0<k<N}\frac{k^{k-1}}{k!}C_k\frac{(N-k)^{N-k-1}}{(N-k)!}$$

From this it can be proven by appealing to Abel's binomial theorem that if

$$C_N = \frac{N-1}{M}\left(\frac{N-2}{N}\cdots\frac{N-M}{N}\right)$$

then

$$X_N = \frac{N-2}{N}\cdots\frac{N-M}{N}$$

for any positive M. A motivation for this solution might be that Abel's theorem provides a way to evaluate sums of the type that appear in the recurrence, so we may as well work backwards, using the theorem to

evaluate a general version of the sum for a particular C_N, then solve for X_N. (Knuth and Schonhage give another motivation, based on generating functions.) The result is a family of solutions (parameterized by M) to the recurrence, which leads to a general solution involving linear combinations of members of this family. This solution is expressed in terms of a generalization of the function that we studied in the analysis of hashing with linear probing:

$$Q_i(N) \equiv 1 + \frac{1}{2^{i-1}}\frac{N-1}{N} + \frac{1}{3^{i-1}}\frac{N-1}{N}\frac{N-2}{N} + \frac{1}{4^{i-1}}\frac{N-1}{N}\frac{N-2}{N}\frac{N-3}{N} + \cdots$$

The reader may find it interesting to verify that if

$$X_N = Q_i(N)$$

then

$$C_N = (N-1)Q_{i+1}(N) + NQ_{i+2}(N)$$

This is directly relevant to our analysis for the spanning tree model because $Q_0(N) = N$ (this is a subtle fact which the reader should check). We have proved that

$$C_N = 1/2((N-1)Q_1(N) + NQ_2(N))$$

Now, $Q_1(N)$ is none other than the function that we encountered in Chapter 10:

$$Q_1(N) = \sum_{1\le i\le N}\frac{N!}{N^i(N-i)!} = \sqrt{\pi N/2} - \frac{1}{3} + O(N^{-1/2})$$

Furthermore, we have the simple bound

$$Q_i(N) < \sum_{1\le j\le N}\frac{1}{k^{i-1}}$$

which implies, for example, that $Q_2(N) = O(\log N)$ and $Q_3(N) = O(1)$. Putting these results together completes our analysis for the unweighted algorithm:

$$C_N = \sqrt{\pi N^3/8}$$

The analysis for the weighted algorithm proceeds along very similar lines: the solution in this case is

$$\sqrt{2/\pi}((N-1)Q_2(N) + NQ_3(N))$$

The details of working out a more precise asymptotic value for $Q_2(N)$ are left as an interesting exercise for the reader (see Kruskal 1954).

12.6 RANDOM GRAPH MODEL

The full analysis of the performance of the *union* and *find* programs under the random graph model given by Knuth and Schonhage (1978) involves intricate arguments which we do not have room to consider in detail here. It is an asymptotic analysis similar in spirit to those we have been considering, but much more complicated.

As mentioned above, the most likely outcome when edges are added at random to a set of vertices is that one large component is quickly formed, with the last few edges connecting the remaining few single vertices to the large component. This scenario is a classical result proved by Erdos and Renyi (1959). A quick look at the major steps of their proof will not only provide some insights into the behavior of random graphs (and the performance of *union-find* algorithms) but also will illustrate an important method of algorithmic analysis which is applicable to many difficult problems.

There are $\binom{N}{2}$ possible edges connecting N vertices. By a "random graph" we mean one that is formed by adding edges to the vertices in random order (with all $\binom{N}{2}!$ orderings equally likely). How many edges must be added before the graph becomes connected? For the *union-find* algorithms that we have discussed, this point is moot, because we ignored edges within components already connected. However, suppose one wanted to write a program to check the analytic results presented in Table 12.1 by generating random edges, doing a *find* to discover if each should be ignored, then doing a *union* if not. This result will tell us how long such a program will take to complete.

The Erdos–Renyi proof has three parts. First, they show that, after a certain number of edges are added, the probability that there is one very large connected component with at least $N - \log\log(N)$ edges is $1 - O(1/\log(N))$. Next, they show that the $\log\log N$ remaining vertices are likely to be isolated (with no edges between them) with even higher probability. Finally they calculate the probability that a graph has no isolated vertices. This is asymptotically the same as the probability that a graph is connected, but it is much easier to calculate. (Graphs which are not connected are nearly certain to consist of one large connected component plus some small number of isolated vertices.)

Suppose that C edges have been chosen. The number of graphs with no isolated points can be counted using the principle of inclusion–exclusion. This leads to the following formula for the probability that no isolated points occur:

$$\sum_{0\le k\le N}(-1)^k\binom{N}{k}\frac{\left[\binom{N-k}{2}\atop C\right]}{\left[\binom{N}{2}\atop C\right]}$$

The crux of the proof is to use Stirling's formula to show that if C is about $\frac{1}{2}N\ln N + cN$ for some constant c then this sum is approximated by

$$\sum_{k\ge 0}(-1)^k\frac{e^{-ck}}{k!} = e^{-e^{-c}}$$

The first two parts of the proof use similar, but more complicated, counting arguments and asymptotic estimates, with the final result that, as $N\to\infty$, the probability that more than $1/2N\ln N + cN$ edges are needed before the graph becomes connected approaches $1 - e^{-e^{-c}}$. This is close to zero for only moderately large c: for example $e^{-e^{-10}} = 0.99995\ldots$

This proof is an example of the *probabilistic* method of algorithmic analysis: when we can prove that an algorithm is nearly certain to behave in a particular way, then by assuming that it does so we are sometimes led to an easier way to analyze its performance. Of course, the challenge in such a proof is to be sufficiently precise about the likely behavior of the algorithm to be able to make sharp asymptotic estimates: the above proof depends on the discovery of the $1/2N\ln N + cN$ cut-off point. Nevertheless, this is a powerful general technique that must be considered for algorithms which we cannot analyze directly but for whose performance we have some suspicion of a precise description. For example, Guibas and Szemeredi (1978) use this method to analyze the double hashing method that was described briefly in Chapter 10.

From this result we can get some intuitive feeling about the behavior of the *union-find* algorithms under the random graph model. The last few edges connect isolated vertices to a large connected component with about N nodes. For the unweighted algorithm, this will cost $O(N)$ on the average, while for the weighted algorithm the cost will be only $O(1)$ The challenge in the Knuth–Schonhage proof is to show that these performance characteristics are maintained throughout the execution of the algorithms so that the total cost of the unweighted algorithm is $O(N^2)$ while the weighted algorithm is linear. This turns out to be particularly difficult for the unweighted algorithm: a delicate asymptotic argument is used to prove that the cost is $O(N)$, but the constant of proportionality is not known.

12.7 OTHER MODELS

Despite the fact that the various *union-find* algorithms and probabilistic models provide an interesting review of many of the techniques of algorithmic analysis that we have discussed, it must be pointed out that the analysis of the algorithms of greatest practical interest (those using path compression) still remains open. What is worse, it is apparent that none of the models examined above may be appropriate for this problem, because a practical implementation would do path compression in the *find* routine, so that commands requesting that two already connected elements be made equivalent will have an effect. This is an interesting open problem, but it also makes a sobering concluding comment on the importance of the proper choice of model in the analysis of algorithms.

REFERENCES

ABRAMOWITZ, M. and STEGUN, I. A. (1972). "Handbook of Mathematical Functions". Dover, New York.

BENDER, E. A. (1974). Asymptotic methods in enumeration. *SIAM Review*, **16**, (Oct).

DEBRUIJN, N. G. (1958). "Asymptotic Methods in Analysis". North-Holland Publishing Co., Amsterdam.

DOYLE, J. and RIVEST, R. (1976). Linear expected time of a simple union-find algorithm. *Information Processing Letters*, **5**.

ERDOS, P. and RENYI, A. (1959). On random graphs, I. *Publicationes Mathematicae*, **6**.

FLAJOLET, P., RAOULT, J. C. and VUILLEMIN, J. (1977). On the Average Number of Registers Required for Evaluating Arithmetic Expressions. Proc. 18th Ann. Symp. Fnds. Comp. Sci.

FLAJOLET, P. and ODLYZKO, A. (1980). The Average Height of Binary Trees and Other Simple Trees. Proc. 21st Ann. Symp. Fnds. Comp. Sci.

MATHLABGROUP (1977). MACSYMA Reference Manual, Laboratory for Computer Sciences, MIT.

GUIBAS, L. and SEDGEWICK, R. (1978). A Dichromatic Framework for Balanced Trees. 19th Annual Symposium on Foundations of Computer Sciences, A Decade of Progress: 1970–1980. Xerox Palo Alto Research Center, Palo Alto, CA.

GUIBAS, L. J. and SZEMEREDI (1978). The analysis of double hashing. *J. Computer and System Sciences*, **16** (April).

KNOPP, K. (1945). "Theory of Functions". Dover, New York.

KNUTH, D., DEBRUIJN, N. and RICE, S. O. (1969). On the height of planted plane trees. *In* "Graph Theory and Computing", R. C. Reed (ed.).

KNUTH, D. E. (1971) Mathematical Analysis of Algorithms. IFIP Congress. Lubyiana, Yugoslavia.

KNUTH, D. E. (1973a). "The Art of Computer Programming, Volume I: Fundamental Algorithms", 2nd Edition. Addison-Wesley, Reading, MA.

KNUTH, D. E. (1973b) "The Art of Computer Programming, Volume III: Sorting and Searching". Addison-Wesley, Reading, MA.

KNUTH, D. E. (1976). Big Oh and Big Omicron and Big Theta. *SIGACT News*.

KNUTH, D. E. and SCHONHAGE, A. (1978). The expected linearity of a simple equivalence algorithm. *Theoretical Computer Science* **6**, (3), June.

KNUTH, D. E. (1980). "The Art of Computer Programming. Volume II: Seminumerical Algorithms", 2nd edition. Addison-Wesley, Reading, MA.

KRUSKAL, M. D. (1954). The expected number of components under a random mapping function. *Am. Math. Monthly*, **61**.

ODLYZKO, A. (1979). On the number of 2–3 trees. *Theoretical Computer Science*, **10**.

RAMSHAW, L. and FLAJOLET, P. (1980). Grey code and the odd-even merge. *SIAM. J. Computing*, April.

ROBSON, J. (1979). The average height of binary search trees. *Australian J. Computer Science*, **11** November.

SEDGEWICK, R. (1977a). Quicksort with equal keys. *SIAM. J. Computing* **6** June.

SEDGEWICK, R. (1977b). The analysis of quicksort programs. *Acta Informatica*, **7**, pp. 327–355.

SEDGEWICK, R. (1978a). Implementing quicksort programs. CACM, **21**, October.

SEDGEWICK, R. (1978b). Data movement in odd-even merging. *SIAM J. Computing*, **7** (2). August.

SEDGEWICK, R. (1980). "Quicksort". Garland Publishing Co., New York. (Reprint of the author's Ph.D. thesis, Stanford University, 1975).

SEDGEWICK, R. and HONG, Z. (1982a). Notes on Merging Networks. In preparation.

SEDGEWICK, R. (1982b). The Asymptotic Behavior of Heapsort. In preparation.

SEDGEWICK, R. (1982c). "Algorithms". Addison-Wesley, Reading, MA. In preparation.

SHEPP, L. A. and LLOYD, S. P. (1966). Ordered cycle lengths in a random permutation. *Trans. Am. Math. Soc.*, **121**.

TITSCHMARSH, E. C. (1951). "The Theory of the Riemann Zeta-Function". Clarendon Press, Oxford.

WHITTAKER, E. T. and WATSON, G. N. (1927). "A Course of Modern Analysis". Cambridge University Press.

YAO, A. C. (1976). On the Average Behavior of Set Merging Algorithms. 8th Annual Symposium on Theory of Computation.

YAO, A. C. (1980). "Analysis of (h, k, l) Shellsort". *J. Algorithms*, **1** (2). June.

Index