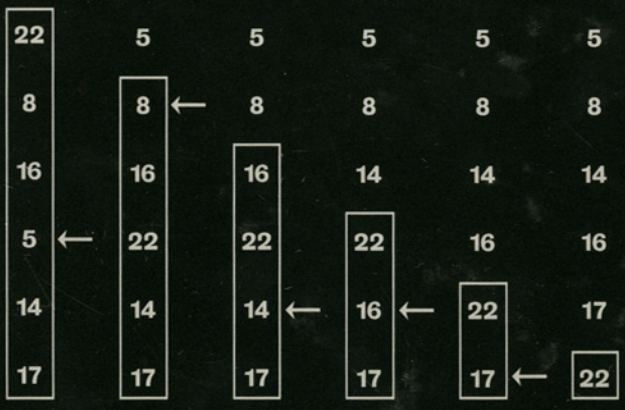
Wiley-Interscience Series in Discrete Mathematics and Optimization

SORTING

A DISTRIBUTION THEORY



HOSAM M. MAHMOUD

This page intentionally left blank

Sorting

WILEY-INTERSCIENCE
SERIES IN DISCRETE MATHEMATICS AND OPTIMIZATION

A complete list of titles in this series appears at the end of this volume.

Sorting
A Distribution Theory

HOSAM M. MAHMOUD
The George Washington University

A Wiley-Interscience Publication
JOHN WILEY & SONS, INC.
New York • Chichester • Weinheim • Brisbane • Singapore • Toronto

This book is printed on acid-free paper. ♾

Published simultaneously in Canada.

For ordering and customer service, call 1-800-CALL-WILEY.

Library of Congress Cataloging-in-Publication Data is available.
ISBN 0-471-32710-7

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

إهداء

إلى فريدة رهنورد محمود

مع حبى و تقديرى

This page intentionally left blank

Contents

This page intentionally left blank

Preface

Putting data in order is an intellectual activity that is perhaps one of the oldest problems of applied mathematics. The wall of the corridor of Abydos Temple in Egypt is padded with a chronologically ordered list (Gallery of the List of Kings). Dating back to around 1250 B.C., this display allegedly lists the Pharaohs who had preceded Siti I (modern research proves that the list makes some false historical claims; a few rulers of Egypt before Siti I are omitted from that list!) The Inakibit-Anu Babylonian tablets, dating back to 200 B.C., contain a sorted list of what seems to be several hundred records, of which only a little over 100 records are preserved. Alphabetical ordering of words dates at least as far back as Western dictionaries. Over the millennia people have been yearning to find fast efficient sorting algorithms. With the appearance of computers in the twentieth century, sorting occupied a central position among computing problems for its numerous business and administrative potentials.

This is a book about sorting methods with a general focus on their analysis. We concentrate on well-known algorithms that are widely used in practice. Of course, to the basic skeleton of any such algorithm small improvements have been proposed. We present these algorithms in their simplest form, even when a modification is known to improve them. This choice is motivated by a desire to make the material lucid. While they may affect lower-order terms, the modifications usually do not change the dominant asymptotic behavior. The analysis techniques are as varied as the sorting algorithms themselves, giving rise to a fascinating variety of methods, ranging from standard treatment such as elementary probability theory, combinatorial counting, and graph-theoretic methods, to instances of more modern techniques such as martingales, Poissonization, Wasserstein's metric space, and the Mellin transform.

We discuss a variety of standard sorting methods that can be readily implemented on a conventional computer. The book takes the following algorithmic view. When a *deterministic* algorithm runs on a set of data of size n, certain computing resources are committed. Among standard resources are the running time and the amount of memory needed to support the algorithm. A suitable combination of these computing resources may be referred to as the *cost*. Running the deterministic algorithm on the same data set will always give the same results. However, running the deterministic algorithm on a different data set of size n may result in different cost. Putting a reasonable probability measure on the set of inputs of size n renders the cost a random variable. Interpreting costs as random variables provides some understanding

of the behavior of the algorithm over the variety of data sets the algorithm may face. Probabilistic analysis addresses the natural questions one usually asks about random variables. What are the average, variance, higher moments, and exact distributions? In many cases it may be prohibitive to get exact answers. Even when such answers are obtained they may give little insight into the algorithm's behavior. Exact answers often take the form of multifolded sums of products or similar complicated forms. Insight is gained by simplifying such expressions asymptotically, that is, finding a simple representation of the leading terms in the form of elementary functions of n, when n gets large. Asymptotics then provide quick ways of estimating how the mean and other moments grow with n, and how probabilities change. Limiting distributions give ways to approximate probabilities from standard distributions that may be well known and tabulated. A question of interest to the practitioner is how large n should be before one can consider limit distributions as reliable approximations. This is a question of finding rates of convergence. We give a few examples of rates of convergence in the book. It is fascinating to find that some phenomena in sorting algorithms may have essential periodic fluctuations in leading asymptotic terms or in rates of convergence to such terms.

The issue we consider is how a deterministic algorithm may behave when it is presented with *random data*. This is to be distinguished from an area of algorithmics that deals with *randomized algorithms*, where the algorithm itself may follow different paths to obtain the same result, or even produce different results when it is run repeatedly on the same data set. Such variability is intrinsic to a randomized algorithm as the algorithm makes its decisions based on random outcomes (like generating random numbers or flipping coins). We touch in passing on the area of randomized algorithms in a few exercises, but it is not the main focus of this book.

The book is intended to be used as a reference by computer professionals, scientists, mathematicians, and engineers. The book may also be used for teaching. The material is accessible to first-year graduate students in fields like computer science, operations research, or mathematics. At least a portion of each chapter is accessible to advanced undergraduates. A reading course or a special topics seminar may be based on the book. If used in this fashion, a one-year course in probability and a one-year course in algorithms are recommended.

The developments that led to this book are numerous. Many research contributions were made toward a distribution theory of sorting algorithms over the past quarter century. These contributions are mostly published in highly specialized research outlets such as scientific journals and conference proceedings. The aim of the book is to organize this massive and esoteric body of research that is accessible only to a few specialized researchers into a coherent, well-founded distribution theory and make it accessible to a wider audience by clarifying and unifying the research and bringing it down a notch to be understood by students and other less-specialized interested readership. This is accomplished via the slower-paced presentation of a textbook, the patient construction of a logical framework for sorting, and careful explanation that appeals to intuition as closely as possible. It is hoped that the manuscript is written in a lively style, starting at the basic principles and building from the ground up, integrating in the process some of the most modern techniques

that have succeeded in cracking some long-standing open problems in sorting, like the distributions associated with QUICK SORT.

The book is organized as follows. It opens with a chapter on the general area of complete and partial sorting (identification of order statistics). The opening chapter sets the tone for the entire monograph: broad meaning, motivation, and applications. The opening sends a message to the reader about the level of complexity of the material and what is expected as background. The opening chapter outlines general methodology (analytic, probabilistic, and combinatorial methods used in later chapters of the book).

The rest of the book dedicates one chapter for every standard sorting algorithm with a careful explanation of the mechanics of the algorithm both as code, for the specialized computer science audience, and verbal description, for the broader readership of scientists, mathematicians, and engineers. A typical chapter delves into the domain of analysis. A broad array of classical and modern techniques converge hand in hand to substantiate a distribution theory of sorting.

Each chapter provides exercises. Chapter 1 is a collection of peripheral topics that are somewhat disparate; the exercises on each topic of this chapter come at the end of the topic. Each of the remaining chapters has exercises at the end of the chapter. The exercises touch on peripheral concepts and even instill new material in small doses. The exercises vary in complexity from the elementary, only reinforcing the basic definitions, to the very challenging, bordering on the threshold of recently understood problems.

HOSAM M. MAHMOUD

This page intentionally left blank

Acknowledgments

Professionals who have made contributions to the study of sorting are too many to be listed individually. I am indebted to the algorithmics community at large. However, in particular I am indebted to the participants of the seminar series *Average-Case Analysis of Algorithms*. Special thanks are in order to Philippe Flajolet, Rainer Kemp, Helmut Prodinger, Robert Sedgewick, and Wojciech Szpankowski, the founding fathers of the seminar series.

The encouraging remarks of Philippe Flajolet, Donald Knuth, and Robert Sedgewick were inspiring. Parts of the book were written while the author was on sabbatical leave at Institut National de Recherche, Rocquencourt, France and Princeton University, New Jersey, U.S. Both places have provided an excellent research atmosphere. The hospitality of Philippe Flajolet, Mireille Régnier, and Robert Sedgewick, who made these visits possible, is greatly acknowledged. Hsien-Kuei Hwang, Philippe Jacquet, Janice Lent, Guy Louchard, Uwe Rösler, Robert Smythe, and Wojciech Szpankowski took interest in earlier drafts of the manuscript and identified several improvements.

The common wisdom of many friends not specialized in the field has been a great inspiration. Many have contributed to the production of the book by providing moral support and unwittingly giving motivation. A few friends not specialized in algorithmics have asked questions that brought to the focus of my attention what the intelligent consumer from other fields may expect from the producers in the field of algorithmics. Talaat Abdin, Kareem and Sherif Hafez, and Christian Hesse may not be aware that they have made contributions to this book. Last but not least, I thank my wife Fari for being an inexhaustible source of patience and understanding.

This page intentionally left blank

1

Sorting and Associated Concepts

1.1 SORTING

Sorting a list of items is a very mundane task. The term generally refers to arranging the items of the list in ascending or descending order according to some ordering relation. Sometimes this activity is desired for its own sake such as the case in ranking Olympic contenders by sorting their scores in an event, or the scores of a class to determine letter grades. Sorting is often intended to facilitate searching, another fundamental activity in data processing. Imagine how difficult it would be to consult a dictionary on the meaning of a word, if the dictionary is not organized in some order to guide our search. With relative ease we can find a word in a dictionary or a person's phone number in a directory when its entries are kept in the standard alphabetical order.

Sorting is a more convoluted task than searching. Generally it is more expensive and time consuming. Nonetheless, we view it as a long-term investment—we take the time and effort to sort a file once to repeatedly search quickly in it for items.

The general problem of sorting can be stated as follows. Say we have a list of items

$$X_1, X_2, \ldots, X_n$$

that we intend to rearrange in increasing order. Our goal is to design efficient algorithms to create a permutation

$$X_{(1)}, X_{(2)}, \ldots, X_{(n)}$$

where $X_{(i)}$ is the ith smallest (or ith *order statistic*) among the items on our list. The term *smallest* refers to relative smallness according to an ordering relation often denoted by $\leq$. For example, $X_1, X_2, \ldots, X_n$ may be real numbers and $\leq$ is the usual arithmetic comparison. The final sorted list in this case will be $X_{(1)}, X_{(2)}, \ldots, X_{(n)}$, with $X_{(1)} \leq X_{(2)} \leq \ldots \leq X_{(n)}$, that is, the numbers arranged in increasing order.

Another way to view the situation is the following. Note that $X_{(1)}$ may turn out to be any of the n numbers—there is an index i_1 so that $X_{(1)} = X_{i_1}$. Generally, the same is true for any other order statistic: $X_{(j)} = X_{i_j}$, for some index i_j. We can think of sorting as an attempt to come up with a map

$$\Pi_n : \{1, 2, \ldots, n\} \rightarrow \{1, 2, \ldots, n\}, \tag{1.1}$$

where $\Pi_n(1) = \pi_1, \Pi_n(2) = \pi_2, \ldots, \Pi_n(n) = \pi_n$, with the integers $\pi_1, \ldots, \pi_n$ being all distinct, and

$$X_{\pi_1} \leq X_{\pi_2} \leq \ldots \leq X_{\pi_n}. \tag{1.2}$$

In other words, $\pi_1, \ldots, \pi_n$ is a permutation of the integers $\{1, 2, \ldots, n\}$, with X_{π_j} being the jth smallest item. We shall use the notation $\Pi_n = (\pi_1, \ldots, \pi_n)$ to represent such a permutation. Permutations can be represented in a number of ways. A standard two-line representation helps in visualizing where the numbers go. This representation consists of a top row of indexes and a bottom row of corresponding values; under each index i, we list π_i. For example, the permutation $\Pi_8 = (7\ 6\ 8\ 3\ 1\ 2\ 5\ 4)$ can also be represented as

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 7 & 6 & 8 & 3 & 1 & 2 & 5 & 4 \end{pmatrix}.$$

Producing the sorted list is an act of actual rearrangement. The task of sorting n elements may effectively be accomplished without moving any elements (if so desired) by specifying the permutation Π_n of the construction (1.2) as a separate list. For in this case the knowledge of $\pi_1, \ldots, \pi_n$ can effectively be used to obtain any order statistic. For example, we can print the still-unsorted list $X_1, \ldots, X_n$ in sorted order by printing

$$X_{\pi_1}, X_{\pi_2}, \ldots, X_{\pi_n}.$$

EXERCISES

1.1.1 Sort each of the following lists in "ascending" order:

(i) 2.6 1.3 7.1 4.4 3.4 2.9.

(ii) 60 33 17 84 47 29 71 56.

(iii) Jim Beatrice Philip Alice Johnny Elizabeth.

1.1.2 For each list in the previous exercise, construct a permutation to effectively sort the list without changing its composition.

1.1.3 Interpret the mapping (1.1) when the list contains repeated elements.

1.2 SELECTION

Another problem of practical interest is that of the selection of some order statistics either ordered by rank or as a set known to have some predesignated ranks but not necessarily in order. An example of finding an ordered group of ranked order statistics is common in competitive sports. It is often the case that from the scores obtained

by the competing athletes the top three contenders with the highest three scores (in order) are to be selected for medals.

To emphasize the distinction between sorting and selection, sorting is sometimes referred to as *complete sorting*, whereby all the elements are moved to their correct positions (or indexes are created to indicate these positions). By contrast, selection is referred to as *partial sorting*, as the selection process imparts only partial information on the correct position of some elements.

Statistics based on a few order statistics (with known orders) abound in the literature of statistical inference. For example, weighted averages of tertiles and quartiles[1] of a sample $X_1, X_2, \ldots, X_n$, are in common use in estimating the center of the distribution from which the sample was gathered. Two such statistics with different weights are Gastwirth's estimator

$$G_n = \frac{3}{10} X_{(\lfloor \frac{n}{3} \rfloor + 1)} + \frac{2}{5} X_{(\lfloor \frac{n}{2} \rfloor)} + \frac{3}{10} X_{(n - \lfloor \frac{n}{3} \rfloor)},$$

and Tukey's tri-mean

$$T_n = \begin{cases} \frac{1}{8}\left(X_{(\frac{n}{4})} + X_{(\frac{n}{4}+1)}\right) + \frac{1}{2} X_{(\frac{n}{2})} \\ \qquad + \frac{1}{8}\left(X_{(\frac{3n}{4})} + X_{(\frac{3n}{4}+1)}\right), & \text{if } n \text{ is a multiple of 4;} \\ \frac{1}{4} X_{(\lfloor \frac{n}{4} \rfloor)} + \frac{1}{2} X_{(\lfloor \frac{n}{2} \rfloor)} + \frac{1}{4} X_{(\lceil \frac{3n}{4} \rceil)}, & \text{otherwise.} \end{cases}$$

The book by Andrews, Bickel, Hampel, Huber, Rogers, and Tukey (1972) gives many other statistical applications involving only a few order statistics. Efficient algorithms for finding quantiles are needed for quick computation of these statistics. Many other applications in statistics need such algorithms to find few order statistics, typically less than six, for a statistical design.

Another flavor of the problem of finding order statistics may request finding a group of unranked order statistics without necessarily knowing the relative ranks in the group. For example, a university that intends to admit 1000 students may want to find the 1000 students with the top 1000 scores (not necessarily in order) among the applicants whose number typically exceeds 10000 each academic season. An example from statistics where an unordered group of order statistics is required is the α-trimmed mean. The *α-trimmed mean* is a statistic that throws out the upper and lower α proportion of the (sorted) data deeming them unreliable outliers, with $0 < \alpha < 1$. The α-trimmed mean is given by

$$\frac{1}{n - 2\lfloor \alpha n \rfloor} \sum_{i=\lfloor \alpha n \rfloor + 1}^{n - \lfloor \alpha n \rfloor} X_{(i)}.$$

[1] An α-quantile of a sample having no duplicates is a real number so that proportion α of the data is less than it; the rest are greater. When n is an exact multiple of 3, the $\frac{1}{3}$- and $\frac{2}{3}$-quantiles are called tertiles. The proportions $\frac{1}{3}$ and $\frac{2}{3}$ of the data are interpreted respectively as $\lfloor \frac{n}{3} \rfloor$ and $\lfloor \frac{2n}{3} \rfloor$ of the points when n is not a multiple of 3. With a similar interpretation of "fourth," "half," and "three-fourths," the $\frac{k}{4}$-quantiles are called quartiles, for $k = 1, 2, 3$ and the $\frac{1}{2}$-quantile is called the sample median.

We may design an algorithm to identify the two bounding order statistics $X_{\lfloor \alpha n \rfloor}$ and $X_{n-\lfloor \alpha n \rfloor+1}$ and in the process place all intermediate order statistics in the list between positions $(\lfloor \alpha n \rfloor + 1), \ldots, (n - \lfloor \alpha n \rfloor)$ in any order, not necessarily sorted. For the computation of the α-trimmed mean we need not sort these intermediate order statistics since we are only seeking their average.

Algorithms for finding a predesignated number of order statistics (ordered or unordered) are generally called *selection algorithms*. Sorting itself is the special case of a selection algorithm, when the selection algorithm finds all order statistics.

One can find order statistics by sorting a data file. The item at position i is then the ith order statistic. This may be much more work than necessary; it is often the case that a more efficient algorithm can be designed for the direct selection of specified order statistics as we shall see. Several sorting algorithms discussed in this book can be adapted for general purpose selection.

EXERCISES

1.2.1 Write an algorithm to find the maximum element in a list. How many comparisons does your algorithm make to pick the largest number in a list of n numbers?

1.2.2 Write an algorithm to simultaneously find the maximum and minimum elements in a list of n elements using no more than $3n/2$ comparisons.

1.3 JARGON

A sorting algorithm is *comparison based* if it sorts an input (a collection of data items called *keys*) by making a series of decisions relying on comparing pairs of the input data. (Many algorithms in this book are comparison based.) Nonetheless, there are sorting algorithms that are not comparison based. For instance, RADIX SORT, an algorithm discussed in a later chapter, bases its decisions on another operation different from comparing pairs of data—this operation is the extraction of digits of the input keys; RADIX SORT makes decisions according to these digits. RADIX SORT is a digital non-comparison-based sorting method.

Broadly speaking, sorting algorithms fall in two categories. The class of *naive* sorting algorithms, the ones that occur first in a natural way to most people, is a class of algorithms with $\Theta(n^2)$ running time[2] when they sort most inputs of size n and that tend to have an average running time of order $\Theta(n^2)$. On close examination we realize that naive algorithms do not effectively use information gained from the earlier stages of the algorithm and may tend to perform redundant or repetitious tasks. Careful parsimonious design that avoids redundancy can introduce a substantial reduction in the order of magnitude of the running time of an algorithm, leading to an algo-

[2]For definition of the asymptotic symbols Θ, O, o, $\sim$, and Ω see the appendix.

rithm in the second class of parsimonious algorithms. We shall see shortly that the best comparison-based algorithm must take at least $\Omega(n \ln n)$ time[3] on some input of size n. That is, the best comparison-based sorting algorithm must take $\Omega(n \ln n)$ time on its worst case of size n, or comparison-based sorting is inherently bounded from below by $\Omega(n \ln n)$ for some input. There are practical algorithms that do take $O(n \ln n)$ time on all inputs. A *parsimonious* sorting algorithm is one that possesses an average running time that is of the order $\Theta(n \ln n)$. A parsimonious algorithm performs on most inputs as the best comparison-based sorting algorithm would on its worst-case instance.

Parsimonious algorithms are often recursive and use the paradigm of *divide-and-conquer*, whereupon the list to be sorted is split into two or more segments according to some splitting criterion, then the parts which are typically smaller than the input list itself are recursively attacked individually. The process continues until very small lists are considered (typically of size 1 or 0) for which the sorting task is trivial or has a directly obvious answer. The solution to the original sorting problem is then assembled by combining the solutions of the parts.

A sorting algorithm is called *stable* if equal elements in the raw data maintain their relative position in the final sorted list. For example, if $X_5 = X_{22} = X_{47}$ are the only keys equal to 18 in an unsorted list of integers, a stable sorting algorithm may move X_5 to position 3, X_{22} to position 4, and X_{47} to position 5. Stability is desired when satellite information is present alongside the primary keys. For example, suppose we have an alphabetically sorted list of passengers out of an international airport on some day. When a stable sorting algorithm is applied to the flight number field of the records, the alphabetical order is not disturbed within one flight; we get the day's activity sorted by flight numbers, and within each flight the passenger names will be sorted alphabetically.

A desirable feature of a sorting algorithm is the capability to finish the job without creating large secondary data structures to hold copies of the data at some stage. Several algorithms discussed in this book are capable of sorting using only a chief host container of data, such as an array. When performing on n data items, the sorting task is accomplished by moving data around within the container, that is, making swaps via a few additional intermediate swap variables (typically $O(1)$ extra space). Recursive sorting algorithms may set up a hidden stack of small size, typically averaging to $O(\ln n)$ additional space. An algorithm that sorts a list of n items "in place" without allocating data structures of size comparable to the list it is sorting—that is, the secondary storage is no more than $o(n)$ on average, is said to sort *in situ*.

EXERCISES

1.3.1 Study the algorithm of Figure 1.6 first. Is this algorithm comparison based? Is it stable? Does it operate in situ?

[3]See the appendix for the convention for the base of the logarithm.

1.4 ALGORITHMIC CONVENTIONS

We specify many algorithms in this book in pseudocode. It is essentially similar to standard conventions of a pedagogical programming language, except that we occasionally take the liberty of relaxing the syntax when convenient or more readable. For example, exponential expressions like 2^k are easier to read than the usual representation by double asterisks or computation via logarithms. Most of such algebraic symbolism is represented within pseudocode in standard mathematical notation. For example, ceils and floors are inserted in our pseudocode and division in the form of a two-level fraction, as in

$$\frac{a+b}{c+d},$$

is preferred to the parenthesized expression $(a + b)/(c + d)$, because the former is more readable and uses fewer symbols.

We also give a brief verbal explanation for each algorithm to make the material accessible to nonprogramming readership of scientists, mathematicians, and engineers.

The reader is assumed to be familiar, at least conceptually, with basic data structures such as arrays and linked lists. Data to be sorted will most of the time be assumed to be already loaded in a typical array structure like the following:

global A: **array** $[1 \,..\, n]$ **of real**;

As an algorithm progresses with its tasks we shall often need to specify a particular stretch (subarray) on which the algorithm operates next. Such a subarray will typically extend from a lower index ℓ, to an upper index u. The convenient notation $A[\ell \,..\, u]$ will refer to this subarray (and to its content).

Arrays and subarrays will be depicted as vertical columns with lower indexes near the top. Sometimes it is more convenient for the typography to depict an array horizontally. In horizontal representation the lower indexes will be on the left.

Many sorting algorithms perform by swapping array elements. In the background of sorting algorithms that need to swap data we shall assume the presence of a procedure

$$swap(x, y)$$

that interchanges the contents of the two variables x and y. A typical call in the context of array sorting will be

call $swap(A[i], A[j])$

to interchange the data at positions i and j. Generally, the background *swap* procedure will be assumed to be capable of handling whatever data type is being sorted.

EXERCISES

1.4.1 Implement the sorting algorithm discussed in Section 1.7 on a computer.

1.4.2 Implement a *swap* procedure on a computer to interchange the content of two variables.

1.5 ORDER

Central to the analysis of sorting and selection is the notion of partial order. We shall briefly review the basic concepts of order. Let A and B be two sets. The *Cartesian product* $A \times B$ is the set of all pairs $\{(a, b) \mid a \in A \text{ and } B \in B\}$; an element of $A \times B$ is an *ordered pair*. A relation R from A to B is a subset of $A \times B$. A relation from A to A is simply called a relation *on* A and is thus a subset of $A \times A$. Let R be a relation on A. If $(x, y) \in R$ we say that x and y are *related* by R and in infix notation we write $x\ R\ y$.

A relation $\leq$ (commonly called less than or equal to) on a set A is a *partial order* if the relation is:

(i) *Reflexive*, that is, $a \leq a$, for all $a \in A$.

(ii) *Antisymmetric*, that is, $a \leq b$ and $b \leq a$ imply $a = b$, for all $a, b \in A$.

(iii) *Transitive*, that is, $a \leq b$ and $b \leq c$ imply $a \leq c$, for all $a, b, c \in A$.

Two related elements a and b are said to be *comparable* (either $a \leq b$, or $b \leq a$). The notation $a \leq b$ is called a *true comparison* when $a \leq b$ is in the partial order. For the comparable pair $a \leq b$, we use expressions like a is less than or equal to b, a has a rank lower than b, a is below b, or a precedes b in the partial order, and so on. Similarly, we often use expressions like b is larger or has rank higher than a, b is above a, or follows a in the partial order. When two elements a and b are not comparable, that is, neither $a \leq b$ nor $b \leq a$ is in the partial order, we say a and b are *incomparable*. The set A and the ordering relation $\leq$ together are called a *partially ordered set* denoted by $(A, \leq)$, also called a *poset*. A true comparison is equivalent to an ordered pair and the partial order may be represented as a set of true comparisons.

A partial order on the universe A is a *total order*, when every pair of the elements of the universe are comparable. For example, the set of integers and the usual $\leq$ arithmetic comparison is a total order.

The following example illustrates several definitions discussed in the preceding paragraphs. Suppose we have the universe $\{X_1, \ldots, X_8\}$. The set of true comparisons

$$Q = \{X_i \leq X_i \mid i = 1, \ldots, 8\} \cup \{X_1 \leq X_2, X_1 \leq X_3, X_1 \leq X_4, X_1 \leq X_5, X_2 \leq X_3, X_2 \leq X_4, X_2 \leq X_5, X_3 \leq X_5, X_6 \leq X_7\} \tag{1.3}$$

is a partial order on the universe and is a subset of the total order induced by indexing, that is, the ordering relation $X_i \leq X_j$ whenever $i \leq j$. If it is not known whether a total indexing order exists, the partial order Q gives only partial information on the relative ranking of some elements.

The goal of a comparison-based sorting algorithm on a finite totally ordered set is to discover the total order. The algorithm does that by asking questions in steps. Every step reveals more information and a larger partial order is attained. The sorting process starts with the empty partial order, as there is no prior information available, then goes on to build nested posets. The algorithm terminates when enough information is gathered to construct the total order.

The set of *minima* in a poset $\mathcal{P} = (A, \leq)$ is the set

$$\{a \in A \mid \text{the only element in } A \text{ that is } \leq a \text{ is } a\};$$

a member in the set of minima is a *minimum* or *minimal element*. We similarly define the set of *maxima* of $\mathcal{P}$ as

$$\{b \in A \mid \text{the only element } x \in A \text{ satisfying } b \leq x \text{ is } b\};$$

a member in the set of maxima is a *maximum* or *maximal element*.

Often a poset is very conveniently represented pictorially by a diagram of the objects of the poset with an arrow going from a to b for any true comparison $a \leq b$ between two different elements $a \neq b$. (The self-loops corresponding to reflexivity are usually not shown but implicitly understood.) As a visual aid to the essence of the relation, the minima are drawn at the bottom of the diagram. Higher-ranked elements usually appear in the drawing above lower-ranked elements of the poset. Figure 1.1 shows the partial order Q of display (1.3).

With every minimal element of a poset $(A, \leq)$ there are associated *chains* of the partial order, where each chain is formed by the following procedure. If $a_1 \in A$ is a minimum, the chains starting at a_1 are formed by first formally writing the string a_1. Whenever a_k is at the rightmost end of a string $a_1 \leq a_2 \leq \ldots \leq a_k$, and a_k is below $b_1, \ldots, b_j$ in the partial order, the string is cloned into j strings; the symbol $\leq$ is added at the right end of each clone followed by one of the elements $b_1, \ldots, b_j$;

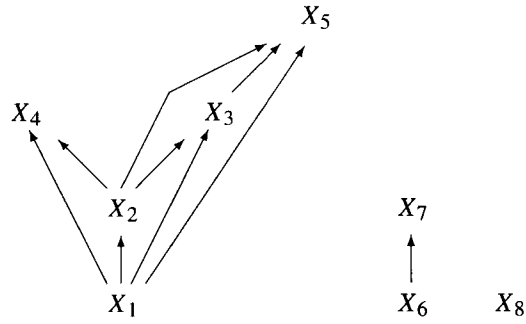


Figure 1.1. A poset.

this step in the process yields

$$\begin{aligned} a_1 \leq a_2 \leq \ldots \leq a_k \leq b_1, \\ a_1 \leq a_2 \leq \ldots \leq a_k \leq b_2, \\ \vdots \\ a_1 \leq a_2 \leq \ldots \leq a_k \leq b_j. \end{aligned}$$

The rules apply recursively to every clone until it is no longer possible to add any elements. Strings that are obtained by removing symbols from longer strings are then discarded. Each remaining formal string is a chain that starts at a minimal element and ends at a maximal element of the partial order. Together, the chains reflect how much order is known. A total order has only one chain.

Finally, suppose A is a subset of some universal set S. The set A is said to be *compatible* with a partial order $(S, \leq)$, if whenever $a \leq b$, and $a \in A$ then $b \in A$. Informally, if a is a member of a chain and $a \in A$, then all the elements of the chain higher than a all the way up to the maximal element of that chain are in A. None of the elements of A is known to be below an element from the complement set $S - A$.

To fix the ideas, let us consider again the partial order Q of display (1.3). The set of minima of Q is $\{X_1, X_6, X_8\}$. The set of maxima is $\{X_4, X_5, X_7, X_8\}$. The elements X_2 and X_5 are comparable; but X_1 and X_7 are incomparable. The chains of Q are $X_1 \leq X_2 \leq X_3 \leq X_5$, $X_1 \leq X_2 \leq X_4$, $X_6 \leq X_7$, and X_8. The set $\{X_2, X_3, X_4, X_5, X_8\}$ is compatible with Q, but the set $\{X_2, X_5, X_6\}$ is not.

The following lemma is important in deriving lower bounds on sorting and selection.

Lemma 1.1 *A comparison-based algorithm for finding the jth order statistic must implicitly determine the set of $j - 1$ items of lower ranks. Consequently the set of $n - j$ items of higher ranks is also determined.*

Proof. We prove the lemma by contradiction. Suppose we have an algorithm that finds the jth smallest in the data set $\{X_1, \ldots, X_n\}$ of distinct elements, without determining the sets $\{X_{(1)}, \ldots, X_{(j-1)}\}$ of smallest $j - 1$ order statistics and $\{X_{(j+1)}, \ldots, X_{(n)}\}$ of largest $n - j$ order statistics. There are indexes k, ℓ, and m such that the algorithm does not know whether $X_{(k)} = X_\ell \leq X_{(j)} = X_m$, or vice versa. No chain of the partial order discovered by the algorithm contains both X_ℓ and X_m; the existence of such a chain would make X_ℓ and X_m comparable. We can then construct an ordering of the input, starting with a chain containing X_m. In this chain let $a \leq X_m$ be the last element that is known to be below X_ℓ, and let b be the first element known to be above X_ℓ such that $X_m \leq b$; one or both of the elements a or b may not exist. In a manner consistent with the partial order, we can complete a linear order by first inserting X_ℓ anywhere between a and b. (If a does not exist and b does, we can place X_ℓ anywhere below b. Similarly, if b does not exist and a does, we can place X_ℓ anywhere above a. If both do not exist we are at liberty to put X_ℓ

anywhere as we completely lack any information on its relative rank in the chain.) Specifically, we insert X_ℓ next to X_m, which is presumed to be at position j. The rest of the elements not already in the chain are then added in any way consistent with the partial order. Transposing X_ℓ and X_m changes the position of X_m in the listing, but remains to be an ordering consistent with the partial order. This uncertainty of the position of X_m, supposedly the jth smallest in the final linear order, contradicts the assumption that the algorithm has identified the jth order statistic. ■

EXERCISES

1.5.1 Let the set $A = \{2, 3, 4, \ldots\}$ be partially ordered by divisibility—for $x, y \in A$, the ordered pair (x, y) is in the partial order if x divides y. What are the minima of A? What are the maxima of A? What are the chains of the poset? Is the set of odd numbers in A compatible with the partial order?

1.6 BINARY TREES

The *binary tree* is a hierarchical structure of *nodes* (also called *vertices*) that underlies many sorting and other combinatorial algorithms. The nodes of the structure are arranged in *levels*. The level count begins at 0. The only node at level 0 is called the *root node*. A node may have up to two nodes above it (viewed as *children*) that are put on the next level—a node (also viewed as a *parent*) may have no children, one left child, one right child, or two children (one left and one right). The children of a node are joined to their parent by links called *edges* or *branches*. It is customary to represent the tree by a drawing in which nodes are black bullets or circles of various sizes to illustrate different purposes, and edges are straight lines. The *depth* of a node in the tree is the node's distance from the root, that is, its level.

Historically trees are drawn upside down with the root at the top of the drawing. We adopt this convention in this book. If a tree has n nodes we say it is of *size* or *order* n. Figure 1.2 shows an example of a binary tree of size 8. We shall continue to use this historical convention; nevertheless, the language of the text will more

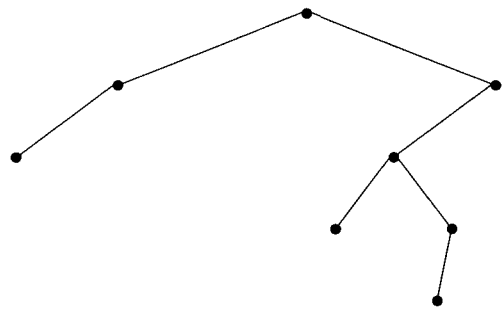


Figure 1.2. A binary tree on 8 nodes.

closely comport with the growth direction of natural trees—the root is at the bottom and tracing a path starting from the root is a process of moving "up" in the tree or climbing it, etc.

It is sometimes useful to think of the binary tree structure in terms of the following inductive definition. A binary tree is either empty, or has two substructures (called *subtrees*) distinguished as a left subtree and a right subtree, with the left and right subtrees both being binary trees. The collection of nodes in the subtree rooted at a node (excluding the node itself) is referred to as its *descendants*. By a parallel terminology, the collection of nodes encountered by climbing down a path from a node to the root of the whole tree is the node's *ancestors* or *predecessors*.

Often the parent-child links in a binary tree are given an orientation with sense leading from parent to child. Thus each node has only one link leading to it, or the so-called *indegree* is 1 for each node (except the root, whose indegree is 0). Each node may have up to two edges emanating out of it leading to its children. The number of edges coming out of a node is called the *outdegree* of the node. The outdegree of any node in a binary tree is at most 2. Level 1 can have up to two nodes; level 2 can have up to 4 nodes, and so forth. The maximum number of nodes that can appear on level i is 2^i. If level ℓ has 2^ℓ nodes, we say that this level is *saturated*. The root is at the lowest level (level 0). As we climb up the branches of the tree moving away from the root we move toward *higher* levels (further down in the inverted drawing).

It is very helpful in many analyses to consider an extension of a binary tree. The *extended binary tree* is obtained by adding to each original node, 0, 1, or 2 children of a new distinct type (called *external nodes* or *leaves*) to make the outdegree of all the original nodes (now called *internal*) exactly equal to 2. Figure 1.3 shows the extension of the tree of Figure 1.2, with internal nodes shown as bullets and leaves as squares.

The extended tree has a number of properties that will be of great use in the analysis of sorting algorithms and will be frequently encountered later in this book. These are invariant properties that are true for all trees of a given size regardless of their shape. We discuss these properties in the following definitions and propositions.

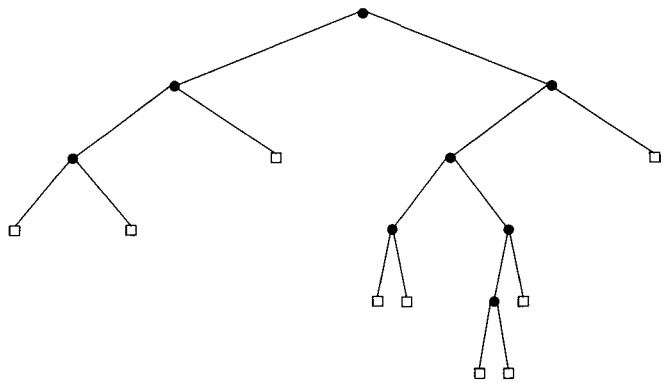


Figure 1.3. An extended binary tree on 8 internal nodes.

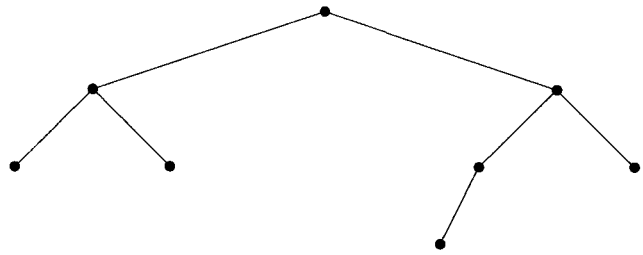


Figure 1.4. A complete binary tree on 8 nodes.

When all the levels of a tree, except possibly the highest level, are saturated, we say the tree is *complete*. The tree of Figure 1.2 is not complete, whereas that in Figure 1.4 is. Furthermore, when all the leaves appear on the same level the tree is said to be *perfect*. The complete tree is so termed because its extension has no "holes" in the levels containing only internal nodes; all these levels are saturated.

The *height* of a binary tree is the length of the longest root-to-leaf path in the tree. So, the height is also the level of the highest leaf. The height of a binary tree T_n on n nodes is denoted by $h(T_n)$ or simply h_n when it is understood which tree is being considered.

Proposition 1.1 *The height h_n of a binary tree on n nodes satisfies:*

$$\lceil \lg(n+1) \rceil \leq h_n \leq n.$$

Proof. The upper bound is trivial. The given tree is at least as high as a complete tree on n nodes; see Exercise 1.6.3. The complete tree is an instance of the lower bound. Letting h_n^* be the height of the complete tree, we have just argued that $h_n^* \leq h_n$. In a complete binary tree of height h_n^*, levels $0, 1, \ldots, h_n^* - 2$ are saturated, and there are internal nodes on level $h_n^* - 1$, and no internal nodes on any higher levels:

$$1 + 2 + \cdots + 2^{h_n^*-2} < n \leq 1 + 2 + \cdots + 2^{h_n^*-1},$$

or

$$2^{h_n^*-1} < n + 1 \leq 2^{h_n^*}.$$

Taking base-two logarithms:

$$h_n^* - 1 < \lg(n+1) \leq h_n^*.$$

However, h_n^* must be an integer. So, $h_n^* = \lceil \lg(n+1) \rceil$. ∎

Proposition 1.2 *An extended binary tree on n internal vertices has $n + 1$ leaves.*