EDWARD M. REINGOLD
JURG NIEVERGELT
NARSINGH DEO

Combinatorial Algorithms:

Theory and Practice

Combinatorial Algorithms: Theory and Practice

EDWARD M. REINGOLD

Department of Computer Science
University of Illinois at Urbana-Champaign

JURG NIEVERGELT

Department of Computer Science
University of Illinois at Urbana-Champaign
and
Swiss Federal Institute of Technology, Zurich

NARSINGH DEO

Department of Electrical Engineering and Computer Science Programme
Indian Institute of Technology, Kanpur

PRENTICE-HALL, INC., ENGLEWOOD CLIFFS, NEW JERSEY 07632

Library of Congress Cataloging in Publication Data

Reingold, Edward M (date)
Combinatorial algorithms.

Includes bibliographical references and index.
1. Combinatorial analysis—Data processing.
I. Nievergelt, Jürg, joint author. II. Deo, Narsingh, joint author. III. Title.
QA164.R43 511'.6 76-46474
ISBN 0-13-152447-X

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

PRENTICE-HALL INTERNATIONAL INC., *London*
PRENTICE-HALL OF AUSTRALIA PTY LIMITED *Sydney*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA PTE LTD *Singapore*
WHITEHALL BOOKS LIMITED, *Wellington, New Zealand*

Preface

The field of combinatorial algorithms concerns the problems of performing computations on discrete, finite mathematical structures. It is a new field, and only in the past few years has it started to emerge as a systematic body of knowledge instead of a collection of unrelated tricks. Its emergence as a new discipline is due to three factors:

An increase in the practical importance of computation of a combinatorial nature, as compared to other computation.

Rapid progress, primarily of a mathematical nature, in the design and analysis of algorithms.

A shift in emphasis from the consideration of particular combinatorial algorithms to the examination of properties shared by a class of algorithms.

The combination of these factors has promoted combinatorial algorithms as an important new discipline on the border between computer science and mathematics. Courses in combinatorial algorithms and related courses in the analysis of algorithms are now being taught in colleges and universities in computer science, mathematics, electrical engineering, and operations research departments.

Combinatorial algorithms can be presented in different ways, and a course or textbook can be directed toward different audiences. This book is aimed at a reader who can best be characterized as having more of a computing background than a mathematics background, a reader who is interested in combinatorial algorithms because of their practical importance. Thus our main goals in writing this book were:

To choose the topics according to their relevance to practical computation (however, we have included some not very practical but mathematically interesting topics).

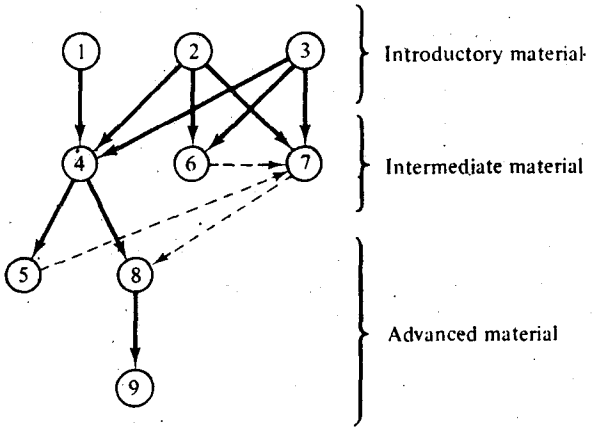
To emphasize those aspects of algorithms that are important from the point of view of their implementation without belaboring details that any competent programmer can supply.

To present mathematical arguments, where they are necessary, with emphasis on insight.

Prerequisites. The prerequisites required by this book vary somewhat from chapter to chapter. The minimal knowledge of programming required is that which can be obtained in a first course on computers in which students write several rather extensive programs. This background should suffice to understand the algorithms discussed, which are presented in a notation similar to current high-level programming languages. The additional background obtained in a second computer course on data structures and list processing is highly desirable. The mathematical maturity required is that typical of a student who has taken several mathematics courses beyond calculus.

Starred Sections. Two sections of this book have been marked by a ★. Section 3.4 is so marked because it requires some familiarity with advanced mathematical concepts, and Section 8.6 is so marked because it is intrinsically complicated. These sections can be considered optional. Exercises related to the starred sections are also starred, as are other exercises that require advanced mathematics or whose solution is unusually complex.

Overall Organization. The dependencies among the nine chapters are indicated by the following diagram in which "strong" dependencies are shown by solid arrows and "weak" dependencies by dashed arrows:



Chapter 1 is designed to display the scope of the material in the book. It also introduces some topics and techniques that reappear in later chapters.

Chapters 2 and 3 discuss data structures and counting techniques, respectively. With the exception of Section 3.4, all the material contained in these two chapters is basic to the rest of the book. It may be more appropriate, however, to cover the topics in these chapters as they are needed in presenting the material in later chapters. The book is not written that way in an effort to make Chapters 4 through 9 as independent of each other as possible.

The material in Chapter 4 on exhaustive search is important in understanding Chapter 5, which is actually a specialization of techniques from Chapter 4 to simple combinatorial objects. For the same reason, the material in Chapter 4 is crucial to understanding many parts of Chapter 8 on graph algorithms. Chapter 9 continues the sequence of material in Chapters 4 and 8 by examining some theoretical questions raised by the failure of computer scientists and mathematicians to find "efficient" algorithms for some of the problems discussed.

Chapters 6 and 7 describe the most common of all combinatorial algorithms: searching and sorting. These chapters rely most heavily on material in Chapters 2 and 3.

ACKNOWLEDGMENTS

The entry "illogicalities" in H. W. Fowler's *A Dictionary of Modern English Usage* (second edition, revised and edited by Sir Ernest Gowers, Oxford University Press, New York and Oxford, 1965) begins with

> The spread of education adds to the writer's burdens by multiplying that pestilent fellow the critical reader. No longer can we depend on an audience that will be satisfied with catching the general drift and obvious intention of a sentence and not trouble itself to pick holes in our wording.

We have been fortunate in having many such critical readers examine all or part of the manuscript for this book, and they have helped to find various instances of "illogicalities." We gratefully acknowledge James R. Bitner, Allan B. Borodin, James A. Fill, W. D. Frazer, Brian A. Hansche, Wilfred J. Hansen, Ellis Horowitz, Alon Itai, Donald B. Johnson, John A. Koch, Der-tsai Lee, Karl Lieberherr, C. L. Liu, Prabhaker Mateti, Yehoshua Perl, David A. Plaisted, Andrew H. Sherman, Robert E. Tarjan, Daniel S. Watanabe, Lee J. White, Thomas R. Wilcox, and Herbert S. Wilf. Without their invaluable assistance this book would be poorer.

We owe an extra debt of gratitude to James A. Fill for his careful preparation of the solutions manual.

We are indebted to the University of Illinois at Urbana-Champaign, Eidgenössische Technische Hochschule, Los Alamos Scientific Laboratory, Weizmann Institute of Science, Washington State University, and Indian Institute

of Technology, Kanpur, for providing facilities for the preparation of this book. Their support made our task much easier.

Finally, we would like to acknowledge and thank Connie Nosbisch and June Wingler, our secretaries at the University of Illinois, for their patient, conscientious, and expert typing, retyping and sometimes even re-retyping of the manuscript.

EDWARD M. REINGOLD

JURG NIEVERGELT

NARSINGH DEO

Contents

chapter 1

What is Combinatorial Computing?

The subject of combinatorial algorithms, frequently called *combinatorial computing,* deals with the problem of how to carry out computations on discrete mathematical structures. It is a new field: only in the past few years has a systematic body of knowledge about the design, implementation, and analysis of algorithms emerged from a collection of tricks and unrelated algorithms.

An analogy with a more established field may be useful. Combinatorial computing bears to combinatorial (discrete, finite) mathematics the same relationship that numerical analysis bears to analysis. We are witnessing today in combinatorial computing the same development that numerical analysis went through in the 1950s—namely,

new algorithms are being invented at a rapid rate.

rapid progress, primarily of a mathematical nature, is being made in our understanding of algorithms, their design, and their analysis.

emphasis is shifting from consideration of a particular algorithm to properties shared by a class of algorithms, and thus unifying patterns are emerging.

The increasing practical importance of computation of a combinatorial nature has undoubtedly contributed to the recent surge of activity in combinatorial computing. There is every reason to believe that the amount of computation of a combinatorial nature that occurs in applications programs will increase faster than the amount of numerical computation. This is because, outside the traditional areas of applications of mathematics to the physical sciences, discrete

mathematical structures occur more frequently than continuous ones, and the fraction of all computing time spent on problems that arise in the physical sciences is decreasing. Hence computer users and applications programmers will probably be called on to solve problems of a combinatorial rather than numerical nature.

Unlike some other fields, combinatorial computing does not have a few "fundamental theorems" that form the core of the subject matter and from which most results can be derived. The entire subject may seem, at first, merely a collection of unrelated, specialized techniques and tricks. Clever tricks do indeed play a role, and in this chapter we will see some examples of "bit pushing" techniques that convey this flavor. However, after examining many combinatorial algorithms, some general principles become apparent. It is these principles that unify the field and make combinatorial computing a coherent subject that can be presented in a systematic way. The purpose of this chapter is to illustrate some of these important principles by means of examples, as well as to serve as an introduction into some of the topics and techniques that will be treated in greater depth in later chapters.

The chapter is organized so as to go from the concrete aspects of computation to more abstract principles, some of which require advanced mathematics for their application to the analysis of algorithms. Therefore the abstract sections of this chapter are more difficult than the others, and in this respect Chapter 1 is a faithful mirror of the field it introduces. Sections 1.1 and 1.2 show examples of the detailed thinking required for efficient implementation of algorithms. Sections 1.3 and 1.4 present general principles of algorithm design. Finally, Sections 1.5 and 1.6 show how algorithms are analyzed. The design, analysis, and implementation of algorithms form the core of combinatorial computing.

1.1 AN EXAMPLE: COUNTING THE NUMBER OF ONES IN A BIT STRING

Bit strings—that is, sequences of zeros and ones—are the basic carriers of information in virtually all modern computers. Most programmers, however, rarely handle information at the detailed level of bit strings. This is certainly true in numerical computation, in which a programmer usually expresses himself in terms of arithmetic operations on numbers and is seldom concerned with the internal representation of these numbers. On the other hand, in areas that are not as well established as numerical computation, certain important operations on data may not be built into computers or high-level programming languages. So, in order to program efficiently, an applications programmer must be familiar with algorithms that operate at the bit level; this is the case with a number of operations that occur frequently in combinatorial computing. Eventually, as these operations become better known, they are likely to be incorporated into computers and programming languages; but until then they are necessary tools of the trade for anyone who programs combinatorial algorithms.

As an example, we consider the problem of counting the number of ones in an n-bit string $B = b_n b_{n-1} \ldots b_2 b_1$. It is a natural operation if we imagine B to represent a subset S of a set U of n elements, the ones indicating which elements of U are in S. This operation, called the *bit sum*, then determines the number of elements in S.

The first algorithm that comes to mind for computing the bit sum of B is to inspect each bit in turn and, if it is a one, to increment a counter.

$c \leftarrow 0$
for $i = 1$ **to** n **do if** $b_i = 1$ **then** $c \leftarrow c + 1$

Algorithm 1.1 Computing the bit sum by looking at each bit.

On some computers, Algorithm 1.1 may be the most reasonable one to use. Most computers, however, have features that permit much faster bit sum algorithms. Assume that the memory consists of cells that can hold an n-bit word and that the computer has logical or boolean operations that operate in parallel on each bit of a word, plus arithmetic operations that interpret these words as unsigned nonnegative integers written in base 2.

$c \leftarrow 0$
while $B \neq 0$ **do** { $c \leftarrow c + 1$; $B \leftarrow B \wedge (B - 1)$ }

Algorithm 1.2 A tricky way to compute the bit sum.

Consider Algorithm 1.2. The statement "$B \leftarrow B \wedge (B - 1)$" contains an interesting operation that makes use of the assumptions stated above. $\wedge$ is the logical "and" operation that operates in parallel on each pair of bits in corresponding positions of its two arguments, B and $B - 1$. $-$ is the arithmetic operation of subtraction on integers represented in base 2. An example best shows that when this operation is executed, the rightmost 1 of B is replaced by 0.

B	11001000
$B - 1$	11000111
$B \wedge (B - 1)$	11000000

The loop in Algorithm 1.2 is executed until $B = 0$, that is, until B contains only zeros. Thus whereas the loop in Algorithm 1.1 is always executed n times, the loop in Algorithm 1.2 is only executed as many times as there are ones in B. The effect is as if Algorithm 1.2 were able to look only at the ones in B, without knowing their positions *a priori*. This algorithm is obviously efficient when applied to "sparse" words—that is, to bit strings that contain few ones and many zeros. It is "tricky" in the sense that it depends intricately on the way that

numbers are represented in the computer (in particular, negative numbers!), something entirely foreign to the original problem of counting the number of ones in a word.

The next algorithm is even more interesting. It shares with Algorithm 1.1 the important property that the loop is executed a fixed number of times, dependent on n but independent of the particular value of B. But unlike Algorithm 1.1, which repeats its loop n times, Algorithm 1.3, which we do not give explicitly, runs through its loop only $\lceil \lg n \rceil$ times.† For a typical word size of $n = 32$ (or 64), the loop is executed five (respectively six) times, which may result in a significant speedup compared to Algorithm 1.1. The assumptions required for Algorithm 1.3 to work on a given computer are about the same as those required for Algorithm 1.2, and there must be a fast way of shifting words by 1, 2, 4, 8, . . . , places.

Algorithm 1.3. is best explained by means of an example,

	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1
B	1	1	0	1	0	0	0	1

1. First, extract the odd-indexed bits $b_7\ b_5\ b_3\ b_1$ and place a zero to the left of each bit to obtain B_{odd}.

		b_7		b_5		b_3		b_1
B_{odd}	0	1	0	1	0	0	0	1

Next, extract the even-indexed bits $b_8\ b_6\ b_4\ b_2$, shift them right by one place into bit positions $b_7\ b_5\ b_3\ b_1$ and place a zero to the left of each bit to obtain B_{even}.

		b_8		b_6		b_4		b_2
B_{even}	0	1	0	0	0	0	0	0

(The newly inserted zeros are shown in small type to distinguish them from the zeros that are extracted from B.)

Then, numerically add B_{odd} and B_{even}, considered as integers written in base 2, to obtain B'.

	b'_8	b'_7	b'_6	b'_5	b'_4	b'_3	b'_2	b'_1
B_{odd}	0	1	0	1	0	0	0	1
B_{even}	0	1	0	0	0	0	0	0
B'	1	0	0	1	0	0	0	1

†The symbol lg x represents $\log_2 x$. $\lceil x \rceil$ is the *ceiling of* x: the least integer k, $k \geq x$. Similarly $\lfloor x \rfloor$ is the *floor of* x: the greatest integer k, $k \leq x$.

2. Extract the alternate pairs of bits $b'_6\ b'_5$ and $b'_2\ b'_1$ and place a pair of zeros to the left of each pair to obtain B'_{odd}.

			b'_6	b'_5			b'_2	b'_1
B'_{odd}	0	0	0	1	0	0	0	1

Next, extract the other pairs $b'_8\ b'_7$ and $b'_4\ b'_3$, shift them right by two places into bit positions $b'_6\ b'_5$ and $b'_2\ b'_1$, respectively, and insert a pair of zeros to the left of each pair to obtain B'_{even}

			b'_8	b'_7			b'_4	b'_3
B'_{even}	0	0	1	0	0	0	0	0

Numerically add B'_{odd} and B'_{even} to obtain B''.

	b''_8	b''_7	b''_6	b''_5	b''_4	b''_3	b''_2	b''_1
B''	0	0	1	1	0	0	0	1

3. Extract bits $b''_4\ b''_3\ b''_2\ b''_1$ and place four zeros to the left to obtain B''_{odd}.

					b''_4	b''_3	b''_2	b''_1
B''_{odd}	0	0	0	0	0	0	0	1

Also, extract bits $b''_8\ b''_7\ b''_6\ b''_5$, shift them right four places into bit positions $b''_4\ b''_3\ b''_2\ b''_1$, and place four zeros to the left to obtain B''_{even}.

					b''_8	b''_7	b''_6	b''_5
B''_{even}	0	0	0	0	0	0	1	1

Finally, numerically add B''_{odd} and B''_{even} to obtain $B''' = (00000100)$.

B''' is the representation in base 2 of the bit sum of B (4 in this example).

The generalization of this algorithm to arbitrary n is easy if we imagine that B is padded with zeros to the left until its length is equal to the first power of 2 greater than or equal to n.

The reader is encouraged to prove that the algorithm (in its general form) is correct. In Section 1.4 we describe a general principle of algorithm design from which the algorithm and a correctness proof follow directly. This principle is a good illustration of our claim that combinatorial computing does have general concepts and techniques from which many special cases follow.

Are there still faster algorithms for computing the bit sum of a word? Is there an "optimal" algorithm? The question of optimality of algorithms is an important one, but it can be treated only in special cases. To show that an algorithm is optimal, one must specify precisely the class of algorithms allowed and the

criterion of optimality. In the case of bit sum algorithms, such specifications would be complicated and largely arbitrary, involving specific details of how computers work. We will discuss optimality of algorithms in Section 1.5, in more simplified settings.

We can however, make a plausible argument that the following bit sum algorithm (Algorithm 1.4) is the fastest possible, since it uses a table lookup to obtain the result in essentially one operation. The penalty for this speed is an extravagant use of memory space (2^n locations), thereby making the algorithm impractical except for small values of n. The choice of an algorithm almost always involves tradeoffs among various desirable properties, and generally the better an algorithm is from one aspect, the worse it is from another.

Algorithm 1.4 is based on the idea that we can precompute the solutions to all possible questions, store the results, and then simply look them up when needed. As an example, for $n = 3$, we would store the information

Word	Bit Sum
000	0
001	1
010	1
011	2
100	1
101	2
110	2
111	3

What is the fastest way of looking up a word B in this table? Under assumptions similar to those used in the preceding algorithms, we may assume that B can be interpreted as an address of a memory cell that contains the bit sum of B, thus giving us an algorithm that requires only one memory reference.

In concluding this example, we notice the great variety of algorithms that exist for computing the bit sum, each one based on entirely different principles. Algorithms 1.1 and 1.4 solve the problem by "brute force": Algorithm 1.1 looks at each bit and so requires much time; Algorithm 1.4 stores the solution for each separate case and thus requires much space. Algorithm 1.3 is an elegant compromise.

1.2 A REPRESENTATION PROBLEM: DIFFERENCE-PRESERVING CODES

A recurring problem of great importance in combinatorial computing is to find efficient representations of the objects to be manipulated. These objects can be as simple as the bit strings of the preceding example or as complicated as