HZ BOOKS

PEARSON Addison Wesley

经典计算机科学著作最新版

计算机程序设计艺术

第4卷 第3册（双语版）

生成所有组合和分划

The Art of Computer Programming, Volume 4 Generating All Combinations and Permutations

Fascicle 3

苏运霖 译

（美）Donald E. Knuth 著

机械工业出版社
China Machine Press

计算机程序设计艺术

第4卷 第3册

生成所有组合和分划

The Art of Computer Programming Volume 4 Fascicle 3

Generating All Combinations and Partitions

（双语版）

（美） Donald E. Knuth 著
斯坦福大学
苏运霖 译

机 械 工 业 出 版 社
China Machine Press

关于算法分析的这多卷论著已经长期被公认为经典计算机科学的定义性描述。这一册以及刚刚出版的第4卷第2册揭开了人们急切等待的《计算机程序设计艺术 第4卷 组合算法》的序幕。作为关于组合查找的冗长一章的一部分，这一册开始关于生成所有组合和分划的讨论。在Knuth讨论这两个主题的过程中，读者不仅会看到很多新内容，并且会发现本册与卷1至卷3及计算机科学和数学的其他方面的丰富联系。一如既往，书中包括了大量的习题和富有挑战性的难题。

本书版权登记号：图字：01-2005-4843

图书在版编目（CIP）数据

计算机程序设计艺术 第4卷 第3册，生成所有组合和分划（双语版）/（美）克努特（Knuth, D. E.）著．苏运霖译．－北京：机械工业出版社，2006.8

书名原文：The Art of Computer Programming, Volume 4, Fascicle 3, Generating All Combinations and Partitions

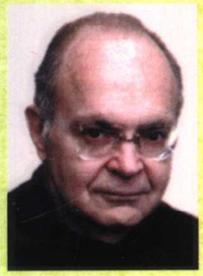
ISBN 7-111-17774-6

Ⅰ.计…　Ⅱ.①克…　②苏…　Ⅲ.程序设计　Ⅳ.TP311.1

中国版本图书馆CIP数据核字（2006）第126014号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）
责任编辑：朱起飞
北京瑞德印刷有限公司印刷·新华书店北京发行所发行
2006年8月第1版第1次印刷
170mm × 242mm · 20.5印张
定价：45.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换
本社购书热线：（010）68326294



Donald E. Knuth

（唐纳德·E.克努特，中文名高德纳）

是算法和程序设计技术的先驱者，并发明了计算机排版系统$\TeX$和METAFONT，他因这些成就和大量创造性的、影响深远的论著而誉满全球。作为斯坦福大学计算机程序设计艺术的荣誉退休教授，Knuth现正投入全部的时间来完成其关于计算机科学的史诗性的七卷集。Knuth教授获得了许多奖项和荣誉，包括美国计算机协会图灵奖 (ACM Turing Award)，美国前总统卡特授予的科学金奖 (Medal of Science)，美国数学学会斯蒂尔奖 (AMS Steele Prize)，以及极受尊重的京都奖 (Kyoto Prize)。

经典计算机科学著作最新版

计算机程序设计艺术
第1卷 第1册 (双语版)
MMIX：新千年的RISC计算机

The Art of Computer Programming,Volume 1
MMIX, A RISC Computer for the New Millennium

1 Fascicle

苏运霖 译

(美) Donald E. Knuth 著

经典计算机科学著作最新版

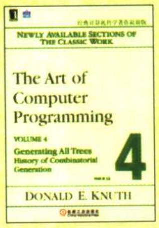
计算机程序设计艺术
第4卷 第2册 (双语版)
生成所有元组和排列

The Art of Computer Programming, Volume 4
Generating All Tuples and Permutations

2 Fascicle

苏运霖 译

(美) Donald E. Knuth 著


经典计算机科学著作最新版

Newly Available Sections of The Classic Work

The Art of Computer Programming

Volume 4
Generating All Trees
History of Combinatorial Generation

4

DONALD E. KNUTH

PREFACE

In my preface to the first edition,
I begged the reader not to draw attention to errors.
I now wish I had not done so
and am grateful to the few readers who ignored my request.
— STUART SUTHERLAND, *The International Dictionary of Psychology* (1996)

THIS BOOKLET is Fascicle 3 of *The Art of Computer Programming*, Volume 4: *Combinatorial Algorithms.* As explained in the preface to Fascicle 1 of Volume 1, I'm circulating the material in this preliminary form because I know that the task of completing Volume 4 will take many years; I can't wait for people to begin reading what I've written so far and to provide valuable feedback.

To put the material in context, this fascicle contains Sections 7.2.1.3, 7.2.1.4, and 7.2.1.5 of a long, long chapter on combinatorial searching. Chapter 7 will eventually fill three volumes (namely Volumes 4A, 4B, and 4C), assuming that I'm able to remain healthy. It will begin with a short review of graph theory, with emphasis on some highlights of significant graphs in The Stanford GraphBase, from which I will be drawing many examples. Then comes Section 7.1, which deals with bitwise manipulation and with algorithms relating to Boolean functions. Section 7.2 is about generating all possibilities, and it begins with Section 7.2.1: Generating Basic Combinatorial Patterns. Details about various useful ways to generate n-tuples appear in Section 7.2.1.1, and the generation of permutations is discussed in Section 7.2.1.2. That sets the stage for the main contents of the present booklet, namely Section 7.2.1.3 (which extends the ideas to combinations of n things taken t at a time); Section 7.2.1.4 (about partitions of an integer); and Section 7.2.1.5 (about partitions of a set). Then will come Section 7.2.1.6 (about trees) and Section 7.2.1.7 (about the history of combinatorial generation), in Fascicle 4. Section 7.2.2 will deal with backtracking in general. And so it will go on, if all goes well; an outline of the entire Chapter 7 as currently envisaged appears on the `taocp` webpage that is cited on page ii.

I had great pleasure writing this material, akin to the thrill of excitement that I felt when writing Volume 2 many years ago. As in Volume 2, where I found to my delight that the basic principles of elementary probability theory and number theory arose naturally in the study of algorithms for random number generation and arithmetic, I learned while preparing Section 7.2.1 that the basic principles of elementary combinatorics arise naturally and in a highly motivated way when we study algorithms for combinatorial generation. Thus, I found once again that a beautiful story was "out there" waiting to be told.

For example, in the present booklet we find many of the beautiful patterns formed by combinations, with and without repetition, and how they relate to famous theorems of extremal combinatorics. Then comes my chance to tell the extraordinary story of partitions; indeed, the theory of partitions is one of the nicest chapters in all of mathematics. And in Section 7.2.1.5, a little-known triangle of numbers, discovered by C. S. Peirce, turns out to simplify and unify the study of set partitions, another vital topic. Along the way I've included expositions of two mathematical techniques of great importance in the analysis of algorithms: Poisson's summation formula, and the powerful saddle point method. There are games and puzzles too, as in the previous fascicles.

My original intention was to devote far less space to these subjects. But when I saw how fundamental the ideas were for combinatorial studies in general, I knew that I could never be happy unless I covered the basics quite thoroughly. Therefore I've done my best to build a solid foundation of theoretical and practical ideas that will support many kinds of reliable superstructures.

I thank Frank Ruskey for bravely foisting an early draft of this material on college students and for telling me about his classroom experiences. Many other readers have also helped me to check the first drafts; I wish to thank especially George Clements and Svante Janson for their penetrating comments.

I shall happily pay a finder's fee of $2.56 for each error in this fascicle when it is first reported to me, whether that error be typographical, technical, or historical. The same reward holds for items that I forgot to put in the index. And valuable suggestions for improvements to the text are worth 32¢ each. (Furthermore, if you find a better solution to an exercise, I'll actually reward you with immortal glory instead of mere money, by publishing your name in the eventual book:–)

Notations that are used here and not otherwise explained can be found in the Index to Notations at the end of Volumes 1, 2, or 3. Those indexes point to the places where further information is available. Of course Volume 4 will some day contain its own Index to Notations.

Machine-language examples in all future editions of *The Art of Computer Programming* will be based on the MMIX computer, which is described in Volume 1, Fascicle 1.

Cross references to yet-unwritten material sometimes appear as '00' in the following pages; this impossible value is a placeholder for the actual numbers to be supplied later.

Happy reading!

Stanford, California D. E. K.
June 2005

CONTENTS

目　录

假若一个词不能在一行的末尾被分开，那它必定要在一个音节的末尾被分划。

——亚历山大 · 何姆(Alexander Hume)，Orthographie···of the Britan Tongue

(约1620年)

7.2.1.3. Generating all combinations. Combinatorial mathematics is often described as "the study of permutations, combinations, etc.," so we turn our attention now to combinations. A *combination of n things, taken t at a time*, often called simply a t-combination of n things, is a way to select a subset of size t from a given set of size n. We know from Eq. 1.2.6–(2) that there are exactly $\binom{n}{t}$ ways to do this; and we learned in Section 3.4.2 how to choose t-combinations at random.

Selecting t of n objects is equivalent to choosing the $n - t$ elements not selected. We will emphasize this symmetry by letting

$$n = s + t \qquad (1)$$

throughout our discussion, and we will often refer to a t-combination of n things as an "(s,t)-combination." Thus, an (s,t)-combination is a way to subdivide $s+t$ objects into two collections of sizes s and t.

If I ask how many combinations of 21 can be taken out of 25,
I do in effect ask how many combinations of 4 may be taken.
For there are just as many ways of taking 21 as there are of leaving 4.
— AUGUSTUS DE MORGAN, *An Essay on Probabilities* (1838)

There are two main ways to represent (s,t)-combinations: We can list the elements $c_t \ldots c_2c_1$ that have been selected, or we can work with binary strings $a_{n-1} \ldots a_1a_0$ for which

$$a_{n-1} + \cdots + a_1 + a_0 = t. \qquad (2)$$

The latter representation has s 0s and t 1s, corresponding to elements that are unselected or selected. The list representation $c_t \ldots c_2c_1$ tends to work out best if we let the elements be members of the set $\{0, 1, \ldots, n-1\}$ and if we list them in *decreasing* order:

$$n > c_t > \cdots > c_2 > c_1 \geq 0. \qquad (3)$$

Binary notation connects these two representations nicely, because the item list $c_t \ldots c_2c_1$ corresponds to the sum

$$2^{c_t} + \cdots + 2^{c_2} + 2^{c_1} = \sum_{k=0}^{n-1} a_k 2^k = (a_{n-1} \ldots a_1a_0)_2. \qquad (4)$$

Of course we could also list the positions $b_s \ldots b_2 b_1$ of the 0s in $a_{n-1} \ldots a_1 a_0$, where

$$n > b_s > \cdots > b_2 > b_1 \geq 0. \tag{5}$$

Combinations are important not only because subsets are omnipresent in mathematics but also because they are equivalent to many other configurations. For example, every (s,t)-combination corresponds to a combination of $s+1$ things taken t at a time *with repetitions permitted*, also called a *multicombination*, namely a sequence of integers $d_t \ldots d_2 d_1$ with

$$s \geq d_t \geq \cdots \geq d_2 \geq d_1 \geq 0. \tag{6}$$

One reason is that $d_t \ldots d_2 d_1$ solves (6) if and only if $c_t \ldots c_2 c_1$ solves (3), where

$$c_t = d_t + t - 1, \quad \ldots, \quad c_2 = d_2 + 1, \quad c_1 = d_1 \tag{7}$$

(see exercise 1.2.6–60). And there is another useful way to relate combinations with repetition to ordinary combinations, suggested by Solomon Golomb [*AMM* **75** (1968), 530–531], namely to define

$$e_j = \begin{cases} c_j, & \text{if } c_j \leq s; \\ e_{c_j - s}, & \text{if } c_j > s. \end{cases} \tag{8}$$

In this form the numbers $e_t \ldots e_1$ don't necessarily appear in descending order, but the multiset $\{e_1, e_2, \ldots, e_t\}$ is equal to $\{c_1, c_2, \ldots, c_t\}$ if and only if $\{e_1, e_2, \ldots, e_t\}$ is a set. (See Table 1 and exercise 1.)

An (s,t)-combination is also equivalent to a *composition* of $n+1$ into $t+1$ parts, namely an ordered sum

$$n+1 = p_t + \cdots + p_1 + p_0, \qquad \text{where } p_t, \ldots, p_1, p_0 \geq 1. \tag{9}$$

The connection with (3) is now

$$p_t = n - c_t, \quad p_{t-1} = c_t - c_{t-1}, \quad \ldots, \quad p_1 = c_2 - c_1, \quad p_0 = c_1 + 1. \tag{10}$$

Equivalently, if $q_j = p_j - 1$, we have

$$s = q_t + \cdots + q_1 + q_0, \qquad \text{where } q_t, \ldots, q_1, q_0 \geq 0, \tag{11}$$

a composition of s into $t+1$ *nonnegative* parts, related to (6) by setting

$$q_t = s - d_t, \quad q_{t-1} = d_t - d_{t-1}, \quad \ldots, \quad q_1 = d_2 - d_1, \quad q_0 = d_1. \tag{12}$$

Furthermore it is easy to see that an (s,t)-combination is equivalent to a path of length $s+t$ from corner to corner of an $s \times t$ grid, because such a path contains s vertical steps and t horizontal steps. Thus, combinations can be studied in at least eight different guises. Table 1 illustrates all $\binom{6}{3} = 20$ possibilities in the case $s = t = 3$.

These cousins of combinations might seem rather bewildering at first glance, but most of them can be understood directly from the binary representation $a_{n-1} \ldots a_1 a_0$. Consider, for example, the "random" bit string

$$a_{23} \ldots a_1 a_0 = 011001001000011111101101, \tag{13}$$

Table 1

THE (3, 3)-COMBINATIONS AND THEIR EQUIVALENTS

$a_5a_4a_3a_2a_1a_0$	$b_3b_2b_1$	$c_3c_2c_1$	$d_3d_2d_1$	$e_3e_2e_1$	$p_3p_2p_1p_0$	$q_3q_2q_1q_0$	path
000111	543	210	000	210	4111	3000	
001011	542	310	100	310	3211	2100	
001101	541	320	110	320	3121	2010	
001110	540	321	111	321	3112	2001	
010011	532	410	200	010	2311	1200	
010101	531	420	210	020	2221	1110	
010110	530	421	211	121	2212	1101	
011001	521	430	220	030	2131	1020	
011010	520	431	221	131	2122	1011	
011100	510	432	222	232	2113	1002	
100011	432	510	300	110	1411	0300	
100101	431	520	310	220	1321	0210	
100110	430	521	311	221	1312	0201	
101001	421	530	320	330	1231	0120	
101010	420	531	321	331	1222	0111	
101100	410	532	322	332	1213	0102	
110001	321	540	330	000	1141	0030	
110010	320	541	331	111	1132	0021	
110100	310	542	332	222	1123	0012	
111000	210	543	333	333	1114	0003	

which has $s = 11$ zeros and $t = 13$ ones, hence $n = 24$. The dual combination $b_s \ldots b_1$ lists the positions of the zeros, namely

$$23\ 20\ 19\ 17\ 16\ 14\ 13\ 12\ 11\ 4\ 1,$$

because the leftmost position is $n - 1$ and the rightmost is 0. The primal combination $c_t \ldots c_1$ lists the positions of the ones, namely

$$22\ 21\ 18\ 15\ 10\ 9\ 8\ 7\ 6\ 5\ 3\ 2\ 0.$$

The corresponding multicombination $d_t \ldots d_1$ lists the number of 0s to the right of each 1:

$$10\ 10\ 8\ 6\ 2\ 2\ 2\ 2\ 2\ 2\ 1\ 1\ 0.$$

The composition $p_t \ldots p_0$ lists the distances between consecutive 1s, if we imagine additional 1s at the left and the right:

$$2\ 1\ 3\ 3\ 5\ 1\ 1\ 1\ 1\ 1\ 2\ 1\ 2\ 1.$$

And the nonnegative composition $q_t \ldots q_0$ counts how many 0s appear between "fenceposts" represented by 1s:

$$1\ 0\ 2\ 2\ 4\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1\ 0;$$

thus we have

$$a_{n-1} \ldots a_1a_0 = 0^{q_t}10^{q_{t-1}}1 \ldots 10^{q_1}10^{q_0}. \qquad (14)$$

The paths in Table 1 also have a simple interpretation (see exercise 2).

Lexicographic generation. Table 1 shows combinations $a_{n-1}\ldots a_1a_0$ and $c_t\ldots c_1$ in lexicographic order, which is also the lexicographic order of $d_t\ldots d_1$. Notice that the dual combinations $b_s\ldots b_1$ and the corresponding compositions $p_t\ldots p_0$, $q_t\ldots q_0$ then appear in *reverse* lexicographic order.

Lexicographic order usually suggests the most convenient way to generate combinatorial configurations. Indeed, Algorithm 7.2.1.2L already solves the problem for combinations in the form $a_{n-1}\ldots a_1a_0$, since (s,t)-combinations in bitstring form are the same as permutations of the multiset $\{s\cdot 0, t\cdot 1\}$. That general-purpose algorithm can be streamlined in obvious ways when it is applied to this special case. (See also exercise 7.1–00, which presents a remarkable sequence of seven bitwise operations that will convert any given binary number $(a_{n-1}\ldots a_1a_0)_2$ to the lexicographically next t-combination, assuming that n does not exceed the computer's word length.)

Let's focus, however, on generating combinations in the other principal form $c_t\ldots c_2c_1$, which is more directly relevant to the ways in which combinations are often needed, and which is more compact than the bit strings when t is small compared to n. In the first place we should keep in mind that a simple sequence of nested loops will do the job nicely when t is very small. For example, when $t=3$ the following instructions suffice:

```
For $c_3 = 2, 3, \ldots, n-1$ (in this order) do the following:
 For $c_2 = 1, 2, \ldots, c_3-1$ (in this order) do the following:
  For $c_1 = 0, 1, \ldots, c_2-1$ (in this order) do the following:
   Visit the combination $c_3c_2c_1$.                                  (15)
```

(See the analogous situation in 7.2.1.1–(3).)

On the other hand when t is variable or not so small, we can generate combinations lexicographically by following the general recipe discussed after Algorithm 7.2.1.2L, namely to find the rightmost element c_j that can be increased and then to set the subsequent elements $c_{j-1}\ldots c_1$ to their smallest possible values:

Algorithm L (*Lexicographic combinations*). This algorithm generates all t-combinations $c_t\ldots c_2c_1$ of the n numbers $\{0,1,\ldots,n-1\}$, given $n\ge t\ge 0$. Additional variables c_{t+1} and c_{t+2} are used as sentinels.

L1. [Initialize.] Set $c_j \leftarrow j-1$ for $1\le j\le t$; also set $c_{t+1}\leftarrow n$ and $c_{t+2}\leftarrow 0$.

L2. [Visit.] Visit the combination $c_t\ldots c_2c_1$.

L3. [Find j.] Set $j\leftarrow 1$. Then, while $c_j+1=c_{j+1}$, set $c_j\leftarrow j-1$ and $j\leftarrow j+1$; eventually the condition $c_j+1\ne c_{j+1}$. will occur

L4. [Done?] Terminate the algorithm if $j>t$.

L5. [Increase c_j.] Set $c_j\leftarrow c_j+1$ and return to L2. ▌

The running time of this algorithm is not difficult to analyze. Step L3 sets $c_j\leftarrow j-1$ just after visiting a combination for which $c_{j+1}=c_1+j$, and the number of such combinations is the number of solutions to the inequalities

$$n > c_t > \cdots > c_{j+1} \ge j; \qquad (16)$$

but this formula is equivalent to a $(t-j)$-combination of the $n-j$ objects $\{n-1,\ldots,j\}$, so the assignment $c_j \leftarrow j-1$ occurs exactly $\binom{n-j}{t-j}$ times. Summing for $1 \le j \le t$ tells us that the loop in step L3 is performed

$$\binom{n-1}{t-1}+\binom{n-2}{t-2}+\cdots+\binom{n-t}{0} = \binom{n-1}{s}+\binom{n-2}{s}+\cdots+\binom{s}{s} = \binom{n}{s+1} \qquad (17)$$

times altogether, or an average of

$$\binom{n}{s+1}\Big/\binom{n}{t} = \frac{n!}{(s+1)!\,(t-1)!}\Big/\frac{n!}{s!\,t!} = \frac{t}{s+1} \qquad (18)$$

times per visit. This ratio is less than 1 when $t \le s$, so Algorithm L is quite efficient in such cases.

But the quantity $t/(s+1)$ can be embarrassingly large when t is near n and s is small. Indeed, Algorithm L occasionally sets $c_j \leftarrow j-1$ needlessly, at times when c_j already equals $j-1$. Further scrutiny reveals that we need not always search for the index j that is needed in steps L4 and L5, since the correct value of j can often be predicted from the actions just taken. For example, after we have increased c_4 and reset $c_3c_2c_1$ to their starting values 210, the next combination will inevitably increase c_3. These observations lead to a tuned-up version of the algorithm:

Algorithm T (*Lexicographic combinations*). This algorithm is like Algorithm L, but faster. It also assumes, for convenience, that $t < n$.

```
T1. [Initialize.] Set $c_j \leftarrow j-1$ for $1 \le j \le t$; then set $c_{t+1} \leftarrow n$, $c_{t+2} \leftarrow 0$, and
    $j \leftarrow t$.
T2. [Visit.] (At this point $j$ is the smallest index such that $c_{j+1} > j$.) Visit the
    combination $c_t \ldots c_2c_1$. Then, if $j > 0$, set $x \leftarrow j$ and go to step T6.
T3. [Easy case?] If $c_1 + 1 < c_2$, set $c_1 \leftarrow c_1 + 1$ and return to T2. Otherwise set
    $j \leftarrow 2$.
T4. [Find $j$.] Set $c_{j-1} \leftarrow j-2$ and $x \leftarrow c_j + 1$. If $x = c_{j+1}$, set $j \leftarrow j+1$ and
    repeat this step until $x \ne c_{j+1}$.
T5. [Done?] Terminate the algorithm if $j > t$.
T6. [Increase $c_j$.] Set $c_j \leftarrow x$, $j \leftarrow j-1$, and return to T2. ▌
```

Now $j = 0$ in step T2 if and only if $c_1 > 0$, so the assignments in step T4 are never redundant. Exercise 6 carries out a complete analysis of Algorithm T.

Notice that the parameter n appears only in the initialization steps L1 and T1, not in the principal parts of Algorithms L and T. Thus we can think of the process as generating the first $\binom{n}{t}$ combinations of an *infinite* list, which depends only on t. This simplification arises because the list of t-combinations for $n+1$ things begins with the list for n things, under our conventions; we have been using lexicographic order on the decreasing sequences $c_t \ldots c_1$ for this very reason, instead of working with the increasing sequences $c_1 \ldots c_t$.

Derrick Lehmer noticed another pleasant property of Algorithms L and T [*Applied Combinatorial Mathematics*, edited by E. F. Beckenbach (1964), 27–30]:

Theorem L. *The combination $c_t \ldots c_2 c_1$ is visited after exactly*

$$\binom{c_t}{t} + \cdots + \binom{c_2}{2} + \binom{c_1}{1} \tag{19}$$

other combinations have been visited.

Proof. There are $\binom{c_k}{k}$ combinations $c'_t \ldots c'_2 c'_1$ with $c'_j = c_j$ for $t \ge j > k$ and $c'_k < c_k$, namely $c_t \ldots c_{k+1}$ followed by the k-combinations of $\{0, \ldots, c_k - 1\}$. ▌

When $t = 3$, for example, the numbers

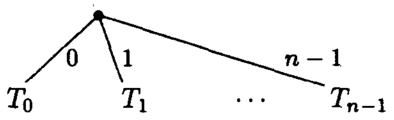
$$\binom{2}{3}+\binom{1}{2}+\binom{0}{1},\ \binom{3}{3}+\binom{1}{2}+\binom{0}{1},\ \binom{3}{3}+\binom{2}{2}+\binom{0}{1},\ \ldots,\ \binom{5}{3}+\binom{4}{2}+\binom{3}{1}$$

that correspond to the combinations $c_3c_2c_1$ in Table 1 simply run through the sequence 0, 1, 2, …, 19. Theorem L gives us a nice way to understand the *combinatorial number system* of degree t, which represents every nonnegative integer N uniquely in the form

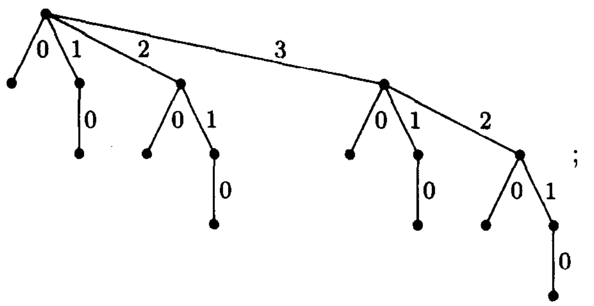
$$N = \binom{n_t}{t} + \cdots + \binom{n_2}{2} + \binom{n_1}{1}, \qquad n_t > \cdots > n_2 > n_1 \ge 0. \tag{20}$$

[See Ernesto Pascal, *Giornale di Matematiche* **25** (1887), 45–49.]

Binomial trees. The family of trees T_n defined by

$T_0 = \bullet$, $T_n =$  for $n > 0$

arises in several important contexts and sheds further light on combination generation. For example, T_4 is

 ; (22)

and T_5, rendered more artistically, appears as the frontispiece to Volume 1 of this series of books.

Notice that T_n is like T_{n-1}, except for an additional copy of T_{n-1}; therefore T_n has 2^n nodes altogether. Furthermore, the number of nodes on level t is the binomial coefficient $\binom{n}{t}$; this fact accounts for the name "binomial tree." Indeed, the sequence of labels encountered on the path from the root to each node on level t defines a combination $c_t \ldots c_1$, and all combinations occur in lexicographic order from left to right. Thus, Algorithms L and T can be regarded as procedures to traverse the nodes on level t of the binomial tree T_n.

The infinite binomial tree T_∞ is obtained by letting $n \to \infty$ in (21). The root of this tree has infinitely many branches, but every node except for the overall root at level 0 is the root of a finite binomial subtree. All possible t-combinations appear in lexicographic order on level t of T_∞.

Let's get more familiar with binomial trees by considering all possible ways to pack a rucksack. More precisely, suppose we have n items that take up respectively $w_{n-1}, \ldots, w_1, w_0$ units of capacity, where

$$w_{n-1} \geq \cdots \geq w_1 \geq w_0; \tag{23}$$

we want to generate all binary vectors $a_{n-1} \ldots a_1 a_0$ such that

$$a \cdot w = a_{n-1}w_{n-1} + \cdots + a_1 w_1 + a_0 w_0 \leq N, \tag{24}$$

where N is the total capacity of a rucksack. Equivalently, we want to find all subsets C of $\{0, 1, \ldots, n-1\}$ such that $w(C) = \sum_{c \in C} w_c \leq N$; such subsets will be called *feasible*. We will write a feasible subset as $c_1 \ldots c_t$, where $c_1 > \cdots > c_t \geq 0$, numbering the subscripts differently from the convention of (3) above because t is variable in this problem.

Every feasible subset corresponds to a node of T_n, and our goal is to visit each feasible node. Clearly the parent of every feasible node is feasible, and so is the left sibling, if any; therefore a simple tree exploration procedure works well:

Algorithm F (*Filling a rucksack*). This algorithm generates all feasible ways $c_1 \ldots c_t$ to fill a rucksack, given $w_{n-1}, \ldots, w_1, w_0$, and N. We let $\delta_j = w_j - w_{j-1}$ for $1 \leq j < n$.

F1. [Initialize.] Set $t \leftarrow 0$, $c_0 \leftarrow n$, and $r \leftarrow N$.

F2. [Visit.] Visit the combination $c_1 \ldots c_t$, which uses $N - r$ units of capacity.

F3. [Try to add w_0.] If $c_t > 0$ and $r \geq w_0$, set $t \leftarrow t+1$, $c_t \leftarrow 0$, $r \leftarrow r - w_0$, and return to F2.

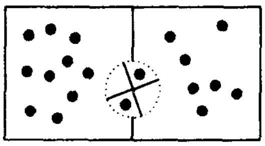
F4. [Try to increase c_t.] Terminate if $t = 0$. Otherwise, if $c_{t-1} > c_t + 1$ and $r \geq \delta_{c_t+1}$, set $c_t \leftarrow c_t + 1$, $r \leftarrow r - \delta_{c_t}$, and return to F2.

F5. [Remove c_t.] Set $r \leftarrow r + w_{c_t}$, $t \leftarrow t - 1$, and return to F4. ▌

Notice that the algorithm implicitly visits nodes of T_n in preorder, skipping over unfeasible subtrees. An element $c > 0$ is placed in the rucksack, if it fits, just after the procedure has explored all possibilities using element $c - 1$ in its place. The running time is proportional to the number of feasible combinations visited (see exercise 20).

Incidentally, the classical "knapsack problem" of operations research is different: It asks for a feasible subset C such that $v(C) = \sum_{c \in C} v(c)$ is maximum, where each item c has been assigned a value $v(c)$. Algorithm F is not a particularly good way to solve that problem, because it often considers cases that could be ruled out. For example, if C and C' are subsets of $\{1, \ldots, n-1\}$ with $w(C) \leq w(C') \leq N - w_0$ and $v(C) \geq v(C')$, Algorithm F will examine both $C \cup \{0\}$ and $C' \cup \{0\}$, but the latter subset will never improve the maximum. We will consider methods for the classical knapsack problem later; Algorithm F is intended only for situations when *all* of the feasible possibilities are potentially relevant.

Gray codes for combinations. Instead of merely generating all combinations, we often prefer to visit them in such a way that each one is obtained by making only a small change to its predecessor.



For example, we can ask for what Nijenhuis and Wilf have called a "revolving door algorithm": Imagine two rooms that contain respectively s and t people, with a revolving door between them. Whenever a person goes into the opposite room, somebody else comes out. Can we devise a sequence of moves so that each (s,t)-combination occurs exactly once?

The answer is yes, and in fact a huge number of such patterns exist. For example, it turns out that if we examine all n-bit strings $a_{n-1}\ldots a_1a_0$ in the well-known order of Gray binary code (Section 7.2.1.1), but select only those that have exactly s 0s and t 1s, the resulting strings form a revolving-door code.

Here's the proof: Gray binary code is defined by the recurrence $\Gamma_n = 0\Gamma_{n-1}, 1\Gamma^R_{n-1}$ of 7.2.1.1–(5), so its (s,t) subsequence satisfies the recurrence

$$\Gamma_{st} = 0\Gamma_{(s-1)t},\ 1\Gamma^R_{s(t-1)} \tag{25}$$

when $st > 0$. We also have $\Gamma_{s0} = 0^s$ and $\Gamma_{0t} = 1^t$. Therefore it is clear by induction that Γ_{st} begins with 0^s1^t and ends with 10^s1^{t-1} when $st > 0$. The transition at the comma in (25) is from the last element of $0\Gamma_{(s-1)t}$ to the last element of $1\Gamma_{s(t-1)}$, namely from $010^{s-1}1^{t-1} = 010^{s-1}11^{t-2}$ to $110^s1^{t-2} = 110^{s-1}01^{t-2}$ when $t \geq 2$, and this satisfies the revolving-door constraint. The case $t = 1$ also checks out. For example, Γ_{33} is given by the columns of

000111	011010	110001	101010	
001101	011100	110010	101100	
001110	010101	110100	100101	(26)
001011	010110	111000	100110	
011001	010011	101001	100011	

and Γ_{23} can be found in the first two columns of this array. One more turn of the door takes the last element into the first. [These properties of Γ_{st} were discovered by D. T. Tang and C. N. Liu, *IEEE Trans.* **C-22** (1973), 176–180; a loopless implementation was presented by J. R. Bitner, G. Ehrlich, and E. M. Reingold, *CACM* **19** (1976), 517–521.]

When we convert the bit strings $a_5a_4a_3a_2a_1a_0$ in (26) to the corresponding index-list forms $c_3c_2c_1$, a striking pattern becomes evident:

210	431	540	531	
320	432	541	532	
321	420	542	520	(27)
310	421	543	521	
430	410	530	510	

The first components c_3 occur in increasing order; but for each fixed value of c_3, the values of c_2 occur in *decreasing* order. And for fixed c_3c_2, the values of c_1 are again increasing. The same is true in general: *All combinations $c_t \ldots c_2c_1$*

appear in lexicographic order of

$$(c_t,\ -c_{t-1},\ c_{t-2},\ \ldots,\ (-1)^{t-1}c_1) \tag{28}$$

in the revolving-door Gray code Γ_{st}. This property follows by induction, because (25) becomes

$$\Gamma_{st} \;=\; \Gamma_{(s-1)t},\ (s{+}t{-}1)\Gamma^R_{s(t-1)} \tag{29}$$

for $st > 0$ when we use index-list notation instead of bitstring notation. Consequently the sequence can be generated efficiently by the following algorithm due to W. H. Payne [see *ACM Trans. Math. Software* **5** (1979), 163–172]:

Algorithm R (*Revolving-door combinations*). This algorithm generates all t-combinations $c_t \ldots c_2c_1$ of $\{0,1,\ldots,n-1\}$ in lexicographic order of the alternating sequence (28), assuming that $n > t > 1$. Step R3 has two variants, depending on whether t is even or odd.

R1. [Initialize.] Set $c_j \leftarrow j-1$ for $t \geq j \geq 1$, and $c_{t+1} \leftarrow n$.

R2. [Visit.] Visit the combination $c_t \ldots c_2c_1$.

R3. [Easy case?] If t is odd: If $c_1 + 1 < c_2$, increase c_1 by 1 and return to R2, otherwise set $j \leftarrow 2$ and go to R4. If t is even: If $c_1 > 0$, decrease c_1 by 1 and return to R2, otherwise set $j \leftarrow 2$ and go to R5.

R4. [Try to decrease c_j.] (At this point $c_j = c_{j-1} + 1$.) If $c_j \geq j$, set $c_j \leftarrow c_{j-1}$, $c_{j-1} \leftarrow j-2$, and return to R2. Otherwise increase j by 1.

R5. [Try to increase c_j.] (At this point $c_{j-1} = j-2$.) If $c_j + 1 < c_{j+1}$, set $c_{j-1} \leftarrow c_j$, $c_j \leftarrow c_j + 1$, and return to R2. Otherwise increase j by 1, and go to R4 if $j \leq t$. ▌

Exercises 21–25 explore further properties of this interesting sequence. One of them is a nice companion to Theorem L: *The combination $c_tc_{t-1}\ldots c_2c_1$ is visited by Algorithm R after exactly*

$$N = \binom{c_t{+}1}{t} - \binom{c_{t-1}{+}1}{t{-}1} + \cdots + (-1)^t\binom{c_2{+}1}{2} - (-1)^t\binom{c_1{+}1}{1} - [t \text{ odd}] \tag{30}$$

other combinations have been visited. We may call this the representation of N in the "alternating combinatorial number system" of degree t; one consequence, for example, is that every positive integer has a unique representation of the form $N = \binom{a}{3} - \binom{b}{2} + \binom{c}{1}$ with $a > b > c > 0$. Algorithm R tells us how to add 1 to N in this system.

Although the strings of (26) and (27) are not in lexicographic order, they are examples of a more general concept called *genlex order*, a name coined by Timothy Walsh. A sequence of strings $\alpha_1, \ldots, \alpha_N$ is said to be in genlex order when all strings with a common prefix occur consecutively. For example, all 3-combinations that begin with 53 appear together in (27).

Genlex order means that the strings can be arranged in a trie structure, as in Fig. 31 of Section 6.3, but with the children of each node ordered arbitrarily. When a trie is traversed in any order such that each node is visited just before or just after its descendants, all nodes with a common prefix — that is, all nodes of