HZ BOOKS
华章科技

PEARSON

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0册 （双语版）

组合算法与布尔函数概论

The Art of Computer Programming, Volume 4
Introduction to Combinatorial Algorithms and Boolean Functions

0

Fascicle

黄林鹏 等译

(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0册

组合算法与布尔函数概论

The Art of Computer Programming
Volume 4, Fascicle 0
Introduction to Combinatorial
Algorithms and Boolean Functions

（双语版）

(美) ——著

斯坦福大学

黄林鹏 等译

机械工业出版社
China Machine Press

本书是《计算机程序设计艺术，第4卷：组合算法》的第0册。本册介绍了组合搜索的历史和演化，涉及组合搜索技术的理论和实践应用；探究了与布尔函数相关的所有重要问题，考察了如何最有效地计算一个布尔函数的值的技术。本册是《计算机程序设计艺术》第7章，即组合搜索这一长篇宏论的起始。

本书版权登记号：图字：01-2009-1359

图书在版编目（CIP）数据

计算机程序设计艺术 第4卷 第0册：组合算法与布尔函数概论（双语版）/（美）克努特（Knuth, D. E.）著；黄林鹏译. —北京：机械工业出版社，2010.6

书名原文：The Art Of Computer Programming, Volume 4, Fascicle 0, Introduction to Combinatorial Algorithms and Boolean Functions

ISBN 978-7-111-30334-3

Ⅰ. 计… Ⅱ. ① 克… ② 黄… Ⅲ. 程序设计—英、汉 Ⅳ. TP311.1

中国版本图书馆CIP数据核字（2010）第063083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：张少波

北京瑞德印刷有限公司印刷

2010年8月第1版第1次印刷

170mm × 242mm . 28印张

标准书号：ISBN 978-7-111-30334-3

定价：69.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991；88361066

购书热线：（010）68326294；88379649；68995259

投稿热线：（010）88379604

读者信箱：hzjsj@hzbook.com

PREFACE

To put all the good stuff into one book is patently impossible,
and attempting even to be reasonably comprehensive
about certain aspects of the subject is likely to lead to runaway growth.
— GERALD B. FOLLAND, "Editor's Corner" (2005)

La dernière chose qu'on trouve en faisant un ouvrage
est de savoir celle qu'il faut mettre la première.
— BLAISE PASCAL, *Pensées* 740 (c. 1660)

THIS BOOKLET is Fascicle 0 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 the opening sections intended to launch a long, long chapter on combinatorial algorithms. Chapter 7 is planned to be by far the longest single chapter of *The Art of Computer Programming*; it will eventually fill at least three volumes (namely Volumes 4A, 4B, and 4C), assuming that I'm able to remain healthy. Like the second-longest chapter (Chapter 5), it begins with pump-priming introductory material that comes before the main text, including dozens of exercises to get the ball rolling. A long voyage lies ahead, and some important provisions need to be brought on board before we embark. Furthermore I want to minimize the shock of transition between Chapter 6 and the new chapter, because Chapter 6 was originally written and published more than thirty years ago.

Chapter 7 proper begins with Section 7.1: Zeros and Ones, which is another sort of introduction, at a different level. It dives into the all-important topics that surround the study of Boolean functions, which essentially underly everything that computers do. Subsection 7.1.1, "Boolean basics," attempts to erect a solid foundation of theoretical and practical ideas on which we shall build significant superstructures later; subsection 7.1.2, "Boolean evaluation," considers how to compute Boolean functions with maximum efficiency.

The remaining parts of Section 7.1 — namely 7.1.3, "Bitwise tricks and techniques," and 7.1.4, "Binary decision diagrams" — will be published soon as Volume 4, Fascicle 1. Then comes Section 7.2, Generating All Possibilities; the fascicles for Section 7.2.1, "Generating basic combinatorial patterns," have already appeared in print. 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.

These introductory sections have turned out to have more than twice as many exercises as I had originally planned. In fact, the total number of exercises in this fascicle (366) is almost unbelievable. But many of them are quite simple, intended to reinforce the reader's understanding of basic definitions, or to acquaint readers with the joys of *The Stanford GraphBase*. Other exercises were simply irresistible, as they cried out to be included here — although, believe it or not, I did reject more potential leads than I actually followed up.

I would like to express my indebtedness to the late Robert W Floyd, who made dozens of valuable suggestions when I asked him to look over the first draft of this material in 1977. Thanks also to Robin Wilson of the Open University for his careful reading and many detailed suggestions; and to hundreds of readers who provided fantastic feedback on early drafts that circulated on the Internet.

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. (See also the entries under "Notation" in the present booklet.) 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.
January 2008

PREFACE TO VOLUME 4

THE TITLE of Volume 4 is *Combinatorial Algorithms*, and when I proposed it I was strongly inclined to add a subtitle: *The Kind of Programming I Like Best.* My editors have decided to tone down such exuberance, but the fact remains that programs with a combinatorial flavor have always been my favorites.

On the other hand I've often been surprised to find that, in many people's minds, the word "combinatorial" is linked with computational difficulty. Indeed, Samuel Johnson, in his famous dictionary of the English language (1755), said that the corresponding noun "is now generally used in an ill sense." Colleagues tell me tales of woe, in which they report that "the combinatorics of the situation defeated us." Why is it that, for me, combinatorics arouses feelings of pure pleasure, yet for many others it evokes pure panic?

It's true that combinatorial problems are often associated with humongously large numbers. Johnson's dictionary entry also included a quote from Ephraim Chambers, who had stated that the total number of words of length 24 or less, in a 24-letter alphabet, is 1,391,724,288,887,252,999,425,128,493,402,200. The corresponding number for a 10-letter alphabet is 11,111,111,110; and it's only 3905 when the number of letters is 5. Thus a "combinatorial explosion" certainly does occur as the size of the alphabet grows from 5 to 10 to 24 and beyond.

Computing machines have become tremendously more powerful throughout my life. As I write these words, I know that they are being processed by a "laptop" whose speed is more than 100,000 times faster than the trusty IBM Type 650 computer to which I'm dedicating these books; my current machine's memory capacity is also more than 100,000 times greater. Tomorrow's computers will be even faster and more capacious. But these amazing advances have not diminished people's craving for answers to combinatorial questions; quite the contrary. Our once-unimaginable ability to compute so rapidly has raised our expectations, and whetted our appetite for more — because, in fact, the size of a combinatorial problem can increase more than 100,000-fold when n simply increases by 1.

Combinatorial algorithms can be defined informally as techniques for the high-speed manipulation of combinatorial objects such as permutations or graphs. We typically try to find patterns or arrangements that are the best possible ways to satisfy certain constraints. The number of such problems is vast, and the art of writing such programs is especially important and appealing because a single good idea can save years or even centuries of computer time.

Indeed, the fact that good algorithms for combinatorial problems can have a terrific payoff has led to terrific advances in the state of the art. Many problems that once were thought to be intractable can now be polished off with ease, and

many algorithms that once were known to be good have now become better. Starting about 1970, computer scientists began to experience a phenomenon that we called "Floyd's Lemma": Problems that seemed to need n^3 operations could actually be solved in $O(n^2)$; problems that seemed to require n^2 could be handled in $O(n \log n)$; and $n \log n$ was often reducible to $O(n)$. More difficult problems saw a reduction in running time from $O(2^n)$ to $O(1.5^n)$ to $O(1.3^n)$, etc. Other problems remained difficult in general, but they were found to have important special cases that are much simpler. Many combinatorial questions that I once thought would never be answered have now been resolved, and these breakthroughs are due mainly to improvements in algorithms rather than to improvements in processor speeds.

By 1975, such research was advancing so rapidly that a substantial fraction of the papers published in leading journals of computer science were devoted to combinatorial algorithms. And the advances weren't being made only by people in the core of computer science; significant contributions were coming from workers in electrical engineering, artificial intelligence, operations research, mathematics, physics, statistics, and other fields. I was trying to complete Volume 4 of *The Art of Computer Programming*, but instead I felt like I was sitting on the lid of a boiling kettle: I was confronted with a combinatorial explosion of another kind, a prodigious explosion of new ideas!

This series of books was born at the beginning of 1962, when I naïvely wrote out a list of tentative chapter titles for a 12-chapter book. At that time I decided to include a brief chapter about combinatorial algorithms, just for fun. "Hey look, most people use computers to deal with numbers, but we can also write programs that deal with patterns." In those days it was easy to give a fairly complete description of just about every combinatorial algorithm that was known. And even by 1966, when I'd finished a first draft of about 3000 handwritten pages for that already-overgrown book, fewer than 100 of those pages belonged to Chapter 7. I had absolutely no idea that what I'd foreseen as a sort of "salad course" would eventually turn out to be the main dish.

The great combinatorial fermentation of 1975 has continued to churn, as more and more people have begun to participate. New ideas improve upon the older ones, but rarely replace them or make them obsolete. So of course I've had to abandon any hopes that I once had of being able to surround the field, to write a definitive book that sets everything in order and provides one-stop shopping for everyone who has combinatorial problems to solve. It's almost never possible to discuss a subtopic and say, "Here's the final solution: end of story." Instead, I must restrict myself to explaining the most important principles that seem to underlie all of the efficient combinatorial methods that I've encountered so far. At present I've accumulated more than twice as much raw material for Volume 4 as for all of Volumes 1–3 combined.

This sheer mass of material implies that the once-planned "Volume 4" must actually become several physical volumes. You are now looking at Volume 4A. Volumes 4B and 4C will exist someday, assuming that I'm able to remain healthy; and (who knows?) there may also be Volumes 4D, 4E, ...; but surely not 4Z.

My plan is to go systematically through the files that I've amassed since 1962 and to tell the stories that I believe are still waiting to be told, to the best of my ability. I can't aspire to completeness, but I do want to give proper credit to all of the pioneers who have been responsible for key ideas; so I won't scrimp on historical details. Furthermore, whenever I learn something that I think is likely to remain important 50 years from now, something that can also be explained elegantly in a paragraph or two, I can't bear to leave it out. Conversely, difficult material that requires a lengthy proof is beyond the scope of these books, unless the subject matter is truly fundamental.

OK, it's clear that the field of Combinatorial Algorithms is vast, and I can't cover it all. What are the most important things that I'm leaving out? My biggest blind spot, I think, is geometry, because I've always been much better at visualizing and manipulating algebraic formulas than objects in space. Therefore I don't attempt to deal in these books with combinatorial problems that are related to computational geometry, such as close packing of spheres, or clustering of data points in n-dimensional Euclidean space, or even the Steiner tree problem in the plane. More significantly, I tend to shy away from polyhedral combinatorics, and from approaches that are based primarily on linear programming, integer programming, or semidefinite programming. Those topics are treated well in many other books on the subject, and they rely on geometrical intuition. Purely combinatorial developments are easier for me to understand.

I also must confess a bias against algorithms that are efficient only in an asymptotic sense, algorithms whose superior performance doesn't begin to "kick in" until the size of the problem exceeds the size of the universe. A great many publications nowadays are devoted to algorithms of that kind. I can understand why the contemplation of ultimate limits has intellectual appeal and carries an academic cachet; but in *The Art of Computer Programming* I tend to give short shrift to any methods that I would never consider using myself in an actual program. (There are, of course, exceptions to this rule, especially with respect to basic concepts in the core of the subject. Some impractical methods are simply too beautiful and/or too insightful to be excluded; others provide instructive examples of what *not* to do.)

Furthermore, as in earlier volumes of this series, I'm intentionally concentrating almost entirely on *sequential* algorithms, even though computers are increasingly able to carry out activities in parallel. I'm unable to judge what ideas about parallelism are likely to be useful five or ten years from now, let alone fifty, so I happily leave such questions to others who are wiser than I. Sequential methods, by themselves, already test the limits of my own ability to discern what the artful programmers of tomorrow will want to know.

The main decision that I needed to make when planning how to present this material was whether to organize it by problems or by techniques. Chapter 5 in Volume 3, for example, was devoted to a single problem, the sorting of data into order; more than two dozen techniques were applied to different aspects of that problem. Combinatorial algorithms, by contrast, involve many different problems, which tend to be attacked with a smaller repertoire of techniques.

I finally decided that a mixed strategy would work better than any pure approach. Thus, for example, these books treat the problem of finding shortest paths in Section 7.3, and problems of connectivity in Section 7.4.1; but many other sections are devoted to basic techniques, such as the use of Boolean algebra (Section 7.1), backtracking (Section 7.2), matroid theory (Section 7.6), or dynamic programming (Section 7.7). The famous Traveling Salesrep Problem, and other classic combinatorial tasks related to covering, coloring, and packing, have no sections of their own, but they come up several times in different places as they are treated by different methods.

I've mentioned great progress in the art of combinatorial computing, but I don't mean to imply that all combinatorial problems have actually been tamed. When the running time of a computer program goes ballistic, its programmers shouldn't expect to find a silver bullet for their needs in this book. The methods described here will often work a great deal faster than the first approaches that a programmer tries; but let's face it: Combinatorial problems get huge very quickly. We can even prove rigorously that a certain small, natural problem will *never* have a feasible solution in the real world, although it is solvable in principle (see the theorem of Stockmeyer and Meyer in Section 7.1.2). In other cases we cannot prove as yet that no decent algorithm for a given problem exists, but we know that such methods are unlikely, because any efficient algorithm would yield a good way to solve thousands of other problems that have stumped the world's greatest experts (see the discussion of NP-completeness in Section 7.9).

Experience suggests that new combinatorial algorithms will continue to be invented, for new combinatorial problems and for newly identified variations or special cases of old ones; and that people's appetite for such algorithms will also continue to grow. The art of computer programming continually reaches new heights when programmers are faced with challenges such as these. Yet today's methods are also likely to remain relevant.

Most of this book is self-contained, although there are frequent tie-ins with the topics discussed in Volumes 1–3. Low-level details of machine language programming have been covered extensively in those volumes, so the algorithms in the present book are usually specified only at an abstract level, independent of any machine. However, some aspects of combinatorial programming are heavily dependent on low-level details that didn't arise before; in such cases, all examples in this book are based on the `MMIX` computer, which supersedes the `MIX` machine that was defined in early editions of Volume 1. Details about `MMIX` appear in a paperback supplement to that volume called *The Art of Computer Programming*, Volume 1, Fascicle 1; they're also available on the Internet, together with downloadable assemblers and simulators.

Another downloadable resource, a collection of programs and data called *The Stanford GraphBase*, is cited extensively in the examples of this book. Readers are encouraged to play with it, in order to learn about combinatorial algorithms in what I think will be the most efficient and most enjoyable way.

Incidentally, while writing the introductory material at the beginning of Chapter 7, I was pleased to note that it was natural to mention some work of

my Ph.D. thesis advisor, Marshall Hall, Jr. (1910–1990), as well as some work of *his* thesis advisor, Oystein Ore (1899–1968), as well as some work of *his* thesis advisor, Thoralf Skolem (1887–1963). Skolem's advisor, Axel Thue (1863–1922), was already present in Chapter 6.

I'm immensely grateful to the hundreds of readers who have helped me to ferret out numerous mistakes that I made in early drafts of this volume, which were originally posted on the Internet and subsequently printed in paperback fascicles. But I fear that other errors still lurk among the details collected here, and I want to correct them as soon as possible. Therefore I will cheerfully pay \$2.56 to the first finder of each technical, typographical, or historical error. The `taocp` webpage cited on page ii contains a current listing of all corrections that have been reported to me.

Stanford, California D. E. K.
April 2008

Naturally, I am responsible for the remaining errors—
although, in my opinion, my friends could have caught a few more.
— CHRISTOS H. PAPADIMITRIOU, *Computational Complexity* (1995)

A note on references. References to *IEEE Transactions* include a letter code for the type of transactions, in boldface preceding the volume number. For example, '*IEEE Trans.* **C-35**' means the *IEEE Transactions on Computers*, volume 35. The IEEE no longer uses these convenient letter codes, but the codes aren't too hard to decipher: '**EC**' once stood for "Electronic Computers," '**IT**' for "Information Theory," '**SE**' for "Software Engineering," and '**SP**' for "Signal Processing," etc.; '**CAD**' meant "Computer-Aided Design of Integrated Circuits and Systems."

The author is especially grateful to the Addison–Wesley Publishing Company for its patience in waiting a full decade for this manuscript from the date the contract was signed.

— FRANK HARARY, *Graph Theory* (1968)

Bitte ein Bit!

— Slogan of Bitburger Brauerei (1951)

CONTENTS

目　录

PREFACE ……*iii*
PREFACE TO VOLUME 4 ……*v*

Chapter 7 Combinatorial Searching

7.1 **Zeros and Ones** ……*47*
7.1.1 Boolean Basics ……*47*
7.1.2 Boolean Evaluation ……*96*
Answers to Exercises ……*134*
Index and Glossary ……*201*

译者序 ……*219*
前言 ……*221*
第4卷前言 ……*223*

第7章 组合搜索

7.1 0和1 ……*274*
7.1.1 布尔基础 ……*274*
7.1.2 布尔求值 ……*321*
习题答案 ……*356*

CHAPTER SEVEN

COMBINATORIAL SEARCHING

You shall seeke all day ere you finde them,
& when you have them, they are not worth the search.
— BASSANIO, in *The Merchant of Venice* (Act I, Scene 1, Line 117)

Amid the action and reaction of so dense a swarm of humanity,
every possible combination of events may be expected to take place,
and many a little problem will be presented which may be striking and bizarre.
— SHERLOCK HOLMES, in *The Adventure of the Blue Carbuncle* (1892)

The field of combinatorial algorithms is too vast to cover
in a single paper or even in a single book.
— ROBERT E. TARJAN (1976)

While jostling against all manner of people
it has been impressed upon my mind that the successful ones
are those who have a natural faculty for solving puzzles.
Life is full of puzzles, and we are called upon
to solve such as fate throws our way.
— SAM LOYD, JR. (1927)

COMBINATORICS is the study of the ways in which discrete objects can be arranged into various kinds of patterns. For example, the objects might be $2n$ numbers $\{1, 1, 2, 2, \ldots, n, n\}$, and we might want to place them in a row so that exactly k numbers occur between the two appearances of each digit k. When $n = 3$ there is essentially only one way to arrange such "Langford pairs," namely 231213 (and its left-right reversal); similarly, there's also a unique solution when $n = 4$. Many other types of combinatorial patterns are discussed below.

Five basic types of questions typically arise when combinatorial problems are studied, some more difficult than others.

i) Existence: Are there any arrangements X that conform to the pattern?
ii) Construction: If so, can such an X be found quickly?
iii) Enumeration: How many different arrangements X exist?
iv) Generation: Can all arrangements X_1, X_2, ... be visited systematically?
v) Optimization: What arrangements maximize or minimize $f(X)$, given an objective function f?

Each of these questions turns out to be interesting with respect to Langford pairs.

For example, consider the question of existence. Trial and error quickly reveals that, when $n = 5$, we cannot place $\{1, 1, 2, 2, \ldots, 5, 5\}$ properly into ten positions. The two 1s must both go into even-numbered slots, or both into odd-numbered slots; similarly, the 3s and 5s must choose between two evens or two odds; but the 2s and 4s use one of each. Thus we can't fill exactly five slots of each parity. This reasoning also proves that the problem has no solution when $n = 6$, or in general whenever the number of odd values in $\{1, 2, \ldots, n\}$ is odd.

In other words, Langford pairings can exist only when $n = 4m-1$ or $n = 4m$, for some integer m. Conversely, when n does have this form, Roy O. Davies has found an elegant way to construct a suitable placement (see exercise 1).

How many essentially different pairings, L_n, exist? Lots, when n grows:

$$\begin{aligned} L_3 &= 1; & L_4 &= 1; \\ L_7 &= 26; & L_8 &= 150; \\ L_{11} &= 17{,}792; & L_{12} &= 108{,}144; \\ L_{15} &= 39{,}809{,}640; & L_{16} &= 326{,}721{,}800; \\ L_{19} &= 256{,}814{,}891{,}280; & L_{20} &= 2{,}636{,}337{,}861{,}200; \\ L_{23} &= 3{,}799{,}455{,}942{,}515{,}488; & L_{24} &= 46{,}845{,}158{,}056{,}515{,}936. \end{aligned} \tag{1}$$

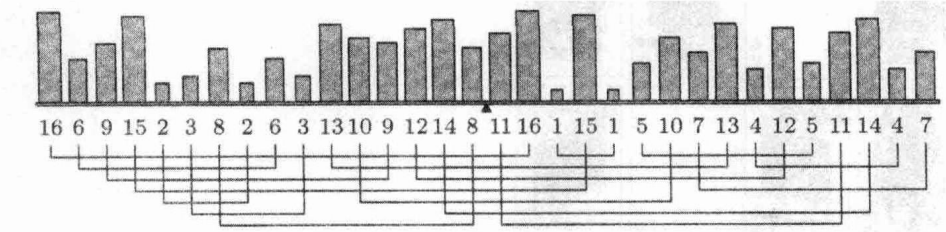
[The values of L_{23} and L_{24} were determined by M. Krajecki, C. Jaillet, and A. Bui in 2004 and 2005; see *Studia Informatica Universalis* 4 (2005), 151–190.] A seat-of-the-pants calculation suggests that L_n might be roughly of order $(4n/e^3)^{n+1/2}$ when it is nonzero (see exercise 5); and in fact this prediction turns out to be basically correct in all known cases. But no simple formula is apparent.

The problem of Langford arrangements is a simple special case of a general class of combinatorial challenges called *exact cover problems*. In Section 7.2.2.1 we shall study an algorithm called "dancing links," which is a convenient way to generate all solutions to such problems. When $n = 16$, for example, that method needs to perform only about 3200 memory accesses for each Langford pair arrangement that it finds. Thus the value of L_{16} can be computed in a reasonable amount of time by simply generating all of the pairings and counting them.

Notice, however, that L_{24} is a *huge* number — roughly 5×10^{16}, or about 1500 MIP-years. (Recall that a "MIP-year" is the number of instructions executed per year by a machine that carries out a million instructions per second, namely 31,556,952,000,000.) Therefore it's clear that the exact value of L_{24} was determined by some technique that did *not* involve generating all of the arrangements. Indeed, there is a much, much faster way to compute L_n, using polynomial algebra. The instructive method described in exercise 6 needs $O(4^n n)$ operations, which may seem inefficient; but it beats the generate-and-count method by a whopping factor of order $\Theta((n/e^3)^{n-1/2})$, and even when $n = 16$ it runs about 20 times faster. On the other hand, the exact value of L_{100} will probably never be known, even as computers become faster and faster.

We can also consider Langford pairings that are *optimum* in various ways. For example, it's possible to arrange sixteen pairs of weights $\{1, 1, 2, 2, \ldots, 16, 16\}$ that satisfy Langford's condition and have the additional property of being "well-

balanced," in the sense that they won't tip a balance beam when they are placed in the appropriate order:



(2)

In other words, $15.5\cdot16+14.5\cdot6+\cdots+0.5\cdot8 = 0.5\cdot11+\cdots+14.5\cdot4+15.5\cdot7$; and in this particular example we also have another kind of balance, $16+6+\cdots+8 = 11+16+\cdots+7$, hence also $16\cdot16+15\cdot6+\cdots+1\cdot8 = 1\cdot11+\cdots+15\cdot4+16\cdot7$.

Moreover, the arrangement in (2) has *minimum width* among all Langford pairings of order 16: The connecting lines at the bottom of the diagram show that no more than seven pairs are incomplete at any point, as we read from left to right; and one can show that a width of six is impossible. (See exercise 7.)

What arrangements $a_1a_2\ldots a_{32}$ of $\{1,1,\ldots,16,16\}$ are the *least* balanced, in the sense that $\sum_{k=1}^{32} ka_k$ is maximized? The maximum possible value turns out to be 5268. One such pairing — there are 12,016 of them — is

$$2\,3\,4\,2\,1\,3\,1\,4\,16\,13\,15\,5\,14\,7\,9\,6\,11\,5\,12\,10\,8\,7\,6\,13\,9\,16\,15\,14\,11\,8\,10\,12. \quad (3)$$

A more interesting question is to ask for the Langford pairings that are smallest and largest in lexicographic order. The answers for $n = 24$ are

$$\begin{tt}\{abacbdecfgdoersfpgqtuwxvjklonhmirpsjqkhltiunmwvx,\\ xvwsquntkigrdapaodgiknqsvxwutmrpohljcfbecbhmfejl\}\end{tt} \quad (4)$$

if we use the letters `a`, `b`, ..., `w`, `x` instead of the numbers 1, 2, ..., 23, 24.

We shall discuss many techniques for combinatorial optimization in later sections of this chapter. Our goal, of course, will be to solve such problems without examining more than a tiny portion of the space of all possible arrangements.

Orthogonal latin squares. Let's look back for a moment at the early days of combinatorics. A posthumous edition of Jacques Ozanam's *Recreations mathematiques et physiques* (Paris: 1725) included an amusing puzzle in volume 4, page 434: "Take all the aces, kings, queens, and jacks from an ordinary deck of playing cards and arrange them in a square so that each row and each column contains all four values and all four suits." Can you do it? Ozanam's solution, shown in Fig. 1 on the next page, does even more: It exhibits the full panoply of values and of suits also on both main diagonals. (Please don't turn the page until you've given this problem a try.)

By 1779 a similar puzzle was making the rounds of St. Petersburg, and it came to the attention of the great mathematician Leonhard Euler. "Thirty-six officers of six different ranks, taken from six different regiments, want to march in a 6×6 formation so that each row and each column will contain one officer of each rank and one of each regiment. How can they do it?" Nobody was able to

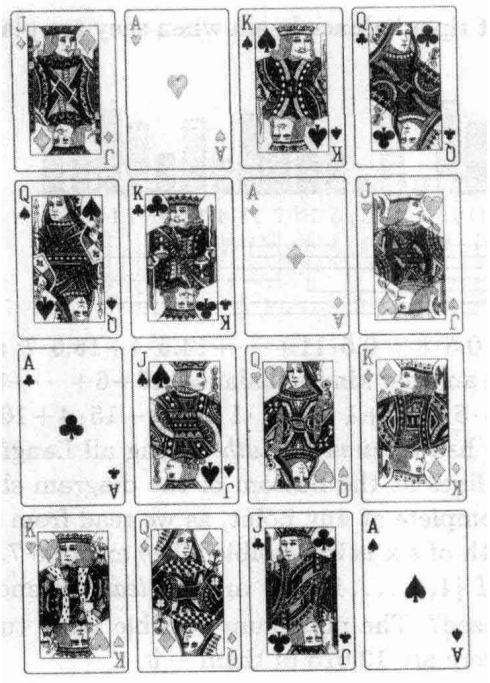


Fig. 1. Disorder in the court cards: No agreement in any line of four. (This configuration is one of many ways to solve a popular eighteenth-century problem.)

find a satisfactory marching order. So Euler decided to resolve the riddle — even though he had become nearly blind in 1771 and was dictating all of his work to assistants. He wrote a major paper on the subject [eventually published in *Verhandelingen uitgegeven door het Zeeuwsch Genootschap der Wetenschappen te Vlissingen* **9** (1782), 85–239], in which he constructed suitable arrangements for the analogous task with n ranks and n regiments when $n = 1, 3, 4, 5, 7, 8, 9, 11, 12, 13, 15, 16, \ldots$; only the cases with $n \bmod 4 = 2$ eluded him.

There's obviously no solution when $n = 2$. But Euler was stumped when $n = 6$, after having examined a "very considerable number" of square arrangements that didn't work. He showed that any actual solution would lead to many others that look different, and he couldn't believe that all such solutions had escaped his attention. Therefore he said, "I do not hesitate to conclude that one cannot produce a complete square of 36 cells, and that the same impossibility extends to the cases $n = 10$, $n = 14$ ... in general to all oddly even numbers."

Euler named the 36 officers $a\alpha$, $a\beta$, $a\gamma$, $a\delta$, $a\epsilon$, $a\zeta$, $b\alpha$, $b\beta$, $b\gamma$, $b\delta$, $b\epsilon$, $b\zeta$, $c\alpha$, $c\beta$, $c\gamma$, $c\delta$, $c\epsilon$, $c\zeta$, $d\alpha$, $d\beta$, $d\gamma$, $d\delta$, $d\epsilon$, $d\zeta$, $e\alpha$, $e\beta$, $e\gamma$, $e\delta$, $e\epsilon$, $e\zeta$, $f\alpha$, $f\beta$, $f\gamma$, $f\delta$, $f\epsilon$, $f\zeta$, based on their regiments and ranks. He observed that any solution would amount to having two *separate* squares, one for Latin letters and another for Greek. Each of those squares is supposed to have distinct entries in rows and columns; so he began by studying the possible configurations for $\{a, b, c, d, e, f\}$, which he called *Latin squares*. A Latin square can be paired up with a Greek square to form a "Græco-Latin square" only if the squares are *orthogonal* to each other, meaning that no (Latin, Greek) pair of letters can be found together in more than one place when the squares are superimposed. For example, if we let $a = \texttt{A}$, $b = \texttt{K}$, $c = \texttt{Q}$, $d = \texttt{J}$, $\alpha = \clubsuit$, $\beta = \spadesuit$, $\gamma = \diamondsuit$, and $\delta = \heartsuit$, Fig. 1 is equivalent

to the Latin, Greek, and Græco-Latin squares

$$\begin{pmatrix} d & a & b & c \\ c & b & a & d \\ a & d & c & b \\ b & c & d & a \end{pmatrix}, \begin{pmatrix} \gamma & \delta & \beta & \alpha \\ \beta & \alpha & \gamma & \delta \\ \alpha & \beta & \delta & \gamma \\ \delta & \gamma & \alpha & \beta \end{pmatrix}, \text{ and } \begin{pmatrix} d\gamma & a\delta & b\beta & c\alpha \\ c\beta & b\alpha & a\gamma & d\delta \\ a\alpha & d\beta & c\delta & b\gamma \\ b\delta & c\gamma & d\alpha & a\beta \end{pmatrix}. \quad (5)$$

Of course we can use *any* n distinct symbols in an $n \times n$ Latin square; all that matters is that no symbol occurs twice in any row or twice in any column. So we might as well use numeric values $\{0, 1, \ldots, n-1\}$ for the entries. Furthermore we'll just refer to "latin squares" (with a lowercase "l"), instead of categorizing a square as either Latin or Greek, because orthogonality is a symmetric relation.

Euler's assertion that two 6×6 latin squares cannot be orthogonal was verified by Thomas Clausen, who reduced the problem to an examination of 17 fundamentally different cases, according to a letter from H. C. Schumacher to C. F. Gauss dated 10 August 1842. But Clausen did not publish his analysis. The first demonstration to appear in print was by G. Tarry [*Comptes rendus, Association française pour l'avancement des sciences* **29**, part 2 (1901), 170–203], who discovered in his own way that 6×6 latin squares can be classified into 17 different families. (In Section 7.2.3 we shall study how to decompose a problem into combinatorially inequivalent classes of arrangements.)

Euler's conjecture about the remaining cases $n = 10$, $n = 14$, ... was "proved" three times, by J. Petersen [*Annuaire des mathématiciens* (Paris: 1902), 413–427], by P. Wernicke [*Jahresbericht der Deutschen Math.-Vereinigung* **19** (1910), 264–267], and by H. F. MacNeish [*Annals of Math.* **23** (1922), 221–227]. Flaws in all three arguments became known, however; and the question was still unsettled when computers became available many years later. One of the very first combinatorial problems to be tackled by machine was therefore the enigma of 10×10 Græco-Latin squares: Do they exist or not?

In 1957, L. J. Paige and C. B. Tompkins programmed the SWAC computer to search for a counterexample to Euler's prediction. They selected one particular 10×10 latin square "almost at random," and their program tried to find another square that would be orthogonal to it. But the results were discouraging, and they decided to shut the machine off after five hours. Already the program had generated enough data for them to predict that at least 4.8×10^{11} hours of computer time would be needed to finish the run!

Shortly afterwards, three mathematicians made a breakthrough that put latin squares onto page one of major world newspapers: R. C. Bose, S. S. Shrikhande, and E. T. Parker found a remarkable series of constructions that yield orthogonal $n \times n$ squares for all $n > 6$ [*Proc. Nat. Acad. Sci.* **45** (1959), 734–737, 859–862; *Canadian J. Math.* **12** (1960), 189–203]. Thus, after resisting attacks for 180 years, Euler's conjecture turned out to be almost entirely wrong.

Their discovery was made without computer help. But Parker worked for UNIVAC, and he soon brought programming skills into the picture by solving the problem of Paige and Tompkins in less than an hour, on a UNIVAC 1206 Military Computer. [See *Proc. Symp. Applied Math.* **10** (1960), 71–83; **15** (1963), 73–81.]