October 1, 2003

Combinatorial Generation

Working Version (1j-CSC 425/520)

no comments printed; pseudo-code version

Frank Ruskey

Department of Computer Science
University of Victoria
Victoria, B.C. V8W 3P6
CANADA
fruskey@csr.csc.uvic.ca

The covers of your book are too far apart. *George Bernard Shaw.*

No man but a blockhead ever wrote except for money. *Dr. Samuel Johnson.*

Everything has been said before by someone. *Alfred North Whitehead*

It is a wise father that knows his own child. *William Shakespeare*, The Merchant of Venice, Act II, Scene II, 73. — good quote for chapter 8.

There are various versions of this book. If there is a comment printed as a footnote to this sentence then comments to myself are printed. These contain bibliographic remarks, sources of info, rewriting suggestions, etc.

Preface

Humanity has long enjoyed making lists. All children delight in their new-found ability to count 1,2,3, etc., and it is a profound revelation that this process can be carried out indefinitely. The fascination of finding the next unknown prime or of listing the digits of π appeals to the general population, not just mathematicians. The desire to produce lists almost seems to be an innate part of our nature. Furthermore, the solution to many problems begins by listing the possibilities that can arise.

There are lists of mathematical objects that have a historical interest, or maybe I should say that there are lists of historical objects that have a mathematical interest. Examples include the Fu Hsi sequencing of the hexagrams of the I Ching (e.g., as described in Martin Gardner - Knotted Donuts [148]) and the Murasaki diagrams of the Tale of the Genji (by Lady Shikibu Murasaki, 1000 A.D.) as explained in Gould [158]. Viewed properly, these are lists of the first 64 binary numbers $0, 1, \ldots, 63$ and all 52 partitions of a 5-set, respectively. They are illustrated in Figures 1 and 2 at the end of this preface.

However, it was not until the advent of the digital computer that the construction of long lists of combinatorial objects became a non-tedious exercise. Indeed, some of the first applications of computers were for producing such lists. With the computer as our able assistant we can construct interesting lists to our hearts content. There is deeply satisfying feeling that one obtains by watching a well ordered list of combinatorial objects marching down the computer screen. The author hopes that the readers of this book will implement some of the algorithms found herein and experience this feeling for themselves!

These lists are not only interesting but can be extremely useful. Nearly every computer scientist and mathematician has encountered some instance where they solved a problem simply by listing all the possibilities and checking for a desired property. One can find applications of the ideas in this book to circuit design, statistical computing, chemistry, and operations research, just to name a few.

Ever since the 1960's there has been a steady flow of new algorithms for constructing lists of combinatorial objects of various types. There is no widely accepted name for the field but I like to call it *Combinatorial Generation*. Many of the papers in the field have a title that contains the word "generating." Unfortunately, "generate" is an overworked word in mathematics and computer science, so the name is not fully satisfying. A widely used alternative is "enumerate", which, unfortunately, also has another widely accepted meaning, "to count", as used in Goulden and Jackson's book *Combinatorial Enumeration* [157]. Some people use "generate" to mean generate uniformly at random. In algebra, generator is a term use for a subset of an algebraic structure that, via some operation, can produce the entire structure, as in the generators of a group. A good alternative title for the book would be "Exhaustive Listing of Combinatorial Objects." Nevertheless, "Combinatorial Generation" is the name chosen and used in this book.

Several books contain chapters about generating combinatorial objects, but there is no book devoted solely to the subject (with pehaps the exception of Wilf's update [451] of [289]). The books of Nijenhuis and Wilf [289], and of Reingold, Nievergelt and Deo [340] contain such chapters and have provided an inspiration for me over the years. Many of the ideas of the present book can be traced back to these books. I would also like to mention the influential books of Even [115], Page and Wilson [296], Rohl [347], Wells [440], and Williamson [453], all of which contain interesting material on generating combinatorial objects. The forthcoming Volume 4 of the Art of Computer Programming by Knuth will contain material on combinatorial generation.

The concern of this book is not with existence questions By an existence question we mean one of the form "find a single instance of a combinatorial object with certain properties". These questions have quite different methods of attack and warrant a separate treatment, although backtracking, which is covered is a widely used technique for solving existence questions. Existence questions arise, for example, in the search for various types of block designs. The recent book of Kreher and Stinson [233] covers such techniques, as well as overlapping some of the exhaustive listing material discussed in this book.

Of course, the particular topics that are treated here reflect my own personal choices about the elegance, importance and accessibility of particular topics — not to mention that they are those with which I am most familiar and find most interesting.

There is something of interest for many types of readers to be found in the chapters to follow. The reader who simply wants a fast algorithm to generate a well-known class of objects will find it here. Many algorithms are fully implemented as Pascal procedures, others with a Pascal-like pseudo-code. The analysis of generation algorithms provides interesting problems for those interested in combinatorial enumeration. The analysis of most of our algorithms involves some sort of counting problem, and many techniques from enumerative combinatorics have been brought to bear in solving these counting problems. The software engineer will find interesting nontrivial, but short, procedures. These provide excellent examples for students on which to try their skill at providing formal proofs of correctness of programs.

Recently, there seems to be an explosion of results on combinatorial generation. This renewed interest is fueled by many sources: the increasing speed of computers, mathematical interest in Hamiltonicity of graphs, the emergence of parallel computers, the complexity theory of counting problems, and so on. It is my belief that a book on combinatorial generation is timely.

Many of the programs presented in this book can be downloaded from the author's "Combinatorial Object Server" (COS) at `http://www.theory.csc.uvic.ca/~cos`. The main purpose of this site is to provide an on-line resource for the generation of small numbers of combinatorial objects. The user first selects a type of object, then supplies input parameters, and output options, and COS returns the appropriate list of objects.

There are very few universities that offer courses on combinatorial generation so I don't expect this to be often used as a textbook (of course, the appearance of a suitable text may change that). The book is mainly intended as a reference and perhaps for supplementary reading in a course. Each chapter contains some exercises, and some contain extensive sets of exercises. It is quite feasible to use the book as a textbook, but it would be essential for prospective students to have had a course in discrete mathematics or combinatorics and enough programming experience to be totally comfortable with recursion. Other than that the background necessary to read the book is minimal. Most of the material should be

accessible to the typical undergraduate Computer Science or Mathematics major.

The first chapter discusses some general issues in combinatorial generation. The second chapter reviews some basic definitions and counting results that are useful in the rest of the book; the reader is advised to skip over this chapter and just refer to it as needed. Chapter 3 is a short introduction to backtracking. The following chapters are broadly organized by type of algorithm. Chapter 4 is Lexicographic Algorithms, Chapter 5 concerns algorithmic aspects of Gray codes and Chapter 6 is a continuation, concentrating on graph theoretic aspects of Gray codes. Chapter 7 considers problems that can be modeled as Eulerian cycles, such as De Bruijn cycles. Chapters 4, 5, 6, and to a lesser extent, Chapter 7 are concerned with elementary combinatorial objects. Orderly Algorithms are covered in Chapter 8 and Chapter 9 looks at problems of generating various types of subgraphs of graphs, such as spanning trees and cliques. Parallel algorithms for generating combinatorial objects are discussed in Chapter **??**. The uniform random selection of objects is the subject of Chapter 10. Some complexity issues are discussed in Chapter **??**. And finally, Chapter **??** contains solutions and hints to the exercises.

I have tried to provide a fairly comprehensive list of references to papers whose main objective is the development of an algorithm for generating some class of combinatorial objects. Inclusion in the references implies no judgement on my part about the merits of the paper. It is evident from the large number of these references that there has been much interest in generating combinatorial objects and that that interest is increasing. It is my hope that this book will provide a guide to those interested in combinatorial generation, both in regards to what has been done before, and in terms of what approaches have proved most successful.

The exercises form an integral part of the book. Many of them expand upon the material of the chapters and contain material about generating other types of combinatorial objects beyond those covered in the main body of the chapter. Many useful generation problems have been relegated to the exercises because of space limitations. The reader is urged to at least read the exercises and understand what they are asking. Solutions or hints or references to the relevant literature are provided for most of the exercises; these solutions may be found in the final chapter at the end of the book. They are classified according to difficulty using a modification of the rating scheme of Stanley [402] as follows.

[**1**] routine, straightforward.

[**2**] somewhat difficult or tricky.

[**3**] difficult.

[**4**] extraordinarily difficult.

[***R***] research problem.

Further gradations are indicated by + and −. An [***R***] indicates that the problem is a research problem, and that the solution is unknown to myself. An [***R***−] or [***R***] rating does not necessarily imply that the problem is extremely difficult; indeed, I suspect that some of them can be solved quite easily. If you do solve any one of these problems, please let me know! An [***R***−] indicates that I don't know the solution or of anybody who has worked on the problem before. Such problems may be quite easy to solve. An [***R***] indicates that at least one person has unsuccessively worked on the problem, and a [***R***+] means that several

well-known researchers have worked on the problem without success. Some of the [***R***+] problems are notoriously difficult!

I would like to thank my favorite list, Susan, Jennifer, and Albert, for putting up with the many lost hours spent away from them as I wrote this book.

Many people have contributed to this book, knowingly or not. In particular I would like to mention Larry Cummings, Brendan McKay, Carla Savage, Ian Roberts, Colin Ramsay, Peter Eades, Malcolm Smith, Gara Pruesse, Joe Sawada, and Donald Knuth.

Frank Ruskey

T'ai Chi

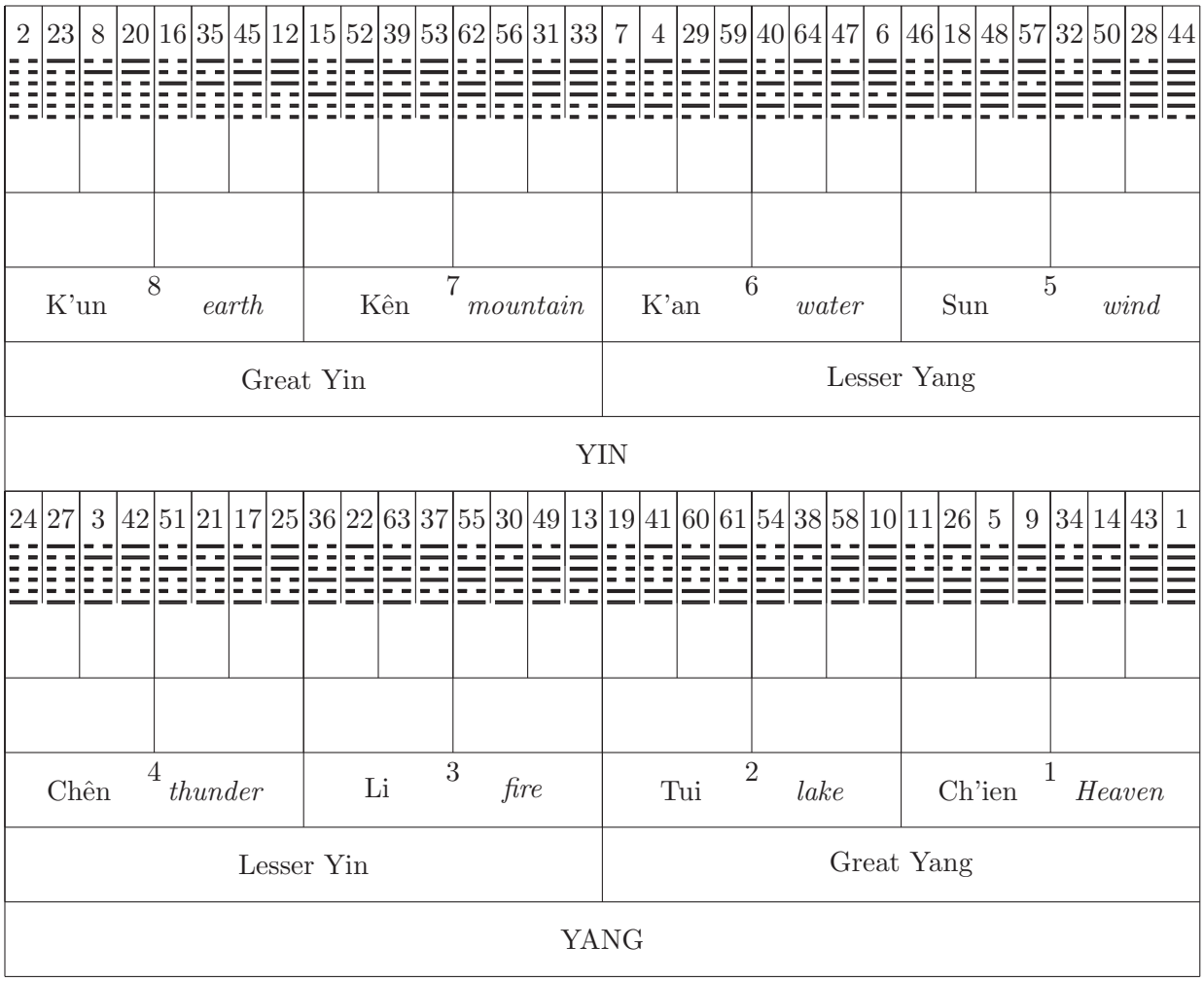


Figure 1: The 64 Hexagrams of the Fu Hsi ordering (from the *I Ching*). The *I Ching* (or *Book of Changes*) is a book that has been used for centuries as an aid to fortune telling in China. The *I Ching* is said to have originated around the eighth century B.C. and the Fu Hsi ordering is from the 11th century. Each of the 2^6 hexagrams consists of six symbols, either a long dash or two short dashes, placed one above the other. Originally the hexagrams were arranged in other, seemingly random, orders; one such order is indicated by the numbers above each hexagram.

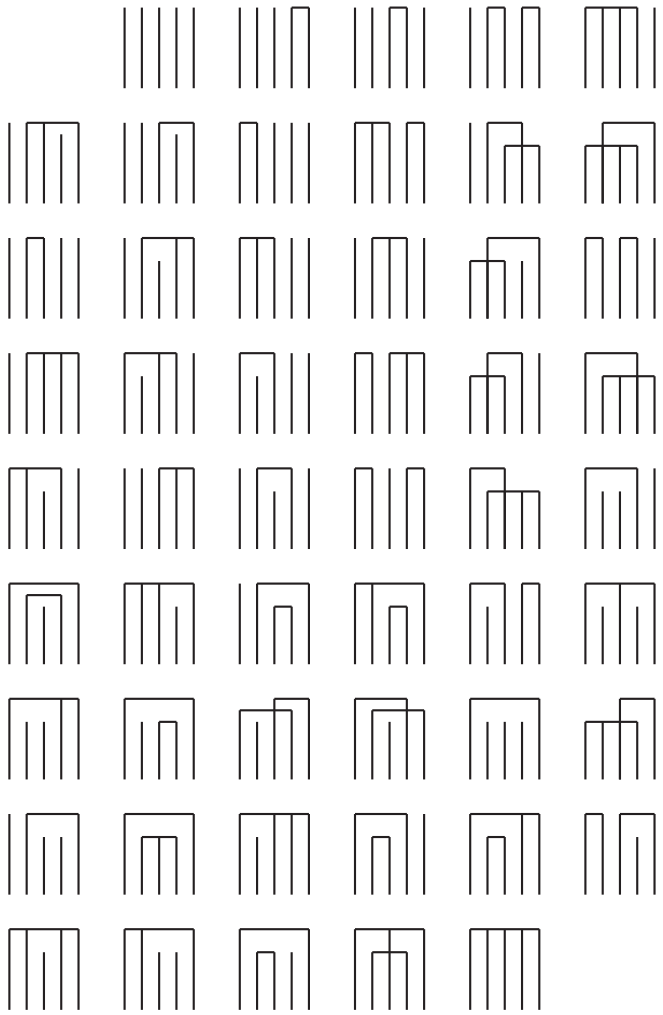


Figure 2: The 52 Murasaki diagrams from *The Tale of the Genji*. The novel *Tale of Genji* was written about 1000 A.D. by Lady Shikibu Murasaki, who is often referred to as the Shakespeare of Japan. The book has 54 chapters and some early copies have one of the Murasaki diagrams at the top of each chapter, except for the first and last. The diagrams themselves derive from a Japanese game involving the identification, by smell alone, of a sequence of 5 incense sticks. Each diagram is formed from 5 vertical bars, where bars representing the same type of incense are connected by a horizontal bar.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

What is Combinatorial Generation?

"Let's look at all the possibilities." This phrase sums up the outlook of this book. In computer science, mathematics, and in other fields it is often necessary to examine all of a finite number of possibilities in order to solve a problem or gain insight into the solution of a problem. These possibilities often have an underlying combinatorial structure which can be exploited in order to obtain an efficient algorithm for generating some appropriate representation of them.

1.1 Some Examples

1.1.1 Fisher's Exact Test

Sir R.A. Fisher described an experiment to test a woman who claimed that she could distinguish whether the milk or tea was poured first in a cup of tea and milk. Eight cups of tea were prepared, 4 in which milk came before tea and 4 with tea before milk. The woman knows that there will be 4 of each type. The results were as shown below in what's called a 2 by 2 contingency table.

	Guess Poured First		
Poured First	*Milk*	*Tea*	
Milk	3	1	4
Tea	1	3	4

The probability that a particular contingency table T occurs follows a multinomial distribution, where x_1 and x_2 are the entries in the first column, n_1 and n_2 are the row totals, and $N = n_1 + n_2$.

$$Prob(T) = \frac{\binom{n_1}{x_1}\binom{n_2}{x_2}}{\binom{N}{x_1+x_2}}$$

The typical questions that a statistician wishes to answer are: (1) How many tables have a lower probability of occurring than the observed table? (2) What is the sum of probabilities of tables having a value of x_1 at least as large as what's observed?

In our example, the value of $\binom{N}{x_1+x_2}$ is 70, and the possible probabilities are given in the table below.

x_1	x_2	numerator	probability
0	4	$\binom{4}{0}\binom{4}{4} = 1$	$1/70 = 0.0142857$
1	3	$\binom{4}{1}\binom{4}{3} = 16$	$16/70 = 0.2285714$
2	2	$\binom{4}{2}\binom{4}{2} = 36$	$36/70 = 0.5142857$
3	1	$\binom{4}{3}\binom{4}{1} = 16$	$16/70 = 0.2285714$
4	0	$\binom{4}{4}\binom{4}{0} = 1$	$1/70 = 0.0142857$

Answering each of these questions involves the generation of combinatorial objects. In this case the generation is particularly simple. We wish to compute the value of $Prob(T)$ for all those values of x_1 and x_2 for which $x_1 + x_2 = 4$, with the additional constraints that $x_1 \leq n_1 = 4$ and $x_2 \leq n_2 = 4$. The combinatorial objects being generated are the pairs (x_1, x_2).

In the more general setting of a k by 2 contingency table, we need to generate all solutions to

$$x_1 + x_2 + \cdots + x_k = r_1 \text{ subject to } 0 \leq x_i \leq n_i.$$

Algorithms for generating these objects, which we call *combinations of a multiset*, are presented in Section 4.5.1.

1.1.2 A Second Example

This subsection is yet to be written.

1.2 Elementary Objects

There is no precise definition of an elementary combinatorial object. Our intuitive notion is that if a class of combinatorial objects satisfies a simple recurrence relation then it is elementary. We make no attempt to define "simple recurrence relation." However, we consider permutations, combinations, set partitions, numerical partitions, binary trees and labeled graphs to all be elementary combinatorial objects; while unlabeled graphs, room squares, and unlabeled partially ordered sets are not elementary.

Many recurrence relations many be stated in a form that involves no division or subtraction, only multiplication and addition, and further that involve only positive values, even the base cases. Such recurrence relations are said to be *positive*. Given a simple recurrence relation describing an elementary combinatorial object, it is typically straightforward to develop an algorithm for generating a natural representation of that object. If the recurrence relation is positive, then the algorithm is often efficient in an amortized sense. This point of view was perhaps first explored by Wilf in two papers [448], [449], and we have more to say on this subject in Section 4.9.

Most of the book is devoted to elementary objects, namely Chapters 4, 5, and 6. Non-elementary objects are generated in Chapters 3 and 8.

1.3 Four Basic Questions

We consider in this book 4 basic questions: listing, ranking, unranking, and random selection, of which the listing question is of paramount importance.

[listing] Algorithms for generating combinatorial objects come in two varieties. Either there is a recursive procedure, call it *GenerateAll*, that typically has the same recursive

structure as a recurrence relation counting the objects being generated, or there is a procedure, call it *Next*, that takes the current object, plus possibly some auxiliary information, and produces the next object. Typically, but not always, *Next* is iterative. Throughout the book we assume that *Next* is used by a section of code as shown below.

```
Initialize; {includes done := false }
repeat
      PrintIt;
      Next;
until done;
```

The boolean variable *done* is global and is eventually set true by *Next*. If *Next* contains no loop then the generation algorithm is said to be *loopless*. Another term one sometimes sees in connection with iterative generation algorithms is *memoryless*. This simply means that *Next* contains no global variables whose value changes; the algorithm can be started with any object.

Many older papers start with a nice recursive decomposition of the class of objects to be generated and then jump immediately into the development of a iterative *Next* routine. This unfortunate habit has turned many a beautiful decomposition into an ugly program. One would sometimes see this uglification justified by a sentence stating that recursion was slow because of the overhead involved in making procedure calls and passing parameters.[1] This justification can no longer be accepted! It's a throwback to a by-gone era. Most modern machines include hardware to speed procedure calls. The Sun workstation sitting on the author's desk has banks of "window" registers that make recursive programs written in C faster than their iterative counterparts! For these reasons, the vast majority of generation algorithms presented in this book are recursive.

However, there are at least two compelling reasons why iterative *Next* routines are useful — these reasons are due to more modern trends in computing. First is the cause of modularization. Iteration and recursion generation procedures present two opposite views to the user. Iteration says "Give me and object and I'll give you the next one". Recursion says "Give me the procedure that uses the object and I'll apply it to every object". The second reason is that iterative procedures are often more amenable to parallel computation.

Sometimes it is desirable to have a listing of objects in which successive objects are "close" in some well-defined sense. Such listings are called Gray Codes, and are the subjects of Chapter 5 and Chapter 6.

[**ranking**] Relative to an ordering of a set of combinatorial objects, such as the ordering imposed by a generation algorithm, the *rank* of an object is the position that the object occupies in the ordering. Our counting begins at 0 so another way of defining the rank is as the number of objects that precede it in the list[2]. One of the primary uses of ranking algorithms is that they provide "perfect hashing functions" for a class of combinatorial objects. The existence of a ranking algorithm allows you to set up an array indexed by the objects. Ranking (and unranking) is generally only possible for some of the elementary combinatorial objects.

[**unranking**] Unranking is the inverse process of ranking. To *unrank* an integer r is to produce the object that has rank r. Unranking algorithms can be used to implement

[1] Or perhaps that the language being used didn't support recursion.

[2] In the literature many ranking algorithms begin counting at 1.

parallel algorithms for generating combinatorial objects, when used in conjunction with a *Next* procedure. For more on this see Chapter **??** For this reason we try ensure that our unranking algorithms return enough information for *Next* to be restarted from any object.

[**random selection**] Here we want to produce a combinatorial object uniformly at random. An unranking algorithm can be used, but often more direct and efficient methods can be developed. This is an important topic, and is treated in Chapter 10, but is not a main focus of this book.

1.4 A Word about the Algorithms

It is hoped that the contents of this book are not only interesting but also useful. The algorithms in this book have been presented in a pseudo-code. In most cases it should be a trivial exercise to translate these algorithms into languages such as C, C++, Java, or Pascal. The most important difference in our pseudo-code from those languages is that the scope of statements *is indicated by indentation*, rather than by the use of parentheses, as in C, C++, and Java, or by the use of **begin**, **end**pairs, as in Pascal. In spirit our algorithms are closest to Pascal. Procedures that do not return a value are indicated by the reserved word **procedure** (these are like void procedures and methods in C and Java. Those that return a value are indicated by the reserved word **function** . The return type of the function is indicated after the parameter list, as in
function ($\langle$ parameter list $\rangle$) : $\langle$ return type $\rangle$;
Following Knuth, we use $x :=: y$ to indicate the swap of the values of two variables; i.e., it is shorthand for $t := x;\ x := y;\ y := t$. Also, we allow for multiple (parallel) assignment statements as $[x, y] := [e, f]$ to indicate the assignments $x := e$ and $y := f$, executed in parallel. The notations a_i and $a[i]$ will be used interchangeably and $a[i..j]$ indicates the subarray of a indexed from i to j. Arrays can be declared over an arbitrary range; i.e., not necessarily starting from 0, as in C and Java.

Linked structures are not often used in this book, but when they are we adopt simple "dot" notation to denote fields within nodes.

1.5 The Representation Issue

There are typically many ways to represent a combinatorial object, and these different representations may lead to wildly differing algorithms for generating the objects. For example, permutations may be represented in one-line notation, in cycle notation, by inversion tables, or even by permutation matrices; binary trees may be represented as a linked data structure, as well-formed parentheses strings, as triangulations of convex polygons, or as ordered trees, as well as many other sequence representations. Which representation is most useful usually depends upon the application that requires the objects to be generated. This is a matter over which the author of the book has no control.

On the other hand, each object usually has some small number of standard representations, and it is these representations that we try to generate, and that other developers of generation algorithms should try to generate. These standard representations we use are almost always sequences, usually of fixed length, but not always.

The user of an algorithm for generating combinatorial objects should specify the representation most useful to them. Assume that we wish to generate some combinatorial family

S_n indexed by numbers n, and where $s_n = |S_n|$ is known or easily computable. Left up to the the lazy generator, you might get an algorithm like the following.

"Compute s_n";
for $i := 1$ **to** s_n **do** Output(i);

Typically the computation of s_n is efficient as a function of s_n, so that the above algorithm is very efficient in an amortized sense. But have we generated S_n? No reasonable person would think so, since all we are doing is counting, and to get a useful representation, some unranking algorithm still has to be developed.

The point of this discussion is that representations are important and the generator (person developing a generation algorithm) should be careful to use a representation that is useful to others, and not just convenient because it makes the algorithm simple or efficient.

1.6 Complexity Measures

Not much attention has been paid to the model of computation in the area of combinatorial generation. For the most part authors use the random access machine model. This is more or less like counting operations in a Pascal program, with the understanding that integers can be arbitrarily large and the assumption that standard arithmetic operations can be done in constant time. For large integers this is an unrealistic assumption. It is generally the case that integers are small in generation algorithms, but that they are large in ranking and unranking algorithms. We will therefore measure ranking and unranking algorithms in terms of the number of arithmetic operations used.

The complexity of generating combinatorial objects is not well addressed by the classical theory of computational complexity, with its emphasis on the polynomial versus non-polynomial time question, and polynomial and log-space reductions. Most classes of combinatorial objects with which we are concerned have at least exponentially many elements, and useful reductions of any kind are rare. Some complexity questions are addressed in Chapter **??**.

An enormous amount of research has gone into getting away from the "brute-force" approach to solving discrete optimization problems, and very fruitful approaches to solving a wide variety of these problems have been developed. Nevertheless, one is occasionally forced into the approach of examining all the possibilities and the results of this book should be useful in those instances. Because of the aversion to the brute-force approach, research in combinatorial generation has never been popular; refining a generation algorithm to more efficiently solve some discrete optimization problem is like an admission of defeat in this view. Combinatorial generation is really a part of theoretical computer science, certainly more so than the complexity of counting, which is now a well-established part of theoretical computer science. Perhaps surprisingly, in the 500+ references in the fairly comprehensive bibliography, there is an almost a total absence of references from the preeminent STOC and FOCS conferences. Perhaps this will change; I hope so.

1.7 Analyzing the Algorithms

There are two terms that are used throughout the book. We strive to develop algorithms that have these properties.

CAT Algorithms The holy grail of generating combinatorial objects is to find an algorithm that runs in Constant Amortized Time. This means that the amount of computation, after a small amount of preprocessing, is proportional to the number of objects that are listed. We do not count the time to actually output or process the objects; we are only concerned with the amount of data structure change that occurs as the objects are being generated.

BEST Algorithms This means Backtracking Ensuring Success at Terminals. In other words, the algorithm is of the backtracking type, but every leaf of the backtracking tree is an object of the desired type; it is a "success".

Suppose that the input to our generation algorithm is n, and that this will produce N objects, each of "size" n. A CAT algorithm has running time $O(N)$. In the typical application of combinatorial generation, each object, as it is produced, is processed somehow. If this processing takes time $O(n)$, then having a CAT algorithm has no advantage over having one with running time $O(nN)$, since the total running time is $O(nN)$ in either case.

There are are many papers about generating combinatorial objects that describe an $O(nN)$ running time as "optimal", because this is the amount of output produced by the algorithm — each object has size n and there are N objects in total. This is a misguided notion that has its roots in the way lower bounds are discussed in the traditional introduction to complexity theory course. There the "trivial lower bound" says that the amount of output gives a lower bound on the amount of computation — of course it says the same thing in our setting, but the amount of output is the wrong thing to measure.

> **Don't Count the Output Principle:** In combinatorial generation it is the amount of data structure change that should be measured in determining the complexity of an algorithm; the time required to output each object should be ignored.

The "trivial lower bound" for combinatorial generation is $\Theta(N)$; it is independent of n. This may seem suspicious, since it appears to take no account of the time that is frequently necessary to initialize the various data structures that are used, but there are many algorithms that require only a constant amount of initialization — we will encounter some, for example, in Chapter 4 on lexicographic generation. Now back to our typical application, where processing each object took time $O(n)$ even though our generation algorithm was CAT. Wouldn't it be nice if the $O(nN)$ could be brought down to $O(N)$? This is frequently possible! If you go back and observe how successive objects are processed, it is often the case that *the amount of processing required is proportional to the amount of change that successive objects undergo.*

1.8 Exercises

1. [***R−***] Explore the use of the accounting and/or potential function methods in analyzing algorithms for generating combinatorial objects.

2. [***R−***] In the listing problem we ask that each object be produced exactly once. For some applications, uniqueness is not required, only that each object is produced at least once. Explore the idea of listing objects with repeats allowed, with the goal of producing an algorithm that is simpler or more efficient than known algorithms for listing the objects uniquely.

3. [***R−***] Can every CAT algorithm in which successive objects change by a constant amount be converted into an equivalent (in the sense that the objects are produced in the same order) loopfree algorithm?

4. [***R−***] Investigate the number of steps required by various types of Turing machines to generate well-known lists of combinatorial objects. For example, in Turing-CAT time we can count in binary on a one-tape machine. Along the same lines, investigate the bit complexity of generating some well-known combinatorial lists.

1.9 Bibliographic Remarks

The aggregate, accounting, and potential function methods for amortized algorithm analysis are discussed in Cormen, Leiserson, and Rivest [70].

Perhaps Lehmer [244] was the first to propose that combinatorial objects be studied by imposing an order on them and then developing ranking and unranking algorithms with respect to this order.

The use of ranking functions for source encoding is discussed in Cover [71]. Ranking functions naturally give rise to numeration systems. Some of these are discussed in Fraenkel [137], [138].

The CAT and BEST acronyms are due to the author. What we call CAT has been called *linear* (because $O(N)$ is linear in N) by Reingold, Nievergelt, and Deo [340], and in several papers.

Chapter 2

Basic Combinatorial Objects

PLEASE! Don't read this section first. You'll get the wrong impression about the rest of the book. This section is best ignored in an initial reading of the book and just referred to as need be in later chapters.

The purpose of this chapter is to review the definitions and basic properties of the combinatorial objects to be generated in succeeding chapters. For the most part proofs are omitted or relegated to the exercises. Exceptions are made for a couple of results that the author feels should be more well-known (e.g., Lemma 2.4 and the proof of Theorem 2.2) or peculiarly relevant to the material of this book (e.g., inequality (2.17) and Lemma 2.5).

2.1 Alphabets, Strings, Languages, and Lists

Usually the objects being generated are finite sequences or are represented by finite sequences which are then generated. We use the language of strings to describe these sequences. The output of our programs are also sequences; sequences of strings. To avoid confusion, these sequences are referred to as lists. Thus we *always generate lists of strings*.

An *alphabet* is set, often a finite set. A *string* of length k over an alphabet Σ is a finite sequence, $a_1, a_2, \ldots, a_k$, where each a_i is in Σ. Where no ambiguity arises the commas will be omitted and the string written $a_1 a_2 \cdots a_k$.

The empty string as length 0, and is denoted ε. If Σ is an alphabet, then Σ^n denotes the set of all strings of length n over Σ, the notation Σ^* means the set of all strings over Σ, and $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ denotes the set of all non-empty strings over Σ.

By *number* we mean a natural number, a member of the set $\{0, 1, 2, \ldots\}$. In most cases the alphabet under consideration is either the set of the first n positive integers, which is denoted $[n]$, or is the set consisting of the first k numbers, which is denoted Σ_k. We adopt very different notations for these two sets so no confusion should arise. To be more specific, $[n] = \{1, 2, \ldots, n\}$ and $\Sigma_k = \{0, 1, \ldots, k-1\}$. Strings over Σ_k are called *k-ary strings*. Strings over Σ_2 are called *bitstrings*. The *Hamming distance*, $\mathrm{Ham}(\mathbf{x}, \mathbf{y})$, between two bitstrings $\mathbf{x}$ and $\mathbf{y}$ of the same length is the number of positions in which they differ. We also define the Hamming distance between two finite sets to be $\mathrm{Ham}(A, B) = |A \cup B| - |A \cap B|$.

A *list* is a finite sequence, usually a sequence of strings. For a list L, let *first*(L) denote the first element on the list and let *last*(L) denote the last element on the list L. If L is a list $l_1, l_2, \ldots, l_n$, then $\overline{L}$ or L^{-1} denotes the list obtained by listing the elements of L in reverse order; i.e., $\overline{L} = L^{-1} = l_n, \ldots, l_2, l_1$. We have the obvious relations *first*$(\overline{L})$ = *last*(L)

and $last(\overline{L}) = first(L)$. If L_1 and L_2 are lists then the *interface* between L_1 and L_2 is the ordered pair $(last(L_1), first(L_2))$. If L is a list of *strings* and x is a symbol, then $L{\cdot}x$ denotes the list of strings obtained by *appending* an x to each string of L. By *prepending* an x to each string of L we obtain $x \cdot L$. For example, if $L = 01, 10$ then $L \cdot 1 = 011, 101$ and $1 \cdot L = 101, 110$. If L and L' are lists then $L \circ L'$ denotes the *concatenation* of the two lists. For example, if $L = 01, 10$ and $L' = 11, 00$, then $L \circ L' = 01, 10, 11, 00$. If x is a string then x^n denotes the n-fold concatenation of x with itself, $x^n = xx \cdot x$. In particular if x is a symbol, then x^n is the string consisting of n occurances of x.

!!! These next two paragraphs should be move later, since lex, relex, tree, etc. not defined yet !!!

We say that a list of strings has the *prefix property* if those strings with a common prefix appear contiguously on the list; the *suffix property* is defined analogously. Lex and relex lists have the prefix property and colex lists have the suffix property. Given a closeness relation for which the underlying graph has a Hamilton path, there may not exist a prefix listing corresponding to a Hamilton path. Consider, for example, the strings 000, 100, 101, 111, 011, with strings "close" if they differ by a single bit.

With every finite language L is associated a *prefix tree*, $T(L)$. Each node of the tree is labelled by a symbol and every prefix of a string in L corresponds to a unique path in $T(L)$ starting at the root. The tree $T(L)$ is a rooted labelled tree; it is not an ordered tree. There is a corresponding *suffix tree* , $T^R(L)$.

2.2 Relations and Functions

If A and B are sets then $A \times B$ denotes the set of all ordered pairs (a, b) where $a \in A$ and $b \in B$. A relation on a set A is a subset R of $A \times A$. The relation R is *reflexive* if $(a, a) \in R$ for all $a \in A$. The relation R is *anti-reflexive* if $(a, a) \notin R$ for all $a \in A$. The relation R is *symmetric* if $(a, b) \in R$ implies that $(b, a) \in R$. The relation R is *anti-symmetric* if $(a, b) \in R$ implies that $(b, a) \notin R$. The relation R is *transitive* if $(a, b) \in R$ and $(b, c) \in R$ implies that $(a, c) \in R$. The relation R is an *equivalence relation* if it is reflexive, symmetric, and transitive. The relation R is a *partial order* if it is anti-reflexive, anti-symmetric, and transitive (partial orders are discussed further in Section 2.9).

A *function* from a set A to a set B is a subset F of $A \times B$ with the property that for each $a \in A$ there is exactly one element in $\{(a, b) \in F \mid b \in B\}$. In other words, with each value of $a \in A$ there is associated a unique value of $b \in B$. We write $f : A \to B$ to denote a function from A to B and $f(a) = b$ if $(a, b) \in F$. By $f(A)$ we denote the set $\{b \in B \mid (a, b) \in F\}$. If $f(A) = B$ then f is said to be *surjective*. If $f(a) = f(b)$ implies that $a = b$ then f is said to be *injective*. A function f that is both surjective and injective is said to be a *bijection*. Another name for a surjective function is an *onto* function. Another name for an injective function is a *one-to-one* function.

Clearly, if A and B are finite sets and $f : A \to B$ is a bijection then $|A| = |B|$. Thus if the cardinality of A is known and a bijection between A and B is established, then the cardinality of B is known. We shall use this style of proof several times. If the bijection is made explicit, for example by means of an algorithm, it is called *bijective proof* or *combinatorial proof*. Combinatorial proofs are particularly useful for translating between various representations of combinatorial objects.

2.3 Asymptotic Notation

In analyzing algorithms it is generally accepted that running time matters most for big values of the input parameters and that constant factors are usually of secondary importance. This has lead to notations useful for comparing algorithms; these notations arose out of the mathematical literature for comparing the growth rate of functions.

Let $f(n)$ and $g(n)$ be functions from the natural numbers to the non-negative real numbers. We say that $f(n)$ *is of order (at most)* $g(n)$ and write $f(n) \in O(g(n))$ if there are positive constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. We say that $f(n)$ *is of order at least* $g(n)$ and write $f(n) \in \Omega(g(n))$ if there are positive constants c and n_0 such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$. We say that $f(n)$ *is of order exactly* $g(n)$ and write $f(n) \in \Theta(g(n))$ if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$. We say that $f(n)$ and $g(n)$ are *asymptotically equal* and write $f(n) \sim g(n)$ if the limit of $f(n)/g(n)$ is 1 as n tends to infinity.

2.4 Subsets and Combinations

An n-set is a set with n elements. An n-set has 2^n subsets. The number of k element subsets (k-subsets) of an n-set is denoted $\binom{n}{k}$; this notation is called a *binomial coefficient*.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1} \tag{2.1}$$

If n and k do not satisfy $0 \leq k \leq n$, then the value of $\binom{n}{k}$ is taken to be 0. A table of binomial coefficients may be found in Appendix A. Below is the "Pascal Triangle" recurrence relation, valid for $0 < k < n$. Note that $\binom{n}{0} = \binom{n}{n} = 1$.

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \tag{2.2}$$

Two useful sums are obtained by iterating the first and second terms of (2.2).

$$\sum_{j=0}^{n} \binom{k+j}{k} = \binom{n+1}{k+1} \tag{2.3}$$

$$\sum_{j=0}^{k} \binom{n+j}{j} = \binom{n+k+1}{k} \tag{2.4}$$

A *k-composition* of n is a solution of $x_1+x_2+\cdots+x_k = n$ where $x_i \geq 0$ for $i = 1, 2, \ldots, n$. The number of such compositions is the same as the number of ways of selecting k objects from n objects with repetitions allowed; i.e., it is

$$\binom{n+k-1}{k}.$$

The binomial coefficients are unimodal in k. Asymptotically, the value of the largest coefficient is

$$\binom{2n}{n} \sim \frac{2^{2n}}{\sqrt{\pi n}}.$$

2.5 Permutations

Permutations are a fundamental combinatorial object and they many variants, some of which are introduced below. A *permutation* of a set S is an arrangement of the elements of S in some order. Normally, our permutations will be of the set $S = [n] = \{1, 2, \ldots, n\}$; and we assume this unless stated otherwise. The set of all permutations of $[n]$ is denoted $\mathbb{S}_n$. The mathematical definition of a permutation is that it is a bijection $\pi : S \to S$. We usually write a permutation as a sequence $\pi(1), \pi(2), \ldots, \pi(n)$ or a string $\pi(1)\pi(2)\cdots\pi(n)$; this is often called "one-line notation". To simplify the notation we sometimes write π_i for $\pi(i)$. As is well-known, the number of permutations of a set with n elements is $n! = n(n-1)\cdots 1$. A useful asymptotic is given by Stirling's approximation

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + O\left(\frac{1}{n^2}\right)\right)$$

The *inverse* π^{-1} of a permutation π is the unique permutation π^{-1} for which $\pi^{-1}(\pi(i)) = \pi(\pi^{-1}(i)) = i$ for $i = 1, 2, \ldots, n$. For example, the inverse of 84235617 is 73425681.

Another notation for writing permutations is *cycle notation*. A *cycle* of length k, or a k-cycle, in a permutation is a sequence of distinct elements $a_1, a_2, \ldots, a_k$ such that $a_i = \pi(a_{i-1})$ for $i = 2, 3, \ldots, k$ and $a_1 = \pi(a_k)$. Such a cycle is written $(a_1\ a_2\ \cdots\ a_k)$. Clearly any permutation can be written as a collection of disjoint cycles. For example, a cycle notation for 84235617 is (5)(8 7 1)(4 3 2)(6). It is customary to omit 1-cycles from the notation, to write cycles with the smallest element listed first, and to order the cycles by their smallest elements. Thus the previous example would be written as (1 8 7)(2 4 3). A *transposition* is a permutation that is a 2-cycle. A permutation without 1-cycles is a *derangement*. In one-line notation a derangement π of $[n]$ is characterized by the property that $\pi_i \neq i$ for all $i \in [n]$.

The number of permutations of $[n]$ with k cycles is the *signless Stirling number of the first kind*, $c(n, k)$. The numbers $c(n, k)$ also count the number of permutations of n with k left-to-right maxima (i.e., elements π_i such that $\pi_i > \pi_j$ for all $j < i$). For example, the 4 left-to-right maxima in the permutation $\underline{4}32\underline{56}8\underline{7}1$ are underlined. Exercise 3 asks for a proof. Clearly, $c(n, n) = 1$ and $c(n, 1) = (n-1)!$. If $1 < k < n$, then

$$c(n, k) \;=\; (n-1) \cdot c(n-1, k) + c(n-1, k-1). \tag{2.5}$$

A table may be found in Appendix A.

An *inversion* of a permutation π is a pair (π_i, π_j) where $i < j$ and $\pi_i > \pi_j$. Let $I(\pi)$ denote the number of inversions of π. For example, classifying the inversions by decreasing values of i, the permutation 84235617 has $0 + 1 + 1 + 1 + 1 + 3 + 7 = 14$ inversions. The *sign* of π is defined to be $(-1)^{I(\pi)}$. A permutation π is said to be *even* or *odd* as $I(\pi)$ is even or odd. The *parity* of a permutation π is the parity (either even or odd) of $I(\pi)$. The *parity difference* of a set S of permutations is the number of permutations in S with even parity minus the number with odd parity. The number of inversions of a permutation is the same as the number of inversions of its inverse; i.e., $I(\pi^{-1}) = I(\pi)$.

The *index* of a permutation π is the sum of indices j (where $1 \leq j \leq n-1$) such that $\pi_j > \pi_{j+1}$, and is denoted $J(\pi)$. For example, the index of $\underline{84}235\underline{6}17$ is $1 + 2 + 6 = 9$. A remarkable theorem (of MacMahon) states that the number of permutations with k inversions is the same as the number of permutations with index k; i.e., that

$$|\{\pi \in \mathbb{S}_n : I(\pi) = k\}| \;=\; |\{\pi \in \mathbb{S}_n : J(\pi) = k\}| \tag{2.6}$$

for all $k = 1, 2, \ldots, n$. Exercise 8 asks for a proof.

A permutation π for which $\pi^{-1} = \pi$ is said to be an *involution* ; i.e., it is a permutation such that $\pi(\pi(i)) = i$ for all $i = 1, 2, \ldots, n$. Thus an involution is characterized by the property that all of its cycles have length 1 or 2. Every transposition is an involution but not vice-versa. More generally, a bijection $\phi : S \to S$ is an involution if $\phi(\phi(x)) = x$ for all $x \in S$. The number of standard Young tableau (Section 2.8) with n cells is the same as the number of involutions of $[n]$.

A *run* in a permutation is a maximal length sequence $\pi(i) < \pi(i+1) < \cdots < \pi(j)$. For example, the permutation

$$|3\ 5\ 7|1\ 6\ 8\ 9|4|2|$$

has four runs. The number of permutations of $1, 2, \ldots, n$ with k runs is the *Eulerian number*, and is denoted $\left\langle {n \atop k} \right\rangle$. A table may be found in Appendix A. Clearly, $\left\langle {n \atop 1} \right\rangle = \left\langle {n \atop n} \right\rangle = 1$. If $1 < k < n$, then

$$\left\langle {n \atop k} \right\rangle \;=\; k\left\langle {n-1 \atop k} \right\rangle + (n-k+1)\left\langle {n-1 \atop k-1} \right\rangle. \tag{2.7}$$

A permutation $\pi_1\pi_2\cdots\pi_n$ is *up-down* if $\pi_1 < \pi_2 > \pi_3 < \pi_4 \cdots$. Let $\mathbf{E}_n$ denote the set of up-down permutations of $\{1, 2, \ldots, n\}$. The cardinality of $\mathbf{E}_n$ is denoted E_n. The numbers E_n are called the *Euler numbers* (not to be confused with the Eulerian numbers discussed earlier) and the following recurrence relationmay be found in Comtet [68].

$$E_{n+1} = \frac{1}{2}\sum_{k=0}^{n} \binom{n}{k} E_k E_{n-k} \tag{2.8}$$

The recurrence relation (2.8) may be proven by considering the $2E_{n+1}$ permutations that alternate in either direction. The permutations are then classified by the number of integers, k, to the left of the $n+1$ in the permutation.

A *k-permutation* of an n-set S is an arrangement of some k elements of S. The number of such arrangements is

$$(n)_k \;=\; n(n-1)\cdots(n-k+1).$$

The notation $(n)_k$ is called the "falling factorial". There is an analogous "rising factorial" notation $(n)^k = n(n+1)\cdots(n+k-1)$, but we shall have no occasion to use it.

2.5.1 Permutations of a Multiset

A *multiset* is a like a set, except that repeated elements are allowed. In the literature, sometimes *bag* is used instead of multiset. Suppose that the elements of a multiset S consist of natural numbers between 0 and t, and that there are n_i occurences of i. Then the number of permutations of S is given by the *multinomial coefficient*

$$\binom{n}{n_0, n_1, \ldots, n_t} \;=\; \frac{n!}{n_0!n_1!\cdots n_t!},$$

where $n = |S| = n_0 + n_1 + \cdots + n_t$.

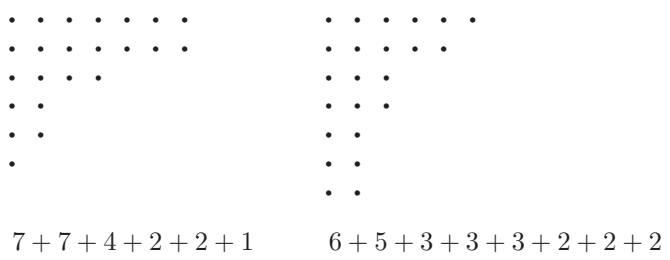


Figure 2.1: Ferrer's diagram and conjugate partition.

2.6 Partitions

Partitions come in two varieties, partitions of an integer and partitions of a set.

2.6.1 Numerical Partitions

A (numerical) *partition* of n into k parts is a sequence $p_1 \geq p_2 \geq \cdots \geq p_k \geq 1$ such that $n = p_1 + p_2 + \cdots + p_k$. Let $P(n)$ denote the number of partitions of the integer n. Here's a table of $P(n)$ for $1 \leq n \leq 17$.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$p(n)$	1	2	3	5	7	11	15	22	30	42	56	77	101	135	176	231	297

Let $p(n,k)$ denote the number of partitions of n into k parts. Then $p(n,1) = p(n,n) = 1$ and for $1 < k < n$,

$$p(n,k) = p(n-1,k-1) + p(n-k,k).$$

A *Ferrer's diagram* of a partition $p_1 \geq p_2 \geq \cdots \geq p_k > 0$ is an arrangement of n dots (on a square lattice) where there are p_i dots in the ith row and the leftmost dots on each row are aligned in a column. See Figure 2.1 for an example. The *conjugate* of a partition is obtained by flipping the Ferrer's diagram about the diagonal running through the upper left dot. Clearly, the conjugate of a partition with k parts results in a partition whose largest part is k, and vice-versa. This observation leads us to the following lemma.

LEMMA 2.1 *The number of partitions of n whose largest part is k is $p(n,k)$.*

Number Theoretic Functions

The *greatest common divisor*, $\gcd(n,m)$, of two natural numbers n and m is the largest natural number that divides both n and m. The *least common multiple*, $\mathrm{lcm}(n,m)$, of n and m is the smallest natural number that is divisible by both n and m. Numbers n and m are *relatively prime* if $\gcd(n,m) = 1$. A proof of the following useful lemma may be found in [159].

LEMMA 2.2 *The multiset of n numbers $\{jm \bmod n : j = 0, 1, \ldots, n-1\}$, consists of d copies of the m/d distinct numbers $\{kd : k = 0, 1, \ldots, m/d - 1\}$, where $d = \gcd(m,n)$.*

Euler's *totient function*, $\phi(n)$, is the number of natural numbers less than n and relatively prime to n. For example, $\phi(1) = \phi(2) = 1$, $\phi(3) = \phi(4) = \phi(6) = 2$, $\phi(5) = \phi(8) = \phi(10) = \phi(12) = 4$, and $\phi(7) = \phi(9) = 6$. An explicit formula is given below, where the product is taken over all primes p that divide n.

$$\phi(n) \;=\; n\prod\left(1-\frac{1}{p}\right). \tag{2.9}$$

Observe that $d = \gcd(i,n)$ if and only if $1 = \gcd(i, n/d)$. Thus

$$\sum_{i=1}^{n} f(\gcd(i,n)) = \sum_{d\backslash n} \phi(n/d)f(d) = \sum_{d\backslash n} \phi(d)f(n/d). \tag{2.10}$$

Define the *Möbius function*, $\mu(n)$ as follows.

$$\mu(n) \;=\; \begin{cases} (-1)^t & \text{if } n \text{ is the product of } t \text{ distinct primes} \\ 0 & \text{otherwise} \end{cases}$$

Thus $\mu(1) = \mu(6) = 1$, $\mu(4) = 0$, and $\mu(p) = -1$ for any prime p. The following useful lemma is known as the Möbius inversion formula.

LEMMA 2.3 *If g and f are functions from positive integers to positive integers, then*

$$f(n) = \sum_{d\backslash n} g(d) \quad \textit{if and only if} \quad g(n) = \sum_{d\backslash n} \mu(n/d)f(d). \tag{2.11}$$

2.6.2 Set Partitions

A (set) *partition* of S into k blocks is a collection of sets $\{S_1, S_2, \ldots, S_k\}$ where each $S_i \subseteq S$, the sets are pairwise disjoint ($S_i \cap S_j = \emptyset$ for all $i \neq j$), and the union of all sets is S (for each $s \in S$, there is an i for which $s \in S_i$). The number of set partitions of an n-set is denoted B_n which is called a *Bell number*. They satisfy the following recurrence relation. If $n = 0$, then $B_0 = 1$, and for $n \geq 0$,

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k, \tag{2.12}$$

from which we can construct the following table.

n	1	2	3	4	5	6	7	8	9	10	11	12
B_n	1	2	5	15	52	203	877	4140	21147	115975	???	???

There is a very cute tabular way to compute Bell numbers; see Exercise 17. By $\left\{ {n \atop k} \right\}$ we denote the number of partitions of an n-set into exactly k disjoint subsets. The disjoint subsets are traditionally called *blocks*. The numbers $\left\{ {n \atop k} \right\}$ are known as the *Stirling numbers of the second kind*. A table may be found in Appendix A. They satisfy the following recurrence relation which may be proven by classifying partitions according to whether n occurs in a block by itself or not.

$$\left\{ {n \atop k} \right\} \;=\; \left\{ {n-1 \atop k-1} \right\} + k\left\{ {n-1 \atop k} \right\} \tag{2.13}$$

The following explicit expression is known.

$$\left\{ {n \atop k} \right\} \;=\; \frac{1}{k!}\sum_{i=0}^{k} (-1)^{k-i} \binom{k}{i} i^n$$

1	2	3	7	8	18
4	5	10	12		
6	9	13	17		
11	16				
14					
15					

11	8	6	5	2	1
8	5	3	2		
7	4	2	1		
4	1				
2					
1					

Figure 2.2: (a) A standard Young tableau of shape 6,4,4,2,1,1. (b) The hook lengths of that tableau.

2.7 The twelve-fold way

The following table, taken from Stanley's book [402], gives a unified look at the most fundamental numbers of combinatorics. Consider functions $f : N \to X$ where $|N| = n$ and $|X| = x$. The different numbers arise depending upon whether the elements of N and X are regarded as being distinguishable (dist.) or indistinguishable (indist.). Further classification is done depending upon whether f is one-to-one or onto. If P is a proposition, then $[\![P]\!]$ is 1 if P is true and is 0 if P is false.

Elements of N	Elements of X	Any f	Injective f	Surjective f
dist.	dist.	x^n	$(x)_n$	$x!\left\{{n \atop x}\right\}$
indist.	dist.	$\binom{n+x-1}{n}$	$\binom{x}{n}$	$\binom{n-1}{n-x}$
dist.	indist.	$\sum_{i=1}^{x} \left\{{n \atop i}\right\}$	$[\![n \le x]\!]$	$\left\{{n \atop x}\right\}$
indist.	indist.	$\sum_{i=1}^{x} p(n,i)$	$[\![n \le x]\!]$	$p(n,x)$

2.8 Tableau

Let $\lambda = \lambda_1, \lambda_2, \ldots, \lambda_k$ be a partition of the number n. A *standard Young tableau* (or simply a tableau) of shape λ is an arrangement of $1, 2, \ldots, n$ into k rows of numbers, where the ith row contains λ_i numbers, and the numbers along each row and each column are increasing. The numbers are placed in the manner of a Ferrers diagram, the leftmost number in each row aligned in a column.

There is an amazing formula that gives the number of standard tableau of given shape, call is d_λ. The hook length of a "cell" in a tableau is the number of cells to the right of it or below it, including the cell itself. In the figure we show a tableau of shape 6,4,4,2,1,1 and to the right of it, the hook length of each cell. Here's the formula:

$$d_\lambda = \frac{n!}{\prod_{c \in \lambda} h_c}$$

2.9 Partially Ordered Sets

Partially ordered sets are finding increased application in diverse areas of computer science and discrete mathematics. There is even a journal, *Order*, that is devoted to results about them.

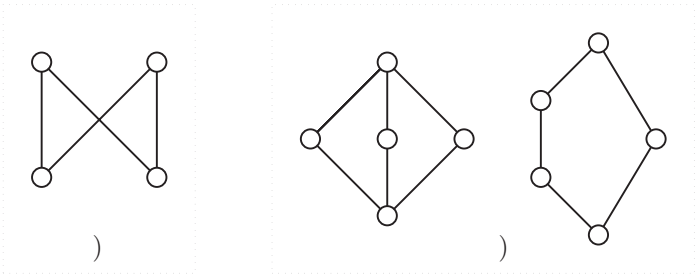


Figure 2.3: (a) Forbidden subposet for lattices. (b) Forbidden sublattices for distributive lattices.

A *poset* (partially ordered set) $\mathcal{P}$ is a pair $\mathcal{P} = (S(\mathcal{P}), R(\mathcal{P}))$, where $S(\mathcal{P})$ is a set and $R(\mathcal{P})$ is a reflexive, antisymmetric, transitive relation on $S(\mathcal{P})$. If $(x, y) \in R(\mathcal{P})$, then we write $x \preceq_{\mathcal{P}} y$, or $x \preceq y$ if $\mathcal{P}$ is understood from context.

If $x \preceq y$ and $x \neq y$, then we write $x \prec y$. If neither of the relations $x \preceq y$ or $y \preceq x$ hold, then we say that x and y are *incomparable* and write $x \parallel y$. A *cover* relation $x \prec y$ is one for which there does not exist a z with $x \prec z \prec y$. A *Hasse diagram* is a representation of a poset as an undirected graph embedded in the plane. Edges connect those elements related by a cover relation. If $x \prec y$, then x is drawn "lower" in the diagram than y. An element x is *minimal* if it covers no other element. An element z is *maximal* if it is covered by no other element.

An *antichain* is a set of incomparable elements. A *chain* is a set of elements, each pair of which is comparable; thus a chain can be regarded as a sequence $x_1 \prec x_2 \prec \cdots \prec x_n$. If $\mathcal{P}$ and $\mathcal{Q}$ are posets over the same set S of elements (i.e., $S(\mathcal{P}) = S(\mathcal{Q})$) and $R(\mathcal{P}) \subseteq R(\mathcal{Q})$, then $\mathcal{Q}$ is said to be an *extension* of $\mathcal{P}$. If $\mathcal{Q}$ is a chain, then $\mathcal{Q}$ is said to be a *linear extension* of $\mathcal{P}$. In other words, a linear extension is a permutation $x_1 x_2 \cdots x_n$ of the elements of $S(\mathcal{P})$ with the property that $x_i \prec x_j$ implies $i < j$. The number of linear extensions of a poset $\mathcal{P}$ is denoted $e(\mathcal{P})$. The probability, $P(x < y)$, that x preceds y is the number of linear extensions in which x appears before y, divided by $e(\mathcal{P})$. Clearly, $P(x < y) + P(y < x) = 1$. The height, $h(x)$, of an element $x \in S(\mathcal{P})$ is the average position that it occupies in a linear extension. Thus, a minimum element has height 1, a maximum element has height $n = |S(\mathcal{P})|$, and if $\mathcal{P}$ is an antichain then all elements have height $(n + 1)/2$.

The graph whose vertices are the linear extensions of $\mathcal{P}$ and with edges connecting those vertices that differ by a transposition of two elements is called the *permutohedron* of $\mathcal{P}$.

Lattices

Let $\mathcal{P}$ be a finite poset and A a subset of its elements. A *lower bound* on A is an element x for which $x \preceq y$ for all $y \in A$. An *upper bound* on A is an element z for which $y \preceq z$ for all $y \in A$. The *greatest lower bound*, $\mathrm{glb}(A)$, of A is the element z' for which $z \leq z'$ for all lower bounds z on A; clearly a glb need not exist. Similarly, the *least upper bound*, $\mathrm{lub}(A)$, of A is the element x' for which $x' \preceq x$ for all upper bounds x on A. A poset for which $\mathrm{lub}(x, y)$ and $\mathrm{glb}(x, y)$ exist for all elements x and y is a *lattice*. Lattices are characterized by the property that they have no induced subposet isomorphic to a "butterfly", as in Figure 2.3(a).

A lattice is *distributive* if it satisfies the the following identities:

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z),$$

$$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z).$$

A set I where $I \subseteq S(\mathcal{P})$ is an *ideal* of $\mathcal{P}$ if $y \in I$ and $x \prec y$ implies that $x \in I$. The set of ideals of $\mathcal{P}$ ordered by set inclusion forms a distributive lattice, which we denote $J(\mathcal{P})$. In fact, all distributive lattices arise as the lattice of ideals of some poset and thus there is a natural one-to-one correspondence between distributive lattices and posets as specified in the following famous theorem of Birkhoff.

THEOREM 2.1 *A lattice L is distributive if and only if it is isomorphic to $J(\mathcal{P})$ for some poset $\mathcal{P}$.*

Distributive lattices also have a forbidden sublattice characterization where the forbidden sublattices are those shown in Figure 2.3(b).

2.9.1 Antimatroids

Antimatroids form an interesting and useful generalization of the ideals and linear extensions of posets. An antimatroid $\mathcal{A}$ is a collection of subsets of some finite set S with the following two properties: (a) For every non-empty $A \in \mathcal{A}$, there is an element $a \in A$ such that $A \setminus \{a\} \in \mathcal{A}$, and (b) if $A, B \in \mathcal{A}$, then $A \cup B \in \mathcal{A}$. We will assume that the union of all sets in $\mathcal{A}$ is S. It is easy to check that the ideals of any finite poset form an antimatroid. The *basic words* of an antimatroid are all strings $a_1 a_2 \cdots a_n$ such that $n = |S|$ and $\{a_1, a_2, \ldots, a_k\} \in \mathcal{A}$ for all $k = 1, 2, \ldots, n$. In the poset case, basic words correspond to linear extensions.

2.10 Trees

There are several important types of trees. We begin with ordered trees, which are the type most frequently encountered in computer science, and then move onto rooted trees and free trees, which are more common in mathematics.

2.10.1 Ordered Trees

DEFINITION 2.1 *Ordered trees are defined recursively as a collection of distinct* nodes *constructed according to the following rules:*

1. *A single node is an ordered tree.*
2. *If $T_1, T_2, \ldots, T_k$ are ordered trees then so is the tree formed by taking a new root r and making the roots of the T_i children of r. The subtrees are ordered from left-to-right.*

When the root r of a rooted tree is removed what remains are called the *principle subtrees*; these are the subtrees $T_1, T_2, \ldots, T_k$ in the definition above. The *degree* of r is k.

There are two ways of visiting all nodes of an ordered tree that are of great importance, called preorder and postorder. In *preorder* we visit first the root r and then the principal subtrees recursively in the order $T_1, T_2, \ldots, T_k$. In *postorder* we visit first the principal subtrees in order and then the root.

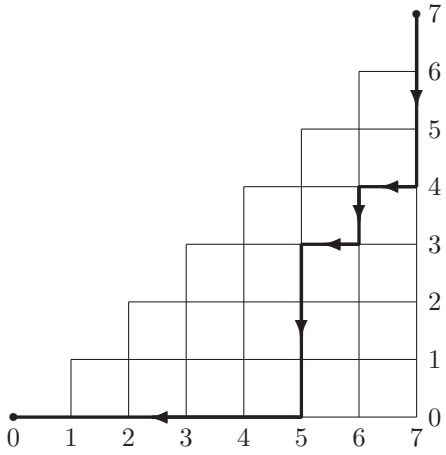


Figure 2.4: The path corresponding to 11101011100000.

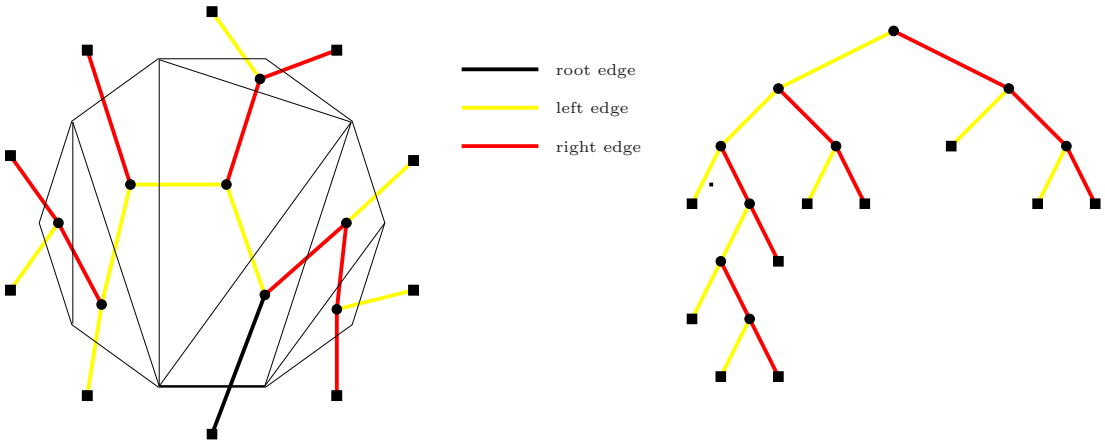


Figure 2.5: A triangulation of a rooted 10-gon and the corresponding extended binary tree with 10 leaves.

Let $\mathbf{T}(n)$ denote the set of binary trees with n nodes, $\mathbf{Ord}(n)$ the set of ordered trees with n nodes and $\mathbf{W}(n)$ the set of well-formed parentheses with n left parentheses and n right parentheses. There are natural bijections between each of the following sets: $\mathbf{T}(n)$, $\mathbf{Ord}(n+1)$, $\mathbf{W}(n)$.

Binary trees are counted by the Catalan numbers The nth Catalan number is denoted C_n and has the following value.

$$C_n = \frac{1}{n+1}\binom{2n}{n}. \tag{2.14}$$

Catalan numbers satisfy the following recurrence relation (with $C_0 = 1$).

$$C_{n+1} = \sum_{k=0}^{n} C_k C_{n-k}. \tag{2.15}$$

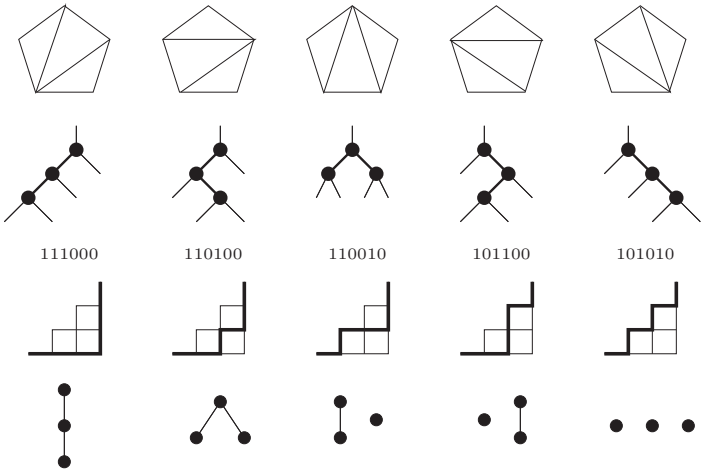


Figure 2.6: Catalan correspondences for $n = 3$: (a) triangulations of a 5-gon, (b) binary trees with 3 internal nodes, (c) tree sequences, (d) lattice walks on 3 by 3 half-grid, (e) ordered forests with 3 nodes.

The usual proof of (2.14) uses (2.15) to derive a generating function equation which is then solved. A much nicer combinatorial proof is based on Lemma 2.4 below.

DEFINITION 2.2 *A* valid *sequence of 0's and 1's is one for which the number of 1's always exceeds the number of 0's in any prefix.*

For example, 11010 is valid but 110011 is not. The following lemma is known as the "Cycle Lemma".

LEMMA 2.4 *Among all the circular shifts of a bitstring with n 0's and m 1's with $n < m$ there are $m - n$ valid bitstrings.*

To prove (2.14) take $m = n+1$. There are $\binom{2n+1}{n}$ bitstrings of length $2n+1$. They may be divided into equivalence classes, each of which contains $2n+1$ members (depending on the position of the unique 1 that starts a valid bitstring). Each eqivalence class corresponds to an element of $\mathbf{T}(n)$ by removing the initial 1 from the single valid bitstring of each equivalence class. The number of equivalence classes (and hence the value of C_n) is therefore

$$C_n \;=\; \frac{1}{2n+1}\binom{2n+1}{n} \;=\; \frac{1}{n+1}\binom{2n}{n}.$$

The cycle lemma may be proven by repeatedly deleting 10 pairs. After performing the deletions there are $m - n$ 1's remaining, which are precisely the starting 1's of the valid bitstrings. The cycle lemma is related to the so-called "trip around the moon" theorem, which is based on the following scenario. Imagine that there are space stations in a fixed orbit around the moon and a rocket ship that takes 100 units of fuel to complete one revolution of the moon. Suppose that a total of 100 units of fuel have been distributed at the stations. The theorem states that there is at least one station at which the rocket ship may start its journey and be able to complete one revolution, no matter how the stations are situated or how the fuel is distributed.

The number of ordered trees with n_i nodes of degree i is

$$\frac{1}{n_0}\binom{n}{n_0, n_1, \ldots, n_t}\left[1 + \sum_{i=0}^{t}(1-i)n_i\right]. \tag{2.16}$$

2.10.2 Rooted Trees

Rooted trees can be defined similarly to ordered trees with the difference that the collection of subtrees of a node are regarded as being unordered. An *in-tree* is a rooted tree in which all edges are directed towards the root, and an *out-tree* is a rooted tree in which all edges are directed away from the root. Our definition of the "degree" of a node in a (rooted or ordered) tree comes from viewing it as an out-tree and then taking degree to mean out-degree.

Let T be a rooted tree with n_i nodes with i children for $i = 0, 1, \ldots, t$. Thus $n = n_0 + n_1 + \cdots + n_t$ is the total number of nodes in the tree. Since the number of edges in a tree is $n - 1$, we have

$$n - 1 \;=\; 0 \cdot n_0 + 1 \cdot n_1 + \cdots + t \cdot n_t$$

It thus follows that

$$n_0 - 1 \;=\; 1 \cdot n_2 + \cdots + (t-1) \cdot n_t \;\geq\; n_2 + \cdots + n_t,$$

with equality in the second relation only if $t = 2$. Thus

$$n \;\leq\; 2n_0 + n_1 - 1. \tag{2.17}$$

The inequality (2.17) is important in analyzing certain recursive algorithms for generating combinatorial objects. In these algorithms n is a measure of the total amount of computation and n_0 is the number of objects produced. Then (2.17) allows us to conclude that an algorithm is CAT if $n_1 = O(n_0)$. In particular, if there are no nodes of degree one, then the algorithm is CAT. Another condition that is useful in proving the CAT property is to be found in the following lemma.

LEMMA 2.5 *If the maximum number of nodes of a chain of degree one nodes is bounded by a constant c, then $n_1 = O(n_0)$.*

PROOF:A node x of degree $k > 1$ has at most k chains of degree one nodes, the roots of which chains are the children of x. Thus

$$\begin{aligned} n_1 &\leq c(2n_2 + 3n_3 + \cdots + tn_t) \\ &= c(n - 1 - n_1) \\ &\leq 2c(n_0 - 1). \end{aligned}$$

□

The number of rooted trees and free trees are asymptotic to

$$a_n \;\sim\; C_1\frac{p^n}{n^{3/2}} \quad \text{and} \quad t_n \;\sim\; C_2\frac{p^n}{n^{5/2}}, \quad \text{respectively,} \tag{2.18}$$

where $C_1 = 0.4399...$, $C_2 = 0.5349...$, and $p = 2.956....$

Labelled Free Trees

There is a simple expression for the total number of labelled free trees.

THEOREM 2.2 *The number of labelled free trees on n nodes is n^{n-2}.*

PROOF:We present here a very nice proof due to Eğecioğlu and Remmel [114] which has a number of important combinatorial consequences. The idea is to establish a bijection θ_{n+1} between the set of all labelled trees on $n+1$ nodes that are rooted at $n+1$ and the set of functions $f : \{2, 3, \ldots, n\} \to \{1, 2, \ldots, n+1\}$. The nodes are assumed to be labelled $\{1, 2, \ldots, n+1\}$ with edges directed towards $n+1$. The bijection is best described by means of an example. Consider the following function.

i	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$f(i)$	3	4	5	3	21	7	12	1	4	4	20	19	19	6	1	16	6	7	12

Now draw the functional digraph of f as illustrated in Figure 2.7. The functional digraph consists of two trees rooted at 1 and $n+1$, together with a number of unicyclic digraphs; in each case edges are either on the cycle or directed toward a cycle or a root. As illustrated in the figure, imagine that the components of the digraph are drawn so that they satisfy the following properties.

- The components are ordered left-to-right by the smallest numbered node that is a root or on a cycle; thus the trees with roots 1 and $n+1$ are drawn on the extreme left and right, respectively.
- Cycles are drawn so their vertices form a directed path on the line between 1 and $n+1$, with one backedge above the line and other edges below the line and directed upwards.
- Each cycle is arranged so that its smallest element is at the right.

Suppose that there are k cycles. Let $l_1, l_2, \ldots, l_k$ and $r_1, r_2, \ldots, r_k$ be the leftmost and rightmost vertices on each of those cycles, respectively. To obtain the corresponding tree remove the edges $[r_i, l_i]$ for each $i = 1, 2, \ldots, k$ and add the edges $[r_{i-1}, l_i]$ for each $i = 1, 2, \ldots, k+1$, where $r_0 = 1$ and $l_{k+1} = n+1$. Note that $r_0 < r_1 < \cdots < r_k$.

We have now described how to obtain the tree from the function. To obtain the function from the tree proceed as follows:

1. Locate the unique path from 1 to $n+1$; call the vertices on this path $1 = a_1, a_2, \ldots, a_m = n+1$.
2. From right to left determine l_i and r_i by the following process. Let $r_k = a_{m-1}$ and $l_{k+1} = n+1$. Inductively assume that r_i and l_{i+1} have been determined. Then let

$$r_{i-1} = a_p \quad \text{where} \quad p = \max\{j : a_j < r_{i-1}\},$$

   and $l_i = a_{p+1}$. Do this for $i = k-1, k-2, \ldots, 1, 0$.
3. Remove the edge $[r_{i-1}, l_i]$ for $i = 1, 2, \ldots, k+1$ and add the edges $[r_i, l_i]$ for $i = 1, 2, \ldots, k$.

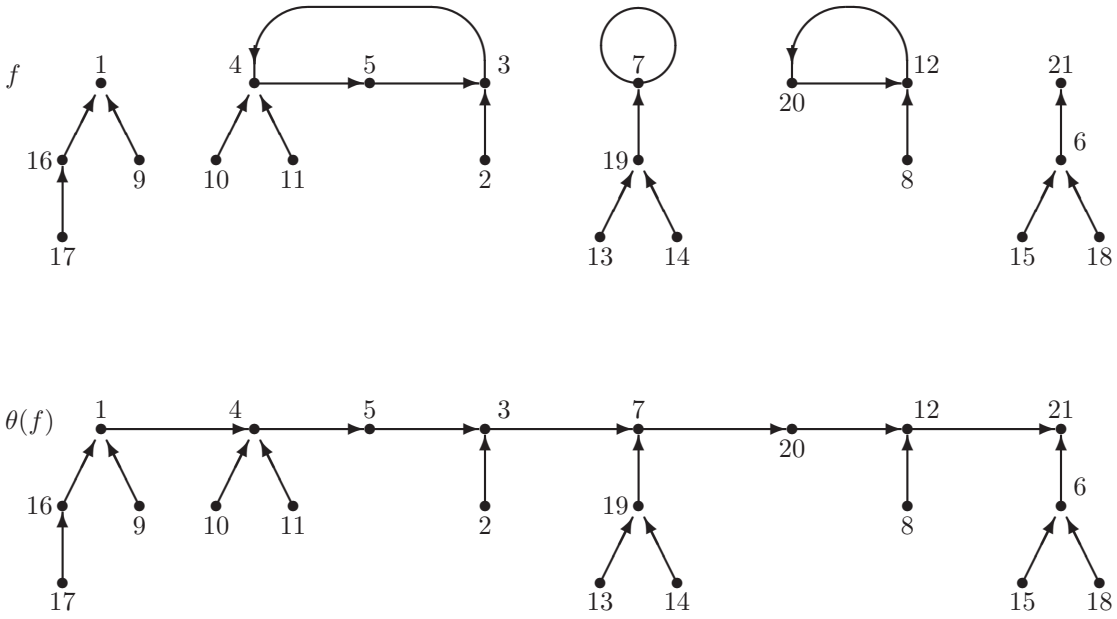


Figure 2.7: The functional digraph of f and the bijection θ.

□

Enumeration of free trees
Lattice path correspondences

2.10.3 Free Trees

A *free tree* (or simply, a *tree*) is a connected graph without cycles.

A vertex v is a *center* of a tree T if its maximum distance to any vertex of T is minimum. Three useful properties of centers are given below.

1. Every tree has either one or two centers. If it has two centers, then it is said to be *bicentral*.

2. If a tree is bicentral, then its centers are adjacent.

3. A vertex is a center if and only if it is a center of every path of maximum length in the tree.

By removing a vertex v from a tree, some subtrees $T_1, T_2, \ldots$ result. Call the *weight* of a v, denoted $w(v)$, the maximum of $|T_1|, |T_2|, \ldots$. A vertex v is a *centroid* if it has minimum weight. Three useful properties of centroids are given below.

1. Every tree has either one or two centroids. If it has two centroids, then it is said to be *bicentroidal*.

2. A tree is bicentroidal if and only if they are adjacent and the removal of the edge joining them results in two trees of equal sizes.

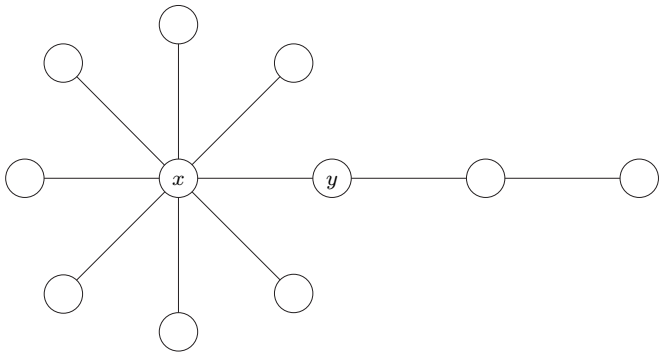


Figure 2.8: For this tree x is the centroid and y is the center.

3. A vertex v is a unique center if and only if $w(v) \leq (n-1)/2$.

The centroid and center of a tree need not coincide; see for example Figure 2.8

2.11 Graphs

Graphs provide an important conceptual tool for us in dealing with Gray codes and, of course, are of deep mathematical interest in their own right. We state here a few basic definitions and results. For further information consult a basic text such as Bondy and Murty [35]. There are a great many terms that are used in graph theory and a good deal of care is required to define everything properly.

A *undirected graph* (or simply a *graph*) G is a pair $(V(G), E(G))$ where $V(G)$ is a finite set and $E(G)$ is a set of 2-subsets of $V(G)$. Where no confusion can arise a graph is denoted (V, E). The elements of V are called the *vertices* of the graph and the elements of E are called the *edges* of the graph. A graph $H = (V(H), E(H))$ is a *subgraph* of G if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. Graph H is an *induced subgraph* of G if $V(H) \subseteq V(G)$ and $E(H) = \{[u, v] \in E(G) \mid u, v \in V(H)\}$. If $[v, w] \in E$ then v and w are said to be *adjacent* (or *joined*). The *degree* of a vertex is the number of vertices to which it is adjacent. A *pendant* vertex is a vertex of degree one. A graph in which every vertex has degree k is said to be *k-regular*. Another name for a 3-regular graphic is *cubic graph*.

A *path* in a graph is a sequence of vertices $v_0, v_1, \ldots, v_l$ such that $[v_{i-1}, v_i] \in E$ for $i \in [l]$. The *length* of the path is l. A *cycle* is a path for which $v_0 = v_l$. Again, the *length* of the cycle is l. An *l-cycle* is a cycle of length l in which the vertices $v_1, v_2, \ldots, v_l$ are all distinct. A graph is *connected* if there is a path between every pair of vertices. It is *acyclic* if there is no cycle of length three or more.

A function ϕ mapping V to V is an *automorphism* of $G = (V, E)$ if $[v, w] \in E$ if and only if $[\phi(v), \phi(w)] \in E$. The set of all automorphisms of a graph form a group under function composition. If, for every pair of vertices v and w, there is an automorphism that maps v to w, then G is said to be *vertex-transitive*.

A *directed graph* G is a pair $(V(G), E(G))$ where $V(G)$ is a finite set and $E(G)$ is a subset of $V(G) \times V(G)$. An edge $(v, w) \in E(G)$ is said to be directed *from* v to w, and is drawn in the figures as $v \rightarrowtail w$.

The *out-degree*, $d^-(v)$, of a vertex v is the number of edges of the form (v, w) where

$w \in V$. The *in-degree*, $d^+(v)$, of a vertex v is the number of edges of the form (w, v) where $w \in V$.

With each directed graph $G = (V(G), E(G))$ there is associated an undirected graph $G' = (V(G), E(G'))$ where $[v, w] \in E(G')$ if $(v, w) \in E(G)$ with $v \neq w$. A directed graph is *connected* if the associated undirected graph is connected. Paths and cycles are defined as they were for undirected graphs except that the directions of the edges must be respected. I.e., a *path* is a sequence of vertices $v_0, v_1, \ldots, v_l$ such that $(v_{i-1}, v_i) \in E$ for $i \in [l]$. A directed graph is *acyclic* if it contains no cycles. Sometimes a directed acyclic graph is called a *dag*. Every dag has a *topological sorting* which is an ordering of the vertices $v_1 v_2 \cdots v_n$ so that $(v_i, v_j) \in E$ implies $i < j$; this concept is closely related to that of a linear extension of a poset.

An *acyclic orientation* of an undirected graph G is a directed acyclic graph G' obtained from G by orienting each edge; i.e., each edge $[u, v] \in E(G)$ becomes either (u, v) or (v, u) in $E(G')$. A directed graph is *strongly connected* if there is a path between every pair of vertices.

A graph is *bipartite* if the vertex set V can be partitioned into two sets A and B such that every edge is incident with an element of A and an element of B. The sets A and B are called *partite sets* . The basic characterization of bipartite graphs is given in the following theorem.

THEOREM 2.3 *A graph is bipartite if and only if it contains no odd cycles.*

The *complete graph* K_n has n vertices and an edge between every pair of vertices. The *complete bipartite graph* $K_{n,m}$ has n vertices in one partite set and m in the other partite set, and an edge between every pair of vertices in different partite sets. A *star* graph is $K_{1,n}$ for some n.

The *cartesian product* $G \times H$ of two graphs G and H is the graph with vertex set $V(G) \times V(H)$ and with edge set

$$\{\{(u, v), (x, y)\} \mid \{u, x\} \in E(G) \text{ and } \{v, y\} \in E(H)\}$$

The *prism* of G is the graph $G \times e$, where e is the graph consisting of a single edge.

An *n-cube*, denoted Q_n is the graph $K_2 \times K_2 \times \cdots \times K_2$, where the product is taken n times. Alternately, we may define Q_n as the graph whose vertices are $\{0, 1\}^n$ and where two edges are joined by an edge if they differ in exactly one position. A *hypercube* is a graph that is an n-cube for some n. The graph Q_4 is known as a *tesseract*.

If G is a graph then by G^k we denote the graph with the same vertex set as G but which has an edge between every pair of vertices that are connected by a path of length at most k in G. In other words, if M is the incidence matrix of G, then M^k is the incidence matrix of G^k, where arithmetic is done logically (1 means *true*, 0 means *false*, $\times$ means $\wedge$, and $+$ means $\vee$). The *cube* of G is G^3 and the *square* of G is G^2.

The concept of a graph is extended to allow for multiple copies of edges and *self-loops*, which are edges of the form $\{v, v\}$. In a *multi-graph* $G = (V(G), E(G))$, the edge set $E(G)$ is a multiset of 1- or 2-subsets of $V(G)$. In a *multi-digraph* $E(G)$ is a multiset of $V(G) \times V(G)$. Many of the previous definitions make sense in the context of multi-graphs and multi-digraphs but there are a few exceptions: the degree of a vertex counts a self-loop twice; paths must be thought of as sequences of edges, rather than vertices. Let $G = (V, E)$ be a multi-graph with n vertices and e edges. An *Eulerian cycle* in a G is a sequence of

vertices of the graph $v_1, v_2, \ldots, v_e$ (with repetitions allowed) such that the following multiset equality holds:

$$E = \{v_e, v_1\} \cup \{\{v_i, v_{i+1}\} \mid i = 1, 2, \ldots, e\}.$$

LEMMA 2.6 *A connected undirected multigraph is Eulerian if and only if every vertex has even degree. A connected directed multi-graph is Eulerian if and only if the out-degree of each vertex is the same as its in-degree.*

A *Hamilton path* H in G is a sequencing of vertices of the graph $H = v_1, v_2, \ldots, v_n$ such that each vertex occurs exactly once and $\{v_i, v_{i+1}\} \in E$ for $i = 1, 2, \ldots, n-1$. If in addition $\{v_n, v_1\} \in E$ then H is a *Hamilton cycle*. A graph that has a Hamilton cycle is said to be *Hamiltonian*. Note that under this definition a graph consisting of a single edge has (actually is) a Hamilton path, and is Hamiltonian. Many results about the Hamiltonicity of particular graphs may be found in Chapter 6.

There are many important variants of Hamiltonicity: A graph is *Hamilton-connected* if there is a Hamilton path between every pair of vertices in the graph. A bipartite graph is *Hamilton-laceable* if there is a Hamilton path between every pair of vertices, where each of the two vertices is in a different partite set. A graph $G = (V, E)$ is *pancyclic* if it possesses l-cycles for $l = 3, 4, \ldots, |V|$. A graph is *hypo-Hamiltonian* if it is not Hamiltonian but every vertex-deleted subgraph $G - \{v\}$ is Hamiltonian. All of the definitions of this paragraph apply without modification to digraphs.

A set $I \subseteq V(G)$ is an *independent set* if there is no edge $e = \{v, w\} \in E(G)$ such that $v \in I$ and $w \in I$. A set $C \subseteq V(G)$ is a *clique* if $(u, v) \in E$ for every distinct pair $u, v \in C$. A set $U \subseteq V(G)$ is a *vertex cover* if, for every edge $(u, v) \in E$, we have $u \in U$ or $v \in U$ (or both). A *k-coloring* of a graph is a function $f : V \to [k]$ such that $f(u) \neq f(v)$ whenever $[u, v] \in E$.

A graph $H = (V(H), E(H))$ is a *spanning subgraph* of $G = (V(G), E(G))$ if $V(H) = V(G)$ and $E(H) \subseteq E(G)$. A *spanning tree* of G is a spanning subgraph T which is a tree.

A directed graph with n vertices is a *tournament* if for every pair i, j with $1 \leq i < j \leq n$ the graph either has the edge (i, j) or has the edge (j, i), but not both.

2.11.1 The Matrix Tree Theorem

The Matrix Tree Theorem provides a elegant and efficient way to count the number of spanning trees in an undirected graph, or the number of spanning out-trees in a digraph. We denote by $\tau(G)$ the number of spanning trees of G. Given a graph G on n vertices labelled from $[n]$, the *Kirchoff matrix* $K(G)$ is the matrix whose ith diagonal entries is the degree of vertex i and where the ij entry for $i \neq j$ is -1 if $[i, j] \in E(G)$ and 0 otherwise.

THEOREM 2.4 (MATRIX TREE) *Let G be a labeled graph on $[n]$. The number of spanning trees of G is equal to any cofactor of $K(G)$.*

2.11.2 Representing Graphs

There are two standard ways of representing graphs, one which uses a matrix and the other which uses linked lists. The *adjacency matrix representation* is a binary matrix $A = [a_{ij}]$, an n by n matrix for which a_{ij} is 1 if there is an edge from i to j and is 0 if not. This

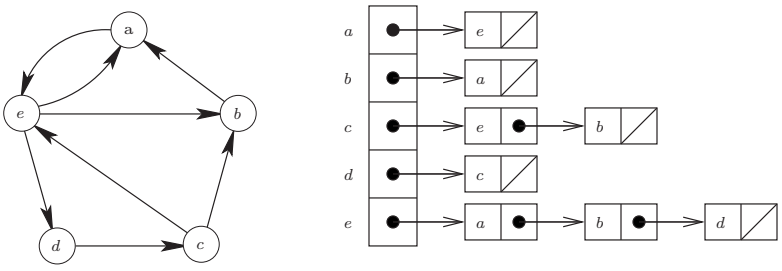


Figure 2.9: A graph and its adjacency list representation.

representation is useful for graphs with many edges. In the case of an undirected graph, A is symmetric.

The *adjacency list representation* is most useful for graphs with smaller numbers of edges, as often arise in practice. A graph together with its adjacency list representation is shown in Figure 2.9. Assume that the vertices of the graph are labelled $1, 2, \ldots, n$. Define an array `Adj[1..n]` of pointers to nodes with containing at least a field `vert` and another field `next`. The `next` fields are used to form a linked list, pointed to by `Adj[v]` containing those vertices pointed to by an edge starting from v. For each edge $(v, w) \in E$, there is a node on the list `Adj[v]` whose `vert` field has the value w. If G is an undirected graph, then each vertex $[u, v] \in E$ is represented twice, as if there were directed edges (u, v) and (v, u). In the case of multi-graphs both representations are easily extended. The adjacency list representation is unchanged; multiple edges occur as multiple occurences on the relevant list. The adjacency matrix representation is changed so that a_{ij} represents the number of times that edge (i, j) occurs.

2.11.3 Depth First Search

Depth-first search is a simple recursive way of exploring, or traversing, a graph. It is often used with graphs represented by adjacency lists. See Algorithm 2.1. Initially the array `num[1..n]` is initialized to zero, as are the variables `start_time` and `finish_time`, and then the loop

```
for v := 1 to n do if num[v] = 0 then dfs( v );
```

is executed. If G is an undirected multi-graph then `dfs` is called exactly once in this for loop for every connected component of G. The final values of `num` give a *depth-first numbering* of G.

Assuming that graph G has n vertices and m edges, depth-first search runs in time $O(n + m)$ since each vertex is visited exactly once, and its edges list is traversed exactly once.

There are many graph algorithms based on depth-first search, including $O(n + m)$ algorithms to find biconnected components, strongly connected components, and for testing whether a graph is planar. We will not have occasion to use such algorithms, but topological sorting is important to us and a depth-first-search algorithm for this is presented next.

Using depth-first-search for topological sorting

A topological sorting of a directed acyclic graph may quite easily be accomplished in time $O(n + m)$ by performing the following two steps.

```
procedure dfs ( v : vertex );
var w : vertex; p : ^Edge;
begin
  start_time := start_time + 1;
  num[v] := start_time;
  p := adj[v];
  while p <> nil do begin
     w := p^.vert;
     if num[w] = 0 then dfs( w );
     p := p^.next;
  end;
  finish_time := finish_time + 1;
  f[v] := finish_time;
end of dfs;
```

Algorithm 2.1: A Pascal procedure for performing depth-first search of a graph.

1. Do a depth-first-search of G computing a postorder numbering f.
2. Arrange the vertices in the order $v_1, v_2, \ldots, v_n$ where $f[v_1] \geq f[v_2] \geq \cdots \geq f[v_n]$.

These two steps can be arranged into a single modified version of the `dfs` algorithm.

2.12 Finite Groups

A *group*, G, is a set together with a binary operation $*$ that satisfies the following laws.

1. There is an identity element $\mathbf{1} \in G$ such that $g * \mathbf{1} = \mathbf{1} * g = g$ for all $g \in G$.
2. For each element $g \in G$ there is is an *inverse* element, denoted g^{-1} such that $g * g^{-1} = g^{-1} * g = \mathbf{1}$.
3. The operation is associative; that is $(a * b) * c = a * (b * c)$ for all $a, b, c \in G$.

Where no confusion can arise the symbol for the operation $*$ is omitted. Thus, for example, the associative law becomes $(ab)c = a(bc)$.

The *order* of a group G, denoted $|G|$, is the number of elements in G. In this book all groups are finite, and so their order is some natural number. Each element g of a finite group has an *order* which is the smallest integer m such that $g^m = \mathbf{1}$.

A subset H of G that is also a group under the operation of G is a *subgroup* of G.

Two groups are said to be *isomorphic* if there is a bijection between their elements that preserves operations.

Some important groups

- The *cyclic group*
- The *dihedral group*
- The *symmetric group* $\mathbb{S}_n$ is the group of permutations of $[n]$ under the operation of composition.

- The *hypo-octahedral group* is the group of symmetries of a hypercube.
- A group is *Abelian* if the group operation is commutative; that is, if $ab = ba$ for all group elements a and b.

New groups may be obtained from old by various operations. Only one *product* will be used in this book. Let G and H be groups then their product, denoted $G \times H$ is the group whose elements are ordered pairs (g, h) with $g \in G$ and $h \in H$ with multiplication defined componentwise. In other words, $(a, x)(b, y) = (ab, xy)$.

THEOREM 2.5 *Every Abelian group is the product of cyclic groups.*

2.12.1 Permutation Groups

Most groups considered in this book are groups whose elements are permutations. What is the multiplication operation? Let π_1 and π_2 be permutations, viewed as bijective functions from $[n]$ to $[n]$. The multiplication operation is just the composition of the functions *in reverse order*. That is, $\pi_1 * \pi_2(x) = \pi_2(\pi_1(x))$. The *symmetric group*, $\mathbb{S}_n$, consists of all $n!$ permutations of $[n]$ with multiplication defined as above. The reason for reversing the order of function application is that we do computation with permutations most often in cycle notation, and then multiplication can be done very simply with a single left-to-right scan.

THEOREM 2.6 *Every finite group is isomorphic to a permutation group.*

A permutation group is *transitive* if for each pair $x, y \in [n]$ there is a permutation $\pi \in G$ such that $\pi(x) = y$. The *automorphism group* ΓG of a graph G is the set of permutations of its vertices which preserve adjacencies; i.e., $\pi \in \Gamma G$ if for every edge $[u, v] \in E(G)$ it is the case that $[\pi(u), \pi(v)] \in E(G)$. A graph is vertex-transitive if and only if its automorphism group is transitive. In other words, a graph is vertex-transitive if, for every pair $u, v \in V(G)$, there is a $\pi \in \Gamma(G)$ such that $\pi(u) = v$.

LEMMA 2.7 *Every permutation is a product of transpositions.*

Group Actions

We say that a group G *acts* on a set S if each group element $g \in G$ is a function $g : S \to S$ with the properties that

1. $g(h(s)) = (gh)(s)$ for all $g, h \in G$ and $s \in S$.
2. $\mathbf{1}(s) = s$ for all $s \in S$.

It follows from the existence of inverses that each group element in fact defines a bijection on S.

We say that a group element g *fixes* an element of s if $g(s) = s$. Define

$$Fix(g) = \{s \in S \mid g(s) = s\}.$$

The *orbit* of $s \in S$ is the following subset of S.

$$Orb(s) = \{g(s) \mid g \in G\}$$

The *stabilizer* of $s \in S$ is the set of group elements which fix s. We denote

$$G_s \;=\; \{g \in G \mid g(s) = s\}.$$

LEMMA 2.8 *For all* $s \in S$

$$|G_s||Orb(s)| = |G|.$$

2.12.2 Burnside's Lemma

Let G act on S and define $x \sim y$ if and only if there is a $g \in G$ such that $g(x) = y$. In other words, $x \sim y$ if and only if x and y are in the same orbit. This defines an equivalence relation on S. Burnside's lemma is used to count the number of equivalence classes of $\sim$ (i.e., the number of orbits of S).

LEMMA 2.9 (BURNSIDE) *The number of equivalence classes of* $\sim$ *is equal to*

$$\frac{1}{|G|}\sum_{g\in G}|Fix(g)|.$$

PROOF:Let $S_1, S_2, \ldots, S_m$ be the orbits of G acting on S. Clearly,

$$S = S_1 \cup S_2 \cup \cdots \cup S_m$$

forms a partition of S. For each $g \in G$ let $F_i(g) = Fix(g) \cup S_i$. Then $F_1(g), F_2(g), \ldots, F_m(g)$ is a partition of $Fix(g)$. Hence,

$$\begin{aligned}
\sum_{g\in G}|Fix(g)| &= \sum_{i=1}^{m}\sum_{s\in S_i}|G_s| \\
&= \sum_{i=1}^{m}\sum_{s\in S_i}\frac{|G|}{|Orb(s)|} \\
&= \sum_{i=1}^{m}\sum_{s\in S_i}\frac{|G|}{|S_i|} \\
&= n|G|.
\end{aligned}$$

□

2.12.3 Polya Theorem

Let G be a permutation group on $[n]$ and define $b_k(g)$ to be the number of cycles of g of length k. Then the *cycle index polynomial* of G is

$$P(x_1, x_2, \ldots, x_n) \;=\; \frac{1}{|G|}\sum_{g\in G} x_1^{b_1(g)} x_2^{b_2(g)} \cdots x_n^{b_n(g)}$$

Let C be a finite set (of colors) and let Ω be the set of all functions $\phi : [n] \to C$ (ways of coloring X). Then G acts on Ω in the following natural manner. For $g \in G$ and $\phi \in \Omega$ define

$$(g\phi)(x) = \phi(g^{-1}x).$$

THEOREM 2.7 *(Pólya) The number of orbits of* G *acting on* Ω *is*

$$P(|C|, |C|, \ldots, |C|)$$

2.12.4 Cayley Graphs

Let G be a group and S a subset of G. The subset S is said to be a *generating set* or *set of generators* for G if every element of G can be written as $g = g_1 g_2 \cdots g_k$ for some k and elements $g_i \in S$. If G is a group and X is a set of generators[1] of G, then the *directed Cayley graph*, $\vec{Cay}(X:G)$, is the graph whose vertex set is G and where the edges leaving g are all of the form $g \rightarrow gx$ for $x \in X$. The *undirected Cayley graph*, $Cay(X:G)$, has vertex set G and where the edges incident to g are either of the form $[g, gx]$ or $[g, gx^{-1}]$ for $x \in X$. In the undirected case we can think of the edges as being "colored" by the corresponding generator; $Cay(X:G)$ is $|X|$ edge-colorable.

2.13 Miscellanea

Suppose that we are generating sets of objects parameterized by numbers n, where there are a_n objects with parameter n. The analysis of many generation algorithms involves comparing the sum $a_1 + a_2 + \cdots + a_n$ with a_n. The following lemma, easily proven by induction, can be useful.

LEMMA 2.10 *If $a_j/a_{j+1} \leq \alpha < 1$ for all $j \geq 1$, then*

$$\sum_{i=1}^{n} a_i \;\leq\; \frac{1}{1-\alpha} a_n$$

As examples of its use, we observe that $\sum C_i = O(C_n)$, $\sum B_i = O(B_n)$, and $\sum T_i = O(T_n)$, where C_n is the Catalan number, B_n is the Bell number, T_n is the number of rooted trees with n nodes, and the summation is over $i = 1, 2, \ldots, n$. In each case, an asymptotic expression for the numbers is used to show that the condition of Lemma 2.10 is satisfied.

2.13.1 Summations

Two useful summation formula are

$$\sum_{i=0}^{n} \sum_{j=0}^{i} f(i,j) = \sum_{0 \leq j \leq i \leq n} f(i,j) = \sum_{j=0}^{n} \sum_{i=j}^{n} f(i,j) \tag{2.19}$$

$$\sum_{i=0}^{n} i \cdot f(n-i) \;=\; \sum_{j=0}^{n-1} \sum_{i=0}^{j} f(i) \tag{2.20}$$

2.13.2 Fibonacci Numbers

The *Fibonacci numbers*, F_n, are the sequence defined by the recurrence relation $F_n = F_{n-1} + F_{n-2}$ with the initial conditions $F_0 = 0$ and $F_1 = 1$.

[1] some authors allow X to be any subset of G, not necessarily a set of generators

2.13.3 Constant Time Array Initialization

Sometimes it is useful to use an array without initialization. This is possible — however, each array access and update becomes more expensive, but only by a constant factor. Let us assume that we are maintaining an array $a[1..n]$, whose values are initialized to 0. Generalization to other array bounds and initial values is trivial.

We introduce a new variable t, and two arrays $\tau[1..n]$ and $\tau^{-1}[1..n]$. Variable t, call it "time", is initialized to 0, and represents the number of array locations that have been modified. Array entry $\tau[i]$ is the time when $a[i]$ was last modified, if it has been modified; otherwise it is garbage. Array entry $\tau^{-1}[i]$ is maintained so that $\tau^{-1}[\tau[i]] = i$ whenever $\tau[i]$ is not garbage.

The initialization

for i:=1 **to** n **do** $a[i]$:=0;

becomes t:=0. The access $a[i]$ becomes

if $(1 \le \tau[i] \le t)$ **and** $(\tau^{-1}[\tau[i]] = i)$ **then** return($a[i]$) **else** return(0);

this code assumes a short-circuit and (i.e., if the first condition fails then the second is not evaluated). The update $a[i] := x$ becomes

```
a[i] := x;
if not ((1 ≤ τ[i] ≤ t) and (τ^{-1}[τ[i]] = i)) then
    t := t + 1;
    τ[i] := t;
    τ^{-1}[t] := i;
```

This technique should not be used indiscriminately since it will usually slow your program down, unless n is very large and large portions of the array are *never* modified.

2.14 Exercises

Questions about permutations

1. **[1]** Develop an $O(n)$ algorithm for computing the inverse of a permutation of $[n]$.

2. **[1+]** Develop an $O(n)$ algorithm that takes as input a permutation π of $\{1, 2, \ldots, n\}$ in one line notation and computes the cycle representation of the permutation. Show that the sign of a permutation can be determined in time $O(n)$.

3. **[2]** (a) Prove recurrence relation (2.5). (b) By direct correspondence prove that $c(n, k)$ is the number of permutations of n with k left-to-right maxima. HINT: order the cycles and elements of cycles in a certain way and then remove parentheses.

4. **[2]** Develop an $O(n \log n)$ algorithm for determining the number of inversions of a permutation π of $\{1, 2, \ldots, n\}$. **[*R*]** Can the inversion vector of a permutation be determined in time $O(n)$?

5. **[1]** Prove that the number of inversions of a permutation is the same as the number of inversions of its inverse.

6. **[1]** Prove that the number of permutations of $[n]$ with an odd number of cycles is equal to the number with an even number of cycles.

7. **[1]** Characterize the P-sequences of alternating permutations.

8. **[2]** Give an inductive proof that the number of permutations π of $[n]$ with $\pi(n) = p$ and k inversions is the same as the number of permutations σ of $[n]$ with $\sigma(n) = p$ and index k, thus proving (2.6).

9. **[2]** Let I_n denote the number of permutations of $[n]$ that are involutions. Find a simple recurrence relation for I_n. Find a explicit expression for I_n.

10. **[1+]** What is the number of even derangements of length n minus the number of odd derangements of length n?

11. **[3]** Prove that the number of permutations with all cycles of even length is equal to the number of permutations with all cycles of odd length. A bijective proof is preferred.

12. **[2]** Prove Ryser's formula $sum_r(-1)^r\binom{n}{r}(n-r)^r(n-r-1)^{n-r}$ for the number of derangements of $[n]$.

Questions about trees

13. **[2+]** Prove (2.16), the formula for the number of ordered trees with given degrees.

14. **[1]** What is the average (out)degree of a node in a rooted tree?

15. **[1]** A sequence $a_1, a_2, \ldots, a_n$ is *unimodal* if there is some value k such that

$$a_1 \le a_2 \le \cdots \le a_k \ge a_{k+1} \ge \cdots \ge a_n.$$

Prove, for fixed n, that the binomial coefficients are unimodal.

16. **[2]** A sequence $x_0, x_1, \ldots, x_n$ of positive real numbers is said to be *log-concave* if $x_i^2 \ge x_{i-1}x_{i+1}$ for all $0 < i < n$. A sequence $x_0, x_1, \ldots, x_n$ of real numbers is said to be *concave* if $2x_i \ge x_{i-1} + x_{i+1}$ Prove that any concave sequence is unimodal and prove that any log-concave sequence is unimodal.

17. **[1+]** If $\mathbf{a} = a_1, a_2, a_3, \ldots$ is a sequence then $\Delta\mathbf{a}$ denotes the sequence $a_2 - a_1, a_3 - a_2, a_4 - a_3, \ldots$ and $\mathbf{a}\langle n\rangle$ denotes the nth element of $\mathbf{a}$; i.e., $\mathbf{a}\langle n\rangle = a_n$. Let $\mathbf{B}$ be the sequence of Bell numbers. Prove that $B_n = \Delta^n\mathbf{B}\langle 1\rangle$ and use this relation to get a tabular way of computing $\mathbf{B}$. The computation of $B_1, B_2, \ldots, B_n$ should use at most $O(n)$ integers (of size at most B_n) and $O(n^2)$ additions, with no other arithmetic operations. Implement such an algorithm.

18. **[3]** Develop a fast algorithm for testing whether two bitstrings are rotationally equivalent; i.e., whether one can be obtained from the other by a rotation.

Questions related to Catalan numbers

19. [**1−**] Construct the binary and ordered trees corresponding to the bitstring 10111010011000.

20. [**1**] Imagine $2n$ persons seated around a circular table. How many ways are there for all of them to shake hands simultaneously so that no pair of arms crosses?

21. [**2**] Prove that C_n is odd if and only if $n = 2^k - 1$ for some positive integer k.

22. [**2**] Show that, as n tends to infinity, the ratio $(C_1 + C_2 + \cdots + C_n)/C_n$ tends to $4/3$.

Questions about spanning trees

23. [**1**] What spanning trees $\theta(f)$ arise from the functions (a) $f(i) = i$, (b) $f(i) = i + 1$, and (c) $f(i) = i - 1$?

24. [**1+**] Let labelled tree T with n vertices be specified by listing its edges in some order. Given $f : \{2, 3, \ldots, n\} \to [n + 1]$, how fast can $T = \Theta(f)$ be computed? Given T how fast can $f = \Theta^{-1}(T)$ be computed?

25. [**1+**] Give a bijective proof that the number of spanning trees of the complete bipartite graph $K_{n,m}$ is $n^{m-1}m^{n-1}$.

26. [**2+**] Another way to prove Cayley's theorem bijectively is using what is known as Prüfer's correspondence. The correspondence is a bijection

$$P : T(n) \to \{a_1 a_2 \cdots a_{n-2} : 1 \le a_i \le n\}$$

as specified by the procedure below.

for $i := 1$ **to** $n - 2$ **do**
    $v :=$ lowest numbered leaf;
    $a[i] :=$ vertex adjacent to v;
    $T := T - \{v\}$;

(a) What is the array a corresponding to the tree of Figure 2.7? (b) What is the tree corresponding to the function f used in Figure 2.7? (c) Write an efficient procedure for computing P^{-1}. What is the running time of your procedure?

27. [**2**] Use Prüfer's correspondence to prove that (a) the number of labelled trees with t leaves and (b) the number of labelled trees where vertex 1 has degree k are given by the following expressions, respectively.

$$\text{(a)}\ \ \frac{n!}{t!}\left\{{n-2 \atop n-t}\right\} \quad \text{and} \quad \text{(b)}\ \ \binom{n-2}{k-1}(n-1)^{n-k-1}.$$

Questions about numerical partitions

28. **[1]** Give an $O(n)$ algorithm to compute the conjugate of a partition.

29. **[3]** Let $PDO(n)$ and $PDE(n)$ denote the set of partitions of n into distinct parts where the number of parts is odd and even, respectively. Prove that $|PDE(n)| - |PDO(n)|$ is $(-1)^k$ if $n = (3k^2 \pm k)/2$ for some number k and is 0 otherwise. This result is known as the "Euler pentagonal number theorem".

30. **[2]** Use the result of the previous exercise to prove that

$$\sum_{k=-\infty}^{\infty} (-1)^k p(n - (3k^2 + k)/2) \;=\; 0$$

and use this equality to develop a "fast" algorithm for determining $p(n)$.

31. **[1]** Let $N(n, k)$ denote the number of parts of size k taken from the set of all partitions of n. Find a simple expression for $N(n, k)$ in terms of $p(n)$.

32. **Add an exercise about the Squire result??**

Questions about set partitions

33. **[1+]** Give a bijective proof showing that the number of ways of placing k nontaking rooks on the n by n "half-chessboard", obtained by cutting the chessboard along a diagonal, is $\left\{ {n \atop n-k} \right\}$.

34. **[1+]** Let $T(n, k)$ denote the number of partitions of $[n]$ into k blocks where no block contains two consecutive numbers. Show that $T(n + 1, k) = \left\{ {n \atop k-1} \right\}$.

35. **[1+]** Find a simple expression for the average number of blocks in a partition of an n-set.

36. **[1]** Prove that the Bell numbers are convex: $B_n \leq \frac{1}{2}(B_{n-1} + B_{n+1})$.

Miscellaneous questions

37. **[2]** Lemma 2.7 states that every permutation π is a product of transpositions. Prove that the least number of transpositions in the product is $n - r$, where r is the number of cycles in π.

38. **[2]** Given a partition λ of n, develop a fast algorithm to comput d_λ, the number of standard tableau with shape λ. As a function of the number of parts and the largest part, how many operations does your algorithm use?

39. **[2]** The *fence poset*, $\mathbf{F}_n$ on the set $[n]$ has cover relations $(i-1, i), (i+1, i)$ for odd i. How many linear extensions does $\mathbf{F}_n$ have? How many ideals does $\mathbf{F}_n$ have?

40. **[1−]** Show that a bipartite Hamiltonian graph must have partite sets of equal size.

41. **[2]** Prove the following two sums for $n > 1$.

$$\sum_{d \backslash n} \phi(d) = n \text{ and } \sum_{d \backslash n} \mu(d) = 0$$

42. **[2]** Prove the following generalization of Möbius inversion: Let f and g be functions from t-tuples of positive integers. Then

$$f(n_1, n_2, \ldots, n_t) = \sum_{d \mid \gcd(n_1, n_2, \ldots, n_t)} g(\frac{n_1}{d}, \frac{n_2}{d}, \ldots, \frac{n_t}{d})$$

if and only if

$$g(n_1, n_2, \ldots, n_t) = \sum_{d \mid \gcd(n_1, n_2, \ldots, n_t)} \mu(d) f(\frac{n_1}{d}, \frac{n_2}{d}, \ldots, \frac{n_t}{d}).$$

43. **[2]** Prove Lemma 2.2

44. **[*R*+]** Show that in any union closed collection of sets that there is some element that occurs in at least half of the sets. This problem is due to Frankl.

45. **[1+]** Show that the antimatroid graph is connected. The antimatroid graph has vertices that are the sets of the antimatroid, and two sets are adjacent if they differ by only one element.

46. **[1+]** Modify the dfs Algorithm 2.1 so that it outputs a topological sorting of a graph whose vertices are labelled $1, 2, \ldots, n$.

47. **[2]** Imagine an infinite deck of cards $1, 2, 3, \ldots$. At step n, pick up the top n cards and interlace them with the next n cards. For example, after step 2 we have $3, 2, 4, 1, 5, 6, 7, \ldots$. It is conjectured that eventually every number becomes the top card of the deck. It can take a long time before a card gets to the top. Develop as efficient a method as you can to determine the step at which card n gets to the top. At what step does card 54 get to the top?

48. **[2]** Find a graph that is edge-transitive but is not vertex-transitive. Find a graph that is vertex-transitive but is not edge-transitive.

2.15 Bibliographic Remarks

The table known as Pascal's triangle actually predates Pascal, having appeared in a 14th century Chinese text. A 12th century Hindu text contains an explanation of the binomial coefficients. Edwards [98] has written a book devoted entirely to Pascal's triangle.

Many further references and interesting commentary on the Bell and Catalan numbers may be found in Gould [158]. An excellent discussion of Fibonacci numbers may be found in Knuth [218].

For more on the cycle lemma see Dershowitz and Zaks [81], Ehrenfeucht, Haemer, and Haussler [99], and Graham, Knuth, and Patashnik [159].

Theorem 2.2 is often attributed to Cayley, but Cayley attributes it to Borchardt. See Moon [285] for further bibliographic remarks, and a wealth of other information about labelled trees.

Further material on antimatroids may be found in the book of Korte, Lovász, and Schrader [231] and in the survey chapter of Björner and Ziegler [34].

Chapter 3

Backtracking

3.1 Introduction

Backtracking is a very general technique that can be used to solve a wide variety of problems in combinatorial enumeration. Many of the algorithms to be found in succeeding chapters are backtracking in various guises. It also forms the basis for other useful techniques such as branch-and-bound and alpha-beta pruning, which find wide application in operations research (and, more generally, discrete optimization) and artificial intelligence, respectively.

As an introductory problem, consider the puzzle of trying to determine all ways to place n non-taking queens on an n by n chessboard. Recall that, in the game of chess, the queen attacks any piece that is in the same row, column or either diagonal. To gain insight, we first consider the specific value of $n = 4$. We start with an empty chessboard and try to build up our solution column by column, starting from the left. We can keep track of the approaches that we explored so far by maintaining a *backtracking tree* whose root is the empty board and where each level of the tree corresponds to a column on the board; i.e., to the number of queens that have been places so far. Figure 3.1 shows the backtracking tree.

This tree is constructed as follows: Starting from an empty board, try to place a queen in column one. There are four positions for the queen, which correspond to the four children of the root. In column one, we first try the queen in row 1, then row 2, etc., from left to right in the tree. After successfully placing a queen in some column we then proceed to the next column and recursively try again. If we get stuck at some column k, then we *backtrack* to the previous column $k - 1$, where we again try to advance the queen in column $k - 1$ from its current position. Observe that the tree is constructed in preorder. All nodes at level n represent solutions; for $n = 4$ there are two solutions.

Let us consider now the case of a standard 8 by 8 chessboard. Here the problem is too large to do the backtracking by hand, unless you are extremely patient. It is easier to write a program.

The backtracking process, as described above, is recursive, so it is not surprising that we can use a recursive procedure to solve the eight queens problem. A pseudo-Pascal procedure for doing this is developed next. The solution x consists of a permutation of 1 through 8, representing the row locations of the queens in successive columns. For the 4-queens problem the permutations giving solutions were $x = [2, 4, 1, 3]$ and $x = [3, 1, 4, 2]$. As the algorithm is proceeding we need some way to determine whether a queen is being attacked by another queen. The easiest way to do this is to maintain three boolean arrays, call them `a, b, c`. The array `a` indicates if a row does not contain a queen. The array

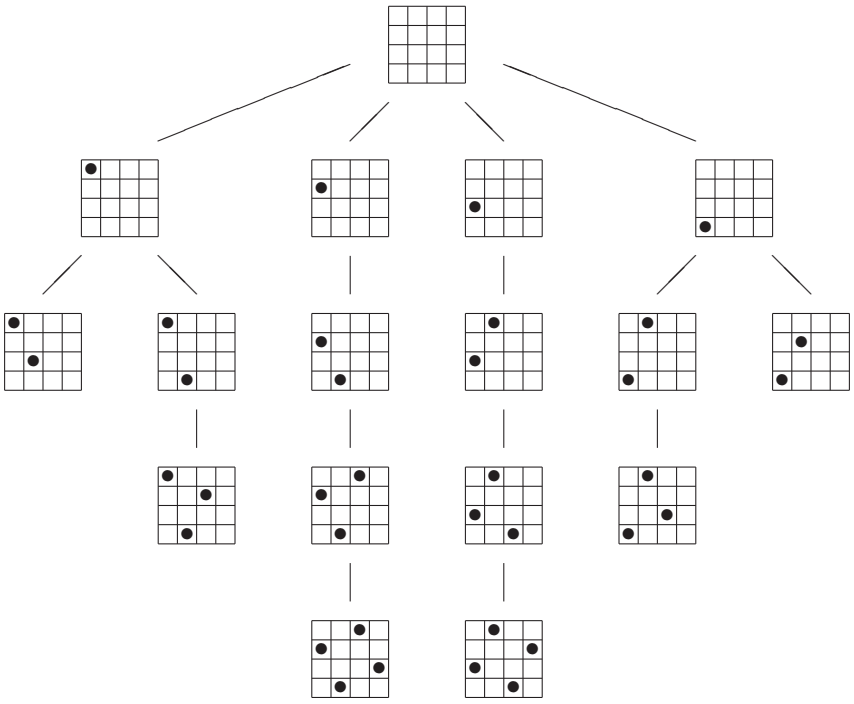


Figure 3.1: The four queens backtracking tree.

`b` indicates if a diagonal does not contain a queen, and `c` indicates if a diagonal does not contain a queen. The sum of the row and column indices is constant along diagonals, and the difference of the row and column indices is constant along diagonals. Thus `a` is indexed `1..8`, array `b` is indexed `2..16`, and `c` is indexed `-7..7`.[1] These arrays are initialized to be `true`, and then we call `Queen( 1 )`. The solution is given in the global array `x`, which need not be initialized.

```
(Q1)  procedure Queen ( col : ℕ )
(Q2)  local row : ℕ;
(Q3)      for row := 1 to 8 do
(Q4)          if a[row] and b[row+col] and c[row−col] then
(Q5)              x[col] := row;
(Q6)              a[row] := b[row+col] := c[row−col] := false;
(Q7)              if col < 8 then Queen( col + 1 ) else PrintIt;
(Q8)              a[row] := b[row+col] := c[row−col] := true;
```

Algorithm 3.1: Algorithm for the 8 queens problem.

It turns out that there are 92 solutions to the 8 by 8 puzzle. Only 12 of the solutions are non-isomorphic in the sense that all other solutions may be obtained from these 12 by rotating and/or flipping the board.

[1] In other languages, the indexing of `c` may have to be offset.

3.2 Backtracking Algorithms

In the general backtracking scenario we wish to generate all strings S that satisfy some predicate P.

$$S = \{(x_1x_2x_3\cdots) \in A_1 \times A_2 \times A_3 \times \cdots | P(x_1x_2x_3\cdots)\}$$

Each A_i is a finite set. Of course, the strings must be of finite length and there must be finitely many of them. In most applications the strings will have a fixed length. In order to apply the backtracking technique the predicate P must be extended to a predicate Q defined on "prefix" strings such that the following properties hold:

$$\begin{array}{l} P(x_1x_2x_3\cdots) \text{ implies } Q(x_1x_2x_3\cdots), \text{ and} \\ \neg Q(x_1x_2\cdots x_{k-1}) \text{ implies } \neg Q(x_1x_2\cdots x_{k-1}x) \text{ for all } x \in A_k \end{array} \tag{3.1}$$

In other words, if a string does not satisfy the predicate Q then it cannot be extended to a string that does satisfy the predicate P. Given a partial solution $\mathbf{x} = x_1x_2\cdots x_{k-1}$ we let $S_k(\mathbf{x})$ denote all the valid ways of extending the string by one element. If the string $\mathbf{x}$ is understood we simply write S_k. In other words,

$$S_k(\mathbf{x}) = \{x \in A_k \mid Q(\mathbf{x}x)\}.$$

The vertices of the *backtracking tree* are all those strings that satisfy Q. The children of a vertex are all those sequences obtained from the parent sequence by appending a single element.

As an illustration of the preceeding definitions consider the 8 by 8 non-taking queens problem. The sets A_i are {1,2,...,8} for each $i = 1, 2, \ldots, 8$. The predicate Q is simply whether the queens placed so far are non-taking. An example in which the sets A_i differ for different i will be given in the next section.

A recursive procedure for the general backtracking scenario is deceptively simple. It is given in Algorithm 3.2. The initial call is $\mathsf{Back}(1, \varepsilon)$; no initialization is necessary. The call $\mathsf{Back}(k, \mathbf{x})$ generates all strings of the form $\mathbf{xy}$ for which $\mathbf{xy} \in S$.

The parameter $\mathbf{x}$, representing a partial solution, is often left as a global variable, as it was in our procedure for the 8 queens problem. Another property that the 8 queens solutions have is that they are all strings of the same length. This is usually the case and results in a simplification of the algorithm. Suppose that all solutions have length n. Then Back can be modified by making lines (B5) and (B6) into the **else** clause of the **if** statement at line (B4). If, in addition, $Q(x_1x_2\cdots x_n)$ implies $P(x_1x_2\cdots x_n)$, then the test for $P(\mathbf{x})$ at line (B4) can be replace by the test "$k > n$". This is done in many of the procedures to be found in succeeding chapters.

```
(B1)   procedure Back ( k : ℕ; x :  string);
(B2)   local x : A_k;   S_k : set of A_k
(B3)   begin
(B4)       if P(x) then PrintSolution( x );
(B5)       compute S_k;
(B6)       for x ∈ S_k  do Back( k + 1, xx );
(B7)   end {of Back};
```

Algorithm 3.2: Recursive backtracking algorithm.

We also present a non-recursive version of the algorithm; see Algorithm 3.3. This version is of interest when speed is of utmost concern — for example, if the backtracking routine is being written in assembly language.

```
k := 1;
compute S_1;
while k > 0 do
    while S_k ≠ ∅ do
        {Advance to next position}
        x_k := an element of S_k;
        S_k := S_k \ {x_k};
        if P(x_1 x_2 ··· x_k) then PrintSolution;
        k := k + 1;
        compute S_k;
    {Backtrack to previous position}
    k := k − 1;
end;
```

Algorithm 3.3: Non-recursive backtracking algorithm.

3.3 Solving Pentomino Problems with Backtracking.

As a somewhat more complicated problem we consider a *pentomino* problem. A pentomino is an arrangement of five unit squares joined along their edges. They were popularized by Golomb [154] [2]. There are 12 non-isomorphic pentominoes as shown in Figure 3.2. The pieces have been given and arbitrary numbering; traditionally letters have also been associated with the pieces and we show those as well. We will develop an algorithm to find all ways of placing the 12 pentomino pieces into a 6 by 10 rectangle. It turns out that there are 2,339 non-isomorphic ways to do this. One of the solutions is shown in Figure 3.3. Each piece can be rotated or flipped, which can give different *orientations* of a piece. Piece 1 has 1 orientation, piece 2 has two orientations, pieces 3,4,5,6,7 have 4 orientations, and pieces 8,9,10,11,12 have 8 orientations. Each solution to the problem is a member of an equivalence class of 4 other solutions, obtained by flipping and rotating the 6 by 10 board.

Figure 3.4 shows how the board is numbered. It also shows a particular placement of the "X" piece. This placement is represented by the set [8,13,14,15,20]. Every placement of a piece is represented by some 5-set. The *anchor* of a placement is the smallest number in the 5-set.

The backtracking algorithm successively tries to place a piece in the lowest numbered unoccupied square; call it k. Once k is determined we try to place an unused piece over it. The pieces are tried in the order $1, 2, \ldots, 12$ as numbered in Figure 3.2. Thus we wish to know, for each piece and number k, all placements that are anchored at k. These placements are precomputed and stored in the array `List` from the type and variable declarations of Figure 3.4. For example, the placement of "X" in Figure 3.4 is the only set in `List[1,8]`. Of course, the list usually contains more than one element; `List[12,8]` has eight members,

[2] The term *pentomino*, for a 5-omino, is a registered trademark of Solomon W. Golomb (No. 1008964, U.S. Patent Office, April 15, 1975).

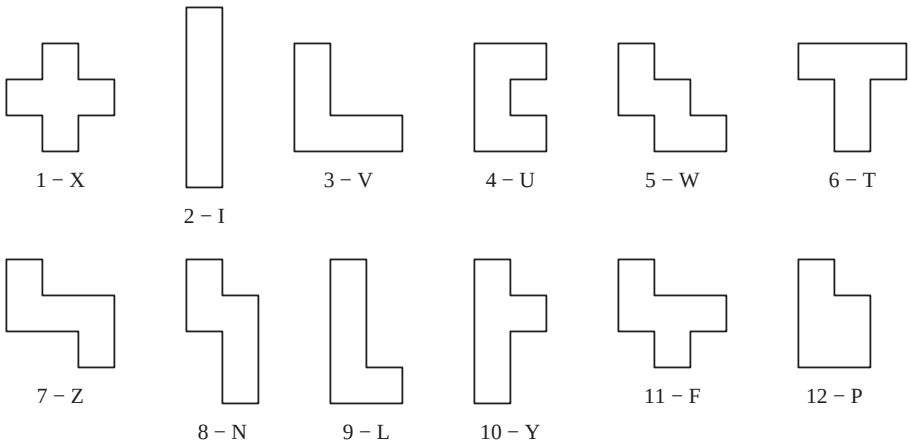


Figure 3.2: The 12 pentomino pieces.

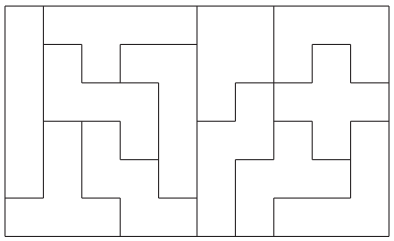


Figure 3.3: A solution to the 6 by 10 pentomino problem.

0	6	12	18	24	30	36	42	48	54
1	7	13	19	25	31	37	43	49	55
2	8	14	20	26	32	38	44	50	56
3	9	15	21	27	33	39	45	51	57
4	10	16	22	28	34	40	46	52	58
5	11	17	23	29	35	41	47	53	59

Figure 3.4: Numbering the board (and the placement of a piece).

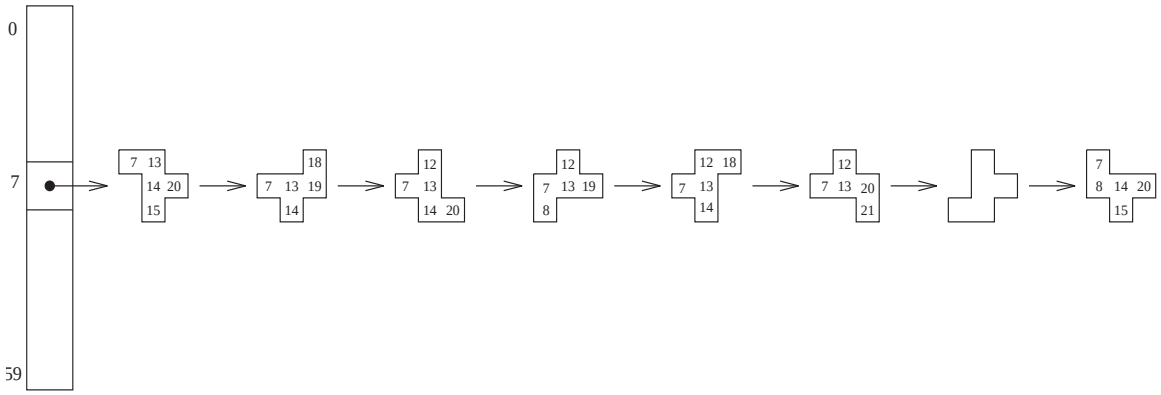


Figure 3.5: Lists for piece 11, with those anchored at location 7 shown explicitly.

the maximum possible number. The list for `List[11,7]` is shown in Figure 3.5. Piece 7 has 8 orientations but only 7 of them appear on the list because one of them would extend outside of the board. The one not appearing is blank in Figure 3.5. The others show the 5 numbers that are in the set correspoding to piece 11 anchored at position 7.

```
type
    PieceNumber = ℤ(1 ... 12);
    BoardNumber = ℤ(0 ... 63);
    Board = word(BoardNumber);
    ListPtr = pointer to ListElement;
    ListElement = node(Position : Board,Link : ListPtr);
global
    TheBoard : Board;
    List : array [PieceNumber,BoardNumber] of ListPtr;
    PieceAvail : word(PieceNumber);
    Solution : array[PieceNumber] of ListPtr;
```

Algorithm 3.4: Declarations for pentomino program.

With these global declarations the backtracking procedure is given in Figure 3.5. Intersection ($\cap$) is used to determine whether a piece can be placed without overlap, union ($\cup$) is used to add a new piece to the board, and difference ($\setminus$) is used to remove a piece from the board. If the set operations are correctly implemented using AND, OR, and NOT on machine words, then this program will be quite fast.

It is natural to wonder whether the backtracking can be sped up by checking for *isolated squares*. That is to say, checking whether the unused portion of the board has a number of squares divisible by five before trying to place the next piece. This strategy will certainly cut down on the number of vertices in the backtracking tree, but now more work is done at each vertex. In the author's experience it does not pay to check for isolated squares; the extra work done at each vertex is greater than the savings obtained by having a smaller tree. Backtracking is full of such tradeoffs between the size of the tree and the amount of work done at each vertex.

The more general question that this raises is: Among two predicates Q and R both satisfying (3.1), which is better? Of course the ultimate test is in the running time of the

```
procedure BackTrack ( k : BoardNumber );
local pc : PieceNumber;
    while k ∈ TheBoard do k := k + 1;
    for pc := 1 to 12 do
        if pc in PieceAvail then
            PieceAvail := PieceAvail \ [pc];
            Solution[pc] := List[pc,k];
            while Solution[pc] ≠ null do
                if TheBoard ∩ Solution[pc].Position = ∅ then
                    TheBoard := TheBoard ∪ Solution[pc].Position;
                    if PieceAvail = ∅
                        then PrintSolution
                        else BackTrack( k + 1 );
                    TheBoard := TheBoard \ Solution[pc].Position;
                Solution[pc] := Solution[pc].Link;
            PieceAvail := PieceAvail ∪ [pc];
end {of BackTrack}
```

Algorithm 3.5: Backtracking routine for pentomino problem.

two algorithms that arise. Two extremes are worth noting. If Q is always true then (3.1) is satisfied. No pruning of the backtracking is done; all of $A_1 \times A_2 \times \cdots$ is generated. What if Q is perfect? That is, what if $Q(\mathbf{x})$ implies that there is some $\mathbf{y}$ such that $P(\mathbf{xy})$? Then every leaf of the backtracking tree is in S. As mentioned in Chapter 1, this is the BEST propery: Backtracking Ensuring Success at Terminals.

3.3.1 Eliminating Isomorphic Solutions

Reject isomorphs! The is no reason to generate isomorphic solutions when generating all solutions to pentomino and other "space-filling" puzzles. In particular, for the 6 by 10 pentomino puzzle, each solution has four equivalent solutions (including itself) in the equivalence class obtained by rotating and flipping. Thus, even if we wanted all solutions, isomorphic or not, we could generate the non-isomorphic ones and then flip and rotate to get the others. As we will see, there is absolutely no computational overhead in rejecting isomorphic solutions in the 6 by 10 case, so that we save a factor of 4 in the running time.

In general, the basic idea is to fix the positions and orientations of some selected piece (or pieces). Two properties are necessary: (a) Each equivalence class must have a member with the selected piece in the one of the fixed positions and orientations, and (b) each of the symmetries of the board must move the piece out of the set of fixed positions and orientations.

For the 6 by 10 pentomino problem we will fix the center square of the "X" shaped piece so that it lies in the upper quadrant (thinking of the board as being centered at its center). See Figure 3.6.

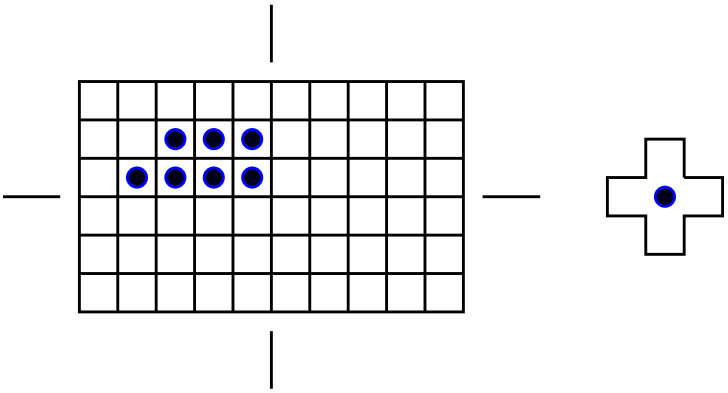


Figure 3.6: Eliminating isomorphs in the 6 by 10 pentomino puzzle.

3.4 Estimating the Running Time of Backtracking.

In many instances it is useful to obtain an estimate of the number of vertices that will occur in a backtracking tree, before the actual algorithm is run. The estimate presented in this section only applies in those instances when we are searching for all possible solutions (i.e. the entire tree is being examined).

The basic idea is to run an experiment in which we follow a random path in the tree from the root to a leaf, and then assume that the entire tree has a similar "parent" - "number of children relationship" as the vertices on this path. The random path is chosen by successively picking a random child of the current vertex to be included next in the path. Each child is regarded as being equally likely. For example, consider the tree shown in Figure 3.7(a) where the thickened edges indicates the random path.

In the experiment the root had three children, the root's child on the thickened path had two children, and so on. Thus we assume that the root has three children, all children of the root have two children, etc. This gives rise to the tree of Figure 3.7(b). This tree has $1 + 3 + 3 \cdot 2 + 3 \cdot 2 \cdot 1 + 3 \cdot 2 \cdot 1 \cdot 2 + 3 \cdot 2 \cdot 1 \cdot 2 \cdot 1 = 40$ vertices. In general, let n_k denote the degree of the $(k-1)$st vertex along the experimental path. There are $n_1 n_2 \cdots n_k$ vertices at level k in the assumed tree. Thus the estimate is the (finite) sum

$$X = 1 + n_1 + n_1 n_2 + n_1 n_2 n_3 + \cdots .$$

A procedure to compute the estimate is given in Algorithm 3.6. A recursive version can also be easily derived.

For any probabilistic estimate a desirable property is that the expected value of the estimate is equal to the quantity being estimated. In probability theory such an estimator

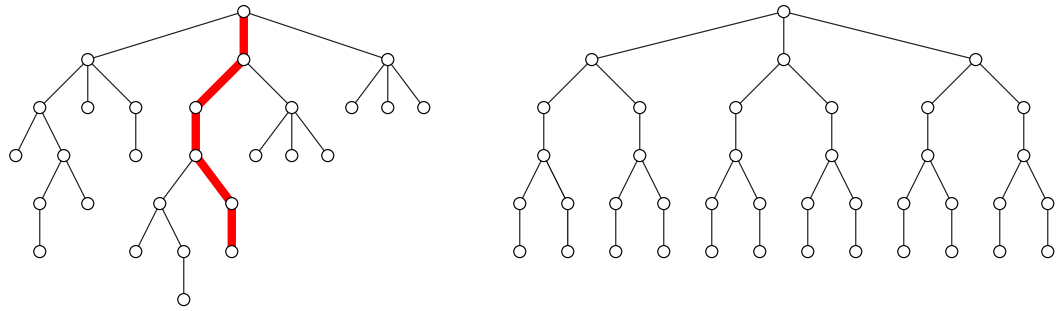


Figure 3.7: (a) True backtracking tree with random root-to-leaf path shown. (b) Assumed tree from that random path.

estimate := *product* := 1;
$k := 1$;
compute S_1;
while $S_k \neq \emptyset$ **do**
    {Advance}
    $n_k := |S_k|$;
    product := $n_k \cdot$ *product*;
    estimate := *estimate* + *product*;
    $x_k :=$ an element of S_k, chosen at random;
    $k := k + 1$;
    compute S_k;

Algorithm 3.6: Estimation algorithm.

is referred to as being *unbiased*. Let T be the tree whose size $|T|$ is being estimated. Intuitively, the reason that the estimate is unbiased is that the probability of reaching a vertex x in the tree whose ancestors have degrees $n_1, n_2, \ldots, n_t$ is equal to $1/(n_1 n_2 \cdots n_t)$, which is the inverse of the weight assigned to that vertex by X. We will now give a more formal argument. Define two functions on the vertices v of the backtracking tree:

$$\tau(v) = \begin{cases} 1 & \text{if } v \text{ is the root} \\ deg(\bar{v}) \cdot \tau(\bar{v}) & \text{if } v \text{ has parent } \bar{v} \end{cases}$$

and

$$I(v) = [\![\text{vertex } v \text{ is visited in the experiment}]\!].$$

Recall the $[\![]\!]$ notation, defined the previous chapter: $[\![P]\!]$ is 1 if P is true and is 0 if P is false. Note that τ depends only on the tree and not the experiment. The random variable X may be rewritten as

$$X = \sum_{v \in T} \tau(v) \cdot I(v).$$

Now take the expected value of the random variable X and use linearity of expectation to obtain

$$\begin{aligned} E(X) &= \sum_{v \in T} \tau(v) \cdot E(I(v)) = \sum_{v \in T} \tau(v) \cdot \frac{1}{\tau(v)} \\ &= \sum_{v \in T} 1 = |T|. \end{aligned}$$

The estimator is therefore unbiased.

Some care should be exercised in applying this estimate. Backtracking trees tend to vary wildly in their structure and subtrees with many vertices may be well hidden in the sense that they are accessed only through paths from the root with vertices of low degree. It is essential in applying the test that a large number of trials are used. Once this is done, however, the test is remarkably effective.

3.5 Exercises.

1. [**1+**] Use backtracking by hand to determine the number of solutions to the 5 by 5 queens problem. How many of the solutions are non-isomorphic?

2. [**1**] Generate an estimate of the number of vertices in the backtracking tree for the 8 by 8 queens problem. In picking your "random" row positions, simply use the lowest numbered valid row.

3. [**1+**] Show that the number of solutions to the n by n Queens problem is less than or equal to $n(n-4) \cdot (n-2)!$. Can you derive a better bound?

4. The following 150 characters of C code outputs the number of solutions to the n-Queens problem. Explain how the code works. What's the largest value of n for which it will produce a correct answer (after a potentially very long wait) on your machine?

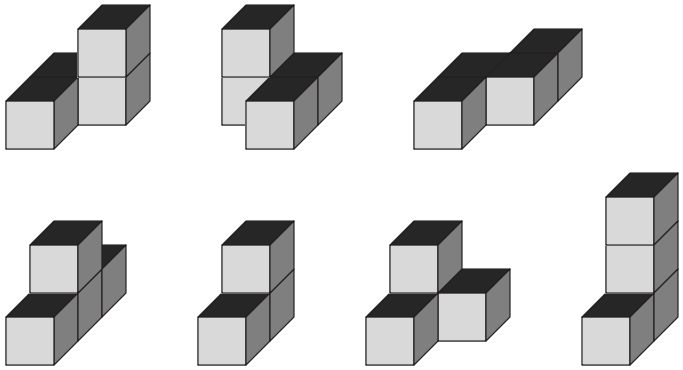


Figure 3.8: The Soma cube pieces.

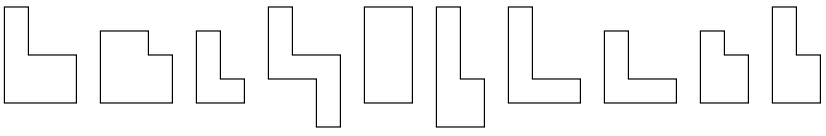
```
t(a,b,c){int d=0,e=a&~b&~c,f=1;if(a)for(f=0;d=(e&=~d)&-e;f+=t(a&~d,(b|d)<<1,
(c|d)>>1));return f;}main(q){scanf("%d",&q);printf("%d\n",t(~(~0<<q),0,0));}
```

5. [**2**] Write a backtracking program to determine a configuration containing the smallest number of queens on a n by n board so that every square of the board is under attack. Is the problem easier if the queens are specified to be non-taking? Give explicit solutions for $1 \leq n \leq 8$.

6. [**1+**] How many hexominoes (6-ominoes) are there? [**2**] Prove that the set of all hexominoes cannot be placed in a rectangular configuration without holes.

7. [**3−**] Define the *order* of a polyomino to be the smallest number of copies of P that will fit perfectly into a rectangle, where rotations and reflections of P are allowed. The figure below shows that the pentomino  has order at most 10. Show that its order is exactly 10. What are the orders of the other two polyomino shapes shown below?

8. [**2**] A polyomino puzzle with the following pieces has been marketed.



The puzzle is to place the pieces on an 8 by 8 board. Write a backtracking program to determine the number of different non-isomorphic solutions.

9. [**2**] The Soma Cube puzzle is to fit the seven pieces listed in Figure 3.8 into a 3 by 3 by 3 cube. Write a backtracking program to generate all 240 non-isomorphic solutions to the Soma Cube puzzle.

10. [**2**] Snake-in-a-cube puzzle: A snake consists of 27 unit cubes arranged in order. They are held together by a shock-cord, but can be rotated around the cord. Each unit cube is either "straight-through" (the cord passed through one face of the cube and exits

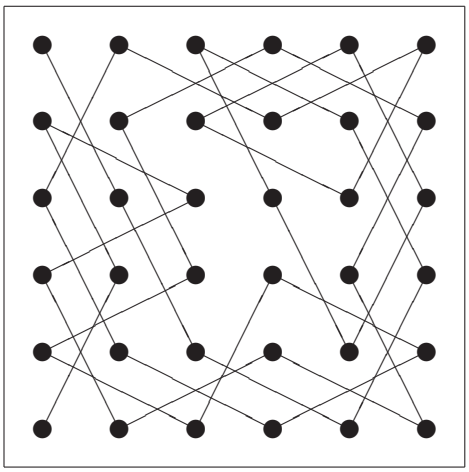


Figure 3.9: A 6 by 6 knight's tour.

throught the opposite face) or is an "elbow" (the cord passed through one face and exists through another perpendicular face). The problem is to find all way of shaping the snake into a 3x3x3 cube. A specific instance of the snake has been marketed and has the following 27 cubes SSESESESEEEESESEEESEESEEESS, where S indicates straight-through, and E indicates elbow. Determine the number of non-isomorphic solutions of this puzzle, and of the puzzle where every unit cube is an elbow.

11. **[2]** A knight's tour of an n by n chessboard is a sequence of moves of the knight on the chessboard so that each of the n^2 squares is visited exactly once. Shown in Figure 3.9 is a knight's tour of a 6 by 6 chessboard. Write a backtracking program to find a single knight's tour on an 8 by 8 chessboard starting from the upper left corner. A knight's tour that returns to the square upon which it started is said to be *re-entrant*. Modify your program from so that it finds a re-entrant knight's tour. Estimate the time that it would take to explore the entire backtracking tree assuming that it takes a millisecond to process one vertex in the tree. Prove that no Knight's tour can exist if n is odd.

12. **[1+]** Draw a graph representing the possible moves of a knight on a 4 by 4 chessboard. Use this graph to prove that there is no knight's tour on a 4 by 4 board.

13. **[2]** In a *noncrossing Knight's tour* the lines drawn for each move of the knight do not cross. The tour shown in Figure 3.9 has many crossings, and in general, a non-crossing tour cannot visit every square of the board. Determine the longest possible non-crossing knight's tour on an 8 by 8 board.

14. **[2]** Write a backtracking program to determine the number of ways to color the vertices of a graph with k colors for $k = 1, 2, \ldots, n$ where n is the number of vertices in the graph. In any such coloring adjacent vertices must receive distinct colors. Deciding whether a graph can be colored with k colors is a well-known NP-complete problem.

15. **[2]** Write a backtracking program to count the number of topological sortings (linear extensions) of a directed acyclic graph (partially ordered set).

16. **[2]** Write a backtracking program to list all Hamilton cycles (if any) in a graph. Use your program to determine the number of Hamilton cycles in the tesseract.

17. **[2]** For given k, a subset $K \subseteq [k]$, is said to *represent* k if there is some unique $I \subseteq K$ such that $\sum_{x \in I} x = k$. For given n, a pair of subsets $P, Q \subseteq [n]$ is said to *dichotomize* $[n]$ if, for each $k \in [n]$, the number k is representable by P or by Q, but not by both. For example $\{1, 2, 6\}, \{4, 5\}$ dichotomizes [9], but [12] is not dichotomized by $\{1, 2, 9\}, \{3, 4, 5\}$ since 3 and 9 are represented twice and 6 is not represented. Write a backtracking program that takes as input n and outputs all pairs of subsets of $[n]$ that dichotomize it. How many solutions are there for $n = 17$?

18. **[3]** Let $\tau = (1\ 2)$ and $\sigma = (1\ 2\ \cdots\ n)$. Write a backtracking program to find a Hamilton cycle in the directed Cayley graph $Cay(S_n : \{\sigma, \tau\})$ when $n = 5$. Can you find a Hamilton path for $n = 6$? **[*R*−]** Find a Hamilton cycle for $n = 7$. It is known that there is no Hamilton cycle for $n = 6$.

19. There is a large family of combinatorial questions involving the packing of squares into squares. **[2+]** For $n \leq 13$ determine the least number of smaller squares that will tile a n by n square. **[2+]** The sum $1^2 + 2^2 + \cdots + 24^2 = 70^2$ suggests that it might be possible to tile a 70 by 70 square with squares of sizes $1, 2, \ldots, 24$. Show that this is impossible.

20. **[1]** Write Algorithm 3.6 as a recursive procedure.

21. **[3]** Extend the unbiased procedure for estimating the size of a backtracking tree to that of estimating the size of a directed acyclic graph that is rooted at r (all vertices are reachable from r).

3.6 Bibliographic Remarks

Perhaps the first description of the general backtracking technique is in Walker [435]. Early papers about backtracking include Golomb and Baumert [156].

The eight queens problem is attributed to Nauck (1850) by Schuh [375]. It is said that Gauss worked on the problem and obtained the wrong answer! Sosic and Gu [396] analyze programs for the 8-queens problem.

A paper of Bitner and Reingold [33] discusses a number of interesting ways to speed up backtracking programs and contains some applications to polyomino problems.

A (very ugly) program for generating all solutions to the Soma cube problem may be found in Peter-Orth [306].

The book that really popularized polyominoes, and pentominoes in particular, is "Polyominoes" by Golomb [154]. The book by Martin [264] contains much interesting material about polyominoes and contains further references on the topic. The two volume book by Berlekamp, Conway and Guy [28] on mathematical games contains interesting chapters about the Soma cube (including a map of all solutions!) as well as scattered references to polyominoes.

The method for estimating the size of the backtracking tree comes from Hall and Knuth [168] and further analysis is carried out in Knuth [220]. The efficiency of the method has been improved by Purdom [324]. The estimation method can be extended from rooted trees to directed acyclic graphs; see Pitt [311].

Other useful references for backtracking include Fillmore and Williamson [127].

Chapter 4

Lexicographic Algorithms

4.1 Introduction

A natural order for any list of strings or sequences is lexicographic order. It is easy to visualize and work with, and is surprisingly useful in a variety of contexts. For many combinatorial objects, the fastest known algorithms for listing, ranking and unranking are with respect to lexicographic order. Many algorithms that are not overtly lexicographic have some underlying lexicographic structure.

Lexicographic order is based on the familiar idea as the ordering of words in dictionaries. The only requirement is that the letters that make up the alphabet of the language be ordered. In the definitions below we use $\prec$ to denote the assumed underlying ordering of the symbols of the alphabet and $<$ to denote orderings of strings. We remark again that in most instances the alphabet is the set of natural numbers under the usual numeric ordering $0 \prec 1 \prec 2 \prec \cdots$.

DEFINITION 4.1 *In* lexicographic *(or lex) order* $a_1a_2\cdots a_n <_l b_1b_2\cdots b_m$ *if either*

1. *for some* k, $a_k \prec b_k$ *and* $a_i = b_i$ *for* $i = 1, 2, \ldots, k-1$, *or*
2. $n < m$ *and* $a_i = b_i$ *for* $i = 1, 2, \ldots, n$.

DEFINITION 4.2 *In* reverse lexicographic *(or relex) order* $a_1a_2\cdots a_n <_r b_1b_2\cdots b_m$ *if* $b_1b_2\cdots b_m <_l a_1a_2\cdots a_n$ *in lex order.*

DEFINITION 4.3 *In* co-lexicographic *(or colex) order* $a_1a_2\cdots a_n <_c b_1b_2\cdots b_m$ *if* $a_n\cdots a_2a_1 <_l b_m\cdots b_2b_1$ *in lex order.*

For example, here is the same set of strings listed in each of the three orders:

lex:	12, 13, 134, 21, 224, 3
relex:	3, 224, 21, 134, 13, 12
colex:	21, 12, 3, 13, 224, 134

4.2 Subsets

There are two common ways to represent a subset of an n-set. One common representation is a list of the elements in the subset. However, for the most part the representation that

we use is a bitstring $b_1 b_2 \cdots b_n$ where b_i is one if and only if the ith element is included in the subset; modifying the algorithms to handle the other representation presents no difficulties. Recall that $\Sigma_2^n = \{0,1\}^n$ is the set of all bitstrings of length n. With the bitstring representation, subsets may be generated very simply by counting in binary. This produces a lexicographic ordering of the bitstrings. In general, a bitstring of the form

$$b_1 b_2 \cdots b_{k-1} 011 \cdots 1 \quad \text{becomes}$$
$$b_1 b_2 \cdots b_{k-1} 100 \cdots 0.$$

This process is easily translated into the procedure Next of Algorithm 4.1. The algorithm is memoryless since the only input is the bitstring; the only variable k is local to Next. Is it CAT?

```
(N1)   procedure Next;
(N2)   {Assumes b_0 = 0}
(N3)   local k : ℕ;
(N4)   begin
(N5)       k := n;
(N6)       while b_k = 1 do
(N7)           b_k := 0;   k := k − 1;
(N8)       b_k := 1;
(N9)       if k = 0 then done := true;
(N10)  end{of Next};
```

Algorithm 4.1: Next for subsets by counting.

The algorithm may be analyzed by computing the number of times the test "$b_k = 1$" is executed at line (N6). The test fails 2^n times, once for each bitstring. How many times does the test succeed? Refine the question by asking how many times it succeeds in position i. Such a success has the form $b_1 b_2 \cdots b_{i-1} 11 \cdots 1$ and there are 2^{i-1} bitstrings of that form. Thus, the total number of times the test is executed is

$$2^n + \sum_{i=1}^{n} 2^{i-1} \;=\; \sum_{i=0}^{n} 2^i \;=\; 2^{n+1} - 1.$$

Dividing by 2^n, the amortized number comparison count is $2 - 2^{-n}$ and thus the amortized amount of work per subset is $O(1)$; the algorithm is CAT.

A recursive algorithm is also simple. Observe that all bitstrings with $b_1 = 0$ precede all bitstrings with $b_1 = 1$ and that, given a fixed value of b_1, the suffixes $b_2 \cdots b_n$ must also appear in lexicographic order. This observation leads us to the procedure `Subset` of Algorithm 4.2. Variable n is global and the initial call is `Subset(1)`.

Unlike algorithm Next, procedure `Subset` does no comparisons with the elements of `a`. In this case the appropriate measure of the amount of computation done by the algorithm is the total number of recursive calls, since a constant amount of computation is done for every recursive call.

The recursion tree of the call `Subset(1)` is an extended binary tree with 2^n leaves. Thus the total number of nodes in the recursion tree is $2^{n+1} - 1$ and consequently the algorithm runs in constant amortized time. We could obtain this same result as the solution to the recurrence relation $t_0 = 1$, and $t_n = 1 + 2t_{n-1}$, which is obtained from examination of `Subset`.

```
procedure Subset ( k : ℕ);
begin
    if k > n then PrintIt
    else
        b_k := 0;  Subset( k + 1 );
        b_k := 1;  Subset( k + 1 );
end{of Subset};
```

Algorithm 4.2: Recursive procedure to generate subsets in lexicographic order.

The *rank* function for subsets in lexicographic order is very simple; compute the value of a number written in binary:

$$rank(b_1 b_2 \cdots b_n) \;=\; \sum_{i=1}^{n} b_i 2^{n-i}$$

This clearly gives rise to an algorithm that uses $O(n)$ arithmetic operations. It is not difficult to construct a $O(n)$ unranking algorithm as well by using the well-known algorithm for converting a number into its binary representation. See Exercise 6.

4.3 Combinations

Combinations are among the most useful of combinatorial objects. We will use them to introduce and illustrate some methods that are very useful for the efficient generation of other types of combinatorial objects.

Suppose that we wish to generate all subsets of size k from a set with n elements. Such subsets are commonly referred to as *combinations* or, more specifically, as k-combinations of an n-set. The number of combinations of k objects chosen from n objects is denoted $\binom{n}{k}$. Recall that

$$\binom{n}{k} \;=\; \frac{n!}{k!(n-k)!} \;=\; \frac{(n)_k}{k!}.$$

There are two representations of combinations that are in common use. One representation is the bitstring representation of subsets presented in the previous section with the restriction that now each bitstring has exactly k 1's. Define $\mathbf{B}(n,k)$ to be the set of bitstrings of length n that contain exactly k 1's.

$$\mathbf{B}(n,k) = \{b_1 b_2 \cdots b_n \in \Sigma_2^n \mid b_1 + b_2 + \cdots + b_n = k\} \tag{4.1}$$

Another representation is obtained by listing in increasing order the k elements (from $\{1, 2, \ldots, n\}$) that are selected. Define

$$\mathbf{A}(n,k) = \{a_1 a_2 \cdots a_k \mid 1 \le a_1 < a_2 < \cdots < a_k \le n\}. \tag{4.2}$$

It is quite simple to list these strings in lexicographic order. As an example, in lexicographic order the strings of $\mathbf{A}(5,3)$ are 123, 124, 125, 134, 135, 145, 234, 235, 245, 345. In general, the lexicographically smallest string is $12\cdots k$, the lexicographically largest is $(n-k+1)\cdots(n-1)n$. Algorithm 4.3 produces the lexicographic successor of a string in $\mathbf{A}(n,k)$. As with all lexicographic algorithms, we simply scan from the right for the rightmost index

```
(C1)   procedure Next;
(C2)   {Assumes 0 < k ≤ n and a_0 < 0}
(C3)   local i, j : ℕ;
(C4)   begin
(C5)       j := k;
(C6)       while a_j = n − k + j  do j := j − 1;
(C7)       if j = 0 then done := true;
(C8)       a_j := a_j + 1;
(C9)       for i := j + 1  to k  do a_i := a_{i−1} + 1;
(C10)  end{of Next};
```

Algorithm 4.3: Procedure Next for combinations in lex order($\mathbf{A}(n,k)$).

j whose corresponding element can be increased. In this case it means that we scan for the largest position j for which $a_j < n-k+j$.

The running time of this algorithm is determined by how many times the comparison $a_j = n-k+j$ at line (C6) is made. Note that the for loop at line (C9) is iterated the same number of times as the while loop. If we generate all $\binom{n}{k}$ combinations then the comparison fails once for each combination and is true for the $\binom{n-p}{k-p}$ combinations for which $a_{k-p+1} = n-p+1$ (where $1 \le p \le k$); thus the number of times the comparison is made is

$$\sum_{p=0}^{k} \binom{n-p}{k-p} \;=\; \sum_{q=0}^{k} \binom{n-k+q}{q} \;=\; \binom{n+1}{k}.$$

The first equality is obtained by the change of variable $q = k-p$ and the second equality follows from (2.4). Dividing $\binom{n+1}{k}$ by $\binom{n}{k}$, the amortized number of times that the algorithm executes the comparison is $(n+1)/(n-k+1)$. Thus the generation algorithm runs in constant amortized time if $k \le n/2$. If $k > n/2$ then it is better to generate the complementary combinations with $n-k$ elements or use another algorithm.

The while loop of Algorithm 4.3 can be eliminated by making j a global variable (initialized to k) and observing that the rightmost changeable position is k if $a_k < n$ and is one less than its previous value if $a_k = n$. The result is Algorithm 4.4. Note that this new algorithm is not memoryless, as was Algorithm 4.3. Furthermore, the for loop at (I6) is iterated the same number of times as the for loop at (C9) so the running times of Algorithms 4.3 and 4.4 differ by only a constant factor. Nevertheless, simple observations like this can often be used to make algorithms faster.

```
(I1)   procedure Next;
(I2)   {Assumes 0 < k ≤ n and a_0 < 0; variable j is global.}
(I3)   local i : ℕ;
(I4)   begin
(I5)       a_j := a_j + 1;
(I6)       for i := j + 1  to k  do a_i := a_{i−1} + 1;
(I7)       if j = 0 then done :=  true;
(I8)       if a_k < n then j := k else j := j − 1;
(I9)   end{of Next};
```

Algorithm 4.4: Improved version of Next for combinations in lex order($\mathbf{A}(n,k)$).

For the sake of contrast, we now switch to generating the elements of $\mathbf{B}(n,k)$, the bitstring representation of combinations. A recursive algorithm is easy to construct and is based on (2.2), the classic "Pascal's triangle" recurrence relation for binomial coefficients. Let us first rewrite the recurrence relation so that there are no possible terms with value zero; i.e., transform it into a *positive* recurrence relation. Making the natural assumption that $0 \le k \le n$, a positive recurrence relation is as follows.

$$\binom{n}{k} = \begin{cases} 1 & \text{if } n = 0 \\ \binom{n-1}{0} & \text{if } k = 0 \\ \binom{n-1}{n-1} & \text{if } k = n \\ \binom{n-1}{k} + \binom{n-1}{k-1} & \text{if } 0 < k < n \end{cases} \tag{4.3}$$

Based on (4.3) and maintaining lexicographic order, we are led to the procedure `Combination` of Algorithm 4.5. There are two parameters. The first, `j`, is the position whose value is to be set to 0 or 1. The second, `m`, is the number of 1's that have been placed so far. The initial call is `Combination(1,0)`; no other initialization is necessary.

```
procedure Combination ( j, m : ℕ);
begin
    if j > n then PrintIt
    else
        if k − m < n − j + 1 then
            b_j := 0;  Combination( j + 1, m );
        if m < k then
            b_j := 1; Combination( j + 1, m + 1 );
end{of Combination};
```

Algorithm 4.5: Recursive procedure for generating $\mathbf{B}(n,k)$ in lex order.

Algorithm 4.5 is somewhat unnatural since the subarray being modified moves to the right the deeper the recursion becomes. This forces `n` and `k` to be global (or left unmodified if they are passed as parameters), and the parameters `j` and `m` to not exactly match those of the recurrence relation. These shortcomings are easily rectified by using colex order. Then the parameters and structure of the procedure exactly match the recurrence relation (4.3). See Algorithm 4.6. The initial call is `Colex(n,k)`; no initialization is necessary. Henceforth, we will try to use colex order instead of lex order whenever it results in more natural algorithms. This observation is important enough to be highlighted.

Colex (or RL) Superiority Principle: Try to use colex order (right-to-left array filling) whenever possible. It will make your programs shorter, faster, and more elegant and natural.

In some sense the principle is misnamed as the "colex superiority principle," since the superiority arises from the right-to-left, instead of left-to-right, array filling, rather than being a variant of lex order. However, in this chapter the name makes sense, and in later chapters we will call it the "RL superiority principle."

Figure 4.1 shows the computation tree of Algorithms 4.5 and 4.6 on input $n = 5$ and $k = 2$. For lex order the levels of the tree correspond to the array indexing of `b`; for

```
procedure Colex ( n,k : ℕ);
begin
    if n = 0 then PrintIt
    else
        if k < n then
            b_n := 0;  Colex( n − 1, k );
        if k > 0 then
            b_n := 1;  Colex( n − 1, k − 1 );
end{of Colex};
```

Algorithm 4.6: A recursive procedure for generating $\mathbf{B}(n, k)$ in colex order.

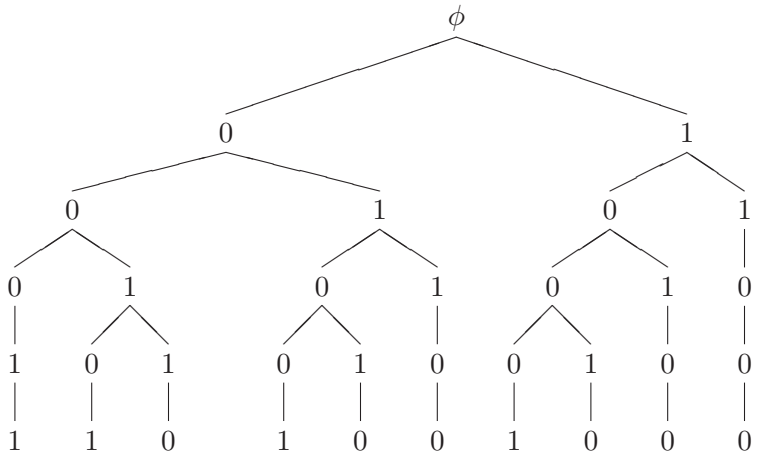


Figure 4.1: Computation tree of $\mathbf{B}(5, 2)$.

colex order tree level i corresponds to array index $5-i+1$. Let $t(n,k)$ denote the number of recursive calls; i.e., the number of nodes in the computation tree of `Colex(n,k)` (or `Combination(1,0)`). The following recurrence relation follows from consideration of Algorithm 4.6.

$$t(n,k) \;=\; \begin{cases} 1 & \text{if } n=0 \\ 1+t(n-1,0) & \text{if } k=0 \\ 1+t(n-1,n-1) & \text{if } k=n \\ 1+t(n-1,k)+t(n-1,k-1) & \text{if } 0<k<n \end{cases} \tag{4.4}$$

The following lemma may be proven by a simple induction (see Exercise 10).

LEMMA 4.1 *For all $0 \le k \le n$, the solution to (4.4) is $t(n,k) = \binom{n+2}{k+1} - 1$.*

Unfortunately,

$$\frac{t(n,k)+1}{\binom{n}{k}} \;=\; \frac{(n+2)(n+1)}{(k+1)(n-k+1)},$$

so the algorithm will not be CAT if k is close to 0 or to n — in those cases there are too many nodes of degree one in the computation tree. The descendants of a node of degree one all have degree one and, furthermore, the descendants of a node of degree one all have the same labels, either all 0's or all 1's. Call these paths of degree one nodes with the same label *0-paths* and *1-paths* (the paths terminate at a leaf). The number of degree one nodes whose children have label 0 is

$$\sum_{i\ge 1} \binom{n-i}{k} \;=\; \binom{n}{k+1}.$$

If calls corresponding to nodes on a 0-path could be eliminated, then the number of calls becomes one less than

$$\binom{n+2}{k+1} - \binom{n}{k+1} = \binom{n+1}{k} + \binom{n}{k},$$

which gives an amortized number of calls $1+(n+1)/(n-k+1)$, the same expression we obtained in analyzing Algorithm 4.3. A similar calculation, but eliminating children of degree one calls labeled 1, gives an amortized number of calls $1+(n+1)/(k+1)$. Fortunately, 0-paths (or 1-paths) can be eliminated by a trick that turns out to be quite useful for generating other objects as well.

The idea for elmininating 0-paths is to stop the recursion at $k=0$ instead of at $n=0$. We still have to ensure that the correct values are in the array `b` whenever `PrintIt` is called. This is done by initializing `b` to be all 0's and by restoring `b[n] := 0` after the call `Colex(n-1,k-1)`. See Algorithm 4.7. Strictly speaking the assignment `b[n] := 0` at the line marked {`*`} is not necessary, although it does obviate the need for initialization.

To illustrate the case of $k \ge n/2$, we revert back to generating $\mathbf{A}(n,k)$. Array `a` is initialized so that `a[i] = i` for $i = 1,2,\ldots,k$. See Algorithm 4.8. Here we stop the recursion at $k=n$.

The technique of eliminating chains of nodes from the computation tree is one of two related tricks that we will use repeatedly. We call them PETs (Path Elimination Techniques). Use PETs to get CAT behavior!

```
procedure Colex( n,k : ℕ);
begin
    if k = 0 then PrintIt
    else
        if k < n then
            {*} b_n := 0;  Colex( n − 1, k );
        b_n := 1;  Colex( n − 1, k − 1 );  b_n := 0;
end{of Colex};
```

Algorithm 4.7: A CAT procedure for generating $\mathbf{B}(n,k)$ in colex order if $k \leq n/2$.

```
procedure Colex ( n,k : ℕ);
begin
    if k = n then PrintIt
    else
        Colex( n − 1, k );
        if k > 0 then
            a_k := n;  Colex( n − 1, k − 1 );  a_k := k;
end{of Colex};
```

Algorithm 4.8: A CAT procedure for generating $\mathbf{A}(n,k)$ in colex order if $k \geq n/2$.

Path Elimination Techniques (PET):

- **#1** Chains in recursive computation trees for generating combinatorial objects are often caused by the values of the parameters reaching a boundary condition. By initializing the array to the values of the string that occur at that boundary condition, and restoring those values as the recursion backs up, all chains caused by that boundary condition can be eliminated from the computation tree.
- **#2** Chains can also be eliminated in many cases by stopping the recursion one (or maybe more) steps earlier than is necessary to get BEST behavior. These one-off boundary conditions often give rise to simple sub-instances of the objects being generated; sub-instances which can be quickly generated by using some other algorithm.

The second technique is implemented with our current example by stopping the recurrence relation when $k = 1$ or $k = n - 1$, and treating these cases using simple for loops. Assume that we are generating $\mathbf{B}(n,k)$. When $k = 1$ we generate the list

$$10^{n-1} \circ 010^{n-2} \circ \cdots \circ 0^{n-1}1$$

and when $k = n - 1$ we generate the list

$$1^{n-1}0 \circ 1^{n-2}01 \circ \cdots \circ 01^{n-1}.$$

These lists are generated with simple for loops, taking time $O(n)$. The resulting algorithm is easily shown to be CAT because now every recursive call has two children or has a child

that satisfies $k = 1$ or $k = n-1$. Thus the number of recursive calls is less than the number of leaves, which is less than $\binom{n}{k}$. The amount of work done at the leaves is proportional to the number of combinations. Exercise 11 asks you to supply the details.

An Application

At this point the reader may feel a little unease with the development as it has been carried out so far. Is it really worth the effort to develop CAT algorithms? Are there really applications where one does not have to look at all of the output and still have a use for a generation algorithm? We present below an application of the ideas developed previously.

A well-known NP-complete problem is the PARTITION problem:

Instance: Positive integers $x_1, x_2, \ldots, x_n$.
Question: Is there a subset $I \subseteq \{1, 2, \ldots, n\}$ such that

$$\sum_{i \in I} x_i \;=\; \sum_{i \notin I} x_i \tag{4.5}$$

We modify this problem by making k part of the instance and in the question insist that $|I| = k$. Call the resulting problem k-PARTITION. Clearly, it is also NP-complete (it is important that k be part of the input) and thus there is no known polynomial time solution. A naïve solution would be to generate combinations and compute the sums (4.5) for each combination generated. The resulting algorithm would be $\Theta(n\binom{n}{k})$. However, we can determine the number of solutions to the k-PARTITION problem in time $O(\binom{n}{k})$ by using our previously developed CAT algorithms for generating combinations. See Algorithm 4.9. We precompute an array `init` where `init[j]` $= \sum_{i=1}^{j} x_i$ and set `totD2 := init[n] div 2`. Both 0-paths and 1-paths have been eliminated and the procedure is thus $O(\binom{n}{k})$ (if a call to PrintIt is given a unit cost) for all values of k. Procedure `PrintIt` needs both the current value of `n` and `k` to actually output the subsets; they are not needed if only a count of the number of solutions is desired.

```
procedure C ( n,k, sum : ℕ);
begin
    if k = n then
        if sum + init[n] = totD2 then PrintIt( n,k );
    else
    if k = 0 then
        if sum = totD2 then PrintIt( n,k );
    else
        b_n := 0;  C( n − 1, k, sum );
        b_n := 1;  C( n − 1, k − 1, sum + val[n] );
end{of C};
```

Algorithm 4.9: A "CAT" algorithm for the k-PARTITION problem.

We can draw two lessons from this example. First, a CAT algorithm is significantly faster than a corresponding algorithm that, say, takes linear time per object generated. If there are $N = 2^n$ objects, then the comparison is between a $\Theta(N \log N)$ algorithm with a

$\Theta(N)$ algorithm. Secondly, in many applications some function needs to be applied to each object as it is generated, and that function need not be computed from scratch for each object. The function can often be updated in the same time that it takes to update the object.

Ranking

In order to develop a lex ranking algorithm let us first consider an example. Suppose that we wish to know the lexicographic rank of 35678 in $\mathbf{C}(9,5)$. Since the ordering is lexicographic we will count the number of combinations that are less than 35678 in lexicographic order. Certainly all combinations beginning with a 1 or a 2 are lexicographically less. There are $\binom{8}{4}$ and $\binom{7}{4}$ such combinations, respectively. Of the combinations that begin with a 3 only the ones that begin 34 are guaranteed to be less than 35678. There are $\binom{5}{3}$ combinations that begin 34. Of the combinations that begin 35, the lexicographically smallest is 35678. Thus the rank is $\binom{8}{4}+\binom{7}{4}+\binom{5}{3} = 70+35+10 = 115$.

Let $R_n(a_1, a_2, \ldots, a_k)$ denote the rank of $a_1 a_2 \cdots a_k$ as a combination of size k taken from n objects. Following our previous discussion, the rank is the number of combinations that begin with an element less than a_1 plus the rank of the remaining string in an altered combination set.

$$R_n(a_1, a_2, \ldots, a_k) = R_{n-a_1}(a_2 - a_1, a_3 - a_1, \ldots, a_k - a_1) + \sum_{i=1}^{a_1-1} \left(\begin{array}{c} n-i \\ k-1 \end{array} \right) \tag{4.6}$$

The terminating condition is $R_n(a_1) = a_1 - 1$. Equation (4.6) can be iterated to obtain (where $a_0 = 0$)

$$\begin{aligned} R_n(a_1, a_2, \ldots, a_k) &= \sum_{j=0}^{k-1} \left[\sum_{i=1}^{a_{j+1}-a_j-1} \binom{n-a_j-i}{k-j-1} \right] && (4.7) \\ &= \sum_{j=0}^{k-1} \left[\binom{n-a_j}{k-j} - \binom{n-a_{j+1}+1}{k-j} \right] && (4.8) \\ &= \binom{n}{k} - 1 - \sum_{j=1}^{k} \binom{n-a_j}{k-j+1}. && (4.9) \end{aligned}$$

Expression (4.8) follows from (4.7) by an application of the identity (2.3). Expression (4.9) follows from (4.8) by breaking the sum into two parts, applying the Pascal's Triangle identity, and cancelling like terms.

Thus, from (4.9), we see that if a table of binomial coefficients is available, then ranking is $O(n)$. Unranking can also be done in $O(n)$ arithmetic operations if a table of binomial coefficients has been pre-computed.

To rank combinations in relex order we clearly obtain

$$Rank_n(a_1, a_2, \ldots, a_k) = \sum_{j=1}^{k} \binom{n-a_j}{k-j+1}.$$

What about colex order? Just as for listing algorithms, colex order can lead to simpler, more natural ranking algorithms. For $\mathbf{A}(n,k)$ a key observation is that, for a fixed value of

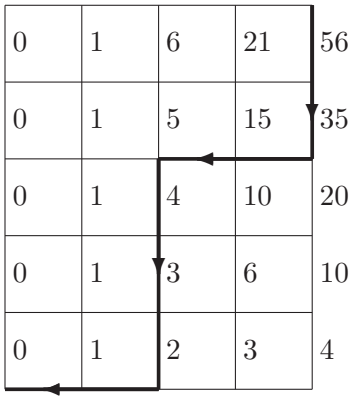


Figure 4.2: The walk corresponding to the rank $100 = 56 + 35 + 4 + 3 + 2$ combination of $\binom{9}{5}$ in colex order.

k, the list for n is the initial $\binom{n}{k}$ strings of the list for $n+1$. Thus the parameter n is not needed for the ranking; one only needs the string $\mathbf{a} = a_1 a_2 \cdots a_k$. Clearly, all combinations whose largest element is smaller than a_k precede $\mathbf{a}$; there are $\binom{a_k - 1}{k}$ of these. We are thus lead to the following equation for the rank.

$$\begin{aligned} RankC(a_1, a_2, \ldots, a_k) &= \binom{a_k - 1}{k} + RankC(a_1, a_2, \ldots, a_{k-1}) \\ &= \sum_{j=1}^{k} \binom{a_j - 1}{j}. \end{aligned} \tag{4.10}$$

There is a very nice way to visualize colex ranking of bitstrings in $\mathbf{B}(n,k)$ in terms of lattice walks on a k by $n-k$ grid. See Figure 4.2 for an example with $k = 6$ and $n = 9$. Regarding the lower left corner as $(0,0)$, the vertical edge from (x,y) to $(x,y+1)$ is labeled $\binom{x+y-1}{y} = \binom{x+y-1}{x-1}$; horizontal edges are unlabeled. The rank of a path from the lower left corner to the upper right corner is the sum of the edge labels encountered. The illustrated path corresponds to the bitstring 001110011 or the element 34589 of $\mathbf{A}(9,5)$.

Keeping the lattice walk interpretation in mind, it is not difficult to derive an unranking algorithm for colex order. See Algorithm 4.10, where the function `C(n,k)` returns $\binom{n}{k}$.

```
procedure Unrank ( r : ℕ);
local i,p : ℕ;
begin
    for i := k downto 1 do
        p := i − 1;
        repeat p := p + 1
        until C(p, i) > r;
        r := r − C(p − 1, i);
        a_i := p;
end{of Unrank};
```

Algorithm 4.10: Unranking combinations ($\mathbf{A}(n,k)$) in colex order.

Computing a table of binomial coefficients with parameters at most n and k requires $\Theta(kn)$ addition operations on rather large integers and this preprocessing dominates the running time of the ranking and unranking algorithms we've presented so far for combinations. For many applications exponentially many rankings or unrankings are required, so this preprocessing is a small part of the total time. However, if only a small number of rankings or unrankings are done then the preprocessing time dominates. The precomputation of the table can be avoided and the binomial coefficients computed on the fly by first computing $\binom{n}{k}$ and then making use of the equations below.

$$\binom{n}{k} = \frac{n}{n-k}\binom{n-1}{k} = \frac{n}{k}\binom{n-1}{k-1}$$

To use these one must now do multiplies and divides (but only integer arithmetic is required). The resulting ranking and unranking algorithms use $\Theta(n)$ arithmetic operations.

4.4 Permutations

Perhaps more algorithms have been developed for generating permutations than any other kind of combinatorial object. For a survey and many references see Sedgewick [377].

Recall that in the n by n queens problem of the previous chapter each solution is a permutation. In fact, that recursive algorithm will recursively generate all permutations in lexicographic order if the references to arrays `b` and `c` are removed. Analyzing the resulting algorithm is the subject of Exercise 21; it is CAT. We now discuss an iterative algorithm.

To iteratively generate permutations lexicographically scan from right to left until an element smaller than its predecessor is found. That element is then minimally increased and the remainder of the permutation is filled in as lexicographically minimal way as possible. For example, 892157643 becomes 892163457. The general procedure is given in Algorithm 4.11.

```
procedure Next;
{Assumes that π_0 = 0.}
begin
    k := n − 1;
    while π_k > π_{k+1} do k := k − 1;
    if k = 0 then done := trueelse
        j := n;
        while π_k > π_j do j := j − 1;
        π_k :=: π_j;
        r := n; s := k + 1;
        while r > s do
            π_r :=: π_s;
            r := r − 1; s := s + 1;
    end;
end{of Next};
```

Algorithm 4.11: Procedure Next for permutations in lexicographic order.

If $n = 4$ then the output of this algorithm is (read across)

1234 1243 1324 1342 1423 1432,
2134 2143 2314 2341 2413 2431,
3124 3142 3214 3241 3412 3421,
4123 4132 4213 4231 4312 4321.

The algorithm operates by transposing pairs of elements and can be analyzed by counting the number of such transpositions. Let t_n denote the number of transpositions used by this algorithm in generating all $n!$ permutations. Then $t_1 = 0$, and if $n > 1$ then

$$t_n = nt_{n-1} + (n-1)\left\lfloor \frac{n+1}{2} \right\rfloor. \tag{4.11}$$

The reasoning behind this recurrence relation is as follows. There are n groups of permutations, each with a fixed value in the first position. Within those groups the number of transpositions is t_{n-1}. At the $n-1$ interfaces between those groups the number of transpositions is $\lfloor (n+1)/2 \rfloor$. The ith interface has the form shown below.

$$\begin{array}{cccccccccc} i & n & n-1 & \cdots & i+2 & i+1 & i-1 & \cdots & 2 & 1 \\ i+1 & 1 & 2 & \cdots & i-1 & i & i+2 & \cdots & n-1 & n \end{array}$$

The recurrence relation (4.11) can be solved by defining

$$s_n = t_n + \left\lfloor \frac{n+1}{2} \right\rfloor.$$

The recurrence relation then becomes

$$s_n = \begin{cases} 1 & \text{if } n = 1 \\ ns_{n-1} & \text{if } n \text{ even} \\ n(s_{n-1}+1) & \text{if } n \text{ odd,} \end{cases}$$

which can be iterated to obtain

$$s_n = n!\left(1 + \frac{1}{2!} + \frac{1}{4!} + \cdots + \frac{1}{(2\lfloor (n-1)/2 \rfloor)!}\right).$$

The "hyperbolic cosine" is defined by $\cosh(x) = \sum_{k\geq 0} 1/(2k)!$. Thus $t_n \sim n! \cdot \cosh 1 \approx 1.5308 \cdot n!$, and so the algorithm runs in constant amortized time; it is CAT. Note that the algorithm is memoryless.

A ranking function R for permutations in lexicographic order is developed next. What, for example, is $R(251436)$? All permutations that start with a 1 precede 251436. There are $1 \cdot 5!$ of them. We then want the rank of 51436 as a lexicographic permutation of $\{1, 3, 4, 5, 6\}$. This is the same as the rank of 41325, obtained by subtracting 1 from each element greater than 2. Thus $R(251436) = 1 \cdot 5! + R(41325)$. Continuing, $R(41325) = 3 \cdot 4! + R(1324)$, $R(1324) = 0 \cdot 3! + R(213)$, and $R(213) = 1 \cdot 2! + R(12)$. Thus, the rank of $R(251436)$ is $1 \cdot 5! + 3 \cdot 4! + 0 \cdot 3! + 1 \cdot 2! = 194$.

In general,

$$R(\pi_1 \pi_2 \cdots \pi_n) = (\pi_1 - 1) \cdot (n-1)! + R(\pi'_1 \cdots \pi'_{n-1}),$$

where π'_{i-1} is π_i if $\pi_1 > \pi_i$, and is $\pi_i - 1$ if $\pi_1 < \pi_i$. A straightforward implementation of this recurrence relation uses $O(n^2)$ arithmetic operations, yielding a moderately efficient ranking algorithm. Similarly, a $O(n^2)$ unranking algorithm will be derived later in this section.

We call the string $(\pi_1 - 1)(\pi'_1 - 1)\cdots$ the *inversion string* string of π, since its sum is the number of inversions in π. Given a permutation π of $\{1, 2, \ldots, n\}$, the *inversion string*, $P[\pi] = p_1 p_2 \cdots p_n$, of π can be defined as follows. Entry p_i is the number of elements π_j that satisfy $\pi_j < \pi_i$ and $j > i$. In other words, p_i is the number of elements that are less than π_i and to the right of it. For example, $P[351642] = 230210$. Inversion strings are useful since they preserve lexicographic order and free us from certain bookkeeping details in the algorithms. Permutation π is lexicographically less than π' if and only if $P[\pi]$ is lexicographically less than $P[\pi']$. Inversion strings are characterized by the property that $0 \le p_i \le n - i$ for $i = 1, 2, \ldots, n$. The rank of π in lex order can be simply expressed as

$$R(\pi) \;=\; p_1(n-1)! + p_2(n-2)! + \cdots + p_n \cdot 0! \;=\; \sum_{i=1}^{n} p_i(n-i)!. \tag{4.12}$$

The process of converting a permutation to an inversion string (or inversion vector or inversion table), or vice-versa, is easily accomplished with an algorithm that uses $O(n^2)$ arithmetic operations. By being somewhat more tricky the number of arithmetic operations can be reduced to $O(n \log n)$. See Exercise 22.

It is straightforward to implement (4.12). The development of an unranking algorithm is also not difficult. Some indexing is simplified by ranking in reverse colex order; see Algorithm 4.12. The running time of this algorithm is $O(n^2)$, which again can be reduced to $O(n \log n)$. Due to the size of the numbers involved, ranking and unranking are typically done only for small values of n and then Algorithm 4.12 is superior to those algorithms that are asymptotically faster. However, there is an $O(n)$ arithmetic operation ranking algorithm for permutations; it does not use lexicographic order and is very fast. See Exercise 23.

```
function Rank ( p : permutation ) : ℕ;
local f,i,j,c,r : ℕ;
begin
    r := 0;
    f := 1;
    for i := 2 to n do
        c := 0;
        for j := 1 to i − 1 do
            if p[j] > p[i] then c := c + 1;
        r := r + c ∗ f;
        f := f ∗ i;   {now f = i!}
    return( r );
end{of Rank};
```

Algorithm 4.12: Ranking algorithm for permutations in reverse colex order.

4.5 Permutations of a Multiset

Recall that a multiset is a set with repeated elements. Throughout this section we assume that the elements of the multiset are $0, 1, \ldots, t$, that the number of occurrences of i is n_i, and that $N = n_0 + n_1 + \cdots + n_t$. A string of length N with n_i occurrences of symbol i is said to be a *multiset permutation* with *specification* $\langle n_0, n_1, \ldots, n_t \rangle$. An algorithm for

generating multiset permutations will generate combinations if $t = 1$ (i.e., the specification is of the form $\langle n_0, n_1 \rangle$) and will generate permutations if all $n_i = 1$ (i.e., the specification is of the form $\langle 1, 1, \ldots, 1 \rangle$), thus unifying the results of the previous two sections.

As an example, in lexicographic order the 12 permutations of specification $\langle 2, 1, 1 \rangle$ are

0012, 0021, 0102, 0120, 0201, 0210, 1002, 1020, 1200, 2001, 2010, 2100.

It is quite straightforward to generate multiset permutations in lex or colex order, using the knowledge gained previously in this chapter. The only challenge is to develop a CAT algorithm. Let us assume that 0 occurs more frequently than any other symbol, $n_0 \geq n_1, n_2, \ldots, n_t$. We first apply PET #1 to develop an algorithm that eliminates all 0-paths and then show that it is CAT.

```
(M1)   procedure GenMult ( n : ℤ(0...N) );
(M2)   local j : ℤ(0...t);
(M3)   begin
(M4)       if n_0 = n then PrintIt( π[1..N] )
(M5)       else
(M6)           for j ∈ {p | n_p > 0}  do
(M7)               π_n := j;
(M8)               n_j := n_j − 1;
(M9)               GenMult( n − 1 );
(M10)              n_j := n_j + 1;
(M11)              π_n := 0;
(M12)  end{ of GenMult}
```

Algorithm 4.13: Generate permutations of a multiset in colex order. Given $\pi[1..n]$ are all zeroes, the call *GenMult*(n) prints all strings $\alpha\pi[n+1..N]$ where α is a multiset permutation of specification $\langle n_0, \ldots, n_t \rangle$.

Our algorithm, GenMult, is shown as Algorithm 4.13. The array π is initialized to 0's and then GenMult(N) is called. The multiset permutation are produced in colex order. Procedure GenMult can be translated directly into a conventional language like Pascal, C, or Java except for the statement

for $j \in \{p \mid n_p > 0\}$ **do** .

A naïve implementation translates this into something like

for $j := 0$ **to** t **do if** $n[j] > 0$ **then do** ,

but the resulting analysis would give an amortized cost of $\Theta(t)$. To eliminate the dependence on t, we could maintain a linked list of those n_j for which $n_j > 0$. The nodes corresponding to n_j are deleted from the list at line (M8) when n_j becomes 0 and are re-inserted at line (M10). Assuming this linked list representation, the running time of the algorithm is proportional to the number of nodes in its computation tree. The computation tree for $\langle 2, 1, 1 \rangle$ is shown in Figure 4.3; this is the colex suffix tree of the set of permutations. Since we are generating in colex order, level i in the tree corresponds to array index $N - i$.

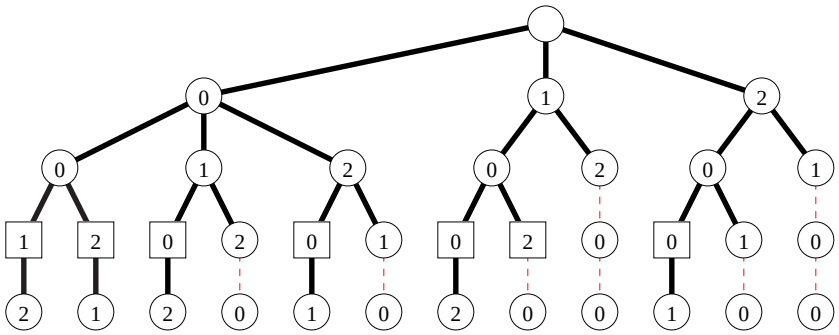


Figure 4.3: Computation tree of *GenMult* on input $\langle 2,1,1\rangle$.

The following two multinomial coefficient identities will prove useful in analyzing this algorithm. The first identity is classic and is simply a generalization of Pascal's triangle (2.2); the second may be verified by induction on n_0.

$$\sum_{i=0}^{t} \binom{N-1}{n_0,\ldots,n_{i-1},n_i-1,n_{i+1},\ldots,n_t} = \binom{N}{n_0,n_1,\ldots,n_t} \tag{4.13}$$

This next identity is a generalization of (2.4).

$$\sum_{i=0}^{n_0} \binom{N-i}{n_0-i,n_1,\ldots,n_t} = \frac{N+1}{n-n_0+1}\binom{N}{n_0,n_1,...,n_t} \tag{4.14}$$

The number of nodes of degree one in the recursion tree T of GenMult is equal to the number of nodes on i-paths in T, where $i>0$. The number of nodes at level $N-j+1$ (with the root at level 0) on i-paths in T is the same as the number of multiset permutations of specification

$$\langle n_0,n_1,\ldots,n_{i-1},n_i-j,n_{i+1},\ldots,n_t\rangle.$$

For the example of Figure 4.3, the number of nodes on 2-paths at level 4 is 3 (corresponding to 001, 010, and 100), there are no 2-paths at level 3, and the total number of degree one nodes is 6 (these are the circled nodes). Thus, the total number of nodes of degree one is given by the following expression.

$$\begin{aligned}
&\sum_{i=1}^{t}\sum_{j=1}^{n_i} \binom{N-j}{n_0,\ldots,n_{i-1},n_i-j,n_{i+1},\ldots,n_t} \\
&= \sum_{i=1}^{t} \frac{N}{N-n_i+1}\binom{N-1}{n_0,\ldots,n_{i-1},n_i-1,n_{i+1},\ldots,n_t} \\
&\le \frac{N}{N-n'+1}\binom{N}{n_0,n_1,...,n_t} \le 2\binom{N}{n_0,n_1,...,n_t},
\end{aligned}$$

where $n'=\max\{n_1,n_2,\ldots,n_t\}$.

All other internal nodes of the tree have degree greater than one, so there cannot be more of them than there are leaf nodes. Thus, the total number of nodes in the tree is at most $4\binom{N}{n_0,\ldots,n_t}$, the algorithm is CAT, independent of both t and N.

4.5.1 Combinations of a Multiset

The k-combinations of an n-set S are obtained by taking all subsets of the set S that have k elements. What if S, instead of being a set, is a multiset with specification, say, $\langle n_0, n_1, \ldots, n_t \rangle$? Such k-subsets of a multiset have several applications, notably in statistics, where they arise in permutation tests with repeated data values and as $2 \times n$ contingency tables.

Let $\mathbf{C}(k; n_0, n_1, \ldots, n_t)$ denote the set of all k-subsets of the multiset consisting of n_i occurrences of i for $i = 0, 1, \ldots, t$. Each combination is itself a multiset. For example, $\mathbf{C}(2; 2, 1, 1) = \{\{0, 0\}, \{0, 1\}, \{0, 2\}, \{1, 2\}\}$. We show that all multiset combinations can be generated by a CAT algorithm, as long as $0 < k < n_0 + \cdots + n_t$. Observe that the k-combinations of the multiset of specification $\langle 1, 1, \ldots, 1 \rangle$ are precisely the k-combinations of a $(t+1)$-set.

Instead of listing the elements of each k-set, we record how many times each element occurs. Thus we will generate all strings $a_0 a_1 \cdots a_t$, where $a_0 + a_1 + \cdots + a_t = k$ and $0 \le a_i \le n_i$ for $i = 0, 1, \ldots, t$. Because of this representation combinations of a multiset are also sometimes called *compositions with restricted parts*. These strings will be listed in colex order. For example, the strings representing $\mathbf{C}(2; 2, 1, 1)$ are listed 200, 110, 101, 011.

Let $C(k; n_0, n_1, \ldots, n_t) = |\mathbf{C}(k; n_0, n_1, \ldots, n_t)|$, and $n = n_0 + n_1 + \cdots + n_t$. If i is the number of times that t occurs in a multiset combination, then clearly we must have $0 \le i \le n_t$, as well as $k - n + n_t \le i \le k$.

Thus, classifying our solutions according to the value of a_t, we obtain the following positive recurrence relation.

$$C(k; n_0, n_1, \ldots, n_t) = \sum_{i=\max(0,k-n+n_t)}^{\min(n_t,k)} C(k-i; n_0, \ldots, n_{t-1}). \tag{4.15}$$

In our discussion the n_i remain fixed, while k and t vary. Thus the $C(k; n_0, \ldots, n_t)$ notation is cumbersome and we shorten it to $C_t(k)$. Hence (4.15) becomes

$$C_t(k) = \sum_{i=\max(0,k-n+n_t)}^{\min(n_t,k)} C_{t-1}(k-i) \tag{4.16}$$

Observe the symmetry $C_t(k) = C_t(n-k)$ and the boundary values $C_t(0) = C_t(n) = 1$.

Our recursive algorithm for listing all k-subsets of a multiset is based on the recurrence relation (4.16). After presenting the algorithm, we show that it is CAT; i.e. that the amount of computation used is proportional to the number of combinations generated.

At each node in the recursion tree at level p, the algorithm decides how many copies of p are included in the subset and array element `a[p]` is set to that number.

The algorithm for listing all k element subsets of a multiset with a total of n elements, $t+1$ of which are distinct, is given in Figures 4.14 and 4.15. The decision to use `gen1` is made if $k \le n/2$, otherwise `gen2` is used. The algorithms take three parameters, `k, t` and `n`.

The degree of a node is the number of different possible values that can be stored in `a[p]`; i.e., the number of terms in (4.16). From the algorithms `gen1` and `gen2` we observe that nodes of degree one in the computation tree occur if $k = n$, or $k = 0$, in the cases of `gen1` and `gen2`, respectively. Unfortunately, there can be many calls of degree one if k is close to 0 or if k is close to n. In the procedure of Figure 4.14, the recursion is terminated when

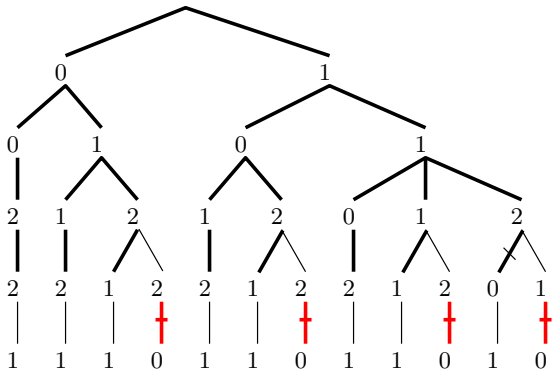


Figure 4.4: Computation tree of gen1 on $\mathbf{C}(5;1,2,2,1,1)$ or of a lex algorithm on $\mathbf{C}(5;1,1,2,2,1)$. Colex order is used in the algorithms, lex order in the analysis. The crossed shaded edges indicate eliminated 0-paths.

$k=0$. The initial call is gen1$(k,0,n)$ and a is initialized so that $a_i=0$ for $i=0,1,\ldots,t$. In the procedure of Figure 4.15, the recursion is terminated when $k=n$. The initial call is also gen2$(k,0,n)$ but a is initialized so that $a_i=n_i$ for $i=0,1,\ldots,t$. The use of the min and max expressions, and the inclusion of the parameter n are necessary to avoid making recursive calls that will not result in a valid subset; i.e., to guarantee that the algorithm is BEST.

```
procedure gen1 ( k, t, n : ℕ);
local i : ℕ;
begin
    if k = 0 then PrintIt else
    for i := max(0, k − n + n_t) to min(n_t, k) do
        a_t := i;
        gen1( k − i, t − 1, n − n_t );
        a_t := 0;
end{ of gen1};
```

Algorithm 4.14: The procedure gen1 (to be used when $k \le n/2$).

In gen1 a call of degree 1 results if $k=n$; after that all descendant calls have degree 1. Similarly, in gen2 a call of degree 1 results if $k=0$; after that all descendant calls have degree 1.

We now analyze the algorithm gen1; the analysis of gen2 is analogous. The notation of the analysis is simplified if we assume that the algorithm produces strings in lex order, rather than colex order; this is simply a matter of indexing. We need the following technical Lemma which is a generalization of the fact that the binomial coefficients $\binom{n}{k}$ are unimodal in k.

LEMMA 4.2 *For fixed t the numbers $C_t(k)$ are unimodal in k.*

PROOF: The numbers $C_t(k)$ are the Whitney numbers (of the second kind) of the finite chain product $(\mathbf{n_0}+\mathbf{1})\times(\mathbf{n_1}+\mathbf{1})\times\cdots\times(\mathbf{n_t}+\mathbf{1})$, where $\mathbf{n_i}$ denotes a chain of i elements.

```
procedure gen2 ( k, t, n : integer );
local i : ℕ;
begin
    if k = n then PrintIt else
    for i := max(0, k − n + n_t) to min(n_t, k) do
        a_t := i;
        gen2( k − i, t − 1, n − n_t );
        a_t := n_t;
end{ of gen2};
```

Algorithm 4.15: The procedure gen2 (to be used when $k > n/2$).

According to Proposition 8.61 of Aigner [3], "the sequence of level numbers of a finite chain product is logarithmically concave." Since any logarithmically concave sequence is unimodal, the proof is complete. □

Let $U_t(k)$ denote the number of calls to `gen1(k,t,n)` that have exactly one child. For example, in Figure 4.4, $U_5(5) = 13$ (and $C_5(5) = 12$).

LEMMA 4.3 *If $k \le n/2$, then $U_t(k) \le 2 \cdot C_t(k)$.*

PROOF: The lemma is clearly true if $k = 0$ or $t = 0$. So let $k > 0$ and $t > 0$; we argue by induction on t.

Every node at the penultimate level has one child (since a_t must be $k-(a_0+\cdots+a_{t-1})$). For a node higher in the tree, say at level p, to have degree one it must be the case that $a_{p+1} = n_{p+1}, \ldots, a_t = n_t$. Thus every non-penultimate node of degree one in the tree for $\mathbf{C}_t(k)$ corresponds to some node of degree one in the tree for $\mathbf{C}_{t-1}(k - n_t)$. For example, in Figure 4.4, the thicker non-crossed edges show the tree for $C_4(5-1)$, and each non-penultimate degree one node in the tree for $C_5(5)$ corresponds to a degree one node in the tree for $C_4(4)$. The number of nodes of degree one that are not at the penultimate level is thus bounded by $U_{t-1}(k - n_t)$. The number of nodes at the penultimate level is $C_t(k)$; the number of degree one nodes at the penultimate level is typically fewer (as in Figure 4.4. Hence we have

$$\begin{aligned} U_t(k) &\le C_t(k) + U_{t-1}(k - n_t) \\ &\le C_t(k) + 2C_{t-1}(k - n_t) \\ &\le C_t(k) + C_{t-1}(k - n_t) + C_{t-1}(k - n_t + 1) \\ &\le 2C_t(k). \end{aligned}$$

Noting that $k \le n/2$ implies $k - n_t \le (n - n_t)/2$, the second inequality follows inductively. The third inequality follows from the symmetry and unimodality of the C_t numbers. The final inequality follows from the recurrence relation (4.16). □

4.6 Trees

Trees play a pivotal role in graph theory and in computer science. There are many types of trees and each type requires a somewhat different approach to generate. We will present lexicographic algorithms for binary trees, ordered trees, rooted trees, free trees, and B-trees. Two Gray codes for binary trees are discussed in the next chapter.

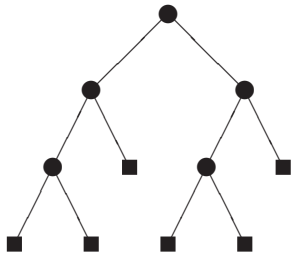


Figure 4.5: Extended binary tree represented by the sequence 1110001100.

10101010	11010010
10101100	11010100
10110010	11011000
10110100	11100010
10111000	11100100
11001010	11101000
11001100	11110000

Figure 4.6: Lexicographic tree sequences for $n = 4$ (read down).

4.6.1 Binary Trees, Ordered Trees

Binary trees are just one of many objects counted by the Catalan numbers. In this section we develop an algorithm for generating all binary trees with n vertices, as represented by certain bitstrings. To represent a binary tree we first convert it to its extended equivalent by adding $n+1$ leaves. Then each internal node is labeled with a 1 and each leaf is labeled with a 0. By performing a preorder traversal of the tree and omitting the final 0, a sequence with n 1's and n 0's is obtained. Such sequences will be called *tree sequences* and are the only representation of binary trees that will be used in this chapter; the set of all tree sequences is denoted $\mathbf{T}(n)$ (see Chapter 2) and may be thought of as the set of all well-formed parentheses with n left parentheses and n right parentheses. Figure 4.5 shows the extended binary tree that yields the tree sequence 1110001100 or well formed parentheses $((()))(())$.

Our generation algorithm will list all tree sequences lexicographically. A tree sequence is characterized by the property that there are exactly n 1's, n 0's, and in any prefix the number of 1's is at least as great as the number of 0's. We call this the *dominating property*. Thus to generate lexicographically we scan from right to left searching for the first 0 that we can change to a 1 and still maintain the dominating property. This 0 is rightmost with the property that a 1 is to its right. That 0 is changed to a 1 and the positions to the right are filled in the lexicographically minimal way.

The general pattern is that

$$\begin{array}{ll} a_1a_2\cdots a_{k-1}0\ \overbrace{11\cdots1}^{p\ 1's}\ \overbrace{00\cdots0}^{q\ 0's} & \text{becomes} \\ a_1a_2\cdots a_{k-1}1\ \underbrace{00\cdots0}_{q-p+2\ 0's}\ \underbrace{1010\cdots10}_{p-1\ 10's}, & \end{array}$$

where q is the number of 0's in the rightmost block of 0's and p is the number of 1's in the rightmost block of 1's. For example, the lexicographic successor of 11101011100000 is 11101100001010. An iterative implementation of this update is given in Algorithm 4.16.

```
procedure Next;
{Assumes a_0 = 0}
local k, q, p, j : ℕ;
begin
    k := 2n;
    while a_k = 0 do k := k − 1;
    q := 2n − k;
    while a_k = 1 do k := k − 1;
    if k = 0 then done := true else
        p := 2n − k − q;
        a_k := 1;
        for j := k + 1 to k + q − p + 2 do a_j := 0;
        j := 2n − 2p + 3;
        while j ≤ 2n − 1 do
        a_j := 1; a_{j+1} := 0; j := j + 2;
end {of Next};
```

Algorithm 4.16: Next for binary trees in lexicographic order.

Let's now analyze this algorithm. Note that $p \leq q$ for any tree sequence, since the reverse complement (flip all bits and reverse the string) of a tree sequence is also a tree sequence, and that the time to go from one tree to the next is $O(p+q) = O(q)$. Also since the reverse complement of a tree sequence is also a tree sequence, the number of tree sequences having a fixed value of q is equal to the number of extended binary trees whose leftmost leaf is at level q. We will show that the average value of this quantity is $3n/(n+2)$ and thus that the listing algorithm is CAT.[1]

Define $\mathbf{T}(n,k)$ to be the set of all bitstrings of length n whose leftmost k bits are 1's and whose $(k+1)$st bit is a 0. This corresponds to the number of extended binary trees on n internal nodes with leftmost leaf at level k. Let $T(n,k)$ be the number of bitstrings in $\mathbf{T}(n,k)$. The numbers $T(n,k)$ satisfy the following recurrence relation, which will be proven a little later.

$$T(n,k) = \begin{cases} T(n,2) & \text{if } k = 1 \\ T(n,k+1) + T(n-1,k-1) & \text{if } 1 < k < n \\ 1 & \text{if } k = n \end{cases} \tag{4.17}$$

Also, a non-recursive expression is known.

$$T(n,k) = \frac{k}{2n-k}\binom{2n-k}{n-k} \tag{4.18}$$

Iterating (4.17) we obtain

$$T(n+1,k+1) \;=\; \sum_{j=k}^{n} T(n,j). \tag{4.19}$$

[1]The exact amortized value of $p+q+1$ is determined in exercise 39.

Thus

$$\sum_{k=1}^{n} k \cdot T(n,k) = \sum_{k=1}^{n}\sum_{j=k}^{n} T(n,j) \tag{4.20}$$

$$= \sum_{k=1}^{n} T(n+1,k+1) \tag{4.21}$$

$$= T(n+2,3) \tag{4.22}$$

To obtain the average we divide this last expression by C_n and obtain the result below.

$$\frac{1}{C_n}\sum_{k=1}^{n} k \cdot T(n,k) = \frac{T(n+2,3)}{T(n+1,1)} \tag{4.23}$$

$$= \frac{\frac{3}{2n+1}\binom{2n+1}{n-1}}{\frac{1}{2n+1}\binom{2n+1}{n}} \tag{4.24}$$

$$= \frac{3n}{n+2} \tag{4.25}$$

This proves the following theorem.

THEOREM 4.1 *The average level number of the leftmost leaf of a binary tree is* $3n/(n+2)$.

Equivalently, a random walk in a random extended binary tree that starts at the root and goes to the left or right subtrees with equal probability ends at a leaf in $3n/(n+2)$ expected number of steps.

To develop a recursive algorithm we could use the recurrence relation (2.15) for the Catalan numbers but the resulting algorithm is not as simple as the one we obtain below. The key idea is to add another parameter k to the problem. A constructive form of the proof of (4.17) is the basis of our recursive algorithm, but we need to introduce some additional notation. For bitstrings x and $y = 1^k0x$ define operations *flip* and *insert* as $\mathit{flip}(y) = 1^{k-1}01x$ and $\mathit{insert}(y) = 1^{k+1}00x$. These definitions are extended to lists of bitstrings in the natural way.

To prove (4.17) classify the elements of $\mathbf{T}(n,k)$ according to whether they are of the form $\alpha = 1^k01x$ or $\beta = 1^k00x$; i.e., according to the value of the $(k+2)$nd bit. In the former case $\mathit{flip}(1^{k+1}0x) = \alpha$ and in the latter $\mathit{insert}(1^{k-1}0x) = \beta$; elements of the argument sets are counted by $\mathbf{T}(n,k+1)$ and $\mathbf{T}(n-1,k-1)$, respectively. In other words,

$$\mathbf{T}(n,k) = \mathit{flip}(\mathbf{T}(n,k+1)) \uplus \mathit{insert}(\mathbf{T}(n-1,k-1)), \tag{4.26}$$

where $\uplus$ denotes union of disjoint sets. Note that (4.26) proves the main case of (4.17).

This discussion leads to the recursive procedure `T` of Algorithm 4.17. To generate $\mathbf{T}(n,k)$, set the first $k+1$ symbols of $\mathbf{x}$ to 1^k0 and call $\mathsf{T}(n,k,k+2)$. To generate $\mathbf{T}(n)$ no initialization is necessary, just call $\mathsf{T}(n+1,1,1)$.

Let $t(n,k)$ denote the number of calls to procedure T with parameters n and k. Since on every level of recursive calls, procedure T takes constant time, the time complexity of

```
procedure T ( n, k, pos : ℕ);
begin
    if k = n then PrintTree else
    if k = 1 then
        x[pos] := 1; T( n, 2, pos + 1 ); x[pos] := 0;
    else
        x[pos] := 1; T( n, k + 1, pos + 1 );
        x[pos] := 0; T( n − 1, k − 1, pos + 1 );
end {of T};
```

Algorithm 4.17: Algorithm to generate the elements of $\mathbf{T}(n,k)$ in relex order.

the algorithm is proportional to $t(n,k)$. The numbers $t(n,k)$ satisfy the recurrence relation given below.

$$t(n,k) = \begin{cases} 1 + t(n,2) & \text{if } k = 1 \\ 1 + t(n,k+1) + t(n-1,k-1) & \text{if } 1 < k < n \\ 1 & \text{if } k = n \end{cases}$$

The following lemma proves that $t(n,k)$ is proportional to $T(n,k)$ and thus that the algorithm is CAT.

LEMMA 4.4 *For $1 < k \leq n$, $t(n,k) \leq 3T(n,k) - 2$, and $t(n,1) \leq 3T(n,1) - 1$.*

Proof: We proceed by induction. If $k = n$, then

$$t(n,n) \;=\; 1 \;=\; 3 \cdot T(n,n) - 2$$

If $k = 1$, then

$$\begin{aligned} t(n,1) &= 1 + t(n,2) \\ &\leq 1 + 3T(n,2) - 2 \\ &= 3T(n,1) - 1 \end{aligned}$$

If $k = 2$, then

$$\begin{aligned} t(n,2) &= 1 + t(n,3) + t(n-1,1) \\ &\leq 1 + 3T(n,3) - 2 + 3T(n-1,1) - 1 \\ &= 3T(n,2) - 2 \end{aligned}$$

If $2 < k < n$, then

$$\begin{aligned} t(n,k) &= 1 + t(n,k+1) + t(n-1,k-1) \\ &\leq 1 + 3T(n,k+1) - 2 + 3T(n-1,k-1) - 2 \\ &\leq 3T(n,k) - 2 \end{aligned}$$

□

Ranking the elements of $\mathbf{T}(n)$ may be visualized conveniently in terms of lattice walks, as was done for combinations. See Figure 4.7, which shows a n by n half grid and a walk that starts at the upper right corner (n,n) and ends at the lower left corner $(0,0)$. At each

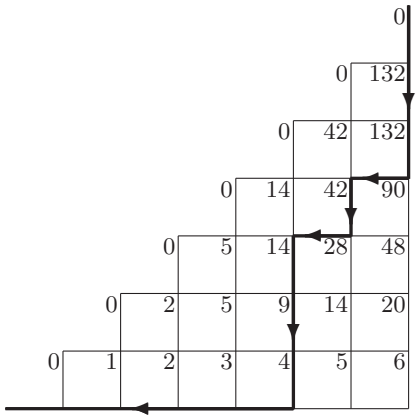


Figure 4.7: The walk, 11101011100000, corresponding to the rank $333 = 132 + 132 + 42 + 14 + 9 + 4$ tree in $\mathbf{T}(8)$.

step of the walk you must either move down or to the left; these two steps correspond to the two two terms in the recurrence relation (4.17) for $T(n,k)$. A step down corresponds to a 1 in a tree sequence and a step left to a 0. Again we attach numbers to vertical edges but not horizontal edges. The edge from $(n, k-1)$ to (n,k) is labeled

$$T(n, n-k) \;=\; \frac{n-k}{n+k}\binom{n+k}{k}.$$

Note that $T(n, n-k)$ is the number of walks from $(n-1,k)$ to $(0,0)$. The rank of a bitstring is simply the sum of the edge labels of the corresponding path. As was the case for combinations, the values of the labels can be computed on the fly (Exercise 43). Thus we have ranking and unranking algorithms that use $O(n)$ arithmetic operations.

4.6.2 Rooted Trees, Free Trees

Recall that a *rooted tree* is a connected acyclic graph with a distinguished vertex called the *root* and a *free tree* is a connected acyclic graph.

Labelled Rooted Trees

In proving Theorem 2.2 we established a bijection between the set of all functions $f : \{2,3,\ldots,n\} \to \{1,2,\ldots,n+1\}$ and labeled trees on $n+1$ nodes with root $n+1$. Those functions can be thought of as sequences $a_2, a_3, \ldots, a_n$ where $1 \le a_i \le n+1$ for $i = 2,3,\ldots,n$. Such sequences are trivial to generate, rank, and unrank lexicographically by an extension of our results on generating subsets. Basically, we're counting in base $n+1$ (See Exercise 8).

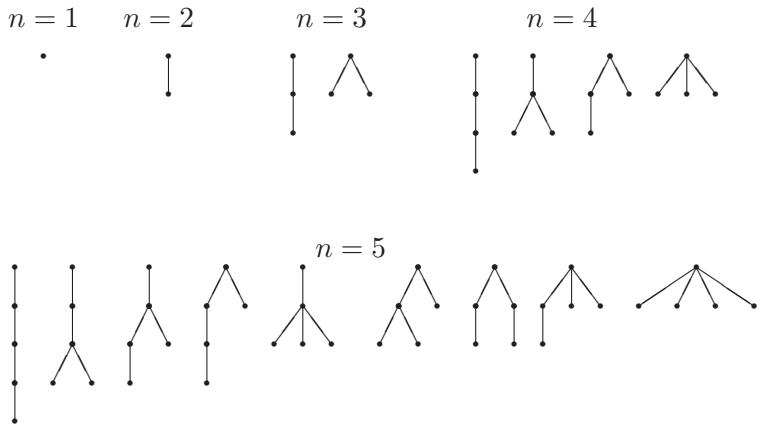


Figure 4.8: The rooted trees on at most 5 nodes.

Unlabeled Rooted Trees

In Figure 4.8 we show the rooted trees on $n = 1, 2, 3, 4, 5$ nodes. We have drawn these trees in what will be referred to as a *canonic* manner. This means that subtrees with greater height are drawn to the left, and that this definition is applied recursively.

To count the number a_n of rooted trees with n nodes we follow the discussion of Knuth [218]. Obviously, $a_1 = 1$. Given a tree with $n > 1$ vertices, suppose that there are j_i subtrees with i vertices. Since the order of the subtrees does not matter and the same subtree may occur repeatedly we have

$$a_n \;=\; \sum_{j_1+2j_2+\cdots=n-1} \prod_{i=1}^{n-1} \binom{a_i + j_i - 1}{j_i}. \tag{4.27}$$

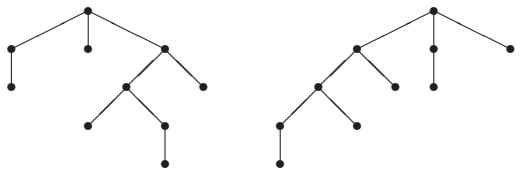
Using this recurrence one can compute a table of the a_n as follows.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a_n	1	1	2	4	9	20	48	115	286	719	1842	4766	12486	32973	87811

One may be tempted to use the recurrence relation (4.27) to generate rooted trees since it involves a summation over integer partitions of "products" of compositions, and there exist CAT algorithms for generating partitions, product spaces, and compositions. However, (4.27) is very awkward to use and a much simpler approach is available. The algorithm and analysis for generating rooted trees we present is essentially due to Beyer and Hedetniemi [30]. One way of representing a rooted tree is obtained by traversing the tree in preorder, recording the level numbers of the nodes. This gives a sequence encoding $e(T) = l_1, l_2, \ldots, l_n$ of T. The encoding also can be defined recursively. If $|T| = 1$ then $e(T) = 0$; otherwise, let the principle subtrees of T be $T_1, T_2, \ldots, T_m$. Then

$$e(T) \;=\; 0, e(T_1) + 1, e(T_2) + 1, \ldots, e(T_m) + 1,$$

where $e(T_i) + 1$ is the sequence resulting from adding 1 to each element of the $e(T_i)$. A canonic tree lexicographically maximizes $e(T)$ over all rooted trees isomorphic to T. For



0121123342 0123432121

Figure 4.9: Two isomorphic rooted trees. The one on the right is canonic.

012345	012321	01234
012344	012312	01233
012343	012311	01232
012342	012222	0123**1**
012341	012221	0123$\underline{2}$
012333	012212	0122**1**
012332	012211	012$\underline{12}$
012331	012121	012**11**
012323	012111	0**11**11
012322	011111	
(a)		(b)

Figure 4.10: (a) The 20 canonic sequences of length $n = 6$ in relex order (read down). (b) The 9 canonic sequences of length $n = 5$ with elements contributing to x indicated in bold and those contributing to y underlined.

example, see Figure 4.9. If T is canonic then $e(T)$ is said to be a *canonic* sequence. Let $h(T)$ denote the height of a tree. In a canonic tree $h(T_1) \geq h(T_2) \geq \cdots \geq h(T_p)$, and thus

$$\max(e(T_1)) \geq \max(e(T_2)) \geq \cdots \geq \max(e(T_p)).$$

We think of the nodes of the tree as being labeled in preorder with 1 as the root.

The algorithm lists canonic sequences in reverse lexicographic (relex) order. Given that $e(T) = l_1, l_2, \ldots, l_n$, let $succ(e(T)) = s_1, s_2, \ldots, s_n$ denote the relex successor of $e(T)$ and let

$$p = \max\{i : l_i > 1\} \text{ and } q = \max\{i : i < p, l_i = l_p - 1\}.$$

In other words, p is the position of the rightmost element larger than 1, and q is the parent of p. Note that p must be a leaf. We wish to decrease l_p by as small amount as possible, and then fill in the entries to the right in a lexicographically maximal manner. This is done by setting s_i as follows (updating in the order $i = 1, 2, \ldots, n$):

$$s_i = \begin{cases} l_i & \text{for } i = 1, 2, \ldots, p-1 \\ s_{i-p+q} & \text{for } i = p, \ldots, n. \end{cases}$$

The subsequence $l_q, l_{q+1}, \ldots, l_{p-1}$ represents the subtree rooted at q. The update repeatedly copies this subsequence. Implementing this update in a straightforward manner (by searching from the right, first for p and then for q) gives rise to an algorithm that is clearly

$O(n)$ per tree generated. Is it, in fact, CAT? That depends on the value of $n-q$, amortized over all sequences. Unfortunately a bit of experimentation reveals that the amortized value of $n-q$ is close to being linear in n (see Exercise 55). On the other hand, the amortized value of $n-p$ appears to be bounded by a constant independent of n. This is proven below.

The observations of the previous paragraph suggest that we maintain some additional data structure that will allow us to quickly determine q, given p. One possibility is to maintain an array par, where $par[i]$ is the parent of i. If $l_i = 1$, then $par[i] = 1$. The following lemma is useful in proving a time bound on the algorithm.

LEMMA 4.5 *If* $\mathbf{l}$ *is a canonic sequence and* $l_{i-1} = l_i = 1$*, then* $l_j = 1$ *for all* $j > i$*.*

PROOF: Note that if the sequence $l_1, l_2, \ldots, l_{i-2}, 1, 1, l_{i+1}, \ldots, l_n$ is valid, then the sequence $l_1, l_2, \ldots, l_{i-2}, 1, l_{i+1}, \ldots, l_n, 1$ is also valid but is lexicographically greater unless $l_{i+1} = \cdots = l_n = 1$. □

```
(r1)    procedure Next;
(r2)    local i, q : ℕ;   global p : NATURAL
(r3)    begin
(r4)        while L[p] = 1  do p := p − 1;
(r5)        if p = 1 then done :=  true else
(r6)        if L[p] = 2  and L[p − 1] = 1 then
(r7)            L[p] := 1; par[p] := par[p − 1];
(r8)        else
(r9)            q := p − par[p];
(r10)           for i := p  to n  do
(r11)               L[i] := L[i − q];
(r12)               if q + par[i − q] < p
(r13)                   then par[i] := par[i − q]
(r14)                   else par[i] := q + par[i − q];
(r15)           p := n;
(r16)   end  {of Next};
```

Algorithm 4.18: Generate rooted trees in relex order.

We now explain Algorithm 4.18. It is an iterative procedure `Next` that transforms the current sequence L (we use capitals since lower case L is confusing in programs) into its lexicographic predecessor. The arrays L and par are initialized so that $L[i] = par[i] = i-1$ for $i = 1, 2, \ldots, n$. If $L[p] = 2$ and $L[p-1] = 1$ at the beginning of an iteration then at the end of the iteration $L[p] = 1$ and all positions to the right also contain 1's. Thus there is no need to reset p to n; we just leave it's value unchanged. This means that p must be left global. In the next subsection, when generating free trees, we will use Next with p initialized to another value. The array `par` is easily updated. As the subsequence $L[q], \ldots, L[p-1]$ is repeatedly copied (line `(r11)`) the values of $par[q+1], \ldots, par[p-1]$ remain the same (line `(r13)`) since they form all nodes of a subtree except the root. The root nodes $p, p+q, p+2q, \ldots$ must be handled separately as is done at line `(r14)`. Line `(r12)` tests whether i is one if the copied root nodes.

There are two loops that we must consider, the while loop at line (r4) and the for loop that begins on line (r10). Let x be the number of times p is decremented by the while loop

at line (r4) and let y be one less than the number of times the body of for loop at line (r10) is executed; this can be viewed as the number of increments of p from its current value up to n. For example, if $n = 5$, then $x = 6$ and $y = 2$ (see Figure 4.10). Observe that

$$x - y = n - 1,$$

since p starts out at n and finishes at 1. We claim that the while loop at line (r4) is executed at most twice. If, on the previous call to `Next`, $l_p = 2$ and $l_{p-1} = 1$, then (by Lemma 4.5) $l_{p-2} > 1$ and so p will be decremented exactly twice. On the other hand, if on the previous call to `Next` $l_p \neq 2$ or $l_{p-1} \neq 1$, then some subsequence, not containing consecutive 1's, has been repeatedly copied. Thus, $l_n \neq 1$ or $l_{n-1} \neq 1$. Hence $x \leq 2a_n$, and so $y \leq 2a_n - n + 1 \leq 2a_n$. We conclude that the algorithm is CAT. The preceding analysis does not tell us a bound on the average value of $n - p$, but its exact asymptotic value has been determined by Kubicka [235].

THEOREM 4.2 (KUBICKA) *The amortized value of $n - p + 1$, taken over all canonic sequences, is $1/(1 - \rho)$, where ρ is the radius of convergence of the generating function $T(x) = \sum a_i x^i$.*

PROOF: The proof of this theorem is outside the scope of this book. We do, however, make a few observations that provide some insight into the result. How many trees are there with a given value of p? The number for which $n - p \geq k$ is a_{n-k} since the rightmost k leaves at level 1 may be removed, yielding an arbitrary rooted tree with $n - k$ nodes. Thus the sum of $n-p+1$ over all canonic sequences is $a_1 + a_2 + \cdots + a_n$. This implies that the generating function for the complexity of the algorithm is

$$T(x) + xT(x) + x^2T(x) + \cdots \;=\; \frac{1}{1-x}T(x).$$

The rest of the proof relies on some asymptotic manipulations. □

The value of ρ is approximately 0.3383219.... This theorem has the interesting consequence that if lines `(r6-8)` are removed from the algorithm, then it is still correct and it is still CAT. With those three lines removed the algorithm depends on L only at line `(r4)` where the test `L[p] = 1` may be replaced with `par[p] = 1`. Thus the array L may be removed and `par` used instead as the representation of a rooted tree.

Free Trees

Let t_n denote the number of free trees with n nodes. The following formula for t_n in terms of the number of rooted trees (the a_n from the previous subsection) is known.

$$t_n = a_n - \sum_{\substack{i+j=n \\ i \leq j}} a_i a_j + [\![n \text{ even}]\!] \binom{a_{n/2}+1}{2}. \tag{4.28}$$

Unfortunately, the combinatorial decomposition upon which it is based is even less suitable for generating than was the one that gave rise to (4.27) — those minus signs are killers! But we can use it to compute a table of the number of free trees.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
t_n	1	1	1	2	3	6	11	23	47	106	235	551	1301	3159	7741	19320

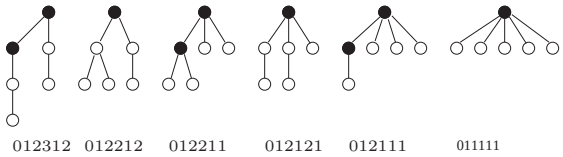


Figure 4.11: The rooted versions of the free trees on 6 vertices, listed in relex order. Central vertices are darkened.

The generation of free trees is more complicated than the generation of rooted trees, due mainly to the absence of the root. Our strategy is to define some unique root for each free tree and then use the level sequence representation of the previous subsection. These sequences will be generated in relex order using the same general algorithm but we need to somehow skip over those trees that are not representatives of a free tree. There are natual candidates for a root, namely the centroid(s) or center(s) of the tree.

For generalizing the algorithm of the previous section it proves to be most convenient to use a center of the tree as the root. Recall that the center(s) of a free tree may be found by repeatedly removing all leaves until one or two vertices remain. If two vertices remain, then they are adjacent and the tree is said to be bicentral. Furthermore, every longest path in the graph passes through the central vertex (or vertices).

The algorithm described in this subsection is due to Wright, Richmond, Odlyzko and McKay [461].

So we use the center of the tree as the root if there is a unique center, but what if T is bicentral? We need a rule that will uniquely identify one of centers. The definition given below will, at first glance, seem more complicated than is necessary (why not, for example, just take the lex maximum equivalent rooted tree?). However, this definition results in simple and efficiently implementable rules for getting the relex successor of a given free tree. By $Succ(e(\langle T\rangle))$ we denote the relex successor of free tree T; i.e., it is the encoding of the free tree whose rooted version follows $e(\langle T\rangle)$ in relex order. Define $|T|$ to be the number of vertices in T. Given two rooted trees S and T, define $S \prec T$ to mean that either $|S| < |T|$, or $|S| = |T|$ and $e(S)$ lexicographically precedes $e(T)$, where $e()$ is the level number encoding introduced in the previous subsection.

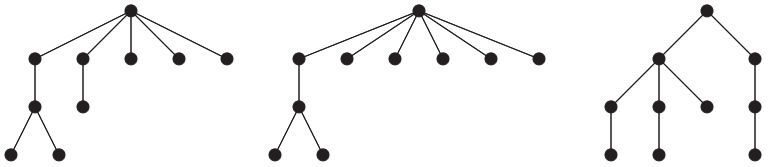
Given an edge $[u, v]$ joining two centers of T, let T_u and T_v denote the two rooted subtrees that remain when $[u, v]$ is deleted. The root of T_u is u and the root of T_v is v. We root a free tree T at u if $T_v \prec T_u$ and root it at v otherwise. Given a free tree T, by $\langle T\rangle$ we denote the rooted version of T with the root chosen as described above. This is called the *rooted version* of T. In Figure 4.11 we show the rooted versions of the 6 free trees on 6 vertices.

Let $\langle T\rangle$ be a canonic rooted tree with principle subtrees $T_1, T_2, \ldots, T_m$. Given an encoding $e(S)$ of some rooted tree S, the following theorem gives a necessary and sufficient condition for there to be a tree T such that $e(S) = e(\langle T\rangle)$.

THEOREM 4.3 *For $n \geq 3$, the encoding $e(S)$ of a rooted tree with principal subtrees $T_1, T_2, \ldots, T_m$ is the encoding $e(\langle T\rangle)$ of the rooted version $\langle T\rangle$ of a free tree T if and only if*

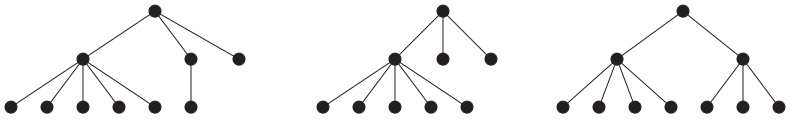
(A) *$m \geq 2$ (i.e., there is more than one principal subtree), and either condition (B1) below or conditions (B2),(C),(D) hold.*

(B1) *$\max(e(T_1)) = \max(e(T_2))$ (i.e., T is unicentral), or*



(a) 0123312111 (b) 0123311111 (c) 0123232123

Figure 4.12: Condition (B2) fails: (The height of T_2 is reduced too much). (a) original tree T. (b) $succ(e(T))$. (c) $Succ(e(T))$.



(a) 0122222121 (b) 0122222111 (c) 0122221222

Figure 4.13: Condition (C) fails: ($|T_1|$ has too many nodes). (a) original tree T. (b) $succ(e(T))$. (c) $Succ(e(T))$.

(B2) $\max(e(T_1)) = 1 + \max(e(T_2))$ *(i.e., T is bicentral), and*

(C) $|T_1| \leq n - |T_1|$, *and*

(D) if equality holds in (C), then $e(T_1) \preceq 0, 1 + e(T_2), \ldots, 1 + e(T_m)$.

The theorem is simply a restatement of our definition of the rooted version of a free tree, stated in terms of the level sequences. Item (A) says that the center is not a leaf, which is obvious for $n \geq 3$. Items (B1) and (B2) are another way saying that a center is a vertex that minimizes the maximum distance to a node. If $\max(e(T_1)) = \max(e(T_2))$ then the tree is unicentral and the central vertex is the root. If $\max(e(T_1)) = 1 + \max(e(T_2))$ then the tree is bicentral; one center is the root and the other is the root of T_1. The last two items, (C) and (D), deal with the bicentral case. Item (C) says that T_1 has at least as many nodes as the remainder of the tree. Item (D) says that the encoding of T_1 is lexicographically smaller than the encoding of the rest of the tree.

Our listing is in relex order so the initial tree is

$$0, 1, \ldots, \lfloor n/2 \rfloor, 1, 2, \ldots, \lceil n/2 \rceil - 1,$$

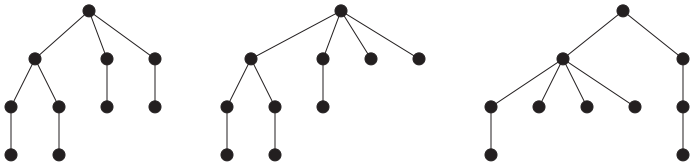
which is the encoding of a path of length n. The final tree is $0, 1, \ldots, 1$ which is the encoding of a star, $K_{1,n-1}$.

Condition (A) can never fail, nor can condition (B1). For condition (B2) to fail $e(T)$ and $succ(e(T))$ must have the form:

$$\begin{aligned} e(T) &= 0, 1 + e(T_1), 1, 2, \ldots, h-1, 1, 1, \ldots, 1 \\ succ(e(T)) &= 0, 1 + e(T_1), 1, 2, \ldots, h-2, h-2, \ldots, h-2 \end{aligned}$$

For condition (C) to fail $e(T)$ and $succ(e(T))$ must have the form:

$$\begin{aligned} e(T) &= 0, 1 + e(T_1), \overbrace{1, 2, \ldots, h, 1, 1, \ldots, 1}^{<|T_1|-1} \\ succ(e(T)) &= 0, 1 + e(T_1), 1, 2, \ldots, h-1, h-1, \ldots, h-1 \end{aligned}$$



(a) 0123231212 (b) 0123231211 (c) 0123222123

Figure 4.14: Condition (D) fails: (The encoding of T_1 is too large). (a) original tree T. (b) $succ(e(T))$. (c) $Succ(e(T))$.

For condition (D) to fail $e(T)$ and $succ(e(T))$ must have the form:

$$\begin{aligned} e(T) &= 0, 1, 2 + e(T_2), \ldots, 2 + e(T_m), 1 + e(T_2), \ldots, 1 + e(T_m) \\ succ(e(T)) &= 0, 1, 2 + e(T_2), \ldots, 2 + e(T_m), 1 + e(T_2), \ldots, 1 + succ(e(T_m)) \end{aligned}$$

These three cases are illustrated in Figures 4.12–4.14. In each case $succ(e(T))$ is not the rooted encoding of a free tree. However, the same general strategy will produce $Succ(e(T))$ from $succ(e(T))$ in each case. Clearly, T_1 is going to have to change in $Succ(e(T))$, even though it didn't change in $succ(e(T))$. We therefore apply the successor function Next (of Algorithm 4.18) with p initialized to $|T_1| + 1$. The result is a tree that is lexicographically smaller than $succ(e(T))$. Is it valid? Not necessarily: if $L[1 + |T_1|] > 2$, then a rooted tree with only one principal subtree is produced. Note that all unicentral trees with a given T_1 precede all bicentral trees with a given T_1. Therefore, if $L[1+|T_1|] > 2$, then we now need to add a second principal subtree to transform it into a unicentral tree. This second principal subtree should be lexicographically as large as possible, which means that it must be a path. The addition of this path is accomplished by changing the last h elements to $1, 2, \ldots, h$, where $h = 1 + \max(e(T_1))$. The overall process of obtaining $Succ(e(T))$ is summarized as Algorithm 4.19. Let $L = e(T)$.

$L := succ(L)$;
if a condition of Theorem 4.3 fails **then begin**
    $L :=$ sequence obtained by running Next with p initialized to $|T_1| + 1$;
    if $L[1+|T_1|] > 2$ **then** $L[n-h+1 \ldots n] := 1, 2, \ldots, h$;

Algorithm 4.19: Relex algorithm for generating free trees. Given $L = e(\langle T \rangle)$ it produces $Succ(e(\langle T \rangle))$.

Intuitively, it is plausible that this algorithm is efficient, since the trees T for which $succ(T)$ violates one of the conditions have a very specific form, and we expect the number of them to be asymptotically smaller than the number of free trees with n nodes. We can detect failures by maintaining variables $m_1 = \max(e(T_1))$, $m_2 = \max(e(T_2))$, and $n_1 = |T_1|$. Clearly, these variables allow us to determine failures of conditions (B2) and (C) with simple tests. Determining a failure of condition (D) is done by maintaining a variable c which is the index of the first element of $e(T_2)$ which is different than the corresponding element of $e(T_1)$. Let A_n be the time used to generate all free trees, aside from the time B_n used in obtaining $Succ(e(T))$ from $succ(e(T))$ when failures occur.

The amortized time $A_n/|\mathcal{T}_n|$ is proportional to the amortized value of $n - p + 1$, by the same reasoning used in the case of generating rooted trees. From the asymptotic (2.18) and Lemma 2.10, $A_n \leq 4t_n$. We now need to deal with B_n. Clearly the time to produce

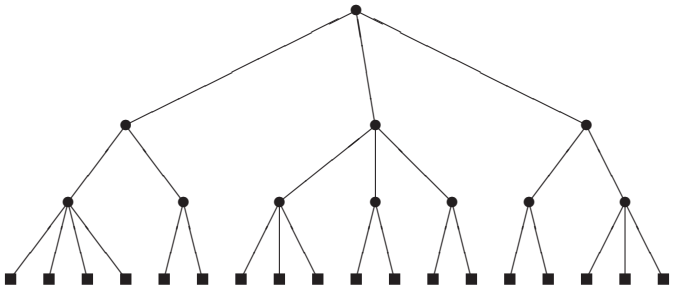


Figure 4.15: B-tree of order 4 with encoding 32223242323.

$Succ(e(T))$ from $succ(e(T))$ is $O(n)$, so we need only show that the number of failures is $o(t_n/n)$. However, proving this requires some delicate asymptotic manipulations that are beyond the scope of this book. See [461] for the details.

THEOREM 4.4 (WRIGHT, RICHMOND, ODLYZKO AND MCKAY) *Properly implemented, Algorithm 4.19 is a CAT algorithm for generating all free trees.*

4.6.3 B-trees

B-trees are used extensively for the storage and retrieval of large amounts of data stored on disk. In particular, they are used in many database applications. Their combinatorial properties are not elegant, but algorithms for generating and ranking them are rather nice and illustrate general useful techniques.

In this section we present a modification of an algorithm, due to Kelsen [209], for generating all B-trees. Recall that a *B-tree of order m* is an ordered tree which satisfies the following properties.

1. Every node has at most m children.
2. Every node, except the root, has at least $m/2$ children.
3. The root has at least 2 children.
4. All leaves occur on the same level.

We will assume throughout this section that $m \geq 3$. As is typical, we generate a sequence encoding of B-trees. Our encoding is the sequence of number of children of each internal node, where the nodes are traversed bottom-up level-by-level and from right-to-left within a level. See Figure 4.15 for an example.

We extend the definition of B-trees to what we call $B(s, d)$ trees. The definition is the same except that instead of all leaves occurring at the same level leaves may occur on two levels; leaves at the lowest level are all in the left part of the tree. There are s leaves at the lowest level and d leaves at the penultimate level. The B-trees with n leaves are just the $B(n, 0)$ trees. Figure 4.16 shows the trees in $B(6, 2)$.

Let $b(s, d) = |B(s, d)|$. The addition of this second parameter to the problem helps immensely, and is a recurrent theme in combinatorial generation. We've seen an instance

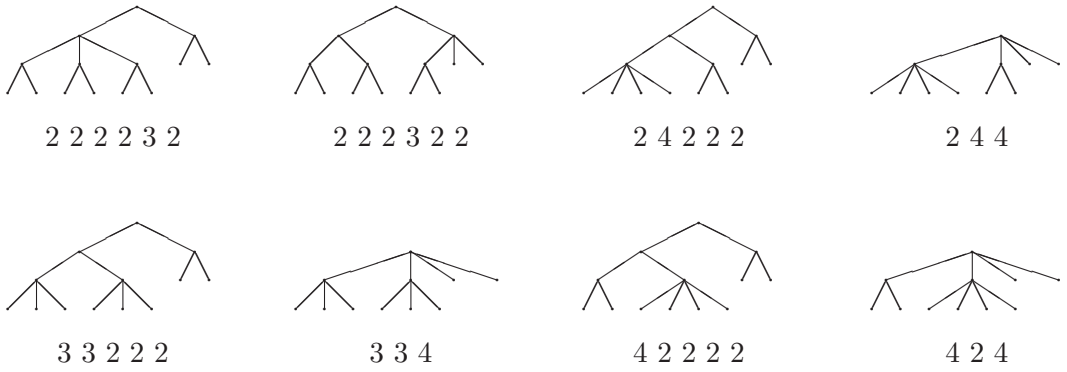


Figure 4.16: The eight trees in $B(6,2)$ (with $m=4$).

$d=$	0	1	2	3	4	5	6	7	8	9	10
$s=1$	1	0	0	0	0	0	0	0	0	0	0
2	1	1	1	2	2	4	5	9	15	28	45
3	1	1	1	2	2	4	5	9	15	28	45
4	2	2	3	4	6	9	14	24	43	73	118
5	2	2	4	4	8	10	18	30	56	90	146
6	4	5	8	10	17	24	42	73	129	208	335
7	5	8	10	16	23	37	63	114	191	309	494
8	9	15	18	31	43	74	127	228	371	599	968
9	15	22	30	48	71	123	217	376	607	975	1608
10	28	36	57	83	135	232	415	691	1116	1797	3031

Table 4.1: The numbers $b(s,d)$ when $m=4$.

of it once before when generating binary trees; i.e., introducing k to get $\mathbf{T}(n,k)$. Table 4.1 contains the values of $b(s,d)$ for $m=4$.

A recurrence relation for $b(s,d)$ and its combinatorial interpretation form the basis of our recursive algorithm for generating B-trees. The trees will be generated in lex order.

$$b(s,d) = \begin{cases} 1 & \text{if } s=1,\ d=0 \\ b(d+1,0)+b(m/2,d+1) & \text{if } m \text{ is even},\ s=m \\ b(d+1,0) & \text{if } m \text{ is odd},\ s=m \\ b(d+1,0) & \text{if } \lceil m/2\rceil \le s < m \\ b(1,0) & \text{if } 2 \le s < m,\ d=0 \\ \displaystyle\sum_{i=\lceil m/2\rceil}^{\min(s-\lceil m/2\rceil,m)} b(s-i,d+1) & \text{if } s>m,\ d\ge 0 \end{cases} \tag{4.29}$$

Each case of this recurrence relation is fairly easy to understand by considering the underlying trees. Clearly if $s=1$ and $d=0$, then there is only one tree, namely the tree with one node. If m is even, then $b(m,d)=b(d+1,0)+b(m/2,d+1)$ because any leftmost m

leaves must have only one parent ($b(d+1,0)$) or have two parents, both with $m/2$ children ($b(m/2,d+1)$). Similarly, s leftmost leaves must have a common parent if $\lceil m/2 \rceil \le s < m$; in this case $b(s,d) = b(d+1,0)$. If $2 \le s < m$ and $d = 0$, then the parent of those s nodes must be the root of the tree. In the general case of $s > m$ and $d \ge 0$, we simply classify the trees according to the number i of children of the parent of the leftmost leaves that are siblings. This discussion proves the recurrence relation (4.29).

The Pascal code to do the generation recursively is based on the recurrence relation and it's proof; see Algorithm 4.20. Some cases have been collapsed. No initialization is necessary. To generate $B(s,d)$ call Bgen$(s,d,1)$; to generate all B-trees with n leaves call Bgen$(n,0,1)$. The parameter $p-1$ to PrintIt indicates the number of nodes in the current tree and is necessary since the number of internal nodes in a tree can vary.

```
procedure Bgen ( s, d, p : ℕ);
local i : ℕ;
begin
    if s = 1 then PrintIt( p − 1 ) else
    if s > m and d ≥ 0 then
        for i := ⌈m/2⌉ to min{s − ⌈m/2⌉, m} do
            a[p] := i; Bgen( s − i, d + 1, p + 1 );
    else
        if s = m and m is even then
            a[p] := ⌈m/2⌉; Bgen( m2, d + 1, p + 1 );
        a[p] := s; Bgen( d + 1, 0, p + 1 );
end {of Bgen};
```

Algorithm 4.20: Generating B-trees in lexicographic order.

To show that this algorithm is CAT we will show that there are at most 4 successive calls of degree one. By Lemma 2.5 we conclude that the listing algorithm is CAT. From the recurrence relation, we observe that calls of degree one occur only if one of the four conditions [A],[B],[C],[D] occurs, as shown in the table below.

	condition	**result**
[A]	$s = m$ is odd	$b(d+1,0)$
[B]	$\lceil m/2 \rceil \le s < m$	$b(d+1,0)$
[C]	$2 \le s < m$ and $d = 0$	$b(1,0)$
[D]	m is odd and $s = m+1$	$b(\lceil m/2 \rceil, d+1)$

Item [D] bears some explanation. The sum in (4.29) has one term if and only if

$$\lceil m/2 \rceil = \min(s - \lceil m/2 \rceil, m).$$

But we cannot have $\lceil m/2 \rceil = m$ unless $m = 1$, which cannot occur. If $\lceil m/2 \rceil = s - \lceil m/2 \rceil$, then $s = 2\lceil m/2 \rceil$. If m is even then we get $s = m$, which is impossible since $s > m$. If m is odd then we obtain $s = m+1$.

Let us now consider what can happen from case [A] in order to obtain further degree one nodes. From [A] we go to $b(d+1,0)$ from which it is possible to obtain cases [A],[C], or [D], but not [B]. If $d = 0$ then we are at a leaf. If $d+1 = m$ or $2 \le d+1 < \lceil m/2 \rceil$, then we go to $b(1,0)$ which is a leaf; this covers subcases [A] and [C]. The most interesting subcase is [D], which occurs when $d = m$; the result is $b(\lceil m/2 \rceil, 1)$, which then goes to $b(2,0)$ and

then to a leaf. Similar exhaustive reasoning will lead us to the conclusion that the longest path of degree one nodes has length 4.

Examples of such maximal paths, with $m = 5$, are

$$[A](5,5) \to [D](6,0) \to [B](3,1) \to [C](2,0) \to (1,0)[\text{leaf}]$$

and

$$[B](3,5) \to [D](6,0) \to [B](3,1) \to [C](2,0) \to (1,0)[\text{leaf}].$$

Algorithm 4.21 contains the corresponding ranking algorithm, which we present without explanation. The structural similarity of the ranking and generating algorithms is obvious, and is something to be strived for.

```
function Rank ( s, d, k : ℕ) : ℕ;
local i, r : ℕ;
begin
    if s = 1 then return(  0  );
    if s > m then
        r := 0;
        for i := ⌈m/2⌉  to a[k] − 1  do r := r + b(s − i, d + 1);
        return(  r+ Rank( s − a[k], d + 1, k + 1 ) );
    if s = m  and m is even then
        if a[k] = m2 then return(  Rank( ⌈m/2⌉, d + 1, k + 1 )  );
        else return(  b(⌈m/2⌉, d + 1)  ) + Rank( d + 1, 0, k + 1 );
    return(  Rank( d + 1, 0, k + 1 )  );
end  {of Rank};
```

Algorithm 4.21: Ranking B-trees in lexicographic order.

4.7 Set Partitions

A partition of a set S is a collection of disjoint subsets of S whose union is S. The number of partitions of an n-set is the Bell number B_n which is discussed in Chapter 2. Below we list the 15 partitions of the set $\{1, 2, 3, 4\}$. We use a slash (/) to separate the subsets and omit all commas and parentheses; 13/24 means $\{1, 3\}, \{2, 4\}$.

1234
123/4 124/3 134/2 234/1 12/34 13/24 14/23
1/2/34 1/3/24 1/4/23 2/3/14 2/4/13 3/4/12
1/2/3/4

Traditionally, the subsets of the union are referred to as *blocks*. In the list above partitions with the same number of blocks are listed on the same row. The number of partitions of a n-set into k blocks is the Stirling number of the second kind, $\left\{ {n \atop k} \right\}$, which satisfy the following positive recurrence relation, which may be proven by classifying n according to whether it occurs in a block by itself or not (c.f. Equation (2.13)).

$$\left\{ {n \atop k} \right\} = \begin{cases} \left\{ {n-1 \atop k-1} \right\} & \text{if } k = n \\ \left\{ {n-1 \atop k} \right\} & \text{if } k = 1 \\ \left\{ {n-1 \atop k-1} \right\} + k \left\{ {n-1 \atop k} \right\} & \text{if } 1 < k < n \end{cases} \tag{4.30}$$

It is awkward to work directly with the partitions; instead we use a sequence encoding which has come to be known as a "restricted growth function". This is a string $a_1 a_2 \ldots a_n$ satisfying $a_1 = 0$ and the restricted growth (RG) condition

$$a_i \;\leq\; 1 + \max\{a_1, \ldots, a_{i-1}\} \quad \text{for } i = 2, \ldots, n. \tag{4.31}$$

There is a one-to-one correspondence between partitions of $\{1, 2, \ldots, n\}$ and restricted growth strings (we will use *string* instead of *function*) of length n. Given a partition, order its blocks $\mathbf{B}_0, \mathbf{B}_2, \ldots, \mathbf{B}_{k-1}$ according to the smallest number in each block. Then set a_i to be the block in which i occurs. For example $\mathbf{B}_0 = \{1, 2, 4, 7\}, \mathbf{B}_1 = \{3, 8\}, \mathbf{B}_2 = \{5, 6, 9\}$ is such an ordering of blocks and the corresponding RG string is 001022012. Here are the restricted growth sequences corresponding to the 15 partitions listed earlier.

```
0000
0001  0010  0100  0111  0011  0101  0110
0122  0121  0112  0120  0102  0012
0123
```

The restricted growth condition (4.31) leads immediately to a recursive procedure that lists all RG strings whose largest value is less than k, as shown in Algorithm 4.22. Variables n and k are global. The initial call is S(1,0). To generate all RG strings, set $k := n$. Variable m keeps track of the largest value in $a[1..l-1]$.

```
procedure gen( l, m : ℕ);
local i : ℕ;
begin
    if l > n then PrintIt
    else
        for i := 0 to m do
            a_l := i;
            gen( l + 1, m );
        if m < k − 1 then
            a_l := m + 1;
            gen( l + 1, m + 1 );
end{of gen};
```

Algorithm 4.22: Procedure to generate RG sequences in lex order.

This algorithm is CAT because there are no calls of degree one except for the root as long as $k \geq 2$. See Figure 4.17 which shows the computation tree for $n = k = 4$. The number of nodes in the tree is clearly the sum of the first n Bell numbers.

However, what if we only wanted the partitions into exactly k blocks? The recurrence relation (4.30) leads to degree one nodes whenever $n = k$ or $k = 1$. A casual inspection of experimental data reveals that the condition $n = k$ occurs more often than $k = 1$, so we apply the PET (#1) technique and eliminate paths along which $n = k$. This yields Algorithm 4.23, which generates RG sequences in a pseudo-colex order. The array `a` is generated right-to-left, but because there are two recursive calls when a_n is set to $k - 1$ the resulting list is not exactly in colex order. The computation tree is not a prefix tree. To initialize set $a[i]{:=}i - 1$ for $i = 1, 2, \ldots, n$ and then call `S(n,k)`. In order to understand the algorithm note that when $k - 1$ is placed at line (s9), it is the leftmost $k - 1$.

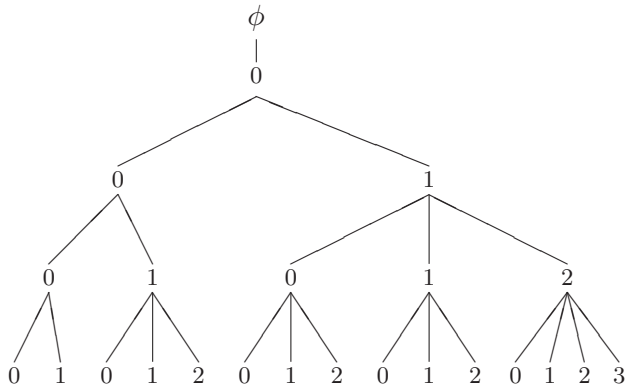


Figure 4.17: Computation tree for $n = k = 4$.

```
(s1)    procedure S( n, k : ℕ);
(s2)    local i : ℕ;
(s3)    begin
(s4)        if n = k then PrintIt
(s5)        else
(s6)            for i := 0 to k − 1 do
(s7)                a_n := i; S( n − 1, k ); a_n := n − 1;
(s8)            if k > 1 then
(s9)                a_n := k − 1; S( n − 1, k − 1 ); a_n := n − 1;
(s10)   end{of S};
```

Algorithm 4.23: Generation of set partitions with exactly k blocks in a pseudo-colex order.

The algorithm takes time $\Theta(n)$ if $k = 1$ and we will show that the amortized number of calls for $k > 1$ is bounded by 2.5. Let $t(n,k)$ denote the number of calls resulting from the call $\mathsf{S}(n,k)$. Then

$$t(n,k) \;=\; \begin{cases} 1 & \text{if } k = n \\ 1 + t(n-1,k) & \text{if } k = 1 \\ 1 + t(n-1,k-1) + k \cdot t(n-1,k) & \text{if } 1 < k < n. \end{cases}$$

LEMMA 4.6 *If $1 < k \le n$, then $t(n,k) \le \frac{5}{2}\left\{{n \atop k}\right\} - 1$.*

Proof: First note that $\left\{{n \atop 1}\right\} = \left\{{n \atop n}\right\} = t(n,n) = 1$, $\left\{{n \atop 2}\right\} = 2^{n-1} - 1$, and $t(n,1) = n$ for all $n \ge 1$. We now argue by induction on n. If $k = n$, then $t(n,n) = 1 \le \frac{5}{2}\left\{{n \atop n}\right\} - 1 = \frac{3}{2}$. If $2 < k < n$, then

$$\begin{aligned} t(n,k) &= 1 + t(n-1,k-1) + k \cdot t(n-1,k) \\ &\le 1 + \frac{5}{2}\left\{{n-1 \atop k-1}\right\} - 1 + \frac{5k}{2}\left\{{n-1 \atop k}\right\} - k \\ &\le \frac{5}{2}\left\{{n \atop k}\right\} - 1. \end{aligned}$$

If $k = 2$, then

$$\begin{aligned} t(n,2) &= 1 + t(n-1,1) + 2t(n-1,2) \\ &= n + 2t(n-1,2) \quad \{\text{Iterate recurrence relation}\} \\ &= 2^{n-2} + \sum_{i=0}^{n-3} (n-i)2^i \\ &= 5 \cdot 2^{n-2} - n - 2 \\ &\le \frac{5}{2}(2^{n-1} - 1) - 1 \\ &= \frac{5}{2}\left\{{n \atop 2}\right\} - 1. \quad \{\text{This is where the } 5/2 \text{ comes from.}\} \end{aligned}$$

Thus we've covered all cases $1 < k \le n$ and the lemma is proved. □

Ranking

We now develop a ranking algorithm for RG sequences in lex order. Consider the example 001022012. Its rank is equal to the number of RG functions of length 9 with length 3 prefix 000, or length 5 prefixes 00100, 00101, or length 6 prefixes 001020, 001021, or length 8 prefix 00102200, or length 9 prefixes 001022010, or 001022011. Note that 00100 and 00101 are prefixes of exactly the same number of RG sequences of a given length; all that matters is the length of the prefix and the maximum value occurring in it.

Define

$$m_i \;=\; \max\{a_1, a_2, \ldots, a_{i-1}\} \quad \text{for } i = 2, \ldots, n;$$

this is the same as the parameter `m` in procedure `gen` of Algorithm 4.22. Also define $R(n,m)$ to be the number of sequences $a_1 a_2 \cdots a_n$ for which $a_1 = m$ and the restricted growth

n/m	0	1	2	3	4	5	6	7	8	9
1	1	1	1	1	1	1	1	1	1	1
2	2	3	4	5	6	7	8	9	10	
3	5	10	17	26	37	50	65	82		
4	15	37	77	141	235	365	537			
5	52	151	372	799	1540	2727				
6	203	674	1915	4736	10427					
7	877	3263	10481	29371						
8	4140	17007	60814							
9	21147	94828								
10	115975									

Table 4.2: Restricted tail numbers $R(n,m)$ for $1 \leq n+m \leq 10$.

condition (4.31) is satisfied. These are sometimes called the *restricted tail coefficients*. Thus $R(n,0)$ is the Bell number B_n. It now follows by our previous discussion that

$$Rank(a_1 a_2 \cdots a_n) \;=\; \sum_{i=2}^{n} a_i \cdot R(n-i+1, m_i). \tag{4.32}$$

For example,

$$\begin{aligned} Rank(001022012) &= R(7,0) + 2R(5,1) + 2R(4,2) + R(2,2) + 2R(1,2) \\ &= 877 + 302 + 154 + 4 + 2 \;=\; 1339. \end{aligned}$$

The $R(n,m)$ numbers satisfy a simple recurrence relation which was used to construct Table 4.2. If $n = 1$, then $R(1,m) = 1$, and if $n > 1$, then

$$R(n,m) \;=\; R(n-1, m+1) + (m+1) \cdot R(n-1, m).$$

To prove the recurrence relation consider a_2; either $0 \leq a_2 \leq m$, in which case $a_2 \cdots a_n$ is counted by $R(n-1,m)$, or $a_2 = m+1$, in which case $a_2 \cdots a_n$ is counted by $R(n-1, m+1)$.

If R is precomputed then ranking takes $O(n)$ arithmetic operations. Observe that the m_i can be computed in $O(n)$ arithmetic operations. Unranking is also $O(n)$, given the R numbers. Computing the restricted tail numbers takes $\Theta(n^2)$ arithmetic operations.

4.8 Numerical Partitions

A numerical partition of a positive integer n is a sequence $p_1 \geq p_2 \geq \cdots \geq p_k > 0$ such that $p_1 + p_2 + \cdots + p_k = n$. Each p_i is called a *part*. For example, $7+4+4+1+1+1$ is a partition of 18 into 6 parts. The number of numerical partitions of n is denoted $p(n)$ and the number of partitions of n into k parts is denoted $p(n,k)$; this is also the number of partitions of n whose largest part is k. These numbers are studied extensively in number theory and combinatorics. See Chapter 2 for further information on $p(n)$ and $p(n,k)$.

There is a simple recurrence relation that the $p(n,k)$ numbers satisfy. This recurrence will form the basis of a recursive algorithm for generating numerical partitions.

$$p(n,k) = p(n-1, k-1) + p(n-k, k) \tag{4.33}$$

To prove (4.33) classify partitions according to whether the smallest part is a 1 or not. If it is a 1, then removing the 1 leaves a partition of $n-1$ into $k-1$ parts. If it is greater than 1, then reducing each part by 1 leaves a partition of $n-k$ into k parts. It is this second term of the recurrence relation that causes trouble when trying to develop an efficient generation algorithm since it seems to require $\Theta(k)$ steps. This recurrence relation may be iterated to obtain

$$p(n,k) \;=\; \sum_{j=1}^{\min(k,n-k)} p(n-k,j). \tag{4.34}$$

Of course, (4.34) may be proven directly by interpreting the right-hand side as a sum over the size j of the second largest part of a partition. One way to represent partitions is by using the string $p_1p_2\cdots p_k$; we call this the *natural representation*. Another representation is obtained by keeping track of how many times each part occurs; this is called the *multiplicity representation*. If

$$p_1+p_2+\cdots+p_k \;=\; \underbrace{v_1+\cdots+v_1}_{m_1}+\underbrace{v_2+\cdots+v_2}_{m_2}+\cdots+\underbrace{v_l+\cdots+v_l}_{m_l}$$

where $v_1>v_2>\cdots>v_l>0$ and $m_i>0$, then clearly $p_1p_2\cdots p_k = v_1^{m_1}v_2^{m_2}\cdots v_l^{m_l}$. For example, $7^14^21^3$ represents the example partition of 18 given in the first paragraph of this section. The multiplicity representation of the partition $p_1+p_2+\cdots+p_k$ is the string of pairs $[v_1,m_1][v_2,m_2]\cdots[v_l,m_l]$; e.g., our example is represented as $[v_1,m_1][v_2,m_2][v_3,m_3] = [7,1][4,2][1,3]$. Below is a list of all integer partitions of 7 in lexicographic order in both representations.

partition	**natural**	**multiplicity**
1+1+1+1+1+1+1	1111111	[1,7]
2+1+1+1+1+1	211111	[2,1][1,5]
2+2+1+1+1	22111	[2,2][1,3]
2+2+2+1	2221	[2,3][1,1]
3+1+1+1+1	31111	[3,1][1,4]
3+2+1+1	3211	[3,1][2,1][1,2]
3+2+2	322	[3,1][2,2]
3+3+1	331	[3,2][1,1]
4+1+1+1	4111	[4,1][1,3]
4+2+1	421	[4,1][2,1][1,1]
4+3	43	[4,1][3,1]
5+1+1	511	[5,1][1,2]
5+2	52	[5,1][2,1]
6+1	61	[6,1][1,1]
7	7	[7,1]

Our first algorithm uses the natural representation and is based upon (4.34). Interpret $p(n,k)$ as the number of partitions of n with largest part k, and the index of summation, j, as the size of the second largest part. The algorithm lists all partitions of n whose largest part is k in lexicographic order; see Algorithm 4.24. The initial call is `P(n,k,1)`; no initialization is necessary. To generate all partitions of n call `P(2*n,n,1)` and modify `PrintIt` so that it doesn't print `p[1]` (or otherwise ignores `p[1]`).

```
procedure P ( n, k, t : ℕ);
local j : ℕ;
begin
    p_t := k;
    if n = k then Printlt( t ) else
    for j := 1 to min(k, n − k) do P( n − k, j, t + 1 );
end{of P}
```

Algorithm 4.24: Lexicographic generation of all partitions of n whose largest part is k.

Unfortunately the algorithm is not CAT when $k = 1$ since then there is only one partition and it takes n recursive calls to create it. This bad behavior propagates and the algorithm is not CAT for small values of k. However, the calls when $k = 1$ can be eliminated by applying the PET (#1) technique, initializing p to be all 1's and restoring the value 1 as the recursion backs up. Details may be found in Algorithm 4.25. To prove that this algorithm is CAT note that every non-leaf node in the computation tree now has degree at least 2 or is the parent of a leaf. The computation tree for all partitions of $n = 7$, which used to have 45 nodes, now has 26 nodes, and the tree for $n = 20$, which used to have 2714 nodes, now has 1117 nodes.

```
procedure P ( n, k, t : ℕ);
local j : ℕ;
begin
    p_t := k;
    if n = k or k = 1 then Printlt( t + n − k ) else
    for j := 1 to min(k, n − k) do P( n − k, j, t + 1 );
    p_t := 1;
end{of P};
```

Algorithm 4.25: CAT generation of all partitions of n whose largest part is k.

It is sometimes useful to be able to generate all partitions with a fixed number of parts or with both a fixed number of parts and given largest part. We now develop an algorithm for these problems. Define $\mathbf{P}(n, k, s)$ to be the set of all partitions of n into k parts with largest part equal to s, and let $p(n, k, s) = |\mathbf{P}(n, k, s)|$. Clearly, in order to have $p(n, k, s) > 0$ we must have at least one part equal to s and at most k parts equal to s. Thus

$$s + k - 1 \le n \le ks.$$

By classifying the partitions of $\mathbf{P}(n, k, s)$ according to the value of the second largest part, call it j, we obtain the following recurrence relation, which has no zero terms; it is a positive recurrence relation.

$$p(n, k, s) \;=\; \sum_{j=\max(1,\lceil \frac{n-s}{k-1} \rceil)}^{\min(s,n-s-k+2)} p(n - s, k - 1, j) \tag{4.35}$$

The resulting procedure is Algorithm 4.26. As before we initialize the array to 1's and stop the recursion if $n = k$ or $k = 1$. The initial call is `P(n,k,s,1)`. To generate all partitions of n into exactly k parts call `P(2*n,k+1,n,1)` and ignore the contents of `p[1]` (which is always n).

```
procedure P ( n, k, s, t : ℕ);
local j, lower : ℕ;
begin
    p_t := s;
    if k = 1 or n = k then PrintIt
    else
        lower := max(1, ⌈(n − s)/(k − 1)⌉)
        for j := lower to min(s, n − s − k + 2) do P( n − s, k − 1, j, t + 1 );
    p_t := 1;
end{of P};
```

Algorithm 4.26: Generation of all partitions of n into k parts with largest part s.

What is the running time of Algorithm 4.26? It depends upon the value of the parameters and is certainly not CAT for some values. This is because another type of degree one path in the computation tree is now possible, namely when $n = ks$ (which gives rise to an s-chain of length k). Even when generating all partitions into a fixed number of parts it is not CAT; the maximum number of calls seems to occur when $k = n/2$. For `P(2*n,n div 2+1,n,1)` the number of recursive calls per partition appears to be $\Theta(\sqrt{n})$. The author is unaware of any *simple* CAT algorithm for generating partitions with a fixed number of parts using the natural representation. However, a rather complicated algorithm is presented in the next chapter in Section 5.8.

We now develop an algorithm for listing numerical partitions in lexicographic order when the multiplicity representation is used. The main idea behind the algorithm is that at most only the last three $[v_i, m_i]$ pairs change from one partition to the next. The index l indicates the last $[v_i, m_i]$ pair; i.e., v_l is the smallest part in the partition. There are four basic cases that must be considered as shown in the table below in the "condition" column; note that they are mutually exclusive and exhaustive. For each case the lexicographically next partition is shown in the "action" column. Each condition has two possible outcomes depending whether there is a part equal to 1 in the lex successor or not.

condition	**action**
$m_l = 1$ and $v_{l-2} > v_{l-1} + 1$	$[v_{l-2}, m_{l-2}][v_{l-1}, m_{l-1}][v_l, m_l]$
	$\rightarrow [v_{l-2}, m_{l-2}][v_{l-1} + 1, 1][1, (m_{l-1} - 1)v_{l-1} + v_l - 1]$
	$\rightarrow [v_{l-2}, m_{l-2}][v_{l-1} + 1, 1]$ if $v_l = m_{l-1} = 1$
$m_l = 1$ and $v_{l-2} = v_{l-1} + 1$	$[v_{l-2}, m_{l-2}][v_{l-1}, m_{l-1}][v_l, m_l]$
	$\rightarrow [v_{l-2}, m_{l-2} + 1][1, (m_l - 1)v_{l-1} + v_l - 1]$
	$\rightarrow [v_{l-2}, m_{l-2} + 1]$ if $v_l = m_{l-1} = 1$
$m_l > 1$ and $v_{l-1} > v_l + 1$	$[v_{l-1}, m_{l-1}][v_l, m_l]$
	$\rightarrow [v_{l-1}, m_{l-1}][v_l + 1, 1][1, (m_l - 1)v_l - 1]$
	$\rightarrow [v_{l-1}, m_{l-1}][2, 1]$ if $v_l = 1$ and $m_l = 2$
$m_l > 1$ and $v_{l-1} = v_l + 1$	$[v_{l-1}, m_{l-1}][v_l, m_l]$
	$\rightarrow [v_{l-1}, m_{l-1} + 1][1, (m_l - 1)v_l - 1]$
	$\rightarrow [v_{l-1}, m_{l-1} + 1]$ if $v_l = 1$ and $m_l = 2$

The procedure *Next* of Algorithm 4.27 is iterative and simply implements the table above. Initially we set $[v_{-1}, m_{-1}]$:=$[0, 0]$, $[v_0, m_0]$:=$[n + 1, 0]$, $[v_1, m_1]$:=$[1, n]$, and l:=1. This algorithm is clearly CAT since there are no loops; the number of operations between

```
procedure Next;
local sum : ℕ;  global l : ℕ;
begin
    sum := v_l m_l;
    if m_l = 1 then
        l := l − 1;   sum:=sum + v_l m_l;
    if v_{l−1} = v_l + 1 then
        l:=l − 1;   m_l:=m_l + 1
    else [v_l, m_l]:=[v_l + 1, 1];
    if sum > v_l then
        [v_{l+1}, m_{l+1}]:=[1, sum − v_l];   l:=l + 1;
    if l = 0 then done := true
end{of Next};
```

Algorithm 4.27: *Next* for lexicographic numerical partitions using multiplicity representation.

successive partitions is bounded by a constant. Usually loopless algorithms are not so easy to come by; further examples will be found in the next chapter.

Ranking numerical partitions

Here we develop a ranking algorithm for partitions of n in lexicographic order (in the natural representation). What is the rank of 7+4+4+1+1+1? All partitions of 18 with largest part less than 7 precede it, as do all partitions of 18 with largest part 7 and second largest part less than 4; this later number is the same as the number of partitions of $18-7=11$ whose largest part is less than 4. Continuing this process, all partitions with largest 2 parts $7+4$ and third largest part less than 4 precede it, and so on.

Define $\bar{p}(n,k)$ to be the number of partitions of n whose largest part is at most k; in other words,

$$\bar{p}(n,k) \;=\; \sum_{j=1}^{k} p(n,j).$$

These numbers inherit the recurrence relation (4.33); in fact from (4.34), $\bar{p}(n,k) = p(n+k,k)$ (combinatorial proofs are also readily derived). Continuing our example,

$$\begin{aligned} \mathit{rank}(7+4+4+1+1+1) &= \bar{p}(18,6) + \mathit{rank}(4+4+1+1+1) \\ &= \bar{p}(18,6) + \bar{p}(11,3) + \mathit{rank}(4+1+1+1) \\ &= \bar{p}(18,6) + \bar{p}(11,3) + \bar{p}(7,3) + \mathit{rank}(1+1+1). \end{aligned}$$

This suggest that we define s_i to be the sum of the parts with index at least i.

$$s_i \;=\; \sum_{j\geq i} p_j$$

The rank is then evidently given by the following equation.

$$\mathit{rank}(p_1+p_2+\cdots) \;=\; \sum_{i\geq 1} \bar{p}(s_i, p_i-1) \;=\; \sum_{i\geq 1} p(s_i+p_i-1, p_i-1)$$

Computing the s_i values takes $O(n)$ additions and if a table of the $\bar{p}$ numbers has been precomputed (a $O(n^2)$ computation), then evaluating the rank takes time $O(n)$ (if arithmetic operations are given unit cost).

4.9 Generalized Settings

4.9.1 Wilf's Generalized Setting

Many of the preceding examples relied upon a simple recurrence relation in two parameters. Wilf in [448] and [449] gave a general setting for handling such recurrences. Extensions of this idea may be found in Kelsen [209] and Williamson [453], [455].

4.9.2 The Generalized Setting of Flajolet, Zimmerman, and Cutsem

4.10 Listing solutions to problems solved by dynamic programming

This subsection is yet to be written. It will include some examples like the LCS problem and it's applications in computational biology.

4.11 Retrospective

Approach to Generation of Elementary Objects

1. Find a recurrence relation for the objects that
   (a) Uses only multiply and add; no divide or subtract.
   (b) Is strictly greater than zero in all cases.
   (c) Admits a combinatorial proof in terms of some natural representation of the objects.
2. Write a recursive procedure with the same structure as the recurrence relation. This is usually easy. If necessary, rewrite the procedure so that the computation can be divided up in such a way that each recursive call is assigned a constant amount of computation. This can be difficult, if not impossible.
3. To analyze the resulting algorithm count recursive calls. The original recurrence relation is modified by adding +1 to each non-constant case.

4.11.1 Why recusive and iterative algorithms have similar analyses

4.12 Ideals and Linear Extensions of Posets

In this section we present some algorithms for generating the linear extensions and ideals of partially ordered sets. These algorithms can be used to solve various scheduling problems. Unlike all of the previously presented algorithms, these algorithms are not or are not known

WHAT WAS MEANT TO GO HERE???

Figure 4.18: A typical computation tree of a recursive algorithm generating strings in lexicographic order.

to be CAT nor are they precisely lexicographic; however, they have a definite lexicographic "feel" to them.

4.12.1 Varol-Rotem Algorithm for Linear Extensions

Let $\mathcal{P}$ be a poset on the set $\{1, 2, \ldots, n\}$ labeled so that $12\cdots n$ is a linear extension of $\mathcal{P}$. Now suppose that a list L_{n-1} of all extensions of $\mathcal{P} - \{n\}$ was available. This list could be expanded to a list L_n of all extensions of $\mathcal{P}$ as follows. Let π be an extension from the list L_{n-1} and k be the maximum index for which $\pi_k \prec n$. Assume that there is a fixed element π_0 that is less than all other elements of the poset. Then, since $\pi_j \parallel n$ for all $j > k$, the permutations

$$\pi_1 \cdots \pi_k \pi_{k+1} \cdots \pi_j n \pi_{j+1} \cdots \pi_{n-1},$$

as j varies from n to k, form a list of all extensions of $\mathcal{P}$ in which the elements of $[n-1]$ appear in the order π. At the end of this process we have the permutation

$$\pi_1 \cdots \pi_k n \pi_{k+1} \cdots \pi_{n-1}.$$

By now rotating to the left the elements $n\pi_{k+1} \cdots \pi_{n-1}$ we obtain the original permutation π with the n appended at the end (and out of the way). We may then proceed to the successor π' of π in the list L_{n-1} and expand π' in exactly the same way. Inductively, we conclude that this recursive process will generate all linear extensions of $\mathcal{P}$. Furthermore, it is easily implemented as illustrated in Algorithm 4.28. Array π is initialized to be the identity permutation $12\cdots n$ and the initial call is VR(1) Recall that $x \parallel y$ means that poset elements x and y are incomparable. A non-recursive implementation is the subject of Exercise 77.

```
procedure VR ( k : integer );
local i : integer;
begin
        if k > n then PrintIt
        else
             VR( k + 1 );
             i:=k;
             while π_{i-1} ∥ π_i  do
                  π_{i-1} :=: π_i;
                  VR( k + 1 );
                  i := i - 1;
             RotateL( π_i, π_{i+1}, ..., π_k );
        end;
```

Algorithm 4.28: Recursive Varol-Rotem algorithm for generating linear extensions.

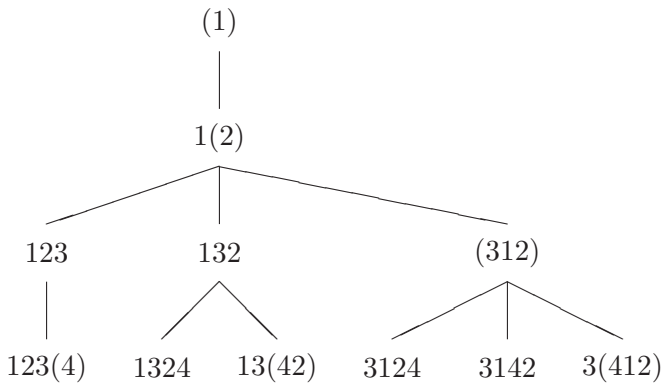
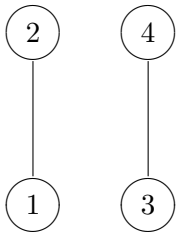


Figure 4.19: Computation tree of Varol-Rotem algorithm on the example poset.

Example: For the poset below the table to its right shows the permutations produced by the algorithm and their inverses.



π	π^{-1}
1234	1234
1324	1324
1342	1423
3124	2314
3142	2413
3412	3421

Figure 4.19 shows the computation tree associated with our example. The digits that are in parentheses are those that get rotated as the computation backs up; i.e., think of them being done in a postorder traversal of the tree. Observe that the rotations have the property that the number of elements involved in the rotation is equal to the number of siblings of that node (a node is a sibling of itself). This observation holds true in general, and thus it follows that the sum of the costs of the rotations is equal to the number of nodes in the computation tree. In the example computation tree there are 11 nodes and there are 11 numbers in parentheses. As an aside, note that if a linked representation is used, then rotations take constant time.

What is the theoretical running time of this algorithm? We just argued that it depends upon the number of nodes in the associated computation tree, which is clearly $O(ne(\mathcal{P}))$. Whether it is $O(e(\mathcal{P}))$ depends on the number of nodes of degree one, which depends upon the poset and the linear extension used to initialize the algorithm. Note that the number of degree one nodes at level $n-1$ is equal to the number of extensions π of $\mathcal{P} - \{n\}$ for which $\pi_{n-1} \prec n$.

Let $\mathcal{Q}$ be the poset consisting of a chain $x_1, x_2, \ldots, x_n$ and an element y incomparable with each element of the chain. If the initial linear extension is $yx_1x_2\cdots x_n$, then $n(n-1)/2+1$ rotations are used to produce the $n+1$ linear extensions of $\mathcal{Q}$. On the other hand, if the initial linear extension is $x_1x_2\cdots x_ny$, then only n rotations are required, and the algorithm runs in time $O(n)$. Thus, the algorithm is quite dependent upon the linear

extension used to initialize the algorithm. However, the algorithm is highly recommended. It is simple and quite fast in practice.

In the example the extensions and their inverses both get listed in lexicographic order, but this is not always the case. Consider an antichain on three elements. Its extensions get listed in the order 123, 132, 312, 213, 231, 321, which is not lexicographic, and neither are its inverses 123, 132, 231, 213, 312, 321. There is a sense, however, in which the algorithm is lexicographic. Let $r(\pi) = r_1 r_2 \cdots r_n$ be the string in which r_i is the number of elements of π that are to the right of i and less than i. In our example, the successive values of r are 0000, 0010, 0011, 0020, 0021, 0022. The following lemma is easily proven by induction.

LEMMA 4.7 *If $\pi_1, \pi_2, \ldots$ is the list of extensions output by procedure VR, then the list $r(\pi_1), r(\pi_2), \ldots$ is lexicographically increasing.*

Since π can be recovered from $r(\pi)$, the converse of Lemma 4.7 is also true.

4.12.2 Algorithms for Listing Ideals

Now consider the problem of generating all ideals (or filters) of a given input poset $\mathcal{P}$. Recall that an ideal I of a poset $\mathcal{P}$ is a subset of $S(\mathcal{P})$ with the property that $x \in I$ and $y \preceq x$ imply that $y \in I$. Given $x \in S(\mathcal{P})$, the ideals of $\mathcal{P}$ may be classified according to whether they contain x or not. An ideal that contains x must contain every element of $\downarrow x$, the set of elements below x. An ideal that doesn't contain x must not contain any element of $\uparrow x$, the set of elements above x. Stated formally,

$$J(\mathcal{P}) \;=\; (\downarrow x \cup J(\mathcal{P} - \downarrow x)) \;\uplus\; J(\mathcal{P} - \uparrow x), \tag{4.36}$$

where $\downarrow x \cup J(\mathcal{P} - \downarrow x)$ means the set $\{\downarrow x \cup I : I \in J(\mathcal{P} - \downarrow x)\}$.

This decomposition leads directly to Algorithm 4.29. The initial call is $\mathsf{GenIdeal}(\mathcal{P}, \varepsilon)$. Note that the computation tree is an extended binary tree since both terms of (4.36) are non-empty. Properly implemented, the amortized running time of this algorithm is $O(n)$ per ideal. Assume that $S(\mathcal{P}) = [n]$ (and thus that $S(\mathcal{Q}) \subseteq [n]$). Ideal I is implemented as a bitstring $\mathbf{b} = b_1 b_2 \cdots b_n$, where $b_i = 1$ if and only if $i \in I$. Then the three parameter updates $I{:=}I \cup \downarrow x$, $\mathcal{Q}{:=}\mathcal{Q} - \downarrow x$, and $\mathcal{Q}{:=}\mathcal{Q} - \uparrow x$ can be implemented in time $O(n)$, given bitstrings representing $\downarrow x$ and $\uparrow x$. The set of all $2n$ bitstrings representing $\downarrow x$ and $\uparrow x$ can be pre-computed and stored in time (and space) $O(n^2)$. Assuming that the selection of x at line (I6) takes time $O(n)$, the overall running time of Algorithm 4.29 is $O(n^2 + nN)$, where $N = |J(\mathcal{P})|$.

```
(I1)   procedure GenIdeal ( Q : Poset ); I : Ideal );
(I2)   local x : PosetElement;
(I3)   begin
(I4)       if S(Q) = ∅ then PrintIt( I );
(I5)       else
(I6)           x:= some element of S(Q);
(I7)           GenIdeal( Q− ↓x, I∪ ↓x );
(I8)           GenIdeal( Q− ↑x, I );
(I9)   end{ of GenIdeal};
```

Algorithm 4.29: Generic algorithm for generating ideals.

In the case were n is less than the word size of the computer, then the three updates may be computed in constant time, and except for the selection of x at line (I6), the algorithm will run in constant amortized time. This is an important consideration, to be kept in mind as we discuss the theoretically faster variants below.

Specific algorithms are obtained by specifying precisely how the element x is to be chosen at line (I6). We discuss two variants of this algorithm, one due to Steiner [404] and the other to Squire [399].

Steiner Variant

In the Steiner variant, x is is chosen to be a minimal element of $\mathcal{P}$. This has no discernable effect on the running time of the algorithm but it does simplify the recursive call at line (I7), which becomes $\mathsf{GenIdeal}(\mathcal{Q}-\{x\}, I\cup\{x\})$. The remaining concern is how to efficiently select a minimal element. One possibility is to maintain, for each element x in the poset, a value $c(x)$, the number of elements that x covers; x is a minimal element if and only if $c(x) = 0$. These $c()$ values can be initialized in time $O(n^2)$ and updated in time $O(n)$, thereby preserving the $O(n^2 + nN)$ running time of the generic algorithm.

Squire Variant

In the Squire variant, x is chosen to be the middle element in some linear extension of $\mathcal{Q}$ — a selection rule that leads to logarithmic cost per ideal.

We were somewhat sloppy in our analysis of Algorithm 4.29. The running time of the algorithm is actually proportional to the sum of $|S(\mathcal{Q})|$, taken over all calls in the computation tree. Imagine a global variable *cost*, initialized to 0, and updated at line (I3.5) by $cost{:=}cost + |S(\mathcal{Q})|$. The total running time to produce all ideals is proportional to the final value of *cost*.

Recall that a *complete* binary tree is an extended binary tree in which all leaves are at the same level. We will show that the computation tree has many complete binary subtrees and use this to get a bound on *cost*.

LEMMA 4.8 *In the computation tree of* $\mathsf{GenIdeal}(\mathcal{P}, \varepsilon)$*, with the Squire rule used to select* x*, the call* $\mathsf{GenIdeal}(\mathcal{Q}, I)$ *is the root of a complete binary subtree of height at least* $\lfloor \lg(1 + |S(\mathcal{Q})|)\rfloor$.

PROOF: If x is chosen to be the middle element in a linear extension of $\mathcal{Q}$ then clearly

$$|S(\mathcal{Q}- \downarrow x)| \geq \left\lfloor \frac{|S(\mathcal{Q})| - 1}{2} \right\rfloor \quad \text{and} \quad |S(\mathcal{Q}- \uparrow x)| \geq \left\lfloor \frac{|S(\mathcal{Q})| - 1}{2} \right\rfloor .$$

From these bounds the lemma is easily proven by induction. □

Imagine the cost $s = |S(\mathcal{Q})|$ of the node $(\mathcal{Q}, I)$ of the computation tree as being distributed evenly over the nodes of the complete binary subtree, call it $T = T(\mathcal{Q}, I)$, of height $\lg s$, as guaranteed by Lemma 4.8. Then each node x of T is assigned a unit cost by the call $\mathsf{GenIdeal}(\mathcal{Q}, I)$, but x may be assigned costs by other calls. How much cost can be assigned to x in total? Note that it can be assigned a cost only by an ancestor, and further, only by an ancestor that occured at most $\lg n$ generations previously. Thus the total cost assigned to a node is bounded above by $\lg n$, which means that the total cost of the entire computation tree is at most $2N \lg n$; i.e., the amortized cost per ideal is $O(\lg n)$.

The only remaining detail is to explain how to implement the selection rule and update the parameters in time $|S(\mathcal{Q})|$. An initial linear extension L of $\mathcal{P}$ may be computed in time $O(n^2)$ using standard algorithms (cf. Section 2.11.3). We may, in fact, assume that the elements of $\mathcal{P}$ have been labeled so that $L = 1, 2, \ldots, n$. The extension L is stored as a linked list, and is used to represent $\mathcal{Q}$ as the algorithm is running. Note that removing x from L results in a linear extension of $\mathcal{P} - x$, so that we simply delete or add elements to L as the recursion is proceeding. For each x, the sets $\downarrow x$ and $\uparrow x$ are precomputed and stored as bitstrings. Parameter I is a bitstring. The structures $\mathcal{Q}$ and I are not explicitly passed as parameters; we may think either as leaving them global or as passing pointers to them. In either case the parameters are changed when the recursive calls are made, and restored when a recursive call is completed. Note that we use a simple linear scan to find the median element of the current extension L; a logarithmic search is not necessary.

Let us consider the update $\mathcal{Q}{:=}\mathcal{Q} - \downarrow x$ in detail. We simply traverse L while updating a local list T, initially empty, and equal to $L \cap \downarrow x$ immediately before the recursive call is made. Upon encountering an element k on L, we check $\downarrow x[k]$, the kth bit of the bitstring representing $\downarrow x$; if it is 1, then delete k from L and add it to the end of T, otherwise, do nothing. At the end of this traversal, list L has been transformed into a new list, L', representing $\mathcal{Q} - \downarrow x$. Now GenIdeal is called. After the call, T and L' are merged, using the standard linear time algorithm for merging sorted linked lists, to obtain the original list L. These updates of L and T clearly take time $O(|S(\mathcal{Q})|)$.

4.13 Exercises

Questions about lexicographic order

1. **[1−]** True or false: In lexicographic order, for strings α, β, γ, and δ, if $\alpha < \beta$ and $\gamma < \delta$, then $\alpha\gamma < \beta\delta$?

2. **[1]** Imagine a rather peculiar dictionary that contains exactly the odd numbers less than 10^{10} listed in lexicographic order.Each number is written out in English without using punctuation or the word "and", or "a" for "one". Thus, 105 is written as *one␣hundred␣five* and not *a␣hundred␣five* or *one␣hundred␣and␣five*; also 45 is written as *forty␣five* and not *forty-five*. Assume the usual ordering of characters with the blank (␣) as the smallest character. What is the first number in the dictionary? What is the last number in the dictionary? Answer the preceding two questions if the words are listed in colex order.

3. **[1]** Under what conditions on a set of strings S are its lexicographic prefix tree and its colex suffix tree identical? The tree of Figure 4.1 is such an example.

Questions about subsets

4. **[1]** The procedure `Subset` has the somewhat undesirable property that the call `Subset(n)` does not generate the subsets of $\{1, 2, \ldots, n\}$. Use colex order and make a few trivial changes to `Subset` so that this undesirable property is eliminated.

5. **[1+]** Show that using *Next* of Algorithm 4.1 to generate the first m ($1 \leq m \leq 2^n$) subsets results in a CAT algorithm.

6. **[1]** Develop a $O(n)$ unranking algorithm for subsets of an n-set in lexicographic order as represented by bitstrings of length n (i.e., the set $\mathbf{B}(n)$).

7. **[1]** Re-write Algorithms 4.1 and 4.2 so that they produce the actual elements of each subset.

8. **[1]** The same ideas that were used to generate subsets can be used to generate what are sometimes called "product spaces". That is, suppose that we wish to generate all t-tuples from $A_1 \times A_2 \times \cdots \times A_t$, where each A_i is a finite set, which may be thought of as $A_i = \{0, 1, \ldots, n_i\}$, for some natural number n_i. Given $n_1, n_2, \ldots, n_t$, show how to generate the associated product space by a CAT algorithm.

9. **[1+]** An (n, m)-*punctured partition* (e.g., Damiani, D'Antona and Naldi [79]) is a string $e_1 e_2 \cdots e_n$ such that the following three conditions hold.

   - $-m + 1 \leq e_i$ for $i \in [n]$, and
   - $e_1 + e_2 + \cdots + e_n = n$, and
   - $e_1 + e_2 + \cdots + e_k = k$ if $e_k > 0$.

   (a) Show that the number of punctured partitions is $(m+1)^n$ by demonstrating an explicit bijection between $(n+1, m)$-punctured partitions and $(m+1)$-ary strings of length n. (b) Develop a CAT algorithm to generate all (n, m)-punctured partitions.

Questions about combinations

10. **[1]** Give a simple proof by induction of Lemma 4.1.

11. **[2]** Develop and implement a CAT algorithm to generate the elements of $\mathbf{B}(n, k)$ where PET #2 is used, but not PET #1. Exactly how many recursive calls are made?

12. **[1+]** The algorithm *Next* below generates the elements of $\mathbf{B}(n, k)$ in lexicographic order. Analyze the amortized running time of this algorithm by computing the sum of $p + q + 1 = n - j + 1$ over all bitstrings in $\mathbf{B}(n, k)$.

```
procedure Next;
{Assumes 0 < k ≤ n and a_0 < 0.}
local j : ℕ;
begin
    j := n;
    while b_j = 0 do j := j − 1;
    p:=n − j;
    while b_j = 1 do j := j − 1;
    q:=n − j − p;
    if j = 0 then done := /TRUE;
    b_j := 1;
    for i := 1 to p + 1 do b_{i+j} := 0;
    for i := q − 1 downto 1 do b_{n−i+1} := 1;
end {of Next};
```

13. This exercise is a continuation of the previous one. Observe that there may be some bits to the right of a_j that do not change, and yet *Next* scans over them and sets them anyways. The scanning can be eliminated by storing the lengths of runs and the resetting to the same value can be avoided as well. (a) **[1+]** Modify the algorithm so that the amount of computation is $O(t(n,k))$, where $t(n,k)$ is one-half of the number of bits that change in a lexicographic listing of the elements of $\mathbf{B}(n,k)$. (b) **[1]** Derive a recurrence relation for $t(n,k)$ and (c) **[2]** prove that $t(n,k)$ is $O(\binom{n}{k})$.

14. **[1+]** Develop a CAT algorithm for generating $\mathbf{F}(n,k)$, the set of all strings in $\mathbf{B}(n,k)$ that do not have two consecutive 1's. For example $\mathbf{F}(n,k) = \{0101, 1001, 1010\}$.

15. **[2]** Find a short recursive algorithm for generating all those k-subsets of an n-set whose sum is p, where each k-subset is represented as a bitstring. In other words, generate the elements of the set $\{b_1 b_2 \cdots b_n \in \mathbf{B}(n,k) \mid \sum_{i=1}^{n} i b_i = p\}$. Is your algorithm CAT?

16. **[2]** Use a run-length encoding of subsets with a given sum to develop a loopless algorithm for generating them.

17. **[1+]** Consider equivalence classes of bitstrings that are equivalent under reversal. For example, $\{011, 110\}$ is one equivalence class; each equivalence class has one or two elements. We wish to generate the lexicographically smallest representative of each class. With $\mathbf{B}(n) = \Sigma_2^n$, let $\mathbf{M}(n) = \{\mathbf{x} \in \mathbf{B}(n) : \mathbf{x} \leq \mathbf{x}^R\}$. What is $|\mathbf{M}(n)|$? Give a CAT algorithm to generate the elements of $\mathbf{M}(n)$.

18. **[2]** Consider equivalence classes of bitstrings with the fixed number of 1's that are equivalent under reversal. Let $\mathbf{M}(p,q) = \{\mathbf{x} \in \mathbf{B}(p+q,q) : \mathbf{x} \leq \mathbf{x}^R\}$. What is $|\mathbf{M}(p,q)|$? Develop a CAT algorithm to generate the elements of $\mathbf{M}(p,q)$.

19. **[3]** We now consider equivalence classes of bitstrings that are equivalent under reversal and/or complementation. Let $\mathbf{Y}(n) = \{\mathbf{x} \in \mathbf{B}(n) : \mathbf{x} \leq \mathbf{x}^R, \mathbf{x} \leq \overline{\mathbf{x}}, \mathbf{x} \leq \overline{\mathbf{x}}^R\}$. What is $|\mathbf{Y}(n)|$? Develop a CAT algorithm to generate the elements of $\mathbf{Y}(n)$.

20. **[*R*]** Develop a CAT algorithm to generate all 0-1 matrices with given row and column sums.

Questions about permutations

21. **[1+]** Prove that the n-queens program (Algorithm 3.1) with references to arrays `b` and `c` removed results in a CAT algorithm for generating all $n!$ permutations of $[n]$.

22. **[2]** Implement a $O(n \log n)$ algorithm for ranking and unranking permutations in lexicographic order.

23. **[3]** Develop an $O(n)$ algorithm for unranking permutations. HINT: Do not base your algorithm on lexicographic order; use an order derived from the algorithm in the first section of Chapter 10.

24. **[1]** Develop and analyze algorithms for listing, ranking, and unranking k-permutations of an n-set.

25. [**1+**] Develop a CAT algorithm for generating all up-down permutations in lexicographic order.

26. [**1**] Characterize the inversion strings of up-down permutations.

27. [**1+**] Defining $E(n,k)$ to be the number of up-down permutations π of $[n]$ for which $\pi_1 = k$, prove the recurrence relation

$$E(n,k) \;=\; E(n,k+1) + E(n-1,n-k).$$

28. [**1+**] Use the $E(n,k)$ numbers of the preceding exercise to develop an $O(n)$ ranking algorithm for the inversion strings of up-down permutations in lex order.

29. [**2**] Using a combinatorial interpretation of the recurrence relation $d_n = (n-1)(d_{n-1} + d_{n-2})$, develop a CAT algorithm for generating all derangements of the set $\{1, 2, \ldots, n\}$.

30. Let $D(n,k)$ denote the number of derangements of $[n]$ with k cycles. [**1**] Prove that $D(n,k) = (n-1)(D(n-1,k) + D(n-2,k-1))$. [**2+**] Develop a CAT algorithm for generating the associated set of derangements.

31. A k by n *Latin rectangle* is a $k \times n$ matrix with entries from $[n]$ such that (a) every row is a permutation of $[n]$ and (b) every column is a k-permutation of $[n]$. A Latin rectangle is *reduced* if the first row is the identity permutation $12\cdots n$ and the first column is $12\cdots k$. The number of $2 \times n$ Latin rectangles is $n!d_n$ where d_n is the number of derangements of n; the number of reduced rectangles is d_{n-1}. [**1**] Show that that there exists at least one reduced Latin rectangle for all $1 \le k \le n$. [***R*−**] Develop a CAT algorithm to generate all reduced k by n Latin rectangles.

32. [***R*−**] Let $M(n,k)$ denote the number of permutations of $[n]$ that have k monotone runs (sometimes called "alternating runs"). Prove that

$$M(n,k) = k \cdot M(n-1,k) + 2 \cdot M(n-1,k-1) + (n-k) \cdot M(n-1,k-2)$$

and use this recurrence relation to develop a CAT algorithm for generating permutations with a fixed number of monotone runs.

33. [**2+**] Develop a CAT algorithm for generating all involutions, where each involution is represented as a permutation in one-line notation.

34. [***R*−**] Develop a CAT algorithm for generating all permutations with a given number of inversions. Develop a CAT algorithm for generating all permutations with a given index.

35. [**1+**] Develop a CAT algorithm for generating all permutations with a given number of cycles. [**2+**] Develop a CAT algorithm for generating all permutations with a given number of left-to-right maxima.

36. [**1+**] A *k-composition* of an integer n is a solution to the equation $x_1 + x_2 + \cdots + x_k = n$ in natural numbers. Develop listing, ranking, and unranking algorithms for compositions.

37. [**1+**] Prove equation (4.14).

Questions about trees

38. **[1]** Develop an iterative CAT algorithm for generating the elements of $\mathbf{T}(n,k)$.

39. **[2]** In reference to Algorithm 4.16, prove that the average value of $p+q+1$ is $C_1 + C_2 + \cdots + C_{n+1}$.

40. **[2]** Use the cycle lemma to prove (4.18).

41. **[1−]** Explain why $T(n,k)$ is the number of ordered forests with n nodes and k trees.

42. **[1]** Prove an orthogonality relation for the $T(n,k)$ numbers; i.e., if T is the matrix of the $T(n,k)$ numbers, then find T^{-1} such that $TT^{-1} = T^{-1}T = I$.

43. **[1+]** Show that the $T(n,n-k)$ numbers from the ranking algorithm for binary trees can be computed on the fly so that the resulting algorithm uses $O(n)$ arithmetic operations $O(n)$ space.

44. **[2+]** What is the average level number of the 2nd leaf of a random binary tree; of the pth leaf as $n \to \infty$?

45. **[1+]** For a binary tree let T, let T_L and T_R denote its left and right subtrees. The following order on binary trees has been called *B-order:* $T < T'$ if

    (a) $T = \emptyset$ and $T' \neq \emptyset$ or
    (b) $T_L < T'_L$ or
    (c) $T_L = T'_L$ and $T_R < T'_R$.

    Show that Algorithm 4.16 generates binary trees in B-order.

46. **[2]** Let $|T|$ denote the number of nodes in a binary tree. The following order on binary trees has been called *A-order* (or natural order): $T < T'$ if

    (a) $|T| < |T'|$ or
    (b) $|T_L| = |T'_L|$ and $|T_R| < |T'_R|$.

    Develop a CAT algorithm for generating binary trees in A-order. [Hint: represent a binary tree as a preorder sequence of the number of descendants of each node.]

47. An ordered forest of $n+1$ nodes may be encoded by traversing the forest in preorder and recording the (out)degree of each node as a sequence $\mathbf{a} = a_1, a_2, \ldots, a_n$. This is a generalization of our encoding of extended binary trees, and we will omit the final 0 as before. Let $F(n_0, n_1, \ldots, n_t) = F(\mathbf{n})$ denote the set of encodings of all ordered forests that have n_i nodes of out-degree i, for $i = 1, \ldots, t$ and $n_0 + 1$ leaves. Let $T(\mathbf{n})$ denote the subset of $F(\mathbf{n})$ consisting only of the encodings of trees. **[1]** Show that $F(\mathbf{n})$ is non-empty if and only if $\sum(1-i)n_i \geq 0$. **[1]** Show that $\mathbf{a} \in T(\mathbf{n})$ if and only if, for $k = 1, \ldots, n$, $\sum_{1 \leq i \leq k} a_i \geq k$. **[2]** Develop a CAT algorithm for listing the elements of $T(\mathbf{n})$. The number of such ordered forests may be inferred from (2.16).

48. **[2+]** Give an $O(n)$ algorithm that takes as input an ordered tree T and transforms it into the equivalent canonic rooted tree. Note that this algorithm implies that we can test for the isomorphism of two rooted trees in $O(n)$ time.

49. [**2**] Prove that the array `par` of Algorithm 4.18 is generated in relex order.

50. [**1+**] Write an unranking algorithm for B-trees.

51. [**2**] Show that the algorithms for generating and ranking B-trees can be extended to the following class of trees. Under what conditions is the generation algorithm CAT?

    The more general case consists of trees with all leaves at one level, the maximum (out) degree m and minimum degree l_0 at the root, and minimum degree l at the other internal nodes, where $1 < l_0 \leq l \leq m$.

52. [**1**] Compute the $b(s, d)$ table when $m = 5$ and $m = 6$.

53. [***R*−**] Develop a CAT algorithm for generating AVL (height-balanced) trees.

54. [**2**] A *red-black* tree is an extended binary tree in which every node is either red or black, every leaf is black, every black internal node has two red children, and every path from a node to a descendant leaf contains the same number of black nodes. Such trees are characterized by the property that the longest path from any node v to a leaf is at most twice as long as the shortest path from v to a leaf. Develop a (CAT) algorithm for generating a natural representation of red-black trees. [Hint: There is a simple relation between red-black trees and B-trees with $m = 4$.]

55. [**1+**] This question refers to the section on generating rooted trees. Compute experimental evidence that the amortized value of $n - q$ is not constant, but that the amortized value of $n - p$ is constant.

56. [**1+**] Prove that `par` is lexicographically decreasing in the procedure *Next* of Algorithm 4.18. from the section on rooted trees.

57. [**3**] Develop a *recursive* version of the Beyer-Hedetniemi algorithm for generating rooted trees. The algorithm should also have the CAT property and need only use the parent array, and not use the L array.

58. [**3**] Develop a CAT algorithm to generate all rooted trees in which no node has more than two children.

59. [**2**] Implement Algorithm 4.19 so that it runs in constant amortized time.

60. [***R*−**] Develop a CAT algorithm to generate all rooted trees in which every non-leaf node has at least two children. [***R*−**] Develop a CAT algorithm to generate all rooted trees with a given degree sequence.

61. [**1**] Show that Wilf's generalized setting can be applied to well-formed parentheses strings; i.e., to generate and rank the set $\mathbf{T}(n)$.

Questions about set partitions

62. [***R***] Is there a closed form expression for the numbers $t(n, k)$ from the set partition section?

63. [**2**] Investigate what happens when the recursion stops at $k = 1$ in Algorithm 4.23 (instead of when $n = k$). Is the resulting algorithm CAT?

64. [**1**] Give a one line proof that $\sum_{i=1}^{n-1} i \cdot R(n-i, i-1) = B_n - 1$.

65. [**1**] Develop an efficient algorithm for finding the rank r RG string of length n in lexicographic order.

Questions about numerical partitions

66. [**1**] What numerical partition of 25 has rank 1000 in lexicographic order?

67. [**1+**] Write an unranking procedure that finds the rank r numerical partition of n in lexicographic order. How many arithmetic operations does your procedure use in the worst case?

68. [**2**] Write a loopless algorithm to generate numerical partitions in the multiplicity representation in relex order and in colex order. Which order resulted in the simplest algorithm: lex, colex, or relex?

69. [**1+**] Give an efficient algorithm to convert a partition in the natural representation to its conjugate. Give an efficient algorithm to convert a partition in the multiplicity representation to its conjugate.

70. [**2**] Write a short procedure to generate all partitions of n into exactly k parts in colex order. Your recursive calls should be modeled after recurrence relation (4.33). Is the resulting algorithm CAT?

71. [***R*−**] Determine the asymptotic number of nodes in the computation trees of Algorithms 4.24 and 4.25 when used to generate all partitions of n.

72. [**1+**] Develop a CAT algorithm for generating all partitions of n into odd parts (all parts are odd) using the natural representation.

    [**1+**] The number of partitions of n into odd parts is equal to the number of partitions of n into distinct parts. Prove this equality.

    [***R*−**] Develop a CAT algorithm for generating all partitions of n into distinct parts using the natural representation.

73. [**2+**] A *binary partition* of 2^n is a solution $a_0, a_1, \ldots, a_n$, in numbers, to the equation $2^n = a_0 2^0 + a_1 2^1 + \cdots + a_n 2^n$; i.e., it is a partition of 2^n into powers of 2. Give a CAT algorithm for listing binary partitions in lex order.

74. [***R*−**] Develop a CAT algorithm for generating all compositions $n = a_1 + a_2 + \cdots$ such that (a) $a_1 = 1$, (b) for $j > 1$, $0 \le n_j \le 2n_{j-1}$, and (c) $a_j = 0$ implies $a_{j+1} = 0$. [**1**] Explain the connection between these compositions and level numbers of binary trees. [**1+**] Explain the connection between these compositions and non-positive powers of two summing to 1.

75. [***R*−**] A *score vector* of a tournament is a sequence $a_1 \le a_2 \le \cdots \le a_n$ satisfying $\sum_{i=1}^{k} a_i \ge \binom{k}{2}$ for $k = 1, 2, \ldots, n$, with equality holding for $k = n$. Develop a CAT algorithm for generating all score vectors.

Questions about linear extensions and ideals

76. [**2**] Develop an algorithm for generating all linear extensions of a partially ordered set in lexicographic order. [***R*−**] Is your algorithm CAT?

77. [**1+**] Implement the Varol-Rotem algorithm iteratively.

78. [**2**] Find an example where the final permutation produced by the Varol-Rotem algorithm is not the linear extension with the greatest number of inversions. [***R*−**] Given an n element poset labeled so that $12\ldots n$ is a linear extension, is it NP-hard to determine an extension that has the greatest number of inversions?

79. [***R*−**] For each poset $\mathcal{P}$, is there an initial linear extension that causes the Varol-Rotem algorithm to run in CAT time?

80. [**1**] Finish the inductive argument of Lemma 4.8.

81. 2 Explain how to compute all the principal upsets and principal downsets in time $O(n^2)$.

82. [***R***] Find a CAT algorithm for generating the ideals of a poset.

Miscellaneous questions

83. [**2**] Develop a CAT algorithm for generating all *contingency tables* with fixed marginals. That is, generate all r by c matrices of natural numbers with given row sums $k_1, k_2, \ldots, k_r$ and given column sums $n_1, n_2, \ldots, n_c$. HINT: base your algorithm on Algorithm 4.14.

84. [***R*−**] A *degree sequence* of a graph G is the sequence of degrees of the vertices of G arranged in non-decreasing order. There are several characterizations of such sequences. Develop a CAT algorithm to generate all degree sequences of length n.

85. A seemingly much harder problem than the previous is to generate all degree sequences of graphs with m edges (and no isolated vertices).These sequences are called *graphical partitions*. [**3+**] Develop a positive recurrence relation (or relations) for the number of graphical partitions. [**2+**] Based on the recurrence relation, develop a CAT algorithm for generating all graphical partitions.

86. [***R***] Given two n-by-n matrices $A = [a_{ij}]$ and $B = [b_{ij}]$, a *stable marriage* is a permutation π such that there does not exist a pair i, j with $a_{i,\pi(i)} < a_{i,\pi(j)}$ and $b_{\pi(j),j} < b_{\pi(j),i}$. Develop a CAT algorithm that takes as input A and B and then generates all stable marriages.

87. [***R*−**] A n by n *magic square* is an arrangement of the integer of $[n^2]$ into a n by n matrix so that all column sums, all row sums, and the two diagonal sums are all equal. Develop a CAT algorithm for generating magic squares..

88. [***R*−**] There are many ways to fold a strip of n stamps, ranging from a zig-zag to a spiral. For example, there are 2 ways to fold 3 stamps, and 5 ways to fold 4 stamps. Devise a suitable representation for such foldings and develop a CAT algorithm for generating the foldings.

4.14 Bibliographic Remarks

Books

The books of Page and Wilson [296] and Rohl [347] contain algorithms for generating various types of combinatorial objects, mostly in lex order. The book of Even [115] also contains algorithms for listing combinatorial objects. The book of Skiena [390] describes algorithms for listing permutations, multiset permutations, combinations, all implemented in "Mathematica". The "Maple V" package "combinat" includes functions to produce exhaustive lists of permutation, combinations (of a multiset), compositions of an integer, and other more esoteric objects such as cosets of a group.

Subsets and Combinations

The algorithm for generating combinations in lexicographic order is by Mifsud [280]; see also Shen [385]. A comparison of algorithms for generating combinations was made by Akl [5] who concludes that Mifsud's algorithm is fastest. Another survey was made by Carkeet and Eades [51]. The idea of eliminating the while loop from Algorithm 4.3 was published by Dvořák [94]. The ranking formula (4.10) is due to Lehmer [244].

Applications of a subset generating algorithm are given by Stojmenović and Miyakawa [408].

Permutations

According to [340], the first "mention" of an algorithm for generating permutations is by L.L. Fisher and K.C. Krause in 1812 [129]. A fast algorithm for generating permutations in lexicographic order is Ord [292] (see also Ives [189]) The analysis of the number of transpositions used by Algorithm 4.11 is from [340].

A close variant of an inversion string is the *inversion table*, $b_1 b_2 \cdots b_n$, of a permutation, where b_i is the number of elements to the left of i that are greater than i; see Knuth [219], pg. 12. Also related is the *inversion vector* defined in Reingold, Nievergelt, and Deo [340]by letting b_i be the number of elements greater than π_i and to its left. Thus an inversion table is an inversion vector indexed differently.

The following references contain algorithms for generating permutations: Golomb [153], Knott [215], Knott [216], Orcutt [290], Rohl [348], Rohl [349], Roy [351], Lipski [250]. Algorithms for generating derangements are discussed in Akl [4].

A *rosemary permutation* is an equivalence class of permutations under rotations and reversal; they can be thought of as Hamilton cycles on the labeled complete graph K_n. The number of rosemary permutations of $[n]$ is $(n-1)!/2$. The generation of rosemary permutations was considered by Harada [169] and Read [334].

Bauslaugh and Ruskey [23] give CAT algorithms for generating up-down permutations lexicographically and for ranking the inversion strings of up-down permutations in time $O(n)$. A CAT algorithm for generating the inversion strings of permutations with given ups and downs in colex order was given in Roelants and Ruskey [430]. Roelants [428] gives a CAT algorithm for generating involutions represented by what he calls I-sequences. The trick for solving (4.11) is from Reingold, Nievergelt, and Deo [340].

A CAT algorithm for generating the permutations of a multiset in lexicographic order may be found in Roelants van Baronaigien and Ruskey [360]. Knuth [222] contains an

algorithm for generating all permutations of a multiset in which the number of inversions is bounded.

Savage and Barnes results get mentioned here.

There is an algorithm for ranking and unranking permutation in $O(n)$ arithmetic operations, but the underlying order is not lexicographic; see [287].

Binary trees, ordered trees

A great many papers have be written about generating binary trees using various sequence representations and many of these papers are closely related. Our discussion closely follows Ruskey [362]. A survey is given by Mäkinen [263] and relationships among several of these algorithms is demonstrated in Lucas, Roelants van Baronaigien and Ruskey [256].Lee, Lee, and Wong [242] give a CAT algorithm for generating binary trees of bounded height. Another algorithm for binary trees of bounded height may be found in Knuth [222] (pp. 153-154). Other references on generating binary trees are Auger and Walsh [16], Bapiraju and Rao [20],Bonnin and Pallo [36], Er [105], [107], [108], [111], [109], [112],Rémy [341],Ruskey and Hu [354], Knott [217], Liu [252], Mäkinen [261],[262], Pallo [297], [298], [300], Pallo and Racca [302], Proskurowski [314], Proskurowski and Laiman [315], Ramanan and Liu [327], Rotem and Varol [350], Roelants van Baronaigien and Ruskey [431], Ruskey [361], Semba [379], Solomon and Finkel [395], Skarbek [389], Trojanowski [416], Vajnovszki and Pallo [425], Zaks [466], Zaks and Richards [467], and Zerling [468].

A CAT algorithm for generating ordered forests with given degree sequence may be found in Ruskey and Roelants van Baronaigien [360]; this topic is also considered by Zaks and Richards [467] and Er [111].

An algorithm for listing all fixed size binary subtrees of a given binary tree is given in Hikita [178]. Generating all ordered subtrees of a given ordered tree is discussed in Ruskey [363] and Koda and Ruskey [226].

Free trees, rooted trees

The remarkable 1969 paper of Scions [376] discussed the generation of various types of trees in lexicographic order. In particular, that paper introduces the encoding $e(T)$ of Section 4.6.2 and observes (but does not prove) the CAT property of a lex algorithm for generating those sequences. The rooted tree algorithm that we presented is due to Beyer and Hedetniemi [30]; the free tree algorithm to Wright, Richmond, Odlyzko and McKay [461].

Another paper that contains an algorithm for generating trees is that of Tinhofer and Schreck [413].Recently, Li and Ruskey [248] have developed algorithms for generating rooted and free trees that are based soley on the familiar parent array representation of trees. These algorithms are recursive and can be adapted to generate rooted trees with degree and height restrictions, and free trees with degree and diameter restrictions — and they are CAT.

Kubicka [235] shows that the average value of $n-p+1$ is $1/(1-\rho)$, where ρ is the radius of convergence of the ordinary generating function $\sum a_i x^i$. Algorithms for generating binary rooted trees have been developed by Kubicka and Kubicki [234]. See also Pallo [300] and Vajnovszki [423].

Additional algorithms for generating and/or ranking rooted trees are given in Wilf and Yoshimura [452], Yoshimura [465], Kozina [232], and Dinits and Zaitzev [86]. Trojanowski [417] considers the problem of ranking unordered binary trees.

Other types of trees

Algorithms for ranking, unranking and listing 2-3 trees and B-trees are given by Gupta, Lee, and Wong [162], [163], , and Belbaraka and Stojmenović [24], but the approach developed by Kelsen [209], and refined in this chapter, are superior.

Algorithms for listing and randomly generating "hybrid" binary trees are given in Pallo [301]. An algorithm for ranking and unranking AVL (height-balanced) trees was given by Li [249].

A *rooted plane tree* is a rooted tree embedded in the plane; i.e., the order of nodes within subtrees is fixed, but the subtrees themselves can move circularly about the root. They can be generated by using fixed density necklaces; see Exercise 7.

Set Partitions

Papers about generating and ranking set partitions in lex order are Djokić, Miyakawa, Sekiguchi, Semba, and Stojmenović [88], Er [110],,Semba [380], and Williamson [454]. Wells [440] gives a listing algorithm and ranking function based on a combinatorial interpretation of (2.12). Williamson [453] contains an algorithm for ranking partitions with a fixed number of blocks.

Numerical Partitions

According to [340], an iterative lexicographic algorithm for generating numerical partitions is due to K.F. Hindenburg in 1778.

The following references contain algorithms for generating numerical partitions: James and Riha [192], Riha and James [344], Fenner and Loizou [120], [121], [122], McKay [275], [276], [277], [278], Narayana, Mathsen and Saranji [288], Stockmal [407], White [446], [447], Ord-Smith [294], [295]. Phillips [309], Schrak and Shimrat [374], Shen [386].

Knuth [222] (pp. 145-149) contains an algorithm for listing all partitions with a bounded number of parts of bounded size. [223]

Linear Extensions and Ideals

The Varol-Rotem algorithm is from [433]; it and other algorithms are discussed and compared in Kalvin and Varol [201]. The recursive formualtion given here is believed to be new. See also Knuth and Szwarcfiter [223]and Wells [440] for other algorithms that generate all linear extensions. Knuth [221] develops an algorithm for listing all linear extensions of forest posets in time $O(M)$, where M is the amount of change that the extensions undergo in a lexicographic listing.

The Steiner algorithm is from Steiner [404]. The improvement of Squire is from [399].

Generalized settings

Lenstra and Rinnooy-Kan present recursive algorithms for generating various objects [245]. Extensions of Wilf's generalized setting were given by Williamson [453] and by Joichi, White, and Williamson [200].

!!!! What does the paper of Bezem and van Leeuwen discuss? !!!!

Miscellaneous

Algorithms for generating stable marriages may be found in the book of Gusfield and Irving [164].

There are a number of ways in which lex order and colex order is useful from a purely combinatorial point of view. See, for example, Erdös and Székely [104]. Other examples include standard proofs of the Kruskal-Katona theorem (e.g., Stanton and White [403]).

Fillmore and Williamson [128] give ranking algorithms for the symmetries and colorations of an n-cube where the underlying order is lexicographic.

Chapter 5

Combinatorial Gray Codes: Algorithmic Issues

In this chapter we introduce combinatorial Gray codes, which are lists of combinatorial objects in which successive objects are "close" in some sense. These lists have many applications and also lead to interesting questions and results of a mathematical nature, many of which are discussed in the chapter following this chapter, "Combinatorial Gray Codes: Graph Theoretic Issues." In this chapter we concentrate on algorithms for generating combinatorial Gray codes and on their applications. The fastest known algorithms for generating certain objects are based on Gray codes; furthermore, they are the basis for an interesting class of algorithms, the loopless algorithms.

5.1 Introductory Example

Consider, as a practical example, the hypothetical rotating device as shown in Figure 5.1. One might imagine the drum in a photocopy machine or a magnetic storage device or the axle of a vehicle, both real examples of the use of this idea. As the device turns it is necessary to know its actual physical location. To help in this determination the device contains a number of *tracks* (rings) each of which is broken down into *sectors* (pieces of pie). Furthermore, there is a fixed set of sensors, one per track. Each (track,sector) pair stores either a 0 or a 1. In the figure there are four tracks and 16 sectors; in general if there are n tracks then there are 2^n sectors. Interpret a white (track,sector) pair as a 0 and a shaded area as a 1. Since each sector intersects n tracks, it corresponds to some binary number in the range $0, 1, \ldots, 2^n - 1$, a number that should be unique in order to unambiguously identify the sector. In what order should those binary numbers be arranged around the sectors? Figures 5.1(a) and (b) illustrate two arrangements and Figure 5.1(c) illustrates the placement of the four fixed sensors. The arrangement (a) lists the numbers in lexicographic order and arrangement (b) lists them in some other order, to be explained later. On the first disk (a), the sensors would be reading 0001 and on the second disk (b), 0011.

If the rotation happens to stop so that the sensors are on the border between two sectors, then the sensors will have trouble distinguishing between 0 and 1 on a track that has different symbols across the border. If the symbols are the same across the border then we assume that the sensor returns the correct value, but if they are different then the sensor

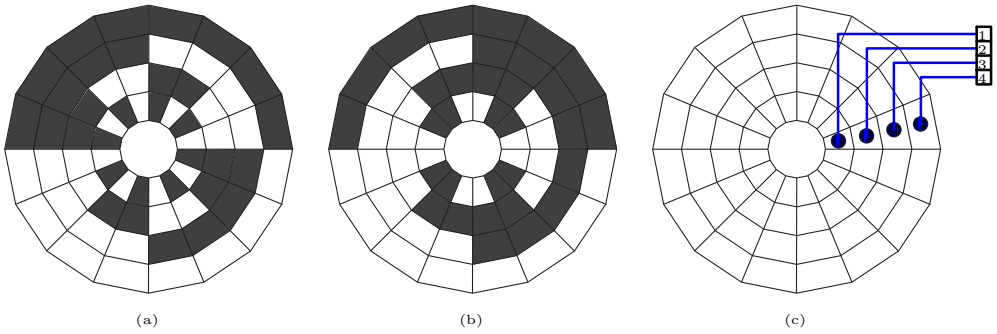


Figure 5.1: Position detection on a rotating device with two different labellings.

may return either value. In lex order the border between 0000 and 1111 displays a very bad behavior because the sensors could be returning any bitstring $b_1b_2b_3b_4$ whatsoever! We consequently have absolutely no idea where the device is positioned.

What is desired is a listing of all 16 bitstrings of length 4 so that successive strings, including the first and last, differ by exactly one bit. Then at the border the sensors can return only one of two values, and those two values are exactly those of the two sectors that share that border. Figure 5.1(b) shows such a listing. The construction of these lists for general values of n is taken up and solved in Section 5.2.

5.1.1 Combinatorial Gray Codes

The listing just asked for is a specific example of a combinatorial Gray code. Here we give a formal definition — just to get it out of the way. Many other examples will be found in this chapter and in the next.

Let S be a finite set of combinatorial objects and C a relation on S, which we call the *closeness relation*. Then a *Combinatorial Gray Code* for S is a listing $s_1, s_2, \ldots, s_N$ of the elements of S, where $|S| = N$, such that $(s_i, s_{i+1}) \in C$ for $i = 1, 2, \ldots, N-1$. Sometimes we also insist that $(s_N, s_0) \in C$; in this case, the code is said to be *cyclic*. There is a graph $G = G(S, C)$, the *closeness graph*, that is naturally associated with a combinatorial Gray code. The vertices of this graph are the objects of S and edges connect those objects that are "close"; in other words, $G(S, C)$ is just the digraph of the relation C. In most instances the closeness relation is symmetric and so G may be regarded as an undirected graph. The question of finding a Combinatorial Gray Code for S with respect to C becomes one of finding a Hamiltonian path (or cycle) in G. Conversely, a Hamilton path in G can be viewed as a Gray code for S. In our introductory example, S is the set of all bitstrings of fixed length, C is the relation of Hamming distance one, and G is the hypercube.

5.2 The Binary Reflected Gray Code

Suppose that we wish to list all 2^n length n bitstrings in such a way that each bitstring differs from its predecessor by a single bit, as required in our introductory example of position detection on rotating devices. Such lists are called *binary (Gray) codes*, named after Frank Gray who used them in a patent he obtained for "pulse code communication".

$$\mathbf{G}_0 = \emptyset, \qquad \mathbf{G}_n = \begin{array}{c|c} 0 & \\ 0 & \\ \vdots & \mathbf{G}_{n-1} \\ 0 & \\ \hline 1 & \\ \vdots & \overline{\mathbf{G}_{n-1}} \\ 1 & \\ 1 & \end{array}$$

Figure 5.2: The definition of $\mathbf{G}_n$.

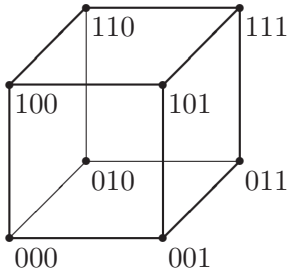


Figure 5.3: A 3-cube.

There is a very simple recursive algorithm for generating one particular Gray code, known as the *binary reflected Gray code* (BRGC).

The binary reflected Gray code on n elements, $\mathbf{G}_n$, is defined recursively by letting $\mathbf{G}_0$ be the single element sequence containing the emptystring, and letting $\mathbf{G}_n$ be the list obtained by prefixing each element of $\mathbf{G}_{n-1}$ with a 0 and then prefixing each element of $\overline{\mathbf{G}}_{n-1}$ with a 1.[1] This definition is illustrated in Figure 5.2. Thus $\mathbf{G}_1 = 0, 1$, $\mathbf{G}_2 = 00, 01, 11, 10$, $\mathbf{G}_3 = 000, 001, 011, 010, 110, 111, 101, 100$, and $\mathbf{G}_4$ is the encoding used in Figure 5.1(b).

The n-dimensional hypercube, or n-cube, Q_n, may be considered as the graph whose vertices consist of all bitstrings of length n and where two vertices are joined by an edge if they differ by a single bit. The problem of finding a Gray code corresponds to finding a Hamilton path in Q_n. Figure 5.3 illustrates a 3-cube. Observe that the binary reflected Gray code actually defines a Hamiltonian *cycle* on the n-cube since the first and last bitstrings only differ by a single bit.

There is an interesting connection between the Towers of Hanoi problem, the binary reflected Gray code, and ordinary counting in binary. Most computer scientists are familiar with the Towers of Hanoi problem, having been introduced to is as an introductory example of recursive programming. The problem is to move n disks of distinct sizes from one peg to another peg. Each disk has a hole in it and the pegs pass through these holes. Initially, the disks are arranged in increasing size on one peg. There are two rules that must be obeyed in moving the disks. First, only one disk at a time may be moved. Secondly, a larger disk

[1]Recall that if $\mathbf{L}$ is a list, then $\overline{\mathbf{L}}$ is its reversal.

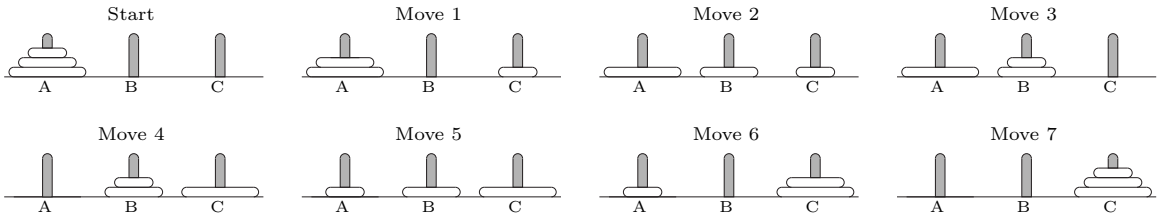


Figure 5.4: Towers of Hanoi for $n = 3$.

Counting	Gray Code	Towers
321	210	
00$\underline{0}$	00$\underline{0}$	1
0$\underline{0}$1	0$\underline{0}$1	2
01$\underline{0}$	01$\underline{1}$	1
$\underline{0}$11	$\underline{0}$10	3
10$\underline{0}$	11$\underline{0}$	1
1$\underline{0}$1	1$\underline{1}$1	2
11$\underline{0}$	10$\underline{1}$	1
111	100	

Figure 5.5: $n = 3$: Counting, Gray Code, and Towers of Hanoi.

may not be placed on a smaller disk. Let $\mathcal{S}_n(a, b, c)$ denote a solution to the problem where the disks are initially on peg a, are moving to peg b, and use peg c as an intermediary. If $n = 1$ then simply move the disk from a to b. Otherwise, a solution may obtained by first doing $\mathcal{S}_{n-1}(a, c, b)$, then moving the largest disk n from a to b, and then doing $\mathcal{S}_{n-1}(c, b, a)$. A solution to the three disk Towers of Hanoi problem is shown in Figure 5.4.

Now suppose that the disks are numbered $1, 2, \ldots, n$ where 1 is the smallest disk and n is the largest disk. Let T_n denote the sequence of disks that are moved in solving the n disk Towers of Hanoi problem. From the example above $T_3 = 1, 2, 1, 3, 1, 2, 1$. From the recursive solution to the Towers of Hanoi problem we see that $T_1 = 1$ and in general $T_{n+1} = T_n, n + 1, T_n$. As a notational convenience we will now assume that the Gray code bitstrings are indexed $g_n g_{n-1} \cdots g_1$. It turns out that the kth number of T_n is the index of the bit that changes in the kth bitstring of $\mathbf{G}_n$, and that this is equal to the position of the rightmost 0, *counting from the right*, in the kth bitstring when counting in binary. These correspondences are illustrated in Figure 5.5, which the reader should digest before proceeding further.

The sequence T_n is called the *transition sequence* of the BRGC. Every Hamilton path on the n-cube has a transition sequence and these paths are often specified by their transition sequences.

We now develop algorithms for generating the binary reflected Gray code, as well as ranking and unranking algorithms. Let *rank* denote the rank function for the BRGC, and $\overline{rank}$ the rank function for the reversal of the BRGC (obtained by prepending with 1's first and then 0's). The following two equations are clear from the recusive definition of the

BRGC.

$$rank(g_n g_{n-1} \cdots g_1) = \begin{cases} 0 & \text{if } n = 0 \\ rank(g_{n-1} \cdots g_1) & \text{if } g_n = 0 \\ 2^{n-1} + \overline{rank}(g_{n-1} \cdots g_1) & \text{if } g_n = 1 \end{cases} \tag{5.1}$$

$$\overline{rank}(g_n g_{n-1} \cdots g_1) = \begin{cases} 0 & \text{if } n = 0 \\ 2^{n-1} + \overline{rank}(g_{n-1} \cdots g_1) & \text{if } g_n = 0 \\ rank(g_{n-1} \cdots g_1) & \text{if } g_n = 1 \end{cases} \tag{5.2}$$

The formula for $\overline{rank}$ is not really needed since

$$\overline{rank}(g_n g_{n-1} \cdots g_1) \;=\; 2^n - 1 - rank(g_n g_{n-1} \cdots g_1).$$

This means that the $g_n = 1$ case of (5.1) could be written as $2^n - 1 - rank(g_{n-1} \cdots g_1)$. Nevertheless, it is convenient to have both (5.1) and (5.2) when deriving an iterative ranking method, which we do next.

Let $rank(g_n g_{n-1} \cdots g_1) = (b_{n-1} \cdots b_1 b_0)_2$. Then the recurrence relations (5.1) and (5.2) for $rank$ and $\overline{rank}$ may be interpreted as setting b_{n-1} to 0 or 1, depending upon whether the term 2^{n-1} in the binary expansion is present or not. The repeated application of these recurrence relations to a bitstring may then be thought of as sweeping a r (for $rank$) or $\overline{r}$ (for $\overline{rank}$) from left-to-right as illustrated below.

	g_9	g_8	g_7	g_6	g_5	g_4	g_3	g_2	g_1
r	1	1	1	0	1	0	1	0	1
1	$\overline{r}$	1	1	0	1	0	1	0	1
1	0	r	1	0	1	0	1	0	1
1	0	1	$\overline{r}$	0	1	0	1	0	1
1	0	1	1	$\overline{r}$	1	0	1	0	1
1	0	1	1	0	r	0	1	0	1
1	0	1	1	0	0	r	1	0	1
1	0	1	1	0	0	1	$\overline{r}$	0	1
1	0	1	1	0	0	1	1	$\overline{r}$	1
1	0	1	1	0	0	1	1	0	r
b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	

With $g_{n+1} = 0$, note that an r always has a 0 to its left and a $\overline{r}$ has a 1 to its left. Thus there are four possibilities for the three symbols $b_i x g_i$ where x is r or $\overline{r}$, namely $0r0$, $0r1$, $1\overline{r}0$, and $1\overline{r}1$. The resulting value of b_{i-1} is $0, 1, 1, 0$ in those four cases, respectively. From this we observe that $b_{i-1} = b_i \oplus g_i$, where $\oplus$ denotes exclusive-or. The bit below an r or $\overline{r}$ is the exclusive-or of the bits left and right of the r or $\overline{r}$.

The preceding discussion proves the following lemma.

LEMMA 5.1 *If* $rank(g_n g_{n-1} \cdots g_1) = (b_{n-1} \cdots b_1 b_0)_2$, *then for* $i = 1, 2, \ldots, n$

$$b_{i-1} = b_i \oplus g_i \quad \textit{and} \quad g_i = b_{i-1} \oplus b_i.$$

For our example, $358 = (101100110)_2$, so the 358th Gray code bitstring is 111010101. Note that Lemma 5.1 may be used to obtain a $O(n)$ unranking algorithm (and constant time when done in parallel).

Practical generation of the BRGC

The equation $g_i = b_i \oplus b_{i-1}$ suggests a rapid way of generating the BRGC as long as n is not to large, in particular if n is at most the number of bits in a computer word. Let b and g be words and use SR to denote shifting a word one position to the right. Since most computers include $\oplus$ and SR as machine operations, the update $g{:=}b \oplus SR(b)$ operating on words will update g in three machine instructions. Putting this update together in a loop with $b{:=}b+1$ will produce the BRGC very efficiently. In algorithm 5.1 we show a C implementation that will produce the BRGC in the unsigned integer `g` (variable `b` is also an unsigned integer).

```
b = 0;
do g = b ^ b++ >> 1;
while (b != 0);
```

Algorithm 5.1: A C implementation of the BRGC on computer words.

Recursively generating the BRGC

Figure 5.2 is a somewhat verbose way of presenting our recursion for the BRGC, although it is the type of diagram most useful for presenting the BRGC on a blackboard in a classroom setting. The recursion below is more compact and is in the style we use subsequently.[2]

$$\mathbf{C}(n) = \begin{cases} \varepsilon & \text{if } n = 0 \\ \mathbf{C}(n-1)\cdot 0 \circ \overline{\mathbf{C}(n-1)}\cdot 1 & \text{if } n > 0 \end{cases} \tag{5.3}$$

Note that we are appending rather than prepending. In developing elegant natural algorithms this has the same advantage that colex order had over lex order in the previous chapter. In subsequent algorithms we will try to append. As in the previous chapter, the recurrence relation leads immediately to an algorithm. Algorithm 5.2 assumes the existence of an array of bits, `b[1..n]`. The initial call is `brgcI(n)`; no intialization is necessary. Note that `brgcI` calls `cgrbI` which produces the reversed list $\overline{\mathbf{B}(n)}$. It is trivially obtained from `brgcI` by inverting the order of the two lines of the else clause.

```
procedure brgcI ( n : ℕ);
begin
    if n = 0 then PrintIt
    else
        g[n] := 0; brgcI( n − 1 );
        g[n] := 1; cgrbI( n − 1 );
end {of brgcI};
```

Algorithm 5.2: Indirect algorithm for generating the BRGC.

This algorithm is CAT and is perfectly suitable for many applications. However, it suffers one drawback, namely that the processing routine `PrintIt` does not know which bit was flipped. This defect can be remedied, but not without modifying the structure of the algorithm. For Gray codes in general, it is often useful to know exactly what the list

[2]Recall that $\circ$ denotes concatenation of lists and $\cdot$ denotes concatenation of strings.

of small changes are that sequence through all objects. From the proof of correctness of the BRGC we know that it's the nth bit that flips between the two recursive calls. This observation leads us to Algorithm 5.3. Algorithms of the first type we call *indirect* and those of the second type are called *direct*. The generation algorithms of the previous chapter were of the indirect style, mostly because the changes between successive objects could be large. However, for combinatorial Gray codes we take the point of view that direct algorithms are preferable.

Direct Gray Code Algorithm Superiority Principle: Try to develop direct algorithms whenever possible. It will make your programs more flexible and easier to analyze.

In Algorithm 5.3, the call flip(n) flips the nth bit and invokes PrintIt. The calling sequence differs from the indirect case in that PrintIt is called once before calling brgcD(n). Algorithm brgcD is also CAT; it is called exactly the same number of times on input n that brgcI(n) and cgrbI are called, namely $2^n - 1$ times. Note that we do not need a separate reversed procedure cgrbD since it would be identical to brgcD. The procedure works correctly no matter how the bitstring a is initialized; that is, the call brgcD(n) produces a Gray code of all strings in Σ_2^n.

```
procedure brgcD ( n : ℕ);
begin
    if n > 0 then
        brgcD( n − 1 ); flip( n ); brgcD( n − 1 );
end {of brgcD};
```

Algorithm 5.3: Direct algorithm for generating the BRGC.

A loopless algorithm: counting without loops

Recall from Section 4.2 that the most straighforward way of counting in binary results in a CAT algorithm (Algorithm 4.1). That algorithm does a right-to-left scan for a 0, the position of that 0 being the bit that should flip in the BRGC. Thus we can generate the BRGC in constant amortized time per bitstring by counting in binary, but occasionally the length of the scan will be proportional to n.

However, it is even possible to generate the BRGC in such a way that a constant amount of computation is used *in the worst case* to go from one bitstring to the next. Such an algorithm will be developed next. We know that if we count in binary and keep track of the leftmost position that changes then we can generate the BRGC. But can we count without doing the right-to-left scan for a zero? The answer is yes. We maintain an array $\tau[n-1..0]$ which represents $b_{n-1} \cdots b_1 b_0$ as follows. Define $b_{-1} = b_n = 0$, and for $0 \leq i \leq n-1$,

$$\tau_i = \begin{cases} i & \text{if } b_i = 0 \text{ or } b_{i-1} = 1 \\ \max\{k \mid \tau_k = 0 \text{ and } k > i\} & \text{if } b_i = 1 \text{ and } b_{i-1} = 0. \end{cases}$$

In other words, $\tau_i = i$ unless i is the rightmost 1 of a run of ones, in which case τ_i is the position of the 0 that immediately precedes the run of 1's. Observe that b may be recovered from τ. Below we show 4 successive bitstrings b and the associated values of τ.

b											τ										
10	9	8	7	6	5	4	3	2	1	0	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	1	1	1	1	10	**11**	8	**8**	6	5	4	3	2	1	**4**
1	1	0	1	0	0	1	0	0	0	0	10	**11**	8	**8**	6	5	**5**	3	2	1	0
1	1	0	1	0	0	1	0	0	0	1	10	**11**	8	**8**	6	5	**5**	3	2	1	**1**
1	1	0	1	0	0	1	0	0	1	0	10	**11**	8	**8**	6	5	**5**	3	2	**2**	0

```
procedure Next;
local i : ℕ;
begin
    i := τ_0;
    g_{i+1} := ¬g_{i+1}; {flip the bit}
    τ_0 := 0;
    τ_i := τ_{i+1};
    τ_{i+1} := i + 1;
    if τ_0 = n then done := true;
end {of Next};
```

Algorithm 5.4: Loop-free procedure *Next* for generating the BRGC.

The bit that flips in $g_n g_{n-1} \cdots g_1$ is simply the bit in position τ_0. There are four cases to consider when updating τ, depending on the values of b_0 and b_{i+1}, where $i = \tau_0$. These cases and the associated updates are shown below.

values of b_0 and b_{i+1}	how to update τ
$b_0 = 1$, $b_{i+1} = 0$	$\tau_i{:=}i+1$; $\tau_0 := 0$;
$b_0 = 1$, $b_{i+1} = 1$	$\tau_i{:=}\tau_{i+1}$; $\tau_{i+1}{:=}i+1$; $\tau_0{:=}0$;
$b_0 = 0$, $b_{i+1} = 0$	$\tau_0{:=}1$;
$b_0 = 0$, $b_{i+1} = 1$	$\tau_0{:=}\tau_1$; $\tau_1{:=}1$;

It turns out that the four cases can be handled without any if statements as shown in the remarkably simple Algorithm 5.4.

5.2.1 Applications of the BRGC

There are a suprising number of applications of the BRGC and other related Gray codes and we mention some of these in this subsection. In particular the BRGC has applications to solving certain puzzles, to hardware position detection, to satellite transmissions, to multiattribute file storage, to the construction of Brunnian links, and to circuit minimization.

Uniform Gray Codes

One problem with certain applications of the Binary Reflected Gray code in particular, and generation algorithms based on backtracking in general, is that a small number of elements do most of the changing (or moving or whatever the basic operation is). In the BRGC the last bit changes on every other bitstring. In some cases we want a listing where the changes are more uniformly distributed over all positions (or elements or whatever).

Figure 5.6: A six ring Chinese rings puzzle.

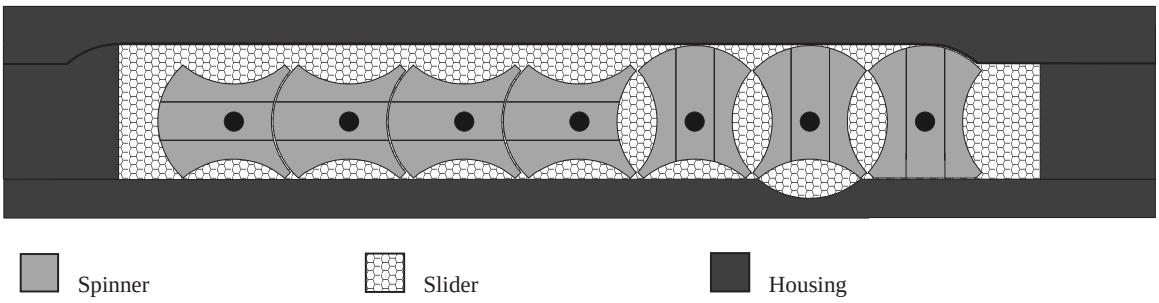


Figure 5.7: The Spin-out puzzle.

Puzzles

There are a number of puzzles whose solution involves the BRGC. We've already seen one of these, the towers of Hanoi. The oldest puzzle solved by the BRGC is known as the *Chinese Rings*.

In the Chinese rings puzzle, the object is to remove the U-shaped piece from the rings. As the puzzle is solved the rings are either on or off the U-shaped piece. If the rings are successively numbered and the on-ness or off-ness of each rings is recorded as a bit, then the solution of the puzzle sequences through that portion of the Gray code from 000000 to 111111.

Other puzzles are "The Brain" (Mag-Nif Inc.) and "Spin-out" (Binary Arts). The Spin-out puzzle is illustrated in Figure 5.7. There are 7 "spinners" which take on one of two orientations, either $\uparrow$ or $\leftarrow$. The figure illustrates the configuration $\leftarrow\leftarrow\leftarrow\leftarrow\uparrow\uparrow\uparrow$, from which two moves are possible. The first is simply to rotate the sixth spinner by 90 degrees to obtain the configuration $\leftarrow\leftarrow\leftarrow\leftarrow\uparrow\leftarrow\uparrow$; the second is to move the slider to the left in the housing and then rotate the seventh spinner to obtain the configuration $\leftarrow\leftarrow\leftarrow\leftarrow\uparrow\uparrow\leftarrow$. Given an initial configuration of $\uparrow\uparrow\uparrow\uparrow\uparrow\uparrow\uparrow$, the puzzle is solved when the configuration is $\leftarrow\leftarrow\leftarrow\leftarrow\leftarrow\leftarrow\leftarrow$. In the solved configuration the plastic piece upon which the spinners are mounted (the slider) can be removed from it's housing, thereby serarating the puzzle into two pieces, as was the goal in the Chinese rings puzzle.

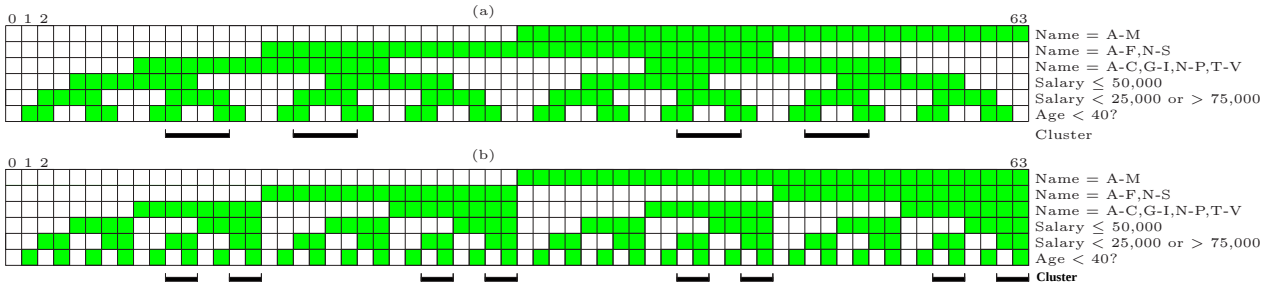


Figure 5.8: Multiattribute file storage: (a) Arranged according to the BRGC, (b) arranged according to lex order. Blank area is 0 and shaded area is 1.

Karnaugh maps

A well-known method of minimizing circuits with a small number of gates is through the use of Karnaugh maps. In such a map the possible inputs are given as the row and column indices of a two dimensional table. Those indices are listed in Gray code order.

Codon rings

The 64 three letter code words, known as codons, that make up DNA can be encoded by a base four number system in a natural way. Surprisingly, when the encoding numbers are sequenced as a 4-ary reflected Gray code the resulting cyclic listing appears to have some biological meaning. It is well known that DNA is composed of 64 codons, each composed of a three letter string made up of one of four bases C (Cytosine), A (Adenine), G (Guanine), U/T (Uracil/Thymine). Map C to 0, A to 3, G to 1, and U/T to 2. With each codon, there is an associated *anti-codon* obtained by adding 2 (mod 4) to each digit. For example, the anti-codon of CAG (012) is GUC (230). The number of hydrogen bonds per codon/anti-codon pair is 9 minus the number of even digits in the codon.

Multi-attribute file storage

Imagine a database application in which records are to be stored in secondary storage as a file and retrieved in response to incoming queries. The file F is a collection of ordered k-tuples, each consisting of k attributes. Denote the attributes by $A_1, A_2, \ldots, A_k$. In a *partial match query* we are given a set of indices I and values v_i for each $i \in I$. The query should return all records for which $A_i = v_i$ for all $i \in I$.

Multiattribute hashing is a popular method of organizing the records. Each record R has a hash value $h(R)$ associated with it, where $h(R)$ is a bitstring of size n. We try to arrange the attributes so that no two records have the same hash value. This is generally impossible, so some method of overflow must be implemented, but we ignore the possiblity of overflow in the discussion below.

As a specific example suppose that each record contains three fields, Name, Salary, and Age, and that there are about 64 records. We specify 6 binary attibutes as follows. Three bits are allocated to Name, allowing 8 subcatagorizations: A-C, D-F, G-I, J-M, N-P, Q-S, T-V, W-Z. Two bits are allocated to Salary, allowing 4 subcatagorizations: $\mathsf{Salary} < 25,000$, $25,000 \leq \mathsf{Salary} < 50,000$, $50,000 \leq \mathsf{Salary} < 75,000$, $75,000 \leq \mathsf{Salary}$. A single bit is

allocated to Age, allowing 2 subcatagorizations: Age < 40 and Age ≥ 40.

A query is specified by a string $q \in \{0, 1, ?\}^k$. The symbol "?" indicates a "don't care". For example, the query ?1?1?? should return all records for which the name is in one of the ranges A-C, G-I, N-P, or T-Z, and the salary is either less than $25,000 or greater than $75,000.

Relative to a query q, a *cluster* is a contiguous collection of records satisfying q, delimited by records not satisfying q. Assuming the records are stored in a file on a computer disk, a reasonable measure of the cost of retreiving all records satisfying q is the number of record clusters. This is because of the way computer disks are constructed — reading sequentially from a disk is fast, whereas finding a particular location on the disk is slow. Thus our aim in storing the files on disk is to arrange them so that the number of clusters returned in a typical query is minimized.

In Figure 5.8 we show the layouts for a 6 bit storage scheme in both the BRGC and in lex order. For the example query ?1?1??, the BRGC organization leads to 4 clusters and the lex organization leads to 8 clusters. It can be shown that the number of clusters returned under the Gray code organization is never more than that returned under the lexicographic organization, and that the lexicographic organization never returns more than twice as many clusters as the Gray code organization. Thus it is always preferable to use the BRGC organization.

Brunn's Brunnian Link

In knot theory a *link* is a finite collection of knots. A Brunnian link is a link for which the removal of any knot causes the link to become trivial; i.e, no remaining pair of knots is linked and each knot is the unknot. In Brunn's paper [43] there is a Brunnian link constructed that has the same underlying structure as the BRGC. See Figure 5.9 which shows his construction for $n = 6$. Note in particular the evident pattern 1,2,1,3,1,2,1,4,1,2,1,3,1,2,1,4, and the way in which the binary over-ness and under-ness gives rise to the BRGC along the 16 radial axes.

5.3 Gray Codes for Combinations

In this section we develop several Gray codes for combinations; that is, for k-subsets taken from the set $[n] = \{1, 2, \ldots, n\}$. For the most part, we identify each subset with a bitstring of length n with exactly k ones; i.e., with an element of $\mathbf{B}(n, k)$ (defined in (4.1). The algorithms developed can be easily modified to generate the elements of the other common representation, $\mathbf{A}(n, k)$ (recall that $\mathbf{A}(n, k)$ is the set of monotonically increasing sequences of length k with elements taken from $[n]$; see (4.2)).

There are many closeness conditions (or relations) that can be applied to bitstrings with a fixed number of ones. In this section we consider five natural closeness relations. These relations can be defined more generally on any set of strings in which the number of occurances of each symbol is the same in each string. We therefore define them on the set of permutations of a multiset.

- **[Trans]** The *transposition* condition: The two permutations differ in exactly two positions. The next three conditions are restrictions of this condition. Call the two positions i and j. Example: $\underline{0}021\underline{1}$ and $\underline{1}021\underline{0}$. In this example the positions that

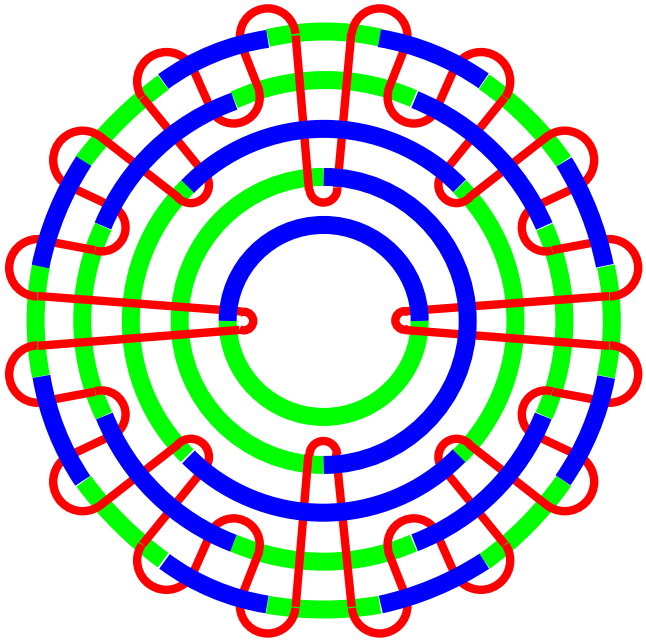


Figure 5.9: Brun's Brunnian link.

change are $i = 1$ and $j = 5$, and are underlined, as they will be in each of the example below.

- **[H-Trans]** The *homogenous transposition* condition: The permutations, say π and π' differ in exactly two positions i and j, and only symbols less than or equal to $\min(\pi_i, \pi_j)$ occur between those two positions. For example, if $t = 1$, then only 0's occur between the 1 and 0 that are transposed. Example: $\underline{0}000\underline{1}12$ and $\underline{1}000\underline{0}12$.

- **[2-Trans]** The *one or two apart transposition* condition: The two permutations differ in positions that are one or two apart. That is, $|i - j| \leq 2$. Example: $0\underline{0}2\underline{1}1$ and $01\underline{2}0\underline{1}$.

- **[A-Trans]** The *adjacent transposition* condition: The two permutations differ in exactly two adjacent positions. That is, $|i - j| = 1$. Example: $00\underline{2}11$ and $02\underline{0}11$.

- **[2A-Trans]** The *one or two adjacent transposition* condition: The two permutations differ by one or two adjacent transpositions. Example: $0\underline{\underline{012}}1\underline{2}2$ and $0\underline{\underline{102}}2\underline{1}2$, or $01\underline{2}0\underline{1}$ and $02\underline{01}1$. In the second example the transpositions are $(2\ 3)(3\ 4) = (2\ 3\ 4)$.

Note that [A-Trans] is the strongest condition, [Trans] is weaker than [2-Trans] or [H-Trans], and [2A-Trans], [2-Trans], and [H-Trans] are incomparable. For combinations, the [A-Trans] condition cannot always be achieved, but the others can be achieved. There are many other natural closeness conditions that could be considered; for example, permutations differing by a 2-cycle or a 3-cycle acting on the positions (this is a restriction of 2A-Trans).

[Trans]

Algorithms satisfying the transposition criteria are somtimes said to be "revolving door" algorithms. Imagine two rooms A and B connected by a revolving door that operates only if occupied by two persons, one from each room. As a person leaves one room, they enter the other, so the door acts to swap two persons. Given k persons in the first room, the problem is to sequence through all possible possible combinations of persons in the first room so that each such combination occurs exactly once. This is precisely the same as the question of generating combinations by transpositions.

The transposition closeness condition is handled in a manner similar to the way subsets were done. That is, we rely on the classic recurrence relation for binomial coefficients, but reverse the list corresponding to one of the terms. Thus our list is

$$\mathbf{C}(n,k) = \begin{cases} 0^n & \text{if } k = 0, \\ 1^n & \text{if } k = n, \\ \mathbf{C}(n-1,k)\cdot 0 \circ \overline{\mathbf{C}(n-1,k-1)}\cdot 1 & \text{if } 0 < k < n. \end{cases} \tag{5.4}$$

Observe that the list defined above has the same form as (5.3), the definition of the BRGC; the substantive difference is the addition of another parameter, k. Thus, taking the BRGC list for n and deleting all strings not containing exactly k 1's results in the $\mathbf{C}(n,k)$ list defined above. However, we still need to prove that it has the [Trans] property. Unlike our proof for the BRGC, here we need to strengthen the inductive statment in order for the proof to proceed. As will be the case in each such proof in this subsection, the strenghening will precisely specify the starting and ending bitstrings on each recursively defined list.

LEMMA 5.2 *The list $\mathbf{C}(n,k)$, defined in (5.4), satisfies the following properties.*

1. *Successive combinations differ by a transposition of two bits; i.e., the list satisfies the [Trans] condition.*
2. *$first(\mathbf{C}(n,k)) = 1^k 0^{n-k}$.*
3. *$last(\mathbf{C}(n,k)) = 1^{k-1}0^{n-k}1$ if $k > 0$.*
4. *$last(\mathbf{C}(n,0)) = 0^n$.*

Note that the first and last bitstrings differ by a transposition. The following table illustrates the various cases that can occur and should serve to convince the reader that the lemma is true. [3]

$k = 1$	$1 < k < n-1$	$k = n-1$	
1000000	11111000000		$\left.\begin{matrix}\\\\\\\end{matrix}\right\} \binom{n-1}{k}$
$\vdots$	$\vdots$	$1111\underline{1}1\underline{0}$	
$0000\underline{1}0$	$111\underline{1}000001\underline{0}$		
	11100000011	1111011	$\left.\begin{matrix}\\\\\\\end{matrix}\right\} \binom{n-1}{k-1}$
0000001	$\vdots$	$\vdots$	
	11110000001	1111101	

[3] Note that the $k = n-1$ case is subsumed by the $1 < k < n$ case.

The two swapped bits are underlined in the table. Note that they are in positions $k-1$ and n if $1 < k < n$, and are in positions $n-1$ and n if $k = 1$. This observation leads us to the direct implementation of Algorithm 5.5. The call `gen(n,k)` generates $\mathbf{C}(n,k)$ and the call `neg(n,k)` generates $\overline{\mathbf{C}(n,k)}$. If only a list of positions of transposed bits are desired, then have `swap(i,j)` print the pair (i,j). If the bitstrings of $\mathbf{C}(n,k)$ are desired, then `swap(i,j)` will swap the bits of $b_1 b_2 \cdots b_n$ that are in positions i and j before printing; also, $\mathbf{b}$ must be initialized to $1^k 0^{n-k}$ and printed before the call to `gen`. To produce the elements of $\mathbf{A}(n,k)$, the call `swap(n,n-1)` in procedure `gen` should replace $a_k = n-1$ with $a_k = n$, and the call `swap(n,k-1)` should replace $a_{k-1} = k-1$ with $n-1$ and replace $a_k = n-1$ with n; in `neg` these actions are reversed.

```
procedure gen ( n, k : ℕ);
begin
    if 0 < k  and k < n then
        gen( n − 1, k );
        if k = 1 then swap( n, n − 1 ) else swap( n, k − 1 );
        neg( n − 1, k − 1 );
end  {of gen};

procedure neg ( n, k : ℕ);
begin
    if 0 < k  and k < n then
        gen( n − 1, k − 1 );
        if k = 1 then swap( n, n − 1 ) else swap( n, k − 1 );
        neg( n − 1, k );
end  {of neg};
```

Algorithm 5.5: Direct transposition generation of $\mathbf{B}(n,k)$.

What is the running time of `gen`(n,k) of Algorithm 5.5? In answering this question we begin to appreciate the potential beauty and power of Gray code algorithms. Only a constant amount of computation is done at each recursive call. Every recursive call spawns two others (or none), so the computation tree is an extended binary tree. Exactly one new combination is generated for each recursive call that is not a leaf. Thus the number of leaves is $\binom{n}{k}+1$, one more than the number of internal nodes. We conclude that the total amount of computation is $O(\binom{n}{k})$; the algorithm is CAT.

To close our discussion of this [Trans] Gray code we note an interesting pattern which it posesses, and use this pattern to develop a loopfree algorithm. Consider the corresponding elements of $\mathbf{A}(n,k)$ that are generated, as illustrated in column 1 of Figure 5.10. We claim that the value of the element a_k always increases, that, *for fixed* a_k, the value of a_{k-1} always decreases, and that in general, for fixed values of $a_{k-j}, \ldots, a_k$, the value of a_{k-j-1} always either decreases or increases, depending on whether j is even or odd.

That this is true follows from a similar property of the BRGC. Below is a left-to-right listing of the BRCG with set elements listed explicitly, bottom-to-top and in increasing order. Note that along the rows, for fixed values below them, the order of elements is alternately increasing ($\rightarrow$) and decreasing ($\leftarrow$).

```
→ . . . . . 1 . . . . . .
← . . . . . 1 . . . . . 1 . 2 2 1 . . . 1 . . . . .
→ . . . . 1 . . . 1 2 2 . 1 . . . 1 2 2 3 3 3 3 . 1 2 2 . 1 . .
← . . 1 . 2 2 1 . 3 3 3 3 2 2 1 . 4 4 4 4 4 4 4 4 3 3 3 3 2 2 1 .
→ . 1 2 2 3 3 3 3 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5
```

Call the $n = 5$ listing shown above **L**. Now imagine the construction of the list for $n = 6$. It is obtained by taking a copy of **L** with an additional row of 32 periods on top, followed by a copy of $\overline{\mathbf{L}}$ with a row of 32 6's on bottom. The alternating property is thus preserved inductively. Of course, the property holds for any sublist; in particular, for the subslists consisting of subsets of fixed size, or falling within a fixed range.

This alternating lexicographic nature of the $\mathbf{A}(n, k)$ list leads us to the recursive Algorithm 5.6. To generate the elements of $\mathbf{A}(n, k)$, set a_{k+1} to be $n + 1$ and call `gen( n )`. It is fairly straightforward to transform this into an iterative algorithm.

```
procedure gen ( n : ℕ);
begin
    if n = 0 then PrintIt else
    if n is odd then
        for j := n  to a_{n+1} − 1  do
            a_n := j;  gen( n − 1 );
    else
        for a[n+1]-1  downto n  do
            a_n := j;  gen( n − 1 );
end  {of gen};
```

Algorithm 5.6: A surprising indirect algorithm for generating $\mathbf{A}(n, k)$ by transpositions.

By a careful simulation of the implicit recursion stack it is possible to obtain a loop-free algorithm ([32]), but a more direct approach is available. Let us examine the list $\mathbf{A}(n, k)$ more closely. For a given combination, define p to be the minimum index for which $a_p > p$. The value of p changes by at most one from a combination to its successor.

If $p = 1$ and k is even then a_1 should be decreased by one; if $a_1 = 1$ after this decrease, then p becomes $p + 1$, otherwise it remains unchanged. Otherwise, if k and $p > 1$ have opposite parity, then a_{p-2} and a_{p-2} both increase by one and p becomes the maximum of 1 and $p - 2$. Finally, if k and p have the same parity, then there are two cases depending on the value of a_{p+1}. If $a_{p+1} = a_p + 1$, then it decreases by one and a_p becomes p. The new value of p is one or two greater, depending on whether $a_{p+1} > p+1$ or not. If $a_{p+1} > a_p + 1$, then a_{p-1} becomes a_p, a_p is incremented, and p becomes the maximum of 1 and $p - 1$.

Algorithm 5.7 is a loop-free procedure `Next` that transforms one combination to the next. Instead of maintaining p, it proves simpler to maintain a variable j whose value is p if it has the same parity as k, and whose value is $p - 1$ if it has parity different from k. Array `a[1..k+1]` is initialized to $1, 2, \ldots, k, n + 1$ and `j` is initialized to k. Then `PrintIt` and `Next` are executed in a repeat loop with terminating condition `a[k+1] < n+1`. Note that `Next` makes no reference to n or k; their only role is in the initialization and termination.

[H-Trans]

An early paper [96] described the problem as follows. Imagine a piano player who wishes to play all chords, with k keys pressed, one after the other and with no repeats, on a piano

```
procedure Next;  {j global and modified}
begin
    if j < 1 then
        a_1 := a_1 - 1;
        if a_1 = 1 then j := j + 2;
    else
    if a[j+1] = a[j] + 1 then
        a[j+1] := a[j];  a[j] := j;
        if a_{j+1} = a_j + 1 then j := j + 2;
    else
        a_j := a_j + 1;
        if j > 1 then
            a_{j-1} := a_j - 1;
            j := j - 2;
end  {of Next};
```

Algorithm 5.7: Loopfree algorithm for generating combinations by transpositions.

with n contiguous keys. Furthermore, in going from one chord to the next, exactly one finger is allowed to change position. The problem just described is solved by generating all k combinations of an n set by transpositions. But now we impose the further natural restriction that fingers never cross. This is exactly the problem of generating combinations via homogenous transpositions.

The homogenous transposition closeness condition is handled by expanding the term $\binom{n-1}{k-1}$ of the classic recurrence relation for binomial coefficients (2.2) into $\binom{n-2}{k-1} + \binom{n-2}{k-2}$. The following list was proposed by Eades and McKay [96] (but see Section 5.7).

$$\mathbf{E}(n,k) = \begin{cases} 0^n & \text{if } k = 0 \\ 10^{n-1} \circ 010^{n-2} \circ \cdots \circ 0^{n-1}1 & \text{if } k = 1 \\ \mathbf{E}(n-1,k)\cdot 0 \circ \overline{\mathbf{E}(n-2,k-1)}\cdot 01 \circ \mathbf{E}(n-2,k-2)\cdot 11 & \text{if } 1 < k < n \\ 1^n & \text{if } k = n \end{cases} \tag{5.5}$$

This list has a number of interesting properties which are embodied in the next lemma.

LEMMA 5.3 *The list $\mathbf{E}(n,k)$, defined in (5.5), satisfies the following properties.*

1. *Successive combinations differ by a transposition of two bits and all bits between those that are transposed are zeroes; i.e., the transpositions are homogeneous — the list satisfies the [H-Trans] condition.*

2. $first(\mathbf{E}(n,k)) = 1^k 0^{n-k}$.

3. $last(\mathbf{E}(n,k)) = 0^{n-k}1^k$.

The lemma may be proven by considering the list shown below for the $1 < k < n$ cases.

$$\left.\begin{array}{l} 111^{k-2}0^{n-k-1}\mathbf{0} \\ \quad\vdots \\ 0^{n-k-1}1^{k-2}1\underline{\mathbf{1}}\underline{\mathbf{0}} \end{array}\right\} \mathbf{E}(n-1,k)\cdot 0$$

$$\left.\begin{array}{l} 0^{n-k-1}1^{k-2}1\overline{\mathbf{01}} \\ \quad\vdots \\ 1^{k-2}\underline{1}0^{n-k-1}\underline{\mathbf{0}}\mathbf{1} \end{array}\right\} \overline{\mathbf{E}(n-2,k-1)}\cdot 01$$

$$\left.\begin{array}{l} 1^{k-2}\overline{0}0^{n-k-1}\overline{\mathbf{1}}\mathbf{1} \\ \quad\vdots \\ 00^{n-k-1}1^{k-2}\mathbf{11} \end{array}\right\} \mathbf{E}(n-2,k-2)\cdot 11$$

This list shows that the change at the first interface transposes the bits in positions $n-1$ and n; the change at the second interface transposes the bits in positions $k-1$ and $n-1$. We are thus led to the direct algorithm `gen` shown in Figure 5.8. Note that the first and last bitstrings do not differ by a transposition.

```
procedure gen ( n, k : ℕ);
begin
    if 1 < k  and k < n then
        gen( n − 1, k ); swap( n − 1, n );
        neg( n − 2, k − 1 ); swap( k − 1, n − 1 );
        gen( n − 2, k − 2 );
    else
    if k = 1 then
        for i := 1  to n − 1  do swap( i, i + 1 ); {*}
end  {of gen};
```

Algorithm 5.8: Direct generation of combinations by homogenous transpositions.

There is a corresponding procedure `neg` which simply executes the statements of `gen` in reverse order and reverses the roles of the two procedures. Of course, after initializing the string to 1^k0^{n-k}, the initial call is `gen`(n,k). The algorithm is CAT since (a) every iteration of the for loop at line {*} produces another combination and (b) when the for loop is not executed, three recursive calls are made and thus the underlying computation tree has more leaves than internal nodes. The algorithm is quite efficient in practice if `swap` is defined as a macro. The simple for loop, which is oft executed, and can be optimized, accounts for this efficiency.

[2-Trans]

The one or two apart transposition closeness condition can be handled by the same list-reversing technique that we used previously, but we need to add an additonal parameter, which takes on one of two values. For $p = 1, 2$, let $\mathbf{R}(n,k,p)$ denote a listing of the bitstrings in $\mathbf{B}(n,k)$ that starts at $0^{n-k-p}1^k0^p$ and ends at $0^{n-k}1^k$ and which satisfies the [2-Trans] condition. Of course, the list $\mathbf{R}(n,k,p)$ exists only if $n-k \geq p$. These lists may be defined

recursively as shown below.

$$\mathbf{R}(n,k,p) = \begin{cases} 0^n & \text{if } k = 0 \\ 1^{n-1}0 \circ \mathbf{R}(n-1,k-1,1)\cdot 1 & \text{if } p = 1 \text{ and } k = n-1 \\ \overline{\mathbf{R}(n-1,k,1)}\cdot 0 \circ \mathbf{R}(n-1,k-1,2)\cdot 1 & \text{if } p = 1 \text{ and } 0 < k < n-1 \\ \mathbf{R}(n-1,k,1)\cdot 0 \circ \mathbf{R}(n-1,k-1,1)\cdot 1 & \text{if } p = 2 \text{ and } 0 < k < n-1 \end{cases}$$

The list for $p = 1$ and $k = n-1$ could also be expressed as $1^{n-1}0 \circ 1^{n-2}01 \circ \cdots \circ 01^{n-1}$; this is what causes the for loop in Algorithm 5.9. That this listing is correct follows from consideration of the following table.

$p = 1$	$p = 2$	
$0^{n-k-2}01^{k-1}1\ \mathbf{0}$	$0^{n-k-2}1^{k-1}10\ \mathbf{0}$	$\binom{n-1}{k}$
$\vdots$	$\vdots$	
$0^{n-k-2}1^{k-1}\underline{1}0\ \underline{\mathbf{0}}$	$0^{n-k-2}01^{k-1}\underline{1}\ \underline{\mathbf{0}}$	
$0^{n-k-2}1^{k-1}00\ \mathbf{1}$	$0^{n-k-2}01^{k-1}0\ \mathbf{1}$	$\binom{n-1}{k-1}$
$\vdots$	$\vdots$	
$0^{n-k-2}001^{k-1}\mathbf{1}$	$0^{n-k-2}001^{k-1}\mathbf{1}$	

Note that the transposition is in positions $n-2$ and n if $p = 1$ and is in positions $n-1$ and n if $p = 2$. This discussion leads directly to the Pascal procedure `gen` of Algorithm 5.9. Also note that both transpositions are homogeneous and that the first and last bitstrings do not differ by a homogeneous transposition.

```
procedure gen ( n, k, p : ℕ);
begin
    if p = 1 then
        if k = n − 1 then
            for i := n − 1 downto 1  do swap( i, i + 1 );
        else
        if 0 < k  and k < n − 1 then
            neg( n − 1, k, 1 ); swap( n − 2, n ); gen( n − 1, k − 1, 0 );
        else {p = 2}
            if 0 < k  and k < n − 1 then
                gen( n − 1, k, 1 ); swap( n − 1, n ); gen( n − 1, k − 1, 1 );
end  {of gen};
```

Algorithm 5.9: Direct [2-Trans] generation of Combinations.

Figure 5.10 shows the lists arising from each of the algorithms presented in the last three subsections as well as from the [A-Trans] subsection, which is to be found below.

[2A-Trans]

This closeness condition will be dealt with in Section 5.10 on linear extensions of posets, where it forms a special case of a more general setting.

1 2 3	1 1 1 0 0 0	1 1 1 0 0 0	0 0 1 1 1 0
1 3 4	1 0 1 1 0 0	1 1 0 1 0 0	1 0 0 1 1 0
2 3 4	0 1 1 1 0 0	1 0 1 1 0 0	0 1 0 1 1 0
1 2 4	1 1 0 1 0 0	0 1 1 1 0 0	0 1 1 0 1 0
1 4 5	1 0 0 1 1 0	0 1 1 0 1 0	1 0 1 0 1 0
2 4 5	0 1 0 1 1 0	1 0 1 0 1 0	1 1 0 0 1 0
3 4 5	0 0 1 1 1 0	1 1 0 0 1 0	1 1 1 0 0 0
1 3 5	1 0 1 0 1 0	1 0 0 1 1 0	1 1 0 1 0 0
2 3 5	0 1 1 0 1 0	0 1 0 1 1 0	1 0 1 1 0 0
1 2 5	1 1 0 0 1 0	0 0 1 1 1 0	0 1 1 1 0 0
1 5 6	1 0 0 0 1 1	0 0 1 1 0 1	0 1 1 0 0 1
2 5 6	0 1 0 0 1 1	1 0 0 1 0 1	1 0 1 0 0 1
3 5 6	0 0 1 0 1 1	0 1 0 1 0 1	1 1 0 0 0 1
4 5 6	0 0 0 1 1 1	0 1 1 0 0 1	1 0 0 1 0 1
1 4 6	1 0 0 1 0 1	1 0 1 0 0 1	0 1 0 1 0 1
2 4 6	0 1 0 1 0 1	1 1 0 0 0 1	0 0 1 1 0 1
3 4 6	0 0 1 1 0 1	1 0 0 0 1 1	0 0 1 0 1 1
1 3 6	1 0 1 0 0 1	0 1 0 0 1 1	1 0 0 0 1 1
2 3 6	0 1 1 0 0 1	0 0 1 0 1 1	0 1 0 0 1 1
1 2 6	1 1 0 0 0 1	0 0 0 1 1 1	0 0 0 1 1 1

Figure 5.10: [Trans],[H-Trans],[2-Trans]($\mathbf{R}(6,3,1)$).

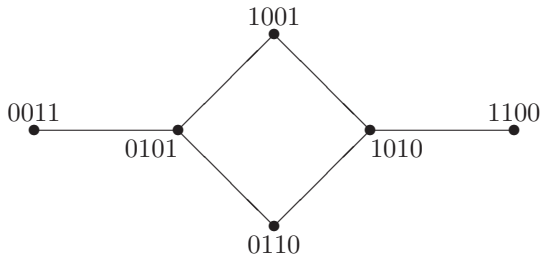


Figure 5.11: Non-Hamiltonian Adjacent Transposition graph.

[A-Trans]

The adjacent transposition closeness relation is that the two bitstrings differ by the interchange of two *adjacent* bits; that is, a 01 becomes a 10, or vice-versa. In this case it is not always possible to have a Gray code. One exists if and only if n is even and k is odd. Consider the case of $n = 4$ and $k = 2$. The underlying graph, which clearly has no Hamilton path, is shown in Figure 5.3; in general this graph is denoted $G(n, k)$.

The graph $G(n, k)$ is bipartite. The vertices may be partitioned into two sets $Even(n, k)$ and $Odd(n, k)$ depending upon the parity of their distance from the vertex $0^{n-k}1^k$. Clearly, an adjacent transposition changes the parity of a bitstring. Now consider the *parity difference*, $D(n, k)$, which is defined to be $Even(n, k) - Odd(n, k)$. We determine this quantity by using an involution ϕ on $\mathbf{B}(n, k)$. Define $\phi(b_1 b_2 \cdots b_n)$ to be

$$b_1 b_2 \cdots b_{2i-2} b_{2i} b_{2i-1} b_{2i+1} \cdots b_n,$$

where i is the smallest index for which $b_{2i-1} \neq b_{2i}$; if there is no such index then the bitstring

is unmodified — it is a fixed point. If the number of 0's and 1's are both odd, then there are no fixed points. Otherwise the fixed points consist of a string of 00 and 11 pairs, followed by a 1 or a 0 if n is odd. Furthermore, all of these fixed points have even parity since it takes 4 adjacent transpositions to transform 1100 into 0011 (and 2 to transform 110 into 011). Thus the number of fixed points is equal to

$$D(n,k) \;=\; \begin{cases} 0 & \text{if } n \text{ even and } k \text{ odd} \\ \binom{\lfloor n/2 \rfloor}{\lfloor k/2 \rfloor} & \text{if } n \text{ odd or } k \text{ even.} \end{cases}$$

Hence $D(n,k) \leq 1$ if and only if n is even and k is odd, or $k = 0, 1, n-1, n$.

THEOREM 5.1 *If n is even and k is odd, then there is a Hamilton path in $G(n,k)$ that starts at 1^k0^{n-k} and ends at $0^{n-k}1^k$.*

Proof: Assume that n is even and k is odd. The proof proceeds by induction on n and k. The theorem is clearly true when $k = 1$ or $k = n-1$ (by moving the single 1 or 0 from left-to-right through the remaining bits).

Let $1 < k < n$. The proof is based on the recurrence relation given below, obtained by iterating (2.2).

$$\binom{n}{k} \;=\; \binom{n-2}{k-2} + 2\binom{n-2}{k-1} + \binom{n-2}{k}$$

First, all $\binom{n-2}{k-2}$ bitstrings with prefix 11 are generated, followed by those beginning 01 or 10 (of which there are $2\binom{n-2}{k-1}$), followed by the $\binom{n-2}{k}$ with prefix 00. The list starts with the bitstring 1^k0^{n-k} and ends with the bitstring $0^{n-k}1^k$. Inductively, those beginning 11 or 00 can be generated. Those beginning 11 are listed from $111^{k-2}0^{n-k}$ to $110^{n-k}1^{k-2}$, and those beginning 00 are listed from 001^k0^{n-k-2} to $000^{n-k-2}1^k$. The complicated part of the proof is in listing those bitstrings that begin 01 or 10. A special type of tree that is used in this part of the proof is defined below.

DEFINITION 5.1 A *comb* is a tree where each vertex has degree at most three and all vertices of degree three lie along a single path which is called the *spine* of the comb. The paths that are attached to the spine are called *teeth*.

Let us consider the specific case of $n = 8$ and $k = 5$. There are $\binom{n-4}{k-2} = \binom{4}{3} = 4$ bitstrings with prefix 1010, and similarly there are 4 with each prefix 1001, or 0110, or 0101. Inductively, the four suffixes are 1110, 1101, 1011, and 0111. Denote the list of $m = \binom{n-4}{k-2}$ bitstrings prepended with 10 by $p_1, p_2, \ldots, p_m$, and when prepended with 01 by $q_1, q_2, \ldots, q_m$. In our example, the p list is 101110, 101101, 101011, 100111, and the q list is 011110, 011101, 011011, 010111. Note that

$$q_1, p_1, p_2, q_2, q_3, \ldots, p_{m-1}, p_m, q_m$$

is a path in $G(n-2, k-1)$. This path is the spine of the comb. The bitstrings of $\mathbf{B}(n-2, k-1)$ that begin 00 or 11 are attached as the teeth of the comb; those that begin 00 are attached to q vertices, and those that begin 11 are attached to p vertices. The tooth attached to a q vertex is obtained by moving its leftmost 1 to the right until it encounters another 1. The tooth attached to a p vertex is obtained by moving the leftmost 0 to the right until it

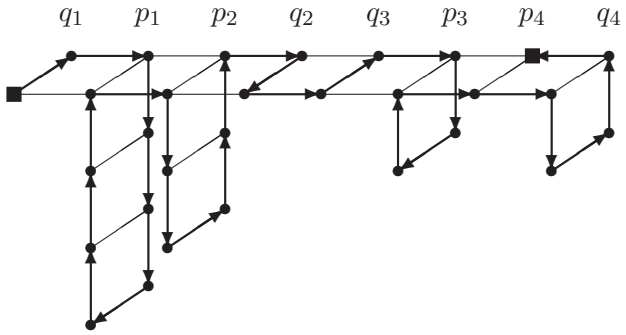
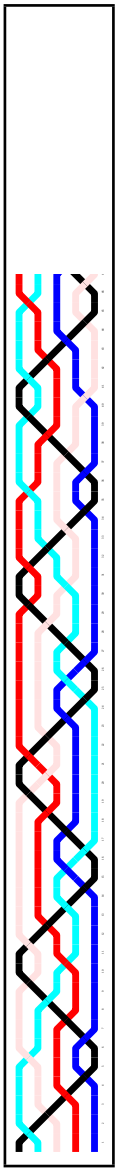


Figure 5.12: Path in the prism of combs for $n = 8$ and $k = 5$.

encounters another 0. The following table lists the vertices of the spine for our example as the leftmost column, and the bitstrings to the right are the teeth of the comb.

q_1	011110			
p_1	101110	110110	111010	111100
p_2	101101	110101	111001	
q_2	011101			
q_3	011011			
p_3	101011	110011		
p_4	100111			
q_4	010111	001111		



There are two combs for $\mathbf{C}(n, k)$ depending on whether the bitstring starts 01 or 10. In other words, we have the product graph of the comb and an edge. Let us call the comb with prefix 10 the *upper* comb and the one with prefix 01 the *lower* comb. It is a simple matter to find a Hamilton path in the two combs that starts at the upper vertex $10p_m$ and ends at the lower vertex $01q_1$. Since $p_m = 10^{n-k-1}1^{k-1}$ and $q_1 = 01^{k-1}0^{n-k-2}$ the proof will be finished. The Hamilton path in prism of combs for our example is illustrated in Figure 5.12.

When viewed along the spines the path sequence starts $10p_m$, $10q_m$, $01q_m$, $01p_m$. Thereafter, the patterns $01p_i$, $10p_i$, $10q_i$, $01q_i$, and $01q_i$, $01p_i$, $10p_i$, $10q_i$ alternate as i decreases from $m-1$ to 1. Of course, the teeth have to be generated along the way as well. This finishes the proof. □

An efficient implementation of this proof a little tricky. See Hough and Ruskey [180] for a CAT implementation.

5.4 Permutations

The most natural closeness condition for permutations is that they differ by a transposition of two elements. There are many algorithms for generating permutations by transpositions. In this section we discuss two such algorithms.

The first algorithm, Algorithm 5.10, is very short and is recursive. It assumes that the array π is initialized to some permutation of $1, 2, \ldots, n$. The initial call is $Permute(n)$.

```
procedure Permute ( m : integer );
var i : integer;
begin
    if m = 0 then PrintIt;
    for i := 1  to m  do
        π_i :=: π_m;
        Permute( m − 1 );
        π_i :=: π_m;
end  {of Permute};
```

Algorithm 5.10: Recursive permutation algorithm.

The number of transpositions use by this algorithm is the subject of Exercise 20, where it is found to use more transpositions than the lexicographic algorithm, Algorithm 4.11 of the previous chapter. However, it is a CAT algorithm.

If we further insist that the transposition be of elements in adjacent positions (just like [A-Trans] in the previous section) then the problem is elegantly solved by the Steinhaus-Johnson-Trotter (SJT) algorithm ([405], [198], [418]).

The basic idea is a recursive sublist expansion. The largest symbol n "sweeps" from right to left and then left to right, etc, through each of the $(n-1)!$ permutations of $1, 2, \ldots, n-1$. When the position of n becomes extreme (either at the right or left), then recursively an adjacent transposition is done between two permutations in the list of permutations of $[n-1]$. If this idea is used with n = 3,4, then we obtain the following list of permutations (read down).

12**3**	**3**12	2**3**1	123**4**	312**4**	231**4**
			124**3**	314**2**	234**1**
			142**3**	341**2**	243**1**
			4123	**4**312	**4**231
1**3**2	**3**21	21**3**	**4**132	**4**321	**4**213
			143**2**	342**1**	241**3**
			134**2**	324**1**	214**3**
			132**4**	321**4**	213**4**

A recursive algorithm based on these ideas appears as Algorithm 5.11. Initially we set $\pi = \pi^{-1} = id$ and $dir[i] := -1$ for $i = 1, 2, \ldots, N$. Variable $dir[i]$ indicates whether element i is currently moving right $(+1)$ or left (-1). The intial call is Perm(1), which generates all permutations of $[N]$. The algorithm is CAT because, except at the root, the degree of each node in the computation has degree greater than one, and a constant amount of computation is done at each node.

We now construct an iterative version Next of the recursive Algorithm 5.11 described above. As before, each element, i, of the permutation will have a direction, $dir[i]$, associated with it. Initially, permutations π (`pi[1..n]`) and π^{-1} are set to be the identity permutation, and all directions are -1, to the left. Call an element *mobile* if it is larger than the element adjacent to it in the appropriate direction. The algorithm simply transposes the largest mobile integer, call it m, with the element to the left or right of it according to the direction $dir[m]$. To stop n from moving beyond the array bounds we set π_0 and π_{n+1} to both be

```
procedure Move( x, d : ℕ);
local j : ℕ;
begin
    j := π^{-1}[x]; π[j] := π[j + d]; π[j + d] := x;
    π^{-1}[x] := j + d; π^{-1}[π[j]] := j;
end {of Move};

procedure Perm ( n : ℕ);
local i : ℕ;
begin
    if n > N then PrintIt
    else
        Perm( n+1 );
        for i := 1 to n − 1 do
            Move( n, dir[n] ); Perm( n + 1 );
    dir[n] := −dir[n];
end {of Perm};
```

Algorithm 5.11: Recursive Pascal implementation of the SJT Algorithm.

$n+1$. The directions of all integers in $[N]$ larger than m are changed and we are ready for the next iteration. These ideas are implemented as Algorithm 5.12.

```
procedure Next;
{Assumes that π_0 = π_{n+1} = n + 1.}
begin
    m := n;
    while π[π^{-1}[m] + d[m]] > m do          {Find largest mobile integer}
        d[m] := −d[m];  m := m − 1;
    π[π^{-1}[m]] :=: π[π^{-1}[m] + d[m]]; {update π}
    π^{-1}[π[π^{-1}[m]]] :=: π^{-1}[m];       {update π^{-1}}
end {of Next};
```

Algorithm 5.12: Iterative Next implementation of the SJT algorithm (assumes $\pi_0 = \pi_{n+1} = n+1$).

We can place Next in a repeat loop with $m = 0$ as the terminating condition, as long as the initial values of $\pi_{-1} = \pi_{-1}^{-1} = -1$ are set. The loop of Algorithm 5.12 can be eliminated by introducing another array $\tau_1, \ldots, \tau_{n+1}$, where τ_j is the largest element smaller than j which is mobile. The development of this loopless algorithm is the subject of Exercise 21.

It is easy to derive a ranking algorithm for the SJT algorithm based on the following recurrence relation, where k is the position of n in π (i.e., $k = \pi^{-1}(n)$).

$$Rank(\pi) \;=\; n \cdot Rank(\pi - n) + \begin{cases} n-k & \text{if } Rank(\pi - n) \text{ is even} \\ k-1 & \text{if } Rank(\pi - n) \text{ is odd} \end{cases}$$

For example, $Rank(1324) = 4 \cdot Rank(132) + 4 - 1 = 7$, since $Rank(12) = 0$ and so $Rank(132) = 3 \cdot Rank(12) + 3 - 2 = 1$. A explicit Pascal implementation is given as Algorithm 5.13. The algorithm uses $\Theta(n^2)$ arithmetic operations in the worst case and is thus slower than the ranking algorithm for lexicographic order.

```
function rank ( n : ℕ) : ℕ;
local j, k, r : ℕ;
begin
    if n = 1 then return( ( ) 0 );
    j := 1;  k := 1;          {Determine k}
    while π[j] ≠ n  do
        if π[j] < n then k := k + 1;
        j := j + 1;
    r := rank( n − 1 );       {Apply recursive formula}
    if r is odd
        then rank := n ∗ r + k − 1
        else rank := n ∗ r + n − k;
end  {of rank};
```

Algorithm 5.13: Ranking algorithm for SJT Algorithm.

Algorithm 5.14 unranks permutations in the SJT order. It sets both π and the directions `dir[i]` and is thus "complete" with respect to *Next*. It also uses $\Theta(n^2)$ arithmetic operations in the worst case.

```
procedure UnrankSJT ( r : ℕ);
local j, k, rem, c : ℕ;
begin
    for j := 1  to n  do π[j] := 0;
    for j := n  downto 1  do
        rem := r mod j;     r := r/j;
        if r is odd then
            k := 0;  dir[j] := +1;
        else
            k := n + 1;  dir[j] := −1;
        c := −1;
        repeat k := k + dir[j];
            if π[k] = 0 then c := c + 1;
        until c = rem;
        π[k] := j;
end  {of UnrankSJT};
```

Algorithm 5.14: Unranking algorithm for SJT algorithm.

5.5 Gray Codes for Binary Trees

The two most common representations of binary trees are as well-formed parentheses strings and the usual computer implementation as a collection of nodes, each consisting of two pointers. We will develop Gray codes for each of these representations.

5.5.1 Well-formed Parentheses

Let $\mathbf{T}(n,k)$ denote the set of all bitstrings in $\mathbf{T}(n)$ with prefix 1^k0. These bitstrings correspond to binary trees with leftmost leaf at level k. If 1^k0s is an element of $\mathbf{T}(n,k)$ then the 1's in s will be called the *free* 1's (since their position within the bitstrings can vary). It can be shown that there is no adjacent transposition ([A-Trans]) Gray code for $\mathbf{T}(n,k)$ for even n or for odd $n > 5$ and odd k (except when $k = 0, 1, n-1,$ or n). A reason for considering a Gray code for $\mathbf{T}(n,k)$ is that a Gray code for $\mathbf{T}(n+1,1)$ can be trivially transformed into a Gray code for $\mathbf{T}(n)$ by ignoring the 10 prefix of every bitstring in $\mathbf{T}(n+1,1)$.

By the proof of (4.17) given in the previous chapter, the following expression gives a recursive way of listing all the elements of $\mathbf{T}(n,k)$.

$$\mathbf{T}(n,k) = \begin{cases} flip(\mathbf{T}(n,2)) & \text{if } k = 1 \\ flip(\overline{\mathbf{T}(n,k+1)}) \circ insert(\mathbf{T}(n-1,k-1)) & \text{if } 1 < k < n \\ 1^n0^n & \text{if } k = n \end{cases} \tag{5.6}$$

In a slight abuse of notation, we let $\mathbf{T}(n,k)$ stand for both the set of bitstrings and the list of bitstrings produced by (5.6). By the remarks above, the corresponding recurrence with $\mathbf{T}$ replaced by $\overline{\mathbf{T}}$, and vice-versa, also holds. The reason for reversing the sublist in (5.6) is, of course, to produce a Gray code list as per the following theorem.

THEOREM 5.2 *For any values of n and k, where $1 \le k \le n$, the list $\mathbf{T}(n,k)$, as defined in (5.6), is a Gray code list satisfying the [Trans] condition.*

PROOF: We will prove the theorem by mathematical induction on increasing values of n and decreasing values of k with the hypothesis, $P(n,k)$, that $\mathbf{T}(n,k)$, for $1 \le k \le n$, is a Gray code, and that (5.7) and (5.8) hold.

$$first(\mathbf{T}(n,k)) = \begin{cases} 101100(10)^{n-3} & \text{if } k = 1 \\ 1^k010^k(10)^{n-k-1} & \text{if } 1 < k < n \\ 1^n0^n & \text{if } k = n \end{cases} \tag{5.7}$$

$$last(\mathbf{T}(n,k)) = 1^k0^k(10)^{n-k} \text{ for } 1 \le k \le n \tag{5.8}$$

By definition, $P(n,k)$ is true for $n = 1$ and for $k = n$. We have to show that if $P(m,l)$ holds for all $m < n$, and for all $l > k$ when $m = n$ then this implies $P(n,k)$. Clearly, the operations $first$ and $last$ commute with $flip$ and $insert$.

To show (5.7) observe for $1 < k < n$ that

$$\begin{aligned} first(\mathbf{T}(n,k)) &= first(flip(\overline{\mathbf{T}(n,k+1)})) \\ &= flip(last(\mathbf{T}(n,k+1))) \\ &= flip(1^{k+1}0^{k+1}(10)^{n-k-1}) \\ &= 1^k010^k(10)^{n-k-1} \end{aligned}$$

And for $k = 1$

$$\begin{aligned} first(\mathbf{T}(n,1)) &= flip(first(\mathbf{T}(n,2))) \\ &= flip(110100(10)^{n-3}) \\ &= 101100(10)^{n-3} \end{aligned}$$

To show (5.8) observe for $1 \leq k \leq n$ that

$$\begin{aligned} last(\mathbf{T}(n,k)) &= insert(last(\mathbf{T}(n-1,k-1))) \\ &= insert(1^{k-1}0^{k-1}(10)^{n-k}) \\ &= 1^k0^k(10)^{n-k} \end{aligned}$$

That $\mathbf{T}(n,k)$ is a Gray code will follow from the inductive assumption and from the proper 'interface' between the last bitstring of the first sublist $last(flip(\overline{\mathbf{T}(n,k+1)}))$ and the first bitstring of the second sublist $first(insert(\mathbf{T}(n-1,k-1)))$. If $k=1$ then there is no interface to consider, and for $2 < k < n-1$,

$$\begin{aligned} last(flip(\overline{\mathbf{T}(n,k+1)})) &= flip(first(\mathbf{T}(n,k+1))) \\ &= flip(1^{k+1}010^{k+1}(10)^{n-k-1-1}) \\ &= 1^k0110^{k+1}(10)^{n-k-2} \\ &= 1^k0\underline{1}10^{k-1}\underline{0}0(10)^{n-k-2} \end{aligned}$$

and

$$\begin{aligned} first(insert(\mathbf{T}(n-1,k-1))) &= insert(first(\mathbf{T}(n-1,k-1))) \\ &= insert(1^{k-1}010^{k-1}(10)^{n-k-1}) \\ &= 1^k0010^{k-1}(10)^{n-k-1} \\ &= 1^k00\underline{1}0^{k-1}\underline{1}0(10)^{n-k-2} \end{aligned}$$

If $k=2$, $last(flip(\overline{\mathbf{T}(n,k+1)})) = 110\underline{1}1\underline{0}00(10)^{n-4}$ and $first(insert(\mathbf{T}(n-1,k-1))) = 110\underline{0}1\underline{1}00(10)^{n-4}$. If $k=n-1$ then $last(flip(\overline{\mathbf{T}(n,n)}) = flip(1^n0^n) = 1^{n-1}0\underline{1}\underline{0}0^{n-2}$ and $first(insert(\mathbf{T}(n-1,k-2)) = 1^{n-1}0\underline{0}\underline{1}0^{n-2}$. In both cases, as indicated by the underlined bits, the two bitstrings differ on exactly two positions. □

The recursive algorithm producing the list $\mathbf{T}(n,k)$ is simple enough to be given explicitly as Algorithm 5.15. The global array `x` is initialized with 1^k0^{2n-k} and the initial call is `T(n,k,k+2,true)`. The parameter `pos` keeps track of where the free 1's are to be positioned.

```
procedure T ( n, k, pos : ℕ; dir : boolean );
begin
    if k = 0 or k = n then PrintTree else
        if k = 1 then
            x[pos] := 1; T( n, k+1, pos+1, dir ); x[pos] := 0;
        else
            if dir then
                x[pos] := 1; T( n, k+1, pos+1, not dir ); x[pos] := 0;
                T( n-1, k-1, pos+1, dir );
            else
                T( n-1, k-1, pos+1, dir );
                x[pos] := 1; T( n, k+1, pos+1, not dir ); x[pos] := 0;
end {of T};
```

Algorithm 5.15: Indirect Gray code algorithm for well-formed parentheses.

Let $t(n,k)$ denote the number of calls to procedure `T` of Algorithm 5.15 with parameters n and k. Since on every level of recursive calls, procedure `T` requires constant amount of computation, the time complexity of the algorithm is proportional to $t(n,k)$. The numbers $t(n,k)$ satisfy the recurrence relation given below.

$$t(n,k) = \begin{cases} 1 + t(n,2) & \text{if } k = 1 \\ 1 + t(n,k+1) + t(n-1,k-1) & \text{if } 1 < k < n \\ 1 & \text{if } k = n \end{cases}$$

It can be shown by an easy induction that $t(n,k) \leq 3T(n,k) - 2$ for $1 < k \leq n$ and $t(n,1) \leq 3T(n,1) - 1$ (see Lemma 4.4). This proves that $t(n,k)$ is proportional to $T(n,k)$ and thus there are constant (on the average) changes necessary to produce the next string in the list.

```
procedure gen ( n, k, p : ℕ);
begin
    if k = 1 then gen( n, 2, p ) else
    if 1 < k  and k < n then
        neg( n, k + 1, p );
        if k = n − 1 then swap( n + 1 + p, n + 2 + p ) else
        if k = 2 then swap( 4 + p, 6 + p )
        else swap( k + 2 + p, 2k + 3 + p );
        gen( n − 1, k − 1, p + 2 );
end  {of gen};
```

Algorithm 5.16: Direct Gray code algorithm for well-formed parentheses.

With the proof of Theorem 5.2, it is not difficult to derive a direct algorithm as well, and one is presented as Algorithm 5.16. The initial call is `gen(n,k,0)`. Parameter `p` keeps track of how far into the sequence we have proceeded as a result of recursive calls. The array holding the well-formed parentheses string must be initialized according to (5.7). This algorithm is also a CAT algorithm even though the call when $k = 1$ causes no call to `swap`. This is because that call results in a call with $k = 2$, which does call `swap`.

Note that the bitstrings $last(\mathbf{T}(n,k))$ and $first(\mathbf{T}(n,k-1))$ are adjacent, as are $last(\mathbf{T}(n,2))$ and $last(\mathbf{T}(n,1))$. Thus the following is a Gray code list for $\mathbf{T}(n)$.

$$\mathbf{T}(n) = \mathbf{T}(n,n) \circ \mathbf{T}(n,n-1) \circ \cdots \circ \mathbf{T}(n,2) \circ \overline{\mathbf{T}(n,1)}$$

This list starts with bitstring 1^n0^n and ends with $101100(10)^{n-3}$. Note that these bitstrings cannot be obtained from one another by a transposition.

As noted before, in the case of $k = 1$, the algorithm for $\mathbf{T}(n+1,1)$ gives rise to yet another Gray code listing of $\mathbf{T}(n)$ by ignoring the initial 10 prefix of the bitstrings in $\mathbf{T}(n+1,1)$. In this case the list starts with the bitstring $1100(10)^{n-2}$ and ends with $1010(10)^{n-2}$; these bitstrings are adjacent and hence a different list is produced from the one described in the preceeding paragraph. Thus we obtain a Hamilton cycle in the underlying graph.

Both of these lists are different from the list in [316]. Since the algorithm for listing $\mathbf{T}(n,k)$ runs in constant amortized time, the algorithms for listing $\mathbf{T}(n)$ also run in constant amortized time.

5.5.2 Binary Trees

A *rotation* in a binary tree is the familiar operation used in maintaining balanced binary search trees (e.g., Knuth [219]). Figure 5.13 shows a left and a right rotation at the node x. Note the asymmetry with respect to x. We also speak of these two rotations as rotations of the edge (w, x). We will develop an algorithm for generating all binary trees so that each successive tree differs from its predecessor by a single left or right rotation. Note that a rotation does not change the inorder numbering of the nodes.

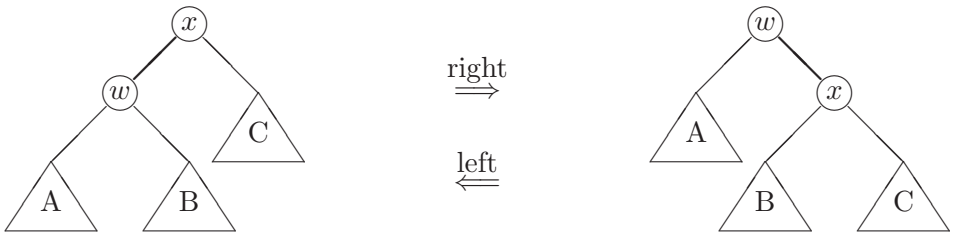


Figure 5.13: A left or right rotation at node x.

Our approach to the problem uses the paradigm of attaching alternating directions to elements or positions in the object in order to obtain "Gray codes" of combinatorial objects. The algorithm does not require the maintenance of any sequence representation of the binary tree and can be implemented to run in constant time per tree on a pointer machine, neglecting the time needed to actually print the tree.

The *rotation graph* G_n has vertex set consisting of all binary trees with n nodes. Two vertices are connected by an edge if a single left or right rotation will transform one tree into the other.

A Hamilton path of G_n is generated by using a recursive strategy. We show how to construct a Hamilton path in G_n from a Hamilton path in G_{n-1}.

The proof uses the idea of an *induced subtree* of a binary tree. Let T be a binary tree of size n. We define $T[i..j]$ to be the tree formed by the nodes i, $i+1$,...,j, where the parent of k, for $i \leq k \leq j$, in $T[i..j]$ is defined to be the node l, for $i \leq l \leq j$, such that l is the nearest proper ancestor of k in T. This is simply the tree formed by contracting any edge in T whose endpoints are not both in the range $[i..j]$. See Figure 5.14.

Consider the effect of a rotation of the edge (x, y) in T on the tree $T[i..j]$. If both x and y are in the range $[i..j]$ then this rotation has the effect of rotating the same edge in $T[i..j]$, otherwise $T[i..j]$ is unchanged. Notice that in the example of Figure 5.14 it is possible to rotate the edge (x, y) in $T[i..j]$ but not in T, because in T the nodes u and v are "in the way".

In the remainder of this section we only use induced subtrees with $i = 1$.

Given $T[1..n-1]$, the only possible way to form a corresponding tree T is to place node n as the root of $T[1..n-1]$, as the right child of $n-1$ in $T[1..n-1]$, or "between" any two nodes along the right path of $T[1..n-1]$. See Figure 5.15.

From this observation we can derive the following result.

THEOREM 5.3 *The rotation graph G_n contains a Hamilton path.*

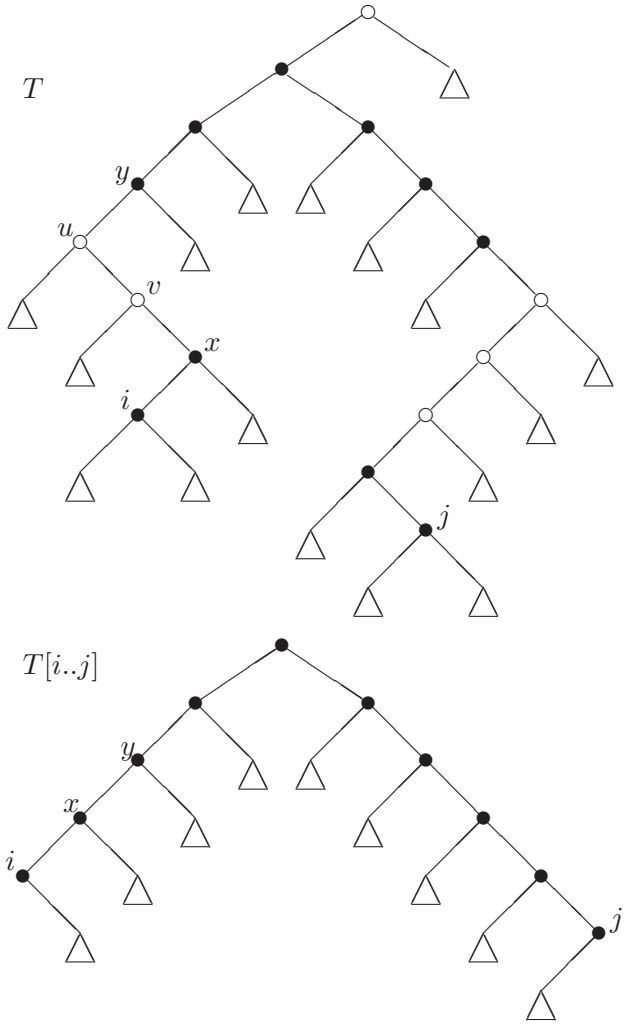
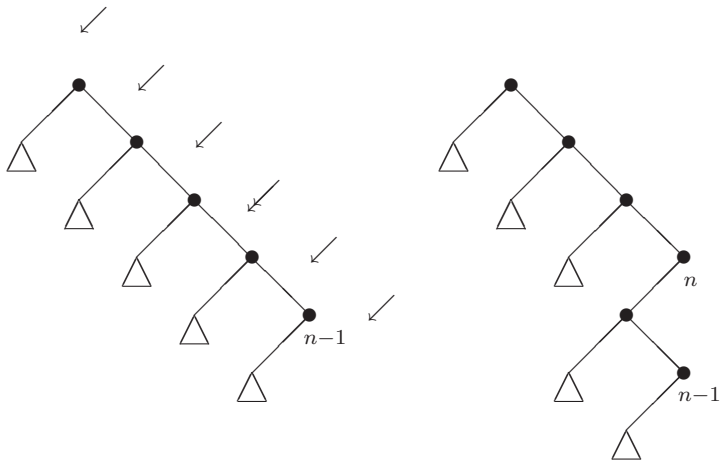


Figure 5.14: Example of induced tree.



(a) $T[1..n-1]$ (b) T

Figure 5.15: (a) Given $T[1..n-1]$ there are 6 places where n could possibly be inserted to get T. (b) The n has been inserted at the position of the double arrow.

Proof: The proof is by induction on n. Clearly there exists a Hamilton path in G_1 since there is only one binary tree with a single node.

Let n be any integer greater than 1. Inductively, there exists a sequence $P_{n-1} = T_0, T_1, \ldots, T_k$ of all trees of size $n-1$ that forms a Hamilton path in G_{n-1}. Using P_{n-1}, we construct a sequence P_n of the trees of size n that forms a Hamilton path in G_n. The initial tree in this sequence is formed by placing node n as the right child of $n-1$ in T_0. We then perform precisely the same sequence of rotations used to generate P_{n-1}, but before performing each rotation of P_{n-1} we first rotate n as follows:

1. If n is not the root, then repeatedly rotate the edge $(parent(n), n)$ until n is the root. These are left rotations at n.

2. If n is the root, then repeatedly rotate the edge $(leftchild(n), n)$ until n has no left child. These are right rotations at n.

Finally, after performing the last rotation of P_{n-1}, we repeat the step above.

Clearly every tree T of size n is generated, since every possible shape of the subtree $T[1..n-1]$ is generated, and for each such shape all the trees of size n whose induced subtree of the nodes 1 to $n-1$ has that shape have been generated.

Clearly it is possible to perform all of the rotations at n. Any rotation that does not contain n as one of the endpoints of the rotated edge can also be performed. Such a rotation is possible as long as n is not "in the way", which is true since n is either the root of the tree or the right child of $n-1$ whenever such a rotation of P_{n-1} is done. □

This proof leads directly to an efficient algorithm for generating all binary trees by rotations. One such implementation is given in Figure 5.17. Procedure call `List(t)` generates all binary trees T whose induced subtree $T[1..t-1]$ equals the induced subtree of the current tree. This procedure is initially called with `t` pointing to node 2. The running time of this

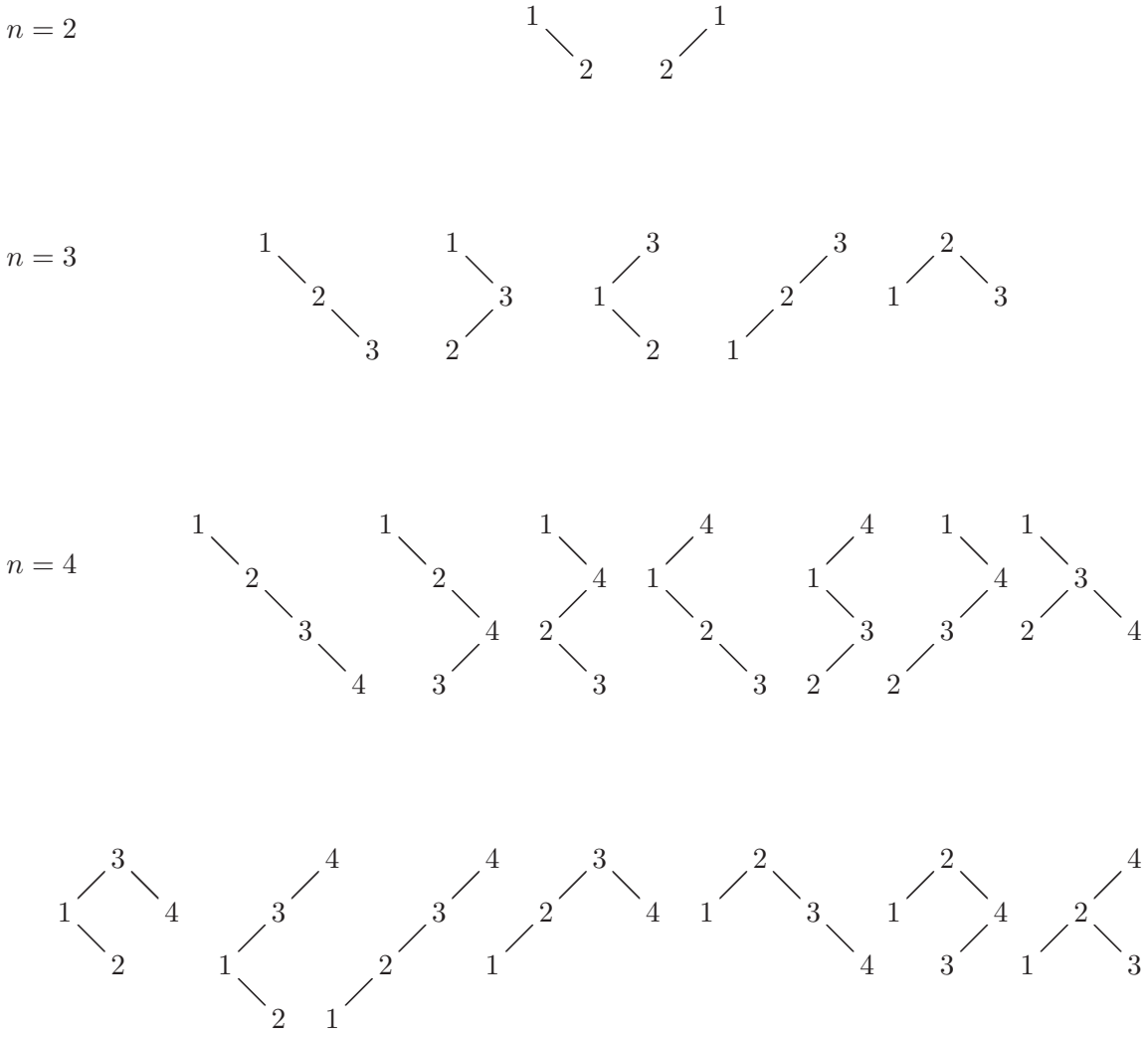


Figure 5.16: List of trees for $n = 2, 3, 4$.

algorithm is determined by the number of recursive calls to `List`, i.e., by the size of the recursion tree. From the proof above we know that the number of nodes in the recursion tree is the sum of the first n Catalan numbers, since the list P_n of all C_n binary trees with n nodes is created from the list P_{n-1} of all C_{n-1} binary trees with $n-1$ nodes, and so on. This sum is bounded above by $1.5C_n$ (for $n \geq 6$), as may be proven by induction from the recurrence relation $C_n = (4n-2)C_{n-1}/(n+1)$. Asymptotically, the ratio of the first n Catalan numbers to C_n is 4/3; see Exercise 22. Thus `List` runs in constant amortized time (i.e., $O(1)$ amortized time) per tree; it is a CAT algorithm.

```
procedure List( t : ptr );
begin
    if t = null then PrintTree
    else
        List( t.sym_succ );
        if t.left = null then {increasing}
            while LeftRotationPossible( t ) do
                RotateLeft( t );
                List( t.sym_succ );
        else {decreasing}
            while RightRotationPossible( t ) do
                RotateRight( t );
                List( t.sym_succ );
end {of List};
```

Algorithm 5.17: A procedure to generate trees by rotations.

5.6 Multisets

5.6.1 Permutations of a multiset

A most natural generalization of k-combinations of an n-set is that of a permutation of a multiset. Combinations correspond to permutation of a multiset with two types of elements. As in the section on combinations we consider permutations of a multiset under various adjacency criteria, but the solutions attained are not so elegant.

We present two principle results. The first is an existence result, based on prefix trees, showing the existence of a Gray code satisfying the [Trans] condition and the prefix property. The second result is a direct CAT algorithm which satisfies the [H-Trans] condition but that does not satisfy the prefix or suffix property. This algorithm is based on the Eades-McKay algorithm for generating combinations by homogeneous transpositions [H-Trans].

A prefix tree Gray code

When we do not wish to specify the label at the root, we write simply n-tree. An n-tree is also called a *multiset tree*. We regard multiset trees as being embedded in the plane with the root on the left and leaves on the right. The ordering of subtrees is from top to bottom. To each n-tree T there is a corresponding list $L(T)$ of all n-permutations; this list is precisely the list of permutations printed by *GenBag*. Each permutation can be obtained by traversing T from its root to a leaf and recording the labels of the nodes encountered.

DEFINITION 5.2 *A multiset tree T is an* interchange tree *if successive permutations in list $L(T)$ differ by an interchange of two elements.*

The following examples illustrate the preceding definitions. For $n = \langle 2, 1\rangle$, there are exactly 4 multiset trees. The list 001, 100, 010 is not $L(T)$ for any multiset tree T. For $n = \langle 2, 2\rangle$, there is a unique multiset tree T such that $L(T) =$ 0011, 0110, 0101, 1010, 1001, 1100; however, T is not an interchange tree.

We call a node of a multiset tree a *plus-node* if its children are labeled in increasing order and a *minus-node* if they are labeled in decreasing order. A node with only one child is both a plus-node and a minus-node.

DEFINITION 5.3 *An* alternating tree *is a multiset tree in which every internal node is either a plus-node or a minus-node and, at each level of the tree, reading top-to-bottom, plus- and minus-nodes alternate.*

We now prove a sufficient condition for a multiset tree to be an interchange tree and use it to derive an interchange algorithm.

THEOREM 5.4 *Any alternating tree T is also an interchange tree.*

PROOF: Let $\pi = \alpha a \beta c \gamma$ and $\pi' = \alpha b \beta d \delta$ be two successive permutations in $L(T)$, where $a, b, c, d \in \{0, 1, \ldots, t\}$, $a \neq b$, $c \neq d$, and $\alpha, \beta, \gamma, \delta \in \{0, 1, \ldots, t\}^*$. In words, the first symbols that differ are a and b and the next symbols that differ are c and d. The nodes labeled a and b have a common parent; call the tree rooted at their parent an m-tree — thus, like n, m is a sequence that represents a multiset. Nodes c and d have different parents; call the tree rooted at c's parent a p-tree and the tree rooted at d's parent a q-tree. There are two symmetric cases, depending upon whether $a < b$ or $a > b$. We discuss only the case $a < b$.

Suppose $a < b$. Since children are labeled monotonically, $m_i = 0$ for all $a < i < b$. This implies $p_i = 0$ and $q_i = 0$ for all $a < i < b$. Since β is common to both π and π', $p_i = q_i$ for $i \neq a$ and $i \neq b$; further, $q_a = p_a + 1$ and $p_b = q_b + 1$. Because T is alternating, the nodes labeled c and d both have either the largest or the smallest labels among their respective siblings.

In the "smallest" case, we have $c = \min\{i \mid p_i > 0\}$ and $d = \min\{i \mid q_i > 0\}$. We argue that $d = a$ — the proof that $c = b$ is similar.

$$
\begin{array}{ll}
 & d \\
= & \quad \langle d \text{ is the smallest sibling} \rangle \\
 & \min\{i \mid 0 \leq i \ \wedge \ q_i > 0\} \\
= & \quad \langle \text{Since } p_i = q_i \text{ for } i < a, \text{ if some such } q_i > 0 \text{ then } c = d; \text{ but } c \neq d \rangle \\
 & \min\{i \mid a \leq i \ \wedge \ q_i > 0\} \\
= & \quad \langle q_a = p_a + 1 > 0 \rangle \\
 & a
\end{array}
$$

The "larger" case is similar and is omitted. Hence, $a = d$ and $b = c$ in either case. Also, $p|_c = q|_d$ and thus, because T is alternating, $\gamma = \delta$. □

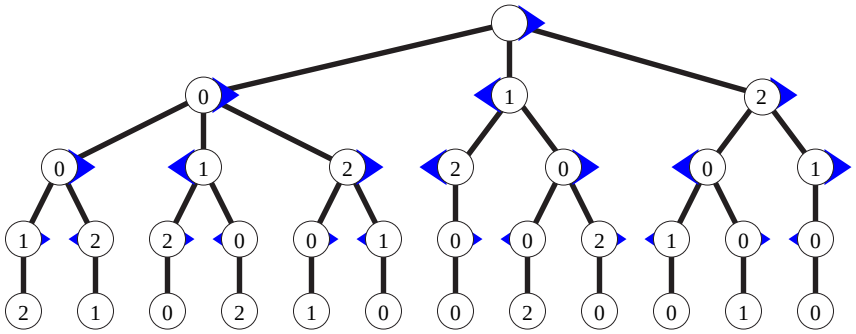


Figure 5.17: Recursion Tree (A) for $n = \langle 2, 1, 1 \rangle$.

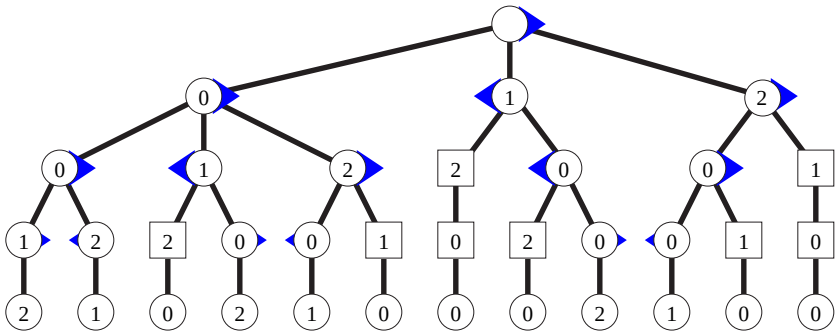


Figure 5.18: Recursion Tree (B) for $n = \langle 2, 1, 1 \rangle$.

A [H-Trans] algorithm

Consider the [H-Trans] algorithm for generating the elements of $\mathbf{B}(n,k)$. In a previous section we developed such an algorithm (Algorithm 5.8), where the initial string was 1^k0^{n-k} and final string was $0^{n-k}1^k$. Now suppose that Algorithm 5.8 was applied to the string $x_1x_2\cdots x_k0^{n-k}$. Because all transpositions are homogeneous, the symbols $x_1, x_2, \ldots, x_k$ always retain their proper order; in particular, the final string will be $0^{n-k}x_1x_2\cdots x_k$. We now have the germ of an idea for generating multiset permutations.

The algorithm is recursive an is reminiscent of the SJT algorithm for generating permutations. Imagine that we have a [H-Trans] listing of the permutations of type $n_1, n_2, \ldots, n_t$, call it

$$l_1, l_2, \ldots, l_m.$$

This listing is expanded into a listing of all permutations of type $n_0, n_1, \ldots, n_t$ by expanding each permutation, l_i, into a list of permutations, L_i, as follows:

$$L_i \;=\; 0^{n_0}l_i, \ldots, l_i0^{n_0} \text{ if } i \text{ odd, and}$$

$$L_i \;=\; l_i0^{n_0}, \ldots, 0^{n_0}l_i \text{ if } i \text{ even.}$$

The L_i lists are produced by using algorithm `gen` (i odd) or `neg` (i even), from the section on generating combinations. If all $n_i = 1$ then this algorithm has the same structure as the SJT algorithm.

Although the main idea is simple, converting it into an efficient algorithm requires a little care. See Algorithm 5.18, which calls `neg` and `gen` of Algorithm 5.8. In addition to n and k, we pass another parameter t to `gen` and `neg`, which is not used directly, but is passed onto `swap`. There are three additional global arrays, `sum`, `dir` and `off`. Array `sum[0..t]` has jth entry `sum`[j] $= \sum_{i=0}^{j-1} n_i$. Array `dir[0..t]` contains boolean values initialized to true, and is used to keep track of whether the the n_i symbols i are moving right or left. Array `off[0..t]` keeps track of the number of blocks of higher symbols that are fixed at the left. Initially, `off`[j] $= \sum_{i=0}^{j-1} n_j$, mirroring the fact that the initial permutation is $0^{n_0}1^{n_1}\cdots t^{n_t}$. The initial call is BigGen(t). In general, the call BigGen(p) generates, in the $\sum_{i=0}^{p} n_i$ locations starting at $1 +$ `off`[p], all multiset permutations of specification $\langle n_0, n_1, \ldots, n_p\rangle$.

5.6.2 Combinations of a multiset

The same general strategy that was used to generate all permutations of a multiset by using a prefix tree Gray code as presented in Section 5.6.1 can also be used to generate the combinations of a multiset. We simply alternate the directions of the subtrees at each level of the computation tree. Combinations of a multiset were generated in lexicographic order in the previous chapter in Section 4.5.1. Recall that a k-combination of a multiset of specification $\langle n_0, n_1, \ldots, n_t\rangle$ is represented as a string $a_0a_1\cdots a_t$ that satifies $a_0 + a_1 + \cdots + a_t = k$ and $0 \le a_i \le n_i$.

5.7 Compositions

A k-composition of n is a solution, in natural numbers, of the following equation.

$$x_1 + x_2 + \cdots + x_k \;=\; n$$

```
procedure BigGen ( t : ℕ);
begin
    if dir[t] then gen( t, sum[t − 1], sum[t] ) else neg( t, sum[t − 1], sum[t] );
    if t > 1 then BigGen( t − 1 );
    dir[t] := not dir[t];
    if dir[t] then off[t] := off[t − 1] + num[t − 1] else off[t] := off[t − 1];
end {of BigGen};

procedure swap ( t,x,y : integer );
local b,temp : integer;
begin
    if t > 1 then BigGen( t − 1 );
    b := off[t − 1];
    a[x + b] :=: a[y + b];
    PrintIt;
end {of swap};
```

Algorithm 5.18: Procedures for generating multiset permutations.

For example, the four 2-compositions of 3 are

$$3 = 0 + 3 = 1 + 2 = 2 + 1 = 3 + 0.$$

Each solution can be thought of as a way of placing n unlabelled balls into k labelled boxes. The number of solutions is $\binom{n+k-1}{n}$. We define a Gray code in which successive compositions have the property that they differ by one in exactly two positions; that is, in the balls into boxes interpretation, exactly one ball moves from one box to another.

Consider the list, $\mathbf{Comp}(n,k)$, of all k-compositions of n, as defined below. If $k = 1$, then $\mathbf{Comp}(n,1) = n$ and if $k > 1$, then $\mathbf{Comp}(n,k)$ is defined recursively as in list (a) below.

$$\begin{array}{r} \mathbf{Comp}(n,k-1)\cdot 0\circ \\ \overline{\mathbf{Comp}(n-1,k-1)}\cdot 1\circ \\ \mathbf{Comp}(n-2,k-1)\cdot 2\circ \\ \overline{\mathbf{Comp}(n-3,k-1)}\cdot 3\circ \\ \vdots \\ \mathbf{Comp}(0,k-1)\cdot n \end{array} \qquad \begin{array}{r} \mathbf{Comb}(n,k-1)\cdot 01^0\circ \\ \overline{\mathbf{Comb}(n-1,k-1)}\cdot 01^1\circ \\ \mathbf{Comb}(n-2,k-1)\cdot 01^2\circ \\ \overline{\mathbf{Comb}(n-3,k-1)}\cdot 01^3\circ \\ \vdots \\ \mathbf{Comb}(0,k-1)\cdot 01^n \end{array}$$

(a) (b)

This list has a number of interesting properties which are embodied in the following Lemma. Recall our convention that lower case Greek letters are strings and x, y are "symbols".

LEMMA 5.4 *The list* $\mathbf{Comp}(n,k)$ *satisfies the following properties.*

1. $first(\mathbf{Comp}(n,k)) = n0\cdots 0 = n0^{k-1}$.
2. $last(\mathbf{Comp}(n,k)) = 0\cdots 0n = 0^{k-1}n$.
3. *If* $\mathbf{a}$ *and* $\mathbf{b}$ *are successive strings in* $\mathbf{Comp}(n,k)$*, then* $\mathbf{a}$ *can be written as* $\mathbf{a} = \alpha, x, \beta, y, \gamma$ *where* $\mathbf{b}$ *is either* $\alpha, x+1, \beta, y-1, \gamma$ *or,* $\alpha, x-1, \beta, y+1, \gamma$ *and where* β *is a string of 0's.*

5 0 0 0	2 1 1 1	11111000	11010101
4 1 0 0	1 2 1 1	11110100	10110101
3 2 0 0	0 3 1 1	11101100	01110101
2 3 0 0	0 4 0 1	11011100	01111001
1 4 0 0	1 3 0 1	10111100	10111001
0 5 0 0	2 2 0 1	01111100	11011001
0 4 1 0	3 1 0 1	01111010	11101001
1 3 1 0	4 0 0 1	10111010	11110001
2 2 1 0	3 0 0 2	11011010	11100011
3 1 1 0	2 1 0 2	11101010	11010011
4 0 1 0	1 2 0 2	11110010	10110011
3 0 2 0	0 3 0 2	11100110	01110011
2 1 2 0	0 2 1 2	11010110	01101011
1 2 2 0	1 1 1 2	10110110	10101011
0 3 2 0	2 0 1 2	01110110	11001011
0 2 3 0	1 0 2 2	01101110	10011011
1 1 3 0	0 1 2 2	10101110	01011011
2 0 3 0	0 0 3 2	11001110	00111011
1 0 4 0	0 0 2 3	10011110	00110111
0 1 4 0	1 0 1 3	01011110	10010111
0 0 5 0	0 1 1 3	00111110	01010111
0 0 4 1	0 2 0 3	00111101	01100111
1 0 3 1	1 1 0 3	10011101	10100111
0 1 3 1	2 0 0 3	01011101	11000111
0 2 2 1	1 0 0 4	01101101	10001111
1 1 2 1	0 1 0 4	10101101	01001111
2 0 2 1	0 0 1 4	11001101	00101111
3 0 1 1	0 0 0 5	11100101	00011111

Figure 5.19: The list (a) $\mathbf{Comp}(5,4)$ (first two columns, read down) and the corresponding list (b) $\mathbf{Comb}(5,4)$ of the elements of $\mathbf{B}(8,5)$ (last two columns).

This lemma is easily proven by induction on k. Property 3 states that in successive strings there are two positions that differ by 1 and that only 0's occur between those two positions. Every k-composition of n corresponds to a element of $\mathbf{B}(n+k-1,n)$. This correspondence is obtained by turning each x_i into a run of x_i 1's and separating the k runs of 1's by $k-1$ 0's. For example 300123 corresponds to 11100010110111. In Figure 5.19 we show the list $\mathbf{Comp}(5,4)$, together with the corresponding bitstrings in $\mathbf{B}(8,5)$. Applying this correspondence to the list definition (a) yields the list $\mathbf{Comb}(n,k)$ of elements of $\mathbf{B}(n,k)$ as shown in (b). Because of property 3 of Lemma 5.4, the corresponding list of bitstrings has the [H-Trans] property. In fact, it turns out that this listing is exactly the same as the Eades-McKay listing discussed in Section 5.3.

LEMMA 5.5 *The list* $\mathbf{E}(n+k-1,n)$*, where* $\mathbf{E}(n,k)$ *are the Eades-McKay lists defined in (5.5), and the list* $\mathbf{Comb}(n,k)$ *are identical.*

PROOF: □

Algorithm 5.19 is an indirect implementation of Knuth's algorithm; a direct version is described in the following paragraph. Procedure `gen` has three parameters, n, k, and `dir`, the last of which indicates whether the list is reversed or not. The initial call is

```
procedure gen ( n, k : ℕ; dir : boolean );
local i : ℕ;
begin
    if k = 1 then
        a[k] := n;  PrintIt;
    else
    if dir then
        for i := 0 to n do
            a[k] := i; gen( n − i, k − 1, dir );
            dir := not dir;
        else
            if n is odd then dir := true;
            for i := n downto 0 do
                a[k] := i; gen( n − i, k − 1, dir );
                dir := not dir;
end {of gen};
```

Algorithm 5.19: Indirect implementation of Knuth's algorithm.

```
procedure updown ( p, q : ℕ);
begin
    a[p] := a[p] + 1; a[q] := a[q] − 1;
    PrintIt; end {of updown};

procedure gen ( n, k : ℕ);
local i : ℕ;
begin
    if k > 1 then
        for i := 0 to n − 1 do
            if i is odd then
                neg( n − i, k − 1 ); updown( k, 1 );
            else
                gen( n − i, k − 1 ); updown( k, k − 1 );
end {of gen};

procedure neg ( n, k : ℕ);
local i : ℕ;
begin
    if k > 1 then
        for i := n − 1 downto 0 do
            if i is odd then
                updown( 1, k ); gen( n − i, k − 1 );
            else
                updown( k − 1, k ); neg( n − i, k − 1 );
end {of gen};
```

Algorithm 5.20: Direct implementation of Knuth's algorithm for compositions.

`gen(n,k,true)`. It is clear that this indirect implementation of the recursive definition of **Comp**(n,k) translates into an algorithm whose computation tree has $t(n,k)$ nodes, where

$$t(n,k) = \begin{cases} 1 & \text{if } k = 1 \\ 1 + \sum_{j=0}^{n} t(j,k-1) & \text{if } k > 1 \end{cases}$$

The $k > 1$ case may be simplified to $t(n,k-1) + t(n-1,k)$ (why?).With this simplification the solution $t(n,k) = \binom{n+k}{k-1}$ to this recurrence relation is readily verified. The amortized running time is thus proportional to $(n+k)/(n+1)$, and the algorithm is not CAT if n is small compared with k (i.e., if $n = o(k)$. By eliminating 0-paths, it is possible to construct a direct CAT algorithm.

Algorithm 5.20 is a direct implementation; the initial call is `gen(n,k)`. The direct algorithm is CAT, since every recursive call is matched with a call to `updown`, which increments the element in one position, decrements an element in another position and then calls `PrintIt`. Further algorithmic questions are pursued in the exercises.

5.8 Numerical Partitions

Recall that a numerical partition can be thought of as a solution, in natural numbers, of the equation

$$x_1 + x_2 + \cdots + x_n \;=\; n,$$

where

$$x_1 \geq x_2 \geq \cdots \geq x_n \geq 0$$

The number of parts of the partition is the number of non-zero x_i. What are the appropriate closeness conditions for defining Gray codes of numerical partitions? Taking our clue from the previous section, define two numerical partitions to be *close* if there are two entries that differ by ± 1. The first column of following table shows the partitions that are close to 53221.

p	$\bar{\mathbf{p}}$
53221	54211
432211	6421
43321	5431
44221	5422
52222	55111
532111	63211
5332	44311
54211	53221
5422	44221
62221	541111
63211	532111
6322	53221

The table also shows the conjugate 54211 and the partitions that are close to it (in the second column). We observe that the partitions in both columns are conjugates of each other. This is no accident as the following lemma states.

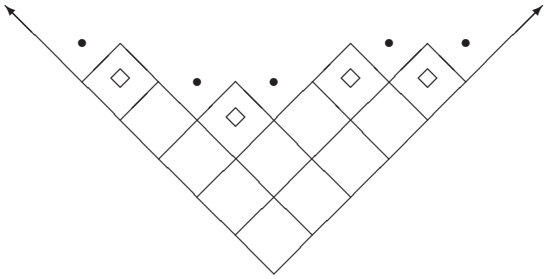


Figure 5.20: Peaks and valleys.

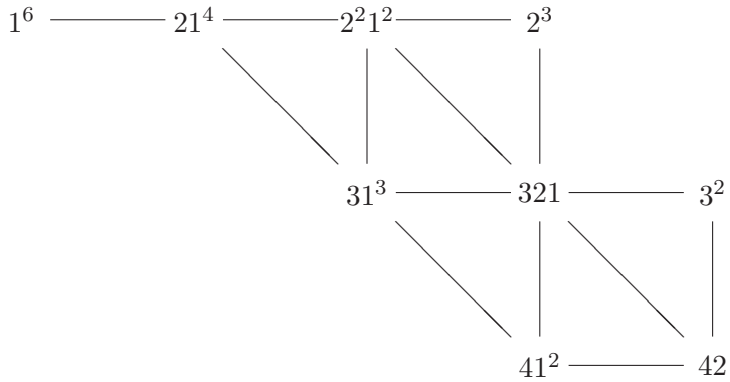


Figure 5.21: The closeness graph $G(6,4)$ of $\mathbf{L}(6,4)$.

LEMMA 5.6 *If* $\mathbf{p}$ *and* $\mathbf{q}$ *are partitions of* n *that are close, then* $\bar{\mathbf{p}}$ *and* $\bar{\mathbf{q}}$ *are also close.*

PROOF: To prove this lemma consider the Ferrer's diagrams of close partitions $\mathbf{p}$ and $\mathbf{q}$. Draw the Ferrer's diagram of p "pointing up" as illustrated in Figure 5.20. That figure represents the partition $5+4+2+1+1$ or its conjugate $5+3+2+2+1$. To each distinct part of the partition there corresponds one "peak", indicated by the small diamonds ($\diamond$). Each peak has a pair of "valleys" on either side; there is one more valley than peak. Valleys are indicated by dots ($\bullet$). Two partitions are close if they can be obtained, one from the other, by moving a square at a peak down into a valley. This operation is the same whether we are considering partitions or their conjugates. □

Define $\mathbf{L}(n,k)$ to be the set of all partitions of n whose largest part is at most k. Thus $\mathbf{L}(n,n)$ is the set of all partitions of n. The *closeness graph* $G(n,k)$ of $\mathbf{L}(n,k)$ has vertices which are the members of $\mathbf{L}(n,k)$ and edges between those vertices that are close. Figure 5.21 shows the closeness graph of $\mathbf{L}(6,4)$. Note that 1^n is always a pendant vertex of $G(n,k)$ for all $0 < k < n$.

Our strategy is to try to find a Hamilton path in the closeness graph. Since 1^n is a pendant vertex, any Hamilton path must start at 1^n, which is the lexicographically least

numerical partition. We will try to make the Hamilton path end at the lexicographically largest partition, which is $k^t r$ where t and r are the quotient and remainder when n is divided by k; i.e., $n = kt + r$ with $0 \le r < k$. Savage [369] has shown that such a Hamilton path always exists except when $(n, k) = (6, 4)$. In that case, the path cannot end at 42, but must end at $2^2 1^2$, 2^3, 31^3, 3^2 or 41^2, as Figure 5.21 clearly shows. Note that generating $\mathbf{P}(n, k)$ reduces to generating $\mathbf{L}(n-k, k)$ by appending a k; i.e., $\mathbf{P}(n, k) = k{\cdot}\mathbf{L}(n-k, k)$.

To show the existence of a Hamilton path in the the general closeness graphs we use recursively constructed lists, some going forwards and some going backwards, as was done for the previous Gray codes of this chapter, but the definitions are quite a bit more complicated in this case. Part of the complication arises because of the (6,4) case and those other cases X that depend upon (6,4). Define

$$X \;=\; \{(6,4),(6,6),(14,4),(15,5),(12,7)\}.$$

All cases of X, depend on (6,4) and they must be handled separately as shown in Figure 5.22. Note that $L(15,5)$ is defined in terms of $M(15,4)$.

In a slight abuse of notation we will use $\mathbf{L}(n, k)$ to stand for both the set defined earlier and the list of partitions to be defined below. It proves convenient to define another set $\mathbf{M}$ which will also be treated as a list. The construction of the Gray code involves a mutual recursion on both $\mathbf{L}$ and $\mathbf{M}$. Define $\mathbf{M}(n, k)$ to be the set of all partitions of n with either largest part at most k or largest part $k+1$ and second largest part less than k. In other words,

$$\mathbf{M}(n,k) \;=\; \mathbf{L}(n,k) \cup (k+1){\cdot}\mathbf{L}(n-k-1, k-1).$$

For example, $M(4,2) = \{2^2, 21^2, 1^4, 31\}$. Just as $\mathbf{L}$ had special cases, so does $\mathbf{M}$. Define

$$Y = \{(11,5),(13,6),(18,4),20,5)\}$$

These lists for these special cases are shown in Figure 5.23.

Here's how the list $\mathbf{L}$ is defined:

$$\mathbf{L}(n,k) = \begin{cases} \varepsilon & \text{if } n = k = 0 \\ \emptyset & \text{if } n > 0 \text{ and } k = 0 \\ 1^n & \text{if } k = 1 \\ \mathit{List}(i,j) & \text{if } (i,j) \in X \\ \mathbf{L}(n,n) & \text{if } n < k \\ \mathbf{M}(n,k-1) & \text{if } n \ge 2k \\ \quad \circ k{\cdot}(k{-}1){\cdot}\overline{\mathbf{L}(n-2k+1,k-1)} & \\ \quad \circ k{\cdot}k{\cdot}\mathbf{L}(n-2k,k) & \\ \mathbf{M}(n,k-1) \circ k{\cdot}(k{-}1) & \text{if } n = 2k{-}1 \\ \mathbf{L}(n,k-2) & \text{if } n = 2k{-}2 \text{ and } n > 4 \\ \quad \circ (k{-}1){\cdot}\overline{\mathbf{L}(k-1,k-3)} & \\ \quad \circ k{\cdot}\mathbf{L}(k-2,k-3) & \\ \quad \circ (k{-}1){\cdot}(k{-}2){\cdot}1 \circ (k{-}1){\cdot}(k{-}1) \circ k{\cdot}(k{-}2) & \\ \mathbf{L}(n,k-2) & \text{if } k \le n < 2k-2 \\ \quad \circ (k{-}1){\cdot}\overline{\mathbf{L}(n-k+1,n-k+1)} & \\ \quad \circ k{\cdot}\mathbf{L}(n-k,n-k) & \end{cases}$$

For the sake of brevity, this definition differs a little from our usual conventions about recursive decompositions. In some cases as sublist may be non-existent. For example, if

$L(6,4) =$

1^6
21^4
31^3
$2^2 1^2$
2^3
321
3^2
42
41^2

$L(6,6) =$

$L(6,3)$
42
41^2
51
6

$L(14,4) =$

1^{14}
21^{12}
$2^2 1^{10}$
$2^3 1^8$
$2^4 1^6$
$2^5 1^4$
$2^6 1^2$
2^7
$32^5 1$
42^5
$42^4 1^2$
$32^4 1^3$
$32^3 1^5$
$42^3 1^4$
$42^2 1^6$
$32^2 1^7$
421^8
41^{10}
31^{11}
321^9
$3^2 1^8$
431^7
$4^2 1^6$
$4^2 21^4$
$4^2 2^2 1^2$
$4^2 2^3$
$43^2 2^2$
$43^2 21^2$
$43^2 1^4$
4321^5
$432^2 1^3$
$432^3 1$
$3^2 2^4$
$3^2 2^3 1^2$
$3^2 2^2 1^4$
$3^2 21^6$
$3^3 1^5$
$3^3 21^3$
$3^3 2^2 1$
$3^4 1^2$
$3^4 2$
$43^3 1$
$4^2 3^2$
$4^2 321$
$4^2 31^3$
$4^3 1^2$
$4^3 2$

$L(15,5) =$

$M(15,4)$
543^2
$54^2 2$
$54^2 1^2$
5431^3
54321
542^3
$542^2 1^2$
5421^4
541^6
$55L(5,5)$

$L(12,7) =$

1^{12}
21^{10}
31^9
$2^2 1^8$
$2^3 1^6$
$2^4 1^4$
$2^5 1^2$
2^6
$32^4 1$
$32^3 1^3$
$32^2 1^5$
321^7
$3^2 1^6$
$3^2 21^4$
$3^3 1^3$
$3^2 2^2 1^2$
$3^2 2^3$
$3^3 21$
3^4
$43^2 2$
$4^2 2^2$
$4^2 21^2$
$4^2 1^4$
431^5
4321^3
$43^2 1^2$
$432^2 1$
42^4
$42^3 1^2$
$42^2 1^4$
421^6
41^8
51^7
521^5
531^4
$52^2 1^3$
$52^3 1$
532^2
5321^2
$53^2 1$
$4^2 31$
4^3
543
5421
$5^2 2$
$5^2 1^2$
541^3
$6\overline{L(6,4)}$
$7L(5,4)$
651
66
75

Figure 5.22: The **L** lists for the special cases in X.

$M(11,5) =$

$L(11,4)$
53^2
542
541^2
531^3
5321
52^3
52^21^2
521^4
51^6
$6L(5,3)$
641
551

$M(13,6) =$

$L(13,5)$
$6\overline{L(7,5)}$
$7L(6,3)$
742
741^2
751
661

$M(18,4) =$

$L(18,3)$
43^42
43^32^21
43^41^2
43^321^3
43^31^5
$43^22L(6,2)$
$4^2\overline{L(10,2)}$
431^{11}
4321^9
43^21^8
$432^2L(7,2)$
$4\overline{L(14,2)}$
$5L(13,3)$
4^23^31
4^33^2
4^3321
$4^32^2\overline{L(2,2)}$
$4^23^22L(2,2)$
$4^232^2\overline{L(3,2)}$
$4^23^21^4$
4^2321^5
4^231^7
4^31^6
4^321^4
4^431^3
$4^4L(2,2)$

$M(20,5) =$

$L(20,4)$
$5\overline{M(15,4)}$
$6L(14,2)$
$632^2\overline{L(7,2)}$
63^21^8
6321^9
631^{11}
$64L(10,2)$
$63^22\overline{L(6,2)}$
$63^3L(5,3)$
$643\overline{L(7,3)}$
$64^2L(6,3)$
64^32
64^31^2
$5^24^21^2$
5^24^22
$5^24\overline{L(6,3)}$
$5^3L(5,5)$

Figure 5.23: The **M** lists for the special cases in Y.

$n = k$, then the sublist $\mathbf{L}(n-k, n-k)$ doesn't exist; this is not a problem because the other two sublists ($\mathbf{L}(n, k-2)$ and $\mathbf{L}(n-k+1, n-k+1)$) do exist. To adhere to our old convention, $n = k$ would have to be split off as a separate subcase. Similar exceptions occur for $(n,k) \in \{(4,3),(2,2)\}$, but these may be considered as base cases.

Here's how the list $\mathbf{M}$ is defined:

$$\mathbf{M}(n,k) = \begin{cases} \emptyset & \text{if } k = 0 \\ 1^n & \text{if } k = 1 \\ List(i,j) & \text{if } (i,j) \in Y \\ \mathbf{L}(n,k-1) & \text{if } 2k+1 \le n < 3k-1 \\ \quad \circ k \cdot \overline{\mathbf{L}(n-k,k-1)} & \\ \quad \circ (k+1) \cdot \mathbf{L}(n-k-1,k-2) & \\ \quad \circ (k+1) \cdot (k-1) \cdot \overline{\mathbf{L}(n-2k,k-2)} & \\ \quad \circ k \cdot k \cdot \mathbf{L}(n-2k,k) & \\ \mathbf{L}(n,k-1) & \text{if } n \ge 3k-1 \\ \quad \circ k \cdot \overline{\mathbf{M}(n-k,k-1)} & \\ \quad \circ (k+1) \cdot \mathbf{L}(n-k-1,k-1) & \\ \quad \circ k \cdot k \cdot (k-1) \cdot \overline{\mathbf{L}(n-3k+1,k-1)} & \\ \quad \circ k \cdot k \cdot k \cdot L(n-3k,k) & \text{only if } n > 3k-1 \end{cases}$$

The indirect algorithm that arises from the recursive definitions of the lists $\mathbf{L}$ and $\mathbf{M}$ is CAT as is now argued. Where are the degree one calls? First note that $\mathbf{M}$ has no degree one calls; the degree of each call is 4 or 5 (or 0 in the case of a leaf). In the definition of $\mathbf{L}$ degree one calls occur only if $k > n$, but then a call $\mathbf{L}(n,n)$ occurs which has degree three. Thus any call of degree one is followed by a call of degree greater than one, from which we conclude that the number of nodes in the computation tree is $O(|\mathbf{L}(n,k)|)$.

THEOREM 5.5 (SAVAGE) *Successive partitions on the lists* $\mathbf{L}$ *and* $\mathbf{M}$ *defined above are close. Unless* $(n,k) = (6,4)$*, the first partition on the list is the lexicographically smallest and the last partition on the list is the lexicographically largest.*

The proof of this theorem is a routine and somewhat tedious induction. The beauty and difficulty of the result lies in the recursive definitions of the lists $\mathbf{L}$ and $\mathbf{M}$. However, there is one aspect of the proof that is of vital interest to us if we wish to construct a direct algorithm: The proof explicitly identifies the two positions of the partition that change. This allows us to easily construct a direct algorithm. Let us illustrate by examining one case of the proof and the code in a direct algorithm that results from that case. Consider the case $k \le n < 2k-2$ in the definition of $\mathbf{L}$; i.e.,

$$\mathbf{L}(n,k-2) \circ (k-1) \cdot \overline{\mathbf{L}(n-k+1,n-k+1)} \circ k \cdot \mathbf{L}(n-k,n-k)$$

The list $\mathbf{L}$(n,k-2) starts at 1^n and ends at $(k-2)(k-2)1$ if $n = 2k-3$ and at $(k-2)(n-k+2)$ if $n < 2k-3$. Note that these partitions are adjacent to $first(\overline{\mathbf{L}(n-k+1,n-k+1)}) = (k-2)(n-k+2)$. At the second interface $last(\overline{\mathbf{L}(n-k+1,n-k+1)}) = (k-1)1^{n-k+1}$ and $first(\mathbf{L}(n-k,n-k)) = k1^{n-k}$ are also adjacent. At the first interface the changes occured in positions 1 and 3 if $n = 2k-3$ and in positions 1 and 2 if $n < 2k-3$. At the second interface the changes occured in positions 1 and $n-k+2$.

In implementing a direct procedure we could write procedures `L(n,k,p)` and `rL(n,k,p)` to produce $\mathbf{L}(n,k)$ and $\overline{\mathbf{L}}(n,k)$, respectively. Since the partition is created left-to-right the

parameter p is necessary to keep track of the indexing offset as the recursion proceeds. The resulting code is shown as Algorithm 5.21. Completing the implementation is a straightforward, if somewhat long, exercise.

```
if k ≤ n and n < 2k − 2 then
    L( n, k − 2, p );
    if n = 2k − 3 then UpDown( p + 1, p + 3 ) else UpDown( p + 1, p + 2 );
    rL( n − k + 1, n − k + 1, p + 1 );
    if n > k then
        UpDown( p + 1, p + n − k + 2 );
        L( n − k, n − k, p + 1 );
```

Algorithm 5.21: Case $k \leq n < 2k-2$ of the recursion for **L**.

5.9 Set Partitions

Let $\mathbf{S}(n)$ be the set of all set partitions of $\{1, 2, \ldots, n\}$ and let $\mathbf{S}(n, k)$ denote those partitions into k blocks. Recall that $B_n = |\mathbf{S}(n)|$ is a Bell number. We say that two partitions are *adjacent* if they differ in only two blocks and those blocks differ by a single element. For example $\{1, 2, 5\}, \{3, 4, 6\}$ and $\{1, 2, 4, 5\}, \{3, 6\}$ are adjacent since the two blocks differ by the single element 4. This definition of closeness allows for the creation (or deletion) of a block of size one; e.g., $\{1, 2, 5\}, \{3, 4, 6\}$ and $\{1, 2, 5\}, \{3, 6\}, \{4\}$ are adjacent.

As in the previous chapter we represent set partitions using RG (restricted growth) sequences. Consider two RG sequences that differ in only one position i, say with the two different values x and y. Then the corresponding partitions are adjacent; the only difference is that in one partition i is in block x and in the other i is in block y. However, this is not the only way that partitions can be adjacent; consider, for example, what happens when the smallest element of one block moves to another block.

If **a** is an RG sequence of length n, define

$$m(\mathbf{a}) \;=\; \max\{a(1), a(2), \ldots, a(n)\}.$$

Clearly, if **a** and **b** are adjacent, then $|m(\mathbf{a}) - m(\mathbf{b})| \leq 1$. Now let

$$l_1, l_2, \ldots, l_{B_{n-1}}$$

be a list of RG sequences of length $n-1$, whose corresponding partitions are adjacent. Expand l_i into

$$L_i \;=\; l_i 0, l_i 1, \ldots, l_i m, l_i(m+1) \quad \text{if } i \text{ is odd, and}$$

$$L_i \;=\; l_i(m+1), l_i m, \ldots, l_i 1, l_i 0 \quad \text{if } i \text{ is even,}$$

where $m = m(l_i)$. This recursive construction is illustrated for $n = 1, 2, 3, 4$ in Figure 5.24(a). We now claim that $L_1, L_2, \ldots, L_{B_{n-1}}$ is a list of all RG sequences of length n and that successive sequences represent adjacent partitions. The basic structure of the construction is the same as that used in Algorithm 4.22 and thus the list contains all RG sequences. The underlying computation tree is the same as that shown in Figure 4.17, except that certain subtrees have been rotated.

0	00	000	0000		0	00	000	0000	
			0001					0001	
		001	0012				001	0011	
			0011					0012	
			0010					0010	
	01	012	0120			01	011	0110	
			0121					0111	
			0122					0112	
			0123				012	0122	
		011	0112					0123	
			0111					0121	
			0110					0120	
		010	0100				010	0100	
			0101					0101	
			0102					0100	
		(a)					(b)		

Figure 5.24: Gray codes for RG functions $n = 1, 2, 3, 4$: (a) Knuth list, (b) differing in only one position.

Within the L_i lists successive partitions are adjacent since they differ only by element n moving to another block. If i is even then $last(L_i) = l_i0$ and $first(L_{i+1}) = l_{i+1}0$ are adjacent since l_i and l_{i+1} are. Similarly, if $m = m(l_i) = m(l_{i+1})$, then $last(L_i) = l_i(m+1)$ and $first(L_{i+1}) = l_{i+1}(m+1)$ are adjacent. The remaining cases occur when $m = m(l_i) = m(l_{i+1}) \pm 1$. Now suppose that $m = m(l_{i+1}) + 1$; the case $m = m(l_{i+1}) - 1$ is symmetric. So $last(L_i) = l_i(m+1)$ and $first(L_{i+1}) = l_{i+1}m$. In both of these partitions n is in a block by itself and thus they are adjacent.

If the definition of m is changed from $m = m(l_i)$ to $m = \min\{k-2, m(l_i)\}$, then the list produced is of all partitions into at most k blocks.

These lists can be generated by a CAT algorithm. The basic structure is like that of Algorithm 4.22, except that we need to determine whether the for loop (and the `gen(l+1,m+1)` call) generates `a[l]` in increasing or decreasing order. This is easily determined by looking at the old value of `a[l]`. The algorithm is CAT for exactly the same reasons that Algorithm 4.22 was CAT.

RG sequences differing in only one position

The list that we have just described has the somewhat undesirable property that successive RG sequences may differ in more than one position, even though the partitions that they represent are adjacent. A simple modification of the construction results in lists where successive RG strings differ in only one position, and by at most 2 in that position. As before, assume that $l_1, l_2, \ldots, l_{B_{n-1}}$ is a list of all RG strings having the required properties. We only need to modify our previous expansion as follows. Let $m' = m(l_{i+1})$. If i is odd, then

$$L_i = l_i0, l_i1, \ldots, l_im, l_i(m+1) \quad \text{if } m' \geq m, \text{ and}$$

$$L_i = l_i0, l_i1, \ldots, l_i(m-1), l_im, l_i(m+1) \quad \text{if } m' = m-1.$$

If i is even then

$$L_i \;=\; l_i(m+1), l_i m, \ldots, l_i 1, l_i 0 \quad \text{if } m' \geq m, \text{ and}$$

$$L_i \;=\; l_i(m+1), l_i m, l_i(m-1), \ldots, l_i 1, l_i 0 \quad \text{if } m' = m-1.$$

The construction for $n = 1, 2, 3, 4$ is illustrated in Figure 5.24(b).

Currently, there is no CAT implementation for generating these lists. In general it is impossible to generate RG strings so that successive strings differ by ± 1 in one position (see Exercise 50).

Partitions with a fixed number of blocks

Generating partitions with a given number of blocks is somewhat more complicated. We define two lists of RG sequences representing partitions of $[n]$ into k blocks, denoted $\mathbf{S}(n,k,0)$ and $\mathbf{S}(n,k,1)$. These definitions have similarities to the [2-Trans] lists defined in an earlier subsection on Gray codes for combinations. The lists possess the following properties.

$$first(\mathbf{S}(n,k,0)) = first(\mathbf{S}(n,k,1)) = 0^{n-k}012\cdots(k-1)$$

$$last(\mathbf{S}(n,k,0)) = 0^{n-k}12\cdots(k-1)0$$

$$last(\mathbf{S}(n,k,1)) = 012\cdots(k-1)0^{n-k}$$

The construction of the lists depends upon the parity of k as shown in the tables below, first for k even and then for k odd.

$\mathbf{S}(n,k,0)$ even k	$\mathbf{S}(n,k,1)$ even k
$\mathbf{S}(n-1,k-1,0)\cdot(k-1)\circ$	$\overline{\mathbf{S}(n-1,k-1,1)}\cdot(k-1)\circ$
$\mathbf{S}(n-1,k,1)\cdot(k-1)\circ$	$\overline{\mathbf{S}(n-1,k,1)}\cdot(k-1)\circ$
$\overline{\mathbf{S}(n-1,k,1)}\cdot(k-2)\circ$	$\mathbf{S}(n-1,k,1)\cdot(k-2)\circ$
$\vdots$	$\vdots$
$\mathbf{S}(n-1,k,1)\cdot 1\circ$	$\overline{\mathbf{S}(n-1,k,1)}\cdot 1\circ$
$\overline{\mathbf{S}(n-1,k,1)}\cdot 0$	$\mathbf{S}(n-1,k,1)\cdot 0$

$\mathbf{S}(n,k,0)$ odd k	$\mathbf{S}(n,k,1)$ odd k
$\overline{\mathbf{S}(n-1,k-1,1)}\cdot(k-1)\circ$	$\mathbf{S}(n-1,k-1,0)\cdot(k-1)\circ$
$\overline{\mathbf{S}(n-1,k,1)}\cdot(k-1)\circ$	$\mathbf{S}(n-1,k,1)\cdot(k-1)\circ$
$\mathbf{S}(n-1,k,1)\cdot(k-2)\circ$	$\overline{\mathbf{S}(n-1,k,1)}\cdot(k-2)\circ$
$\vdots$	$\vdots$
$\overline{\mathbf{S}(n-1,k,1)}\cdot 1\circ$	$\mathbf{S}(n-1,k,1)\cdot 1\circ$
$\mathbf{S}(n-1,k,1)\cdot 0$	$\overline{\mathbf{S}(n-1,k,1)}\cdot 0$

The lists $\mathbf{S}(5,3,0)$ and $\mathbf{S}(5,3,1)$ are illustrated in Figure 5.25.

A simple inductive argument proves the following lemma.

0 0 0 1 2	0 0 0 1 2
0 1 0 1 2	0 0 1 1 2
0 1 1 1 2	0 1 1 1 2
0 0 1 1 2	0 1 0 1 2
0 0 1 0 2	0 1 0 0 2
0 1 1 0 2	0 1 1 0 2
0 1 0 0 2	0 0 1 0 2
0 1 2 0 2	0 0 1 2 2
0 1 2 1 2	0 1 1 2 2
0 1 2 2 2	0 1 0 2 2
0 1 0 2 2	0 1 2 2 2
0 1 1 2 2	0 1 2 1 2
0 0 1 2 2	0 1 2 0 2
0 0 1 2 1	0 1 2 0 1
0 1 1 2 1	0 1 2 1 1
0 1 0 2 1	0 1 2 2 1
0 1 2 2 1	0 1 0 2 1
0 1 2 1 1	0 1 1 2 1
0 1 2 0 1	0 0 1 2 1
0 1 2 0 0	0 0 1 2 0
0 1 2 1 0	0 1 1 2 0
0 1 2 2 0	0 1 0 2 0
0 1 0 2 0	0 1 2 2 0
0 1 1 2 0	0 1 2 1 0
0 0 1 2 0	0 1 2 0 0

Figure 5.25: The lists $\mathbf{S}(5,3,0)$ and $\mathbf{S}(5,3,1)$.

LEMMA 5.7 *The lists* $\mathbf{S}(n,k,0)$ *and* $\mathbf{S}(n,k,1)$ *of RG sequences have the property that successive sequences differ in only one position.*

Note that the partition corresponding to $first(\mathbf{S}(n,k))$ is

$\{1,2,\ldots,n-k\},\{n-k+1\},\ldots,\{n\}$

and to $last(\mathbf{S}(n,k,0))$ and $last(\mathbf{S}(n,k,0))$ are

$\{1,2,\ldots,n-k-1,n\},\{n-k\},\ldots,\{n-1\}$ and

$\{1,k+1,k+2,\ldots,n\},\{2\},\ldots,\{k\}.$

Thus first and last are not adjacent. It is an open question as to whether they can be made adjacent (and still preserve the property of Lemma 5.7).

5.10 Linear Extensions of Posets

In this section we present a CAT algorithm of Pruesse and Ruskey [321] for generating all linear extensions of a poset so that successive extensions differ by one or two adjacent transpositions; i.e., so as to satisfy the [2A Tran] condition. Before getting into the nitty-gritty of the algorithm, we first provide some motivational material, showing that many interesting types of combinatorial objects may be viewed as linear extensions of particular posets.

5.10.1 Combinatorially Interesting Posets

Binary Trees

Consider posets B_n whose Hasse diagrams look like 2 by n grids tilted 45 degrees. An example of B_5 is shown in Figure 5.26. Regard the n elements of the lower chain as being labelled with a left parenthesis and of the upper chain as being labelled with a right parenthesis. Then each linear extension is a well-formed parenthesis string. The structure of the poset guarantees that there will be at least as many left parentheses in any prefix as right parentheses, which is the essential condition for a string of parentheses to be well-formed. Conversely, it is clear that any well-formed parentheses string corresponds to a linear extension of B_n.

Young Tableau

In a more general vein, imagine a Ferrer's diagram rotated to face "up" as illustrated in Figure 5.26. Consider each element to be labelled in the order $1,2,\ldots,n$ that they are listed in a linear extension. Such a labelling gives a Young Tableau of the shape of the Ferrer's diagram. Conversely, any Young Tableau gives a linear extension of the poset. Thus an algorithm for listing linear extensions can be used to generate Young Tableau of a given shape.

Permutations of a Multiset

Let $\mathcal{P}$ be a poset consisting of $t+1$ disjoint chains of sizes $n_0,n_1,\ldots,n_t$. Then each extension of $\mathcal{P}$ may be viewed as a permutation of a multiset of specification $\langle n_0,n_1,\ldots,n_t\rangle$. In particular, if $t=1$ then the extensions correspond to combinations.

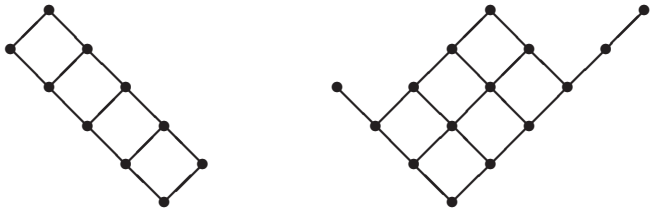


Figure 5.26: (a) Poset whose linear extensions correspond to binary trees. (b) Poset whose linear extensions correspond to Young tableau of shape 6,4,4,1.

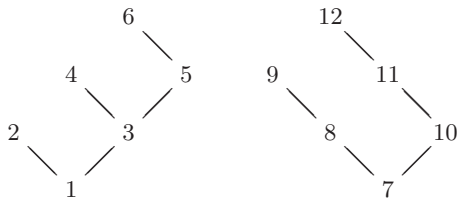


Figure 5.27: Poset whose linear extensions correspond to set partitions of type (2,2,2,3,3).

Alternating Permutations

The fence poset...

Partitions of Given Type

Let $P(n_0, n_1, \ldots, n_t)$ be the set of partitions of the set $\{1, 2, \ldots, n\}$ into blocks of sizes $n_0, n_1, \ldots, n_t$, where $n = n_0 + n_1 + \cdots + n_t$. For example, the partition

$$\{\{2, 8\}, \{3, 5\}, \{6, 12\}, \{1, 10, 11\}, \{4, 7, 9\}\} \tag{5.9}$$

is in $P(2, 2, 2, 3, 3)$. We have listed the elements in order and the blocks of equal size by the magnitude of their smallest element. The problem of generating the elements of P can be reduced to that of generating the linear extensions of a certain forest poset. The cover relations of the poset consist of chains of sizes n_i together with chains of the maximal elements of the previous chains which have equal values of n_i. For our example above the Hasse diagram is shown in Figure 5.27. The linear extension corresponding to our example is $7, 1, 3, 10, 4, 5, 11, 2, 12, 8, 9, 6$. The inverse of this permutation is $2, 8, 3, 5, 6, 12, 1, 10, 11, 4, 7, 9$, which is just our original partition (5.9) but without the curly braces.

To be a little more precise, assume that the n_i are listed such that $n_0 \le n_1 \le \cdots \le n_t$, and let $s_i = \sum_{j=1}^{i-1} n_i$. The cover relations of the poset are indicated by the $t + 1$ chains

$$C_i = s_i + 1, s_i + 2, \ldots, s_i + n_i \quad \text{and}$$

$$D_i = s_i + 1, s_{i+1} + 1, \ldots, s_j + 1,$$

where $n_{i-1} < n_i = n_{i+1} = \cdots = n_j < n_{j+1}$.

There is a formula for the number of extensions of a poset whose Hasse diagram is a forest; see Exercise 37.

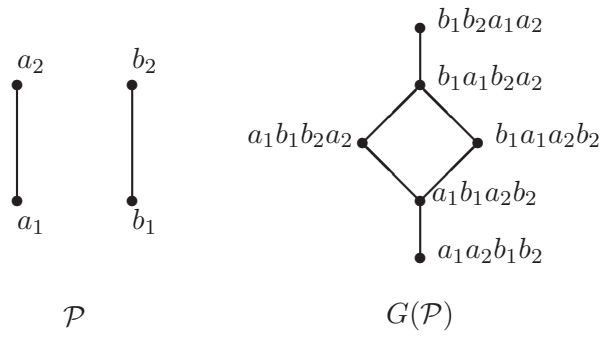


Figure 5.28: A poset and its transposition graph.

5.10.2 Generating Linear Extensions Fast

The Prism of $G'(\mathcal{P})$ is Hamiltonian

For $T \subseteq S(\mathcal{P})$, we let $\mathcal{P}\backslash T$ denote the poset on the set $S(\mathcal{P})\backslash T$ with the relations set $R(\mathcal{P}) \cap (S(\mathcal{P})\backslash T)^2$. Suppose a and b are incomparable elements of $S(\mathcal{P})$ such that a has the same relationship to all other elements of $S(\mathcal{P})$ as b; more precisely, suppose that, for all $c \in S(\mathcal{P})$, $c \prec a$ if and only if $c \prec b$, and $a \prec c$ if and only if $b \prec c$. Then a and b are called *siblings*. For posets $\mathcal{P}$ and $\mathcal{Q}$, if $R(\mathcal{P}) \cup R(\mathcal{Q})$ is antisymmetric, then we let $\mathcal{P} + \mathcal{Q}$ denote the poset on the set $S(\mathcal{P}) \cup S(\mathcal{Q})$ with the relation set which is the transitive closure of $R(\mathcal{P}) \cup R(\mathcal{Q})$. For example, $\mathcal{P} + abc$ is the poset on the set $S(\mathcal{P}) \cup \{a, b, c\}$ with the relation set which is the transitive closure of $R(\mathcal{P}) \cup \{(a, b), (b, c)\}$. If $\mathcal{P} + \mathcal{Q} = \mathcal{P}$, then we say $\mathcal{P}$ *induces* $\mathcal{Q}$. For example, if $\mathcal{P} + abc = \mathcal{P}$, then $\{(a, b), (b, c)\} \subseteq R(\mathcal{P})$, and every linear extension of $\mathcal{P}$ has $a \prec b \prec c$; therefore, we say $\mathcal{P}$ induces abc. For element disjoint total orders $\alpha, \beta, \gamma, \delta$, we let $\alpha(\beta + \gamma)\delta$ denote the poset $\alpha\beta\delta + \alpha\gamma\delta$.

Consider the graph which has $E(\mathcal{P})$ as its vertex set, such that two vertices are adjacent in the graph whenever the corresponding linear extensions differ by a single transposition. This graph is called the *transposition graph* of the poset $\mathcal{P}$ and is denoted $G(\mathcal{P})$. The subgraph of $G(\mathcal{P})$ on the same vertex set but containing only the edges which correspond to adjacent transpositions is called the *adjacent transposition graph* and is denoted $G'(\mathcal{P})$. Generating the linear extensions of $\mathcal{P}$ by (adjacent) transpositions is equivalent to finding a Hamiltonian path in the graph $G(\mathcal{P})$ ($G'(\mathcal{P})$). Figure 5.28 shows a poset and its transposition graph. If α and β are linear extensions of $\mathcal{P}$, then by $D(\alpha, \beta)$ we denote the distance in $G(\mathcal{P})$ from α to β, and by $D'(\alpha, \beta)$ we denote the corresponding distance in $G'(\mathcal{P})$.

Transposition graphs are bipartite and connected. If the partite sets of $G(\mathcal{P})$ are not the same size, then there is no Hamiltonian cycle through the graph; if the difference in the size of the partite sets is more than one, there is no Hamiltonian path through the graph and thus, the linear extensions of $\mathcal{P}$ cannot be generated by transpositions. In Figure 5.28, the partite sets have a size difference of two, so the linear extensions of that poset cannot be generated by transpositions.

Recall the the prism of a graph G is the graph $G \times K_2$ obtained by taking two copies of G, and adding the edges which correspond to an isomorphism between the two copies. To differentiate between the copies of G, we will prefix the vertices of one copy of G with "+" and the other with "−". For example, Figure 5.29 shows $G(\mathcal{P}) \times K_2$, where $\mathcal{P}$ is the poset

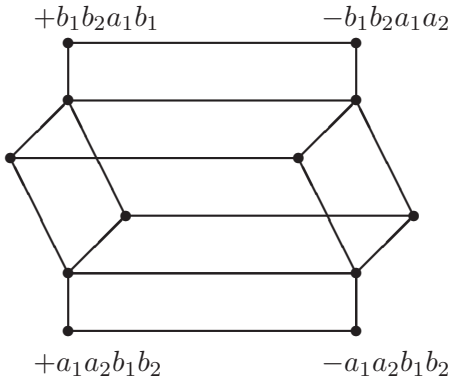


Figure 5.29: The graph $G(\mathcal{P}) \times K_2$.

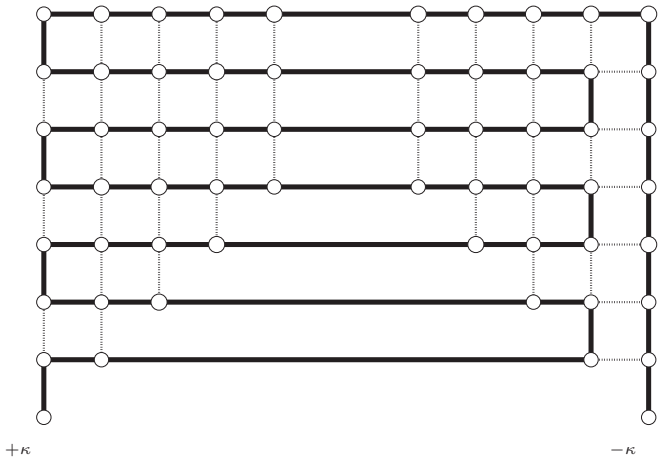


Figure 5.30: Part of a Hamiltonian cycle through $G'(\mathcal{P}) \times K_2$.

shown in Figure 5.28.

LEMMA 5.8 *If a and b are siblings in $\mathcal{P}$, then $G(\mathcal{P}) \cong G(\mathcal{P} + ab) \times K_2$.*

PROOF: Observe that $E(\mathcal{P}) = E(\mathcal{P} + ab) \cup E(\mathcal{P} + ba)$. For any linear extension l of $\mathcal{P}$, transposing a and b in l yields another linear extension of $\mathcal{P}$. Therefore, the operation which transposes a and b in a linear extension provides an isomorphism between $G(\mathcal{P} + ab)$ and $G(\mathcal{P} + ba)$. □

If $e(\mathcal{P}) = 1$ (i.e., if $\mathcal{P}$ is a total order), $G(\mathcal{P}) \times K_2$ is an edge. For the purpose of inductively showing the existence of Hamiltonian cycles, we consider this graph to have a Hamiltonian cycle, since it has a Hamiltonian path such that the endpoints are adjacent.

The Graph $G'(\mathcal{P}) \times K_2$ is Hamiltonian

The proof that $G'(\mathcal{P}) \times K_2$ is Hamiltonian forms the basis of the efficient algorithm to be presented in the next section. That this is true for a certain kind of poset, called a B-poset, was shown by Pruesse and Ruskey [319], and this result will be used in the proof of the general case.

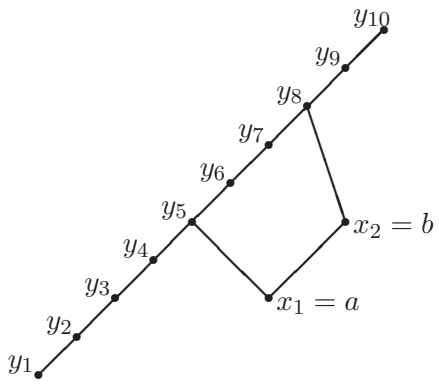


Figure 5.31: A B-poset.

DEFINITION 5.4 *A* B-poset *is a poset $\mathcal{P}$ whose elements can be partitioned into two disjoint chains, $x_1 \prec x_2 \prec \cdots \prec x_n$ and $y_1 \prec y_2 \prec \cdots \prec y_m$, such that $y_j \not\preceq x_i$ for all i and j, where $1 \leq i \leq n$ and $1 \leq j \leq m$.*

An example of a B-poset is shown in Figure 5.31. Note that $\kappa = x_1 x_2 \ldots x_n y_1 y_2 \ldots y_m$ is a linear extension of any B-poset. The extension κ is called the *canonical linear extension* of a B-poset. Define $mr(x_i)$ to be the largest index j such that $x_i \parallel y_j$; if $x_i \prec y_1$, then $mr(x_i) = 0$. For the B-poset of Figure 5.31, $mr(a) = 4$ and $mr(b) = 7$.

The following lemma was proved in [319].

LEMMA 5.9 *Let $\mathcal{P}$ be a B-poset. Then there exists a Hamiltonian cycle in $G'(\mathcal{P}) \times K_2$ which uses the edge $[+\kappa, -\kappa]$.*

Figure 5.29 shows the graph $G'(\mathcal{P}) \times K_2$, where $\mathcal{P}$ is the B-poset shown in Figure 5.31. The edges corresponding to the isomorphism between the two copies of $G'(\mathcal{P})$ have been omitted for the sake of clarity. One can think of traveling up a vertical edge as transposing $b = x_2$ with its neighbor on the right, and traveling along a horizontal edge as transposing $a = x_1$ with one of its neighbors. A Hamiltonian path between $+\kappa$ and $-\kappa$ is shown in Figure 5.30. All B-posets used in the remainder of the paper have $n = 2$; we call these *2B-posets*.

A graph similar to that shown in Figure 5.30 arises whenever $a \parallel y_1$; we call this the *typical case*. If $a \prec y_1$ then $G'(\mathcal{P})$ is a path; we call this the *atypical case*. In other words, the typical case occurs when $mr(a) > 0$ and the atypical case occurs when $mr(a) = 0$.

THEOREM 5.6 *For every poset $\mathcal{P}$, the graph $G'(\mathcal{P}) \times K_2$ is Hamiltonian.*

Proof: The proof of the theorem is by induction on $|S(\mathcal{P})|$. For the base cases of the induction, $\mathcal{P}$ is the poset on zero or one elements; in both of these cases $G'(\mathcal{P}) \times K_2$ is an edge.

Suppose $|S(\mathcal{P})| > 1$. If $\mathcal{P}$ has a unique minimum element a, then $G'(\mathcal{P}) \cong G'(\mathcal{P} \backslash \{a\})$, and by the inductive hypothesis $G'(\mathcal{P} \backslash \{a\}) \times K_2$ is Hamiltonian.

Otherwise, $\mathcal{P}$ has at least two minimal elements, let a and b be two of them. By the inductive hypothesis, the graph $G'(\mathcal{P} \backslash \{a, b\}) \times K_2$ has a Hamiltonian cycle H'. Replace

each signed linear extension $+\alpha_i$ on H' with $ab\alpha_i$; replace each linear extension $-\alpha_i$ with $ba\alpha_i$. The result is a cycle $\beta_1, \beta_2, \ldots, \beta_M$, where $M = 2 \cdot e(\mathcal{P}\backslash\{a,b\})$, in $G'(\mathcal{P}) \times K_2$, which visits exactly those linear extensions in which a and b precede all other elements of $S(\mathcal{P})$. That is, this cycle visits all the linear extensions of

$$\mathcal{P}' = \mathcal{P} + \sum_{x \in S(\mathcal{P})\backslash\{a,b\}} ax + bx.$$

The poset $\mathcal{P}'$ is $\mathcal{P}$ extended so that a and b are covered by every other element of $Min(\mathcal{P})$.

For each $\beta_i = x_i y_i \zeta_i$, where $x_i = a, y_i = b$ or $x_i = b, y_i = a$, the poset $\mathcal{P} + x_i y_i + \zeta_i$ is a B-poset. By Lemma 5.9, there is a Hamiltonian path through $G'(\mathcal{P} + x_i y_i + \zeta_i) \times K_2$ from $+\beta_i$ to $-\beta_i$. We substitute the occurrence of β_i with this path in H', for each odd i. For each even i, we substitute the reverse of this path for the occurrence of β_i. Call the resulting walk H.

To prove that H is a Hamiltonian cycle through $G'(\mathcal{P}) \times K_2$, it is necessary to show that every vertex on the cycle H is a linear extension of $\mathcal{P}$; this is true, since $E(\mathcal{P}') \subseteq E(\mathcal{P})$, and hence each B-poset generated is an extension of $\mathcal{P}$. It is also necessary to show that for each linear extension l of $\mathcal{P}$, $+l$ and $-l$ both occur exactly once on H. Suppose l induces the order xy on $\{a,b\}$ and the order ζ on $S(\mathcal{P})\backslash\{a,b\}$. Then $xy\zeta$ is a linear extension of $\mathcal{P}'$, and l is a linear extension of the B-poset $\mathcal{P} + xy + \zeta$; also, every other B-poset generated either does not induce the order xy or does not induce the order ζ. Therefore, $+l$ and $-l$ are generated only during the generation of $E(\mathcal{P} + xy + \zeta)$; i.e., each $+l$ and $-l$ are generated exactly once. □

Observe that the reference to Lemma 5.9 in the preceding proof was not strictly necessary because the B-posets that occur are all 2B-posets. In the "typical" case, the cycle of Figure 5.30 could be used; in the "atypical" case the cycle is obvious (move b to the right as far as possible, change signs, and then move the b back to the left). If $mr(b)$ is even, the cycle of Figure 5.30 is slightly different and uses the edge $[+a\gamma b, -a\gamma b]$, where $\gamma = y_1 y_2 \cdots y_m$. These cycles are used in the algorithm of the following section.

The Algorithm

The proof of Theorem 5.6 is constructive. In this section, we present the recursive algorithm implicit in the inductive proof. The algorithm runs in *constant amortized time*; i.e., generating all the linear extensions of a poset $\mathcal{P}$ takes time $O(e(\mathcal{P}))$.

We first give an overview of the algorithm and use a small example to give a general understanding of how it works. We then give the details of the algorithm and a proof of its correctness.

The algorithm is an *in-place* algorithm; it maintains an array `le` in which contains the current linear extension, and maintains a variable `IsPlus` which keeps track of the sign ("+" or "−"). We go from one linear extension to the next by making changes to the array or reversing the sign.

The main procedure used by the algorithm, which we call `GenLE`, is recursive and basically follows the path indicated in Figure 5.30. Every level of the recursion has an associated pair of minimal elements of the current subposet. For example, in the poset shown in Figure 5.28, a_1, b_1 are a pair of minimal elements of $\mathcal{P}_1 = \mathcal{P}$, and a_2, b_2 are a pair of minimal elements of $\mathcal{P}_2 = \mathcal{P}_1\backslash\{a_1, b_1\}$. These pairs are determined by some preprocessing which will be described later.

Procedure Call	Linear Extension
	$+a_1b_1a_2b_2$
GenLE(2)	
GenLE(1)	
Move($b_1, \rightarrow$)	$+a_1a_2b_1b_2$
Switch(0)	$-a_1a_2b_1b_2$
Move($b_1, \leftarrow$)	$-a_1b_1a_2b_2$
Switch(1)	$-b_1a_1a_2b_2$
GenLE(1)	
Switch(0)	$+b_1a_1a_2b_2$
Switch(2)	$+b_1a_1b_2a_2$
GenLE(2)	
GenLE(1)	
Move($a_1, \rightarrow$)	$+b_1b_2a_1a_2$
Switch(0)	$-b_1b_2a_1a_2$
Move($a_1, \leftarrow$)	$-b_1a_1b_2a_2$
Switch(1)	$-a_1b_1b_2a_2$
GenLE(1)	
Switch(0)	$+a_1b_1b_2a_2$

Figure 5.32: The trace of the calling sequence for the poset of Figure 1.

The procedures Move and Switch are used to change the current linear extension. They operate in $O(1)$ time as follows:

Switch(i)**:** If $i = 0$, then the sign is changed; that is, `IsPlus` is changed. If $i > 0$, then a_i and b_i are transposed.

`Move(x,left):` This call transposes x with the element on its left.

`Move(x,right):` This call transposes x with the element on its right.

Each time a new linear extension l of $\mathcal{P}_i$ is generated by the call GenLE(i) (i.e., each time Move or Switch is called), GenLE$(i-1)$ is called; the call GenLE$(i-1)$ moves $a_1, b_1, \ldots, a_{i-1}, b_{i-1}$ in all possible ways through l, while maintaining the order $a_{i-1} \prec b_{i-1}$ (or $b_{i-1} \prec a_{i-1}$, depending on their order at the point of calling GenLE$(i-1)$). If $i = 1$, then GenLE$(i-1)$ does nothing.

For example, starting with $+a_1b_1a_2b_2$ and executing the calling sequence GenLE(2), Switch(2), GenLE(2) on the poset shown in Figure 5.28 leads to the trace of calls shown in Figure 5.10.2.

We now follow with the details of our implementation. The reader should refer to the Pascal procedure GenLE of Figure 5.23.

The implementation of the algorithm maintains four global arrays: array `le` is the linear extension; array `li` is its inverse. Arrays `a` and `b` store the elements a_i and b_i. In our discussion of the algorithm, a_i and b_i are considered to be fixed at the outset and unchanging

throughout the run of the algorithm. The arrays will be maintained so that `a[i]` always contains the value of the leftmost of the ith pair, and `b[i]` contains the rightmost. Also, the current sign, either plus (+) or minus (−), is maintained.

The boolean function `Right(x)` is used to determine whether the element `x` can move to the right. It operates in $O(1)$ time as follows:

`Right(b[i])`: Returns true only if `b[i]` is incomparable with the element to its right in the array `le`.

`Right(a[i])`: Returns true only if `a[i]` is incomparable with the element to its right in the array `le` and the element to the right is not `b[i]`.

```
procedure PreProcess
begin
    i := j := 0;
    Q := P;
    while S(Q) ≠ ∅ do
        if Q has exactly one minimal element x then
            j := j + 1; le[j] := x;
            Q := Q\{x};
        else
            Let a', b' be any two minimal elements of Q;
            i := i + 1; j := j + 2;
            a[i] := a'; b[i] := b';
            le[j − 1] := a'; le[j] := b';
            Q := Q\{a', b'};
    MaxPair := i;
end {of PreProcess};
```

Algorithm 5.22: Preprocessing a poset by stripping off minimal pairs.

We now describe our preprocessing. We successively strip off pairs a_i, b_i of minimal elements for $i = 1, 2, \ldots$ until there are no elements left. If a unique minimum element is encountered then it is simply deleted and does not become part of a pair. Let `MaxPair` be the index of last pair of minimal elements we strip from $\mathcal{P}$, the remainder of $\mathcal{P}$ being a total ordering, or empty. This preprocessing is detailed in Algorithm 5.10.2. Note that `MaxPair` is not uniquely determined by the poset, but that it depends on the order in which the elements are stripped from $\mathcal{P}$. The most straightforward way to implement this algorithm is using a breadth-first-search. The running time is proportional to the size of the poset description, $O(n + m)$ where m is the number of cover relations.

We say the linear extension l is in *proper order up to* i if for all $1 \leq j \leq i$, the elements a_j and b_j are adjacent in l, and l induces the orders $a_1 a_2 \ldots a_i a_h$ and $a_1 a_2 \ldots a_i b_h$ for all h, where $i < h \leq$ `MaxPair`. The initial linear extension of the listing must be properly ordered up to `MaxPair`; the preprocessing of Algorithm 5.10.2 does this.

Assuming that Right(b[MaxPair+1]) is false, the initial call is simply GenLE(MaxPair + 1); this is the same as the following procedure calls, which we call the *calling sequence*.

GenLE(MaxPair); Switch(MaxPair); GenLE(MaxPair);

The algorithm consists of executing the preprocessing, setting `IsPlus` to plus (+), and then executing the calling sequence.

A Pascal procedure implementing GenLE is given in Figure 5.23. We now prove the following theorem.

THEOREM 5.7 *The algorithm GenLE generates the linear extensions along a Hamiltonian path in* $G'(\mathcal{P}) \times K_2$.

Proof: In order to prove the theorem, we first prove the following claim.

Claim: Let the linear extension in array `le` be $\xi = \delta a_i b_i \gamma$, and let ξ be properly ordered up to i. Then for each linear extension $l \in E(\mathcal{P} + a_i b_i + \gamma)$, GenLE($i$) generates each of $+l$, $-l$ exactly once. Furthermore, if $i = 1$, then the last extension generated is $-\xi$, and if $i > 1$ then the last extension generated is $+\xi'$, where ξ' differs from ξ by a transposition of a_{i-1} and b_{i-1}.

Proof of Claim: The proof proceeds by induction on i. If $i = 1$ the recursive calls GenLE(0) do nothing, Switch(0) just changes the sign, and δa_1 is induced by $\mathcal{P}$.

It is easy to confirm that the algorithm in Figure 5.23, when stripped of its recursive calls, and in which Switch just changes the sign, simply follows the path indicated in Figure 5.29. In this case, GenLE(1) just finds a Hamiltonian path from $+\xi$ to $-\xi$ through $G'(\mathcal{Q}) \times K_2$, where $\mathcal{Q}$ is the 2B-poset $\mathcal{P} + a_1 b_1 + \gamma$.

If $i > 1$, then assume without loss of generality that the sign in storage when GenLE is invoked is "+". There are α, β, γ such that $+\xi = +\alpha a_{i-1} b_{i-1} \beta a_i b_i \gamma$. Because of the way the preprocessing selects the pairs a_j, b_j, we are assured that $\mathcal{P}$ induces the order $\beta(a_i + b_i)$ (of course, β could be empty).

As mentioned before, the basic structure of the algorithm, stripped of recursive calls, follows the Hamiltonian path in a 2B-poset as indicated in Figure 5.29, where Switch now transposes a_{i-1} and b_{i-1}; therefore it generates $+E' = +E(\mathcal{P} + \alpha(a_{i-1} + b_{i-1})\beta(a_i b_i + \gamma))$. Each linear extension in $+E'$ is properly ordered up to $i - 1$.

As each such linear extension $+l = +\alpha a_{i-1} b_{i-1} \beta \zeta$ (or $+l' = +\alpha b_{i-1} a_{i-1} \beta \zeta$), where $\zeta \in E(a_i b_i + \gamma)$, is generated, GenLE($i-1$) is called on $+l$. By the inductive hypothesis, that call generates $\pm E(\mathcal{P} + a_{i-1} b_{i-1} + \beta\zeta)$ (or $\pm E(\mathcal{P} + b_{i-1} a_{i-1} + \beta\zeta)$, respectively), starting at $+l$ and ending at $+l'$ if $i > 2$, and ending at $-l$ if $i = 2$. Since there are an even number of vertices in the product of the graph with a edge, there are an even number of calls to GenLE($i-1$). Thus if $i > 2$ then the sign of the final permutation is unchanged, and if $i = 2$, then the relative ordering of a_{i-1} and b_{i-1} is unchanged. The union over all such ζ is $\pm E(\mathcal{P} + a_{i-1} b_{i-1} + a_i b_i + \gamma) \cup \pm E(\mathcal{P} + b_{i-1} a_{i-1} + a_i b_i + \gamma) = \pm E(\mathcal{P} + a_i b_i + \gamma)$. □

Let a, b be `a[MaxPair], b[MaxPair]` respectively, and suppose the calling sequence is executed on the preprocessed poset $\mathcal{P}$. By the Claim, the first call to GenLE generates $\pm E(\mathcal{P} + ab)$; then a and b are transposed, and then $\pm E(\mathcal{P} + ba)$ is generated. Therefore, $E(\mathcal{P})$ is generated, and the Theorem is proved. □

In analyzing the time complexity of the algorithm, we assume that `Right`, Switch, and Move can be implemented in constant time. This is easily accomplished as long as the inverse, `li`, of `le` is maintained. Each call to Move and Switch generates one more linear extension. Observe that the call GenLE(i) generates at least two calls to GenLE($i-1$). Each

iteration of a while-loop or for-loop in the algorithm executes a Move, thereby generating a linear extension. The only occasion in which GenLE can be recursively called and no linear extension generated is when $i = 0$, and this happens at most once per linear extension generated. Therefore, the algorithm runs in constant time per linear extension, when generating $\pm E(\mathcal{P})$; the algorithm is CAT. By suppressing the linear extensions which are prefixed with "$-$", we generate $E(\mathcal{P})$ in constant amortized time. Another way to think of the preceding argument is to consider the underlying computation tree, where each internal node is a recursive call and each leaf is a linear extension. The total amount of computation can be divided up so that each node is assigned a constant amount of computation. Since each internal node has at least two children, the number of leaves is greater than the number of internal nodes and therefore the total amount of computation is proportional to the number of leaves.

Observe that the generation of the minus ("$-$") vertices only occurs when $i = 1$ in Algorithm GenLE. This suggests that $i = 1$ be treated as a special case and that minus ("$-$") vertices be omitted entirely by simply skipping to the next plus ("+") vertex. If this is done, then it saves some computation but the same list of extensions is produced as before, and successive extensions can differ by a large number of transpositions.

If one only wants to compute the number of extensions, then some computation can be saved by only computing the number of vertices at the $i = 1$ level of the recursion, and not generating the extensions explicitly; i.e., never moving a_1 and b_1. The number of vertices (extensions) can be determined from $mr(a) = \texttt{mra}$ and $mr(b) = \texttt{mrb}$; the number is $(mr(a)+1)[mr(b)+1-mr(a)/2]$. Furthermore, $mr(a)$ and $mr(b)$ change by at most unity from one extension to the next, since only adjacent transpositions are used. This leads to an algorithm whose running time is $O(e(\mathcal{P} \setminus \{a_1, b_1\}))$. In general, we have

$$2 \cdot e(\mathcal{P} \setminus \{a_1, b_1\}) \;\le\; e(\mathcal{P}) \;\le\; n(n-1) \cdot e(\mathcal{P} \setminus \{a_1, b_1\}).$$

The lower bound is attained when $a_1 \prec c$ and $b_1 \prec c$ for all elements c of $\mathcal{P} \setminus \{a_1, b_1\}$. The upper bound is attained when a_1 and b_1 are maximal, as well as minimal.

Gray Codes for Linear Extensions

We now show that linear extensions can be listed so that successive extensions differ by at most a few adjacent transpositions. We first show the existence of such listings, and then how to modify the results of the previous sections to show that the set of linear extensions of a poset can be listed so that successive extensions differ by only one or two adjacent transpositions. Let us say that an ordering $\alpha_1, \alpha_2, \ldots, \alpha_{e(\mathcal{P})}$ of the extensions of $\mathcal{P}$ has *delay* C if $D'(\alpha_i, \alpha_{i+1}) \le C$ for all $0 \le i < e(\mathcal{P})$, where $\alpha_0 = \alpha_{e(\mathcal{P})}$. Thus we are going to show the existence of a delay 2 ordering of $E(\mathcal{P})$. Furthermore, such a listing can be done in constant amortized time. The existence of a delay 3 ordering is not difficult to show.

A poset $\mathcal{P}$ has a delay k ordering if and only if $G'(\mathcal{P})^k$ is Hamiltonian. A result of Sekanina [378], proved in the next chapter, is that the cube of every connected graph is Hamiltonian. Since $G'(\mathcal{P})$ is always connected, $G'(\mathcal{P})^3$ is Hamiltonian and a delay 3 ordering exists.

The graph $G'(\mathcal{P})$ is not always 2-connected; otherwise the existence of a delay 2 ordering would by implied by a result of Fleischner [134] which states that the square of every 2-connected graph is Hamiltonian.

Even though $G'(\mathcal{P})$ is not in general 2-connected, the posets with 2-connected transposition graphs are easy to characterize. First, consider the question of which transposition

graphs have pendant vertices. If $\mathcal{P}$ consists of two disjoint chains, then $G'(\mathcal{P})$ has two pendant vertices; if $\mathcal{P}$ is a B-poset and is not the disjoint union of two chains, then $G'(\mathcal{P})$ has one pendant vertex; otherwise $G'(\mathcal{P})$ has no pendant vertices.

LEMMA 5.10 *For every poset $\mathcal{P}$, the graph $H(\mathcal{P})$ is 2-connected, where $H(\mathcal{P})$ is $G'(\mathcal{P})$ minus any pendant vertices.*

```
procedure GenLE ( i : ℕ);
local mrb, mra, mla, x : ℕ; typical :  boolean ;
begin
    if i > 0 then
        GenLE( i − 1 );
        mrb := 0; typical := false;
        while Right( b[i] )  do
            mrb := mrb + 1;
            Move( b[i], → ); GenLE( i − 1 );
            mra := 0;
            if Right( a[i] ) then
                typical :=  true;
                repeat
                    mra := mra + 1;
                    Move( a[i], → ); GenLE( i-1 );
                until  not Right( a[i] );
            if typical then
                Switch( i − 1 ); GenLE( i − 1 );
                if mrb is odd then mla := mra − 1 else mla := mra + 1;
                for x := 1  to mla  do
                    Move( a[i], ← ); GenLE( i − 1 );
        if typical  and mrb is odd
            then Move( a[i], ← )
            else Switch( i − 1 );
        GenLE( i − 1 );
        for x := 1  to mrb  do
            Move( b[i], ← ); GenLE( i − 1 );
end  {of GenLE};
```

Algorithm 5.23: Pascal procedure GenLE to generate linear extensions.

This may be proven by showing that every pair of incident edges of $H(\mathcal{P})$ is on a 4, 6, or 8-cycle. This lemma does not help us in finding an efficient algorithm for listing a delay 2 ordering of $E(\mathcal{P})$. Instead, we prove it by applying Theorem 5.6 and the following lemma.

LEMMA 5.11 *If G is bipartite and $G \times K_2$ is Hamiltonian, then G^2 is Hamiltonian.*

Proof: Let G be a bipartite graph on n vertices, and let $(v_1, x_1), (v_2, x_2), \cdots, (v_{2n}, x_{2n})$ be a Hamiltonian cycle through $G \times K_2$, where $v_i \in V(G)$ and $x_i \in V(K_2) = \{1, 2\}$, for all $i, 1 \leq i \leq 2n$. Consider the sequence of vertices $S = v_2, v_4, \ldots, v_{2n}$.

Since G is bipartite, so is $G \times K_2$; thus the vertices of S are all the vertices of one partite set of $G \times K_2$. Also, for a vertex u of G, $(u,1)$ and $(u,2)$ are adjacent are are therefore in different partite sets of $G \times K_2$. Therefore each vertex of G appears exactly once in S. For each i, the vertices v_i and v_{i+2} are either of distance one in G (if $x_i \neq x_{i+2}$) or of distance two in G (if $x_i = x_{i+2}$). Therefore S is a Hamiltonian cycle in G^2. □

THEOREM 5.8 *The linear extensions of any poset can be generated with delay 2 in constant amortized time.*

Proof: We run the Algorithm GenLE given in Figure 8, but instead of suppressing the linear extensions with a negative sign, we suppress every other linear extension; i.e., if we generate the list $l_1, l_2, l_3, l_4, l_5, \ldots$, then we output the list $l_1, l_3, l_5, \ldots$. By the proof of Lemma 5.11, this is a delay 2 listing of the linear extensions. It has the same running time as GenLE, i.e., constant amortized time. □

In the remainder of this section we discuss how to use the algorithm to compute $P(x < y)$ and $h(x)$. We use the version of GenLE that generates each extension exactly twice, where each successive extension differs by an adjacent transposition from its predecessor. We first discuss how to compute $P(x < y)$ for every pair x, y.

Let us define an xy-run to be a maximal sequence of successive extensions where x precedes y. We maintain two arrays of integers, call them S and T. The value of $S[x,y]$ is the sum of the lengths of the previous xy-runs. The value of $T[x,y]$ is the iteration at which the current xy-run started. At each iteration, exactly one adjacent pair, say xy, is transposed. If this occurs at the t-th iteration, then $S[x,y]$ is incremented by $t - T[x,y]$ and $T[y,x]$ is set to t. At the termination of the algorithm, $P(x < y)$ is $S[x,y]$ divided by $2e(\mathcal{P})$. Since only a constant amount of update is done at each iteration the total computation is $O(n^2 + e(\mathcal{P}))$.

To compute $h(x)$ we proceed in a similar fashion. An x-run is a maximal sequence of extensions in which x occupies the same position. Here the value of $S[x]$ is the weighted sum of the lengths of the previous x-runs and $T[x]$ is the iteration at which the current x-run started. At each iteration exactly one adjacent pair, say xy, is transposed. If this occurs at the t-th iteration, then for $z = x, y$, $S[z]$ is incremented by $p[z] * (t - T[z])$ and $T[z]$ is set to t, where $p[z]$ is the position that z occupied in the extension. At termination the value of $h(x)$ is $S[x]$ divided by $2e(\mathcal{P})$.

5.11 Ideals of Posets

In this section we show the existence of a Gray code of ideals where each ideal differs from its predecessor by the flip of one or two bits. Our strategy is the same as in the previous section; we show that the prism of the underlying closeness graph is Hamiltonian. This Gray code can be generated in time $O(\Delta \cdot i(\mathcal{P}))$, where Δ is the maximum number of upper covers of any element of the poset $\mathcal{P}$.

THEOREM 5.9 *Let x be minimal in $\mathcal{P}$. The graph $J(\mathcal{P})$ has a Hamilton cycle that contains the edges $[+\emptyset, -\emptyset]$ and $[+\emptyset, +x]$.*

PROOF:Our proof is by induction on $|\mathcal{P}| = 1$. If $|\mathcal{P}| = 1$ then $J(\mathcal{P}) \times e$ is the 4-cycle $+\emptyset, +x, -x, -\emptyset$. Otherwise, assume that $|\mathcal{P}| > 1$, and let x be minimal.

Inductively, there is a Hamilton cycle

$$+\emptyset = X_1, X_2, \ldots, X_p = -\emptyset$$

in $J(\mathcal{P}/x)$.

If x is the minimum, then the Hamilton cycle

$$+\emptyset, X_1 \cup \{x\}, X_2 \cup \{x\}, \ldots, X_p \cup \{x\}, -\emptyset$$

satisfies the conditions of the theorem.

Otherwise, let x and y be minimal elements of $\mathcal{P}$. There are Hamilton cycles

$$+\emptyset = X_1, X_2, \ldots, X_p = -\emptyset \text{ in } J(\mathcal{P}/x) \text{ and}$$

$$+\emptyset = Y_1, Y_2, \ldots, Y_q = -\emptyset \text{ in } J(\mathcal{P} \setminus x),$$

with $X_2 = Y_2 = \{y\}$. The Hamilton cycle

$$\begin{aligned} +\emptyset \quad = \quad & Y_1, X_1 \cup \{x\}, X_p \cup \{x\}, X_{p-1} \cup \{x\}, \ldots, X_2 \cup \{x\}, \\ & Y_2, Y_3, \ldots, Y_{q-1}, Y_q = -\emptyset \end{aligned}$$

satisfies the conditions of the theorem. □

5.12 Loopless Algorithms

An algorithm for generating combinatorial objects is *loopless* if ...

There is often a natural connection between combinatorial Gray codes, CAT algorithms, and loopless algorithms.

These algorithms are interesting, not only because of the seeming simplicity of the computations, but because such algorithms would be preferred for hardware implementation, particularly in those cases where timing is important.

5.12.1 Binary Trees

The algorithm to generate the sequence of trees can also be implemented on a pointer machine in $O(1)$ *worst case* time per rotation.

The algorithm requires each node to store the usual pointers to its parent, left child and right child in the tree, and in addition two more pointers and a *direction* bit. The direction bit for node i indicates whether i is currently rotating left on its way to becoming the root of $T[1..i]$ (i.e. direction(i) = up) or whether i is currently rotating right on its way to becoming the right child of $i-1$ (i.e. direction(i) = down).

We need only show that the next node to be rotated can be located in $O(1)$ time.

At any given time in the generation procedure each node i is currently in the process of rotating either up or down the right path of $T[1..i-1]$. We say i is *unfinished* if at the current time i has not yet completed its rotations along the right path of $T[1..i-1]$. Otherwise we say i is *finished*.

Initially all the nodes are unfinished except for node 1.

At any point in the generation procedure, if T is the current shape of the tree, then the next node that is to be rotated, node j say, is the largest (rightmost) unfinished node. All

nodes greater than j are either proper ancestors of the root of $T[1..j]$ or in the right subtree of j.

After the rotation at node j in T produces a tree T', every node i, $i > j$ commences to rotate along the right path of $T'[1..i-1]$ in the appropriate direction. But before and after each such rotation at i, all trees T'' whose subtree $T''[1..i]$ is identical to the current induced subtree of the nodes from 1 to i are generated.

Consider a rotation at node j at some point during the generation procedure. Then the state of every node i, $i < j$, remains unchanged but the state of every node i, $i > j$, changes from finished to unfinished. The node j, which was unfinished before this rotation, may be either finished or unfinished after this rotation.

Thus our algorithm can easily identify the next rotation to be performed by maintaining two lists, the list of finished nodes and the list of unfinished nodes. These two lists partition the nodes of the tree. On each list the nodes will appear in decreasing order. Each node i has a pointer *next* to the node following i on the list containing i. The *next* pointers are used for both the list of finished and unfinished nodes, since a node can be on only one list. This facilitates the updating of the lists needed after each rotation. We also maintain for each node i a pointer to its inorder successor $i + 1$.

The next node to be rotated is simply the first node on the list of unfinished nodes. These two lists of nodes can easily be updated in constant time after each rotation by making $O(1)$ pointer changes. The details are given in the code for this generation algorithm which appears in Appendix A.

5.13 Finding a Hamiltonian cycle

So far, even though many problems can be modelled as finding Hamiltonian paths in certain graphs, we have been able to avoid explicitly building the graph before trying to find a Hamiltonian path in it. Unfortunately, we cannot expect to always be so lucky; so in this section we describe how to find a Hamilton path if one exists. We will describe two algorithms, one that works well for dense graphs and another that works well for cubic graphs.

The problem of determining whether a graph has a Hamilton cycle is NP-complete, so we don't expect deterministic polynomial time algorithms that work on all graphs. A brute-force backtracking approach can be employed, but the ugly reality of exponential growth limits the usefulness of backtracking — we therefore describe two other approaches. These algorithms work well in practice (on graphs of moderate size).

5.13.1 Extension-Rotation Algorithms

Here we try a probabilistic approach to finding a Hamilton cycle in undirected graphs that are fairly dense. Roughly speaking, these algorithms will almost always find a Hamilton cycle in a graph that has one, as long as $m \geq n \log n$.It is also true that almost all graphs on that many edges have a Hamilton cycle.Thus we would expect the algorithm to succeed on the n-cube, but not necessarily on planar graphs.

The algorithms are based on two simple operations on a path P in the graph, called *rotation* and *extension*. In an extension the path is simply lengthened by adding a new vertex to the end of P. In a rotation the length of P is unchanged; an existing edge is removed and new one is added. Rotations and extensions are illustrated in Figure 5.33.

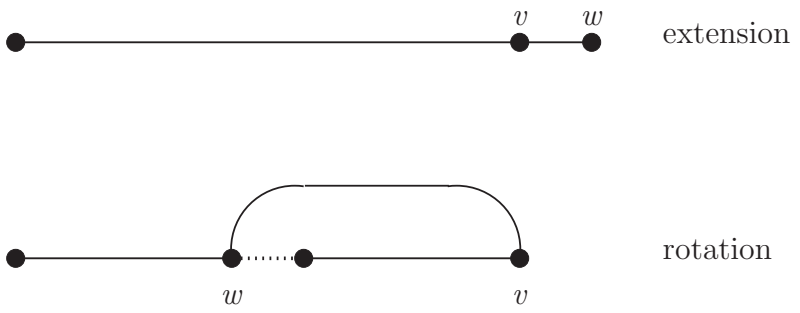


Figure 5.33: The extension and rotation operations.

```
(P1)   P := some vertex of G, call it v_0;
(P2)   repeat
(P3)       v := last(P);
(P4)       if v is adjacent to v_0 and P is a H-cycle then return( P );
(P5)       (v, w) := an edge of E;
(P6)       if w ∉ P then P := P, w;
(P7)       if w ∈ P then P := the path obtained by adding the edge (v, w)
(P8)            and deleting the first edge on the subpath from w to v.
(P9)   until false;
```

Algorithm 5.24: Extension-rotation heuristic algorithm for finding a Hamilton cycle.

See Algorithm 5.24. Some variation of this algorithm can occur, mostly depending on how the edge is chosen at step (P5). The first problem to be overcome is making the algorithm finite, as it is possible that the edge removed on a previous iteration is reinserted at a later iteration, resulting in the same path occuring twice. We will discuss three variations of the algorithm, due to (a) Posa, (b) Angluin and Valiant, and (c) Karp.

In Posa's version, the rotation operation at lines (P7-8) is constrained so that no two paths of the same length have the same final vertex. For example, the removed edge is not again added unless an extension has occured in the interim.

In the Angluin and Valiant version, an edge that is used in P is removed from the underlying graph. Thus each edge is ever considered at most once. Furthermore, they select the edge (v, w) so that w is chosen uniformly at random from $N(v)$, the current neighborhood of v.

In the Karp version of the algorithm a more sophisticated probabilistic selection rule is used. The first step is a randomized preprocessing that creates for each vertex v a set $\mathsf{PN}(v)$ of its neighbors. Let p be the probability that an edge is in E (if this is not known, simply take $p = m/\binom{n}{2}$). Membership in $\mathsf{PN}(v)$ is determined by the following selection rule, applied to each edge $(v, w) \in E(G)$:

$v \in \mathsf{PN}(w)$, $w \in \mathsf{PN}(v)$ with probability $p/4$
$v \in \mathsf{PN}(w)$, $w \notin \mathsf{PN}(v)$ with probability $(1 - p/2)/2$
$v \notin \mathsf{PN}(w)$, $w \in \mathsf{PN}(v)$ with probability $(1 - p/2)/2$
$v \notin \mathsf{PN}(w)$, $w \notin \mathsf{PN}(v)$ with probability $p/4$

For each vertex v, a subset, $\mathsf{OLD}(v)$ of $\mathsf{PN}(v)$, is maintained. Then w at step (P5)

Figure 5.34: Implicated edges on Hamilton cycles in cubic graphs.

is chosen uniformly at random, either from $\mathsf{OLD}(v)$ or $\mathsf{PN}(v) \setminus \mathsf{OLD}(v)$; which of the two sets of used is determined by taking $\mathsf{OLD}(v)$ with probability $|\mathsf{OLD}(v)|/(n-1)$. Initially $\mathsf{OLD}(v) = \emptyset$ for each vertex v. If w is not selected from $\mathsf{OLD}(v)$, then w is added to $\mathsf{OLD}(v)$ at the end of the iteration.

Why do we have three separate algorithms? In each of the succeeding cases the probability of success is greater, at least if certain distributional assumptions about the input graph are made. Precise statements about the success of the algorithms can be made but these are outside of the scope of this book. Appropriate references may found in the Bibliographic Remarks at the end of this chapter.

5.13.2 Cubic graphs

A graph that is regular of degree two has a Hamilton cycle if and only if it is connected. Finding a Hamilton cycle in a cubic (regular of degree three) graph is NP-complete but a simple backtracking approach is remarkably effective on graphs of modest size.

Cubic graphs have the nice property that if two edges of the form $(u, v), (v, w)$ are on a Hamilton cycle then there are two other edges $(x, y), (y, z)$ that are also on the cycle and an edge (v, y) that is not on the cycle. See Figure 5.34. These implicated edges can lead then to further implicated edges and so on. The upshot of this observation is that backtracking algorithms for finding Hamilton cycles in cubic graphs can be made to be remarkably efficient on graphs of size, say 200, a size that is totally impossible on graphs of higher minimum degree.

5.14 Exercises

Questions about the BRGC

1. **[1]** What is the rank of 1^n in the BRGC? Note the solution to this problem gives the number of moves to solve the Chinese Rings puzzle, or "The Brain" puzzle, or the Binary Arts SPIN-OUT puzzle.

2. **[2]** Prove that any two strings in the Binary Reflected Gray Code list whose Hamming distance is at least m have ranks that differ by at least $\lceil 2^m/3 \rceil$.

3. **[2]** Let $\mathbf{b}$ and $\mathbf{c}$ be two bitstrings of length n and assume that $rank(\mathbf{b}) = 2^k + rank(\mathbf{c})$. Prove that the Hamming distance between $\mathbf{b}$ and $\mathbf{c}$ is 1 if $k = 0$, and is 2 if $0 < k < 2^{n-1}$. This result has implications for routing algorithms on the hypercube.

4. [**1+**] In the BRGC, let $rank(g_n g_{n-1} \cdots g_1) = (b_{n-1} \cdots b_1 b_0)_2$. The Hamming distance between $g_n g_{n-1} \cdots g_1$ and $b_{n-1} \cdots b_1 b_0$ is some integer k between 0 and $n-1$. Prove that the number of length n bitstrings at Hamming distance k apart from their corresponding string in the BRGC is $2\binom{n-1}{k}$.

5. [**2**] The transition sequence T_n for the BRGC produces a Gray code, no matter the vertex $a_n \cdots a_2 a_1$ of Q_n from which it starts. Restate Lemma 5.1 so that it applies to such a Gray code, but from the generic vertex $a_n \cdots a_2 a_1$, instead of 0^n.

6. [**2**] The transition sequence of the BRGC could be regarded as an infinite sequence $\mathcal{T} = t_1, t_2, \ldots$, whose first $2^n - 1$ entries give T_n. Find a simple expression for the sum of the first k elements of $\mathcal{T}$. The expression may use the function $\nu(m)$, the number of bits in the binary expansion of m.

7. [**1−**] What is the number of shortest paths from 0^n to 1^n in the n-cube?

8. [**2**] Show that Q_n is Hamilton-laceable by showing the existence of a Hamilton path from 0^n to $b_1 b_2 \cdots b_n$ for any bitstring $b_1 b_2 \cdots b_n$ with $\sum b_i$ odd.

9. [**2+**] The BRGC can be thought of as a permutation of the elements of $\mathbf{B}(n)$, or, alternatively, of the elements of $\{0, 1, \ldots, 2^n - 1\}$. For example, in one line notation for $n = 3$ the permutation is 01326754, or in cycle notation is (0)(1)(2 3)(4 6 5 7). The purpose of this exercise is to determine the cycle lengths of elements in this permutation. Let $G(s)$ be the rank s element of the BRGC. Define $c(s) = \min\{k \mid G^k(s) = s \text{ and } k > 0\}$. Show that $c(s) = 2^{\lceil \lg(1+\lfloor \lg s \rfloor)\rceil}$, for $s > 0$.

10. [**1+**] Prove that any algorithm that solves the Towers of Hanoi problem must make at least $2^n - 1$ moves. [**1**] What happens in the Towers of Hanoi problem if we add the additional requirement that no two disks of the same parity may lie atop each other? [**2**] Devise an efficient divide-and-conquer algorithm that starts from an arbitrary valid configuration of disks. How many moves does your algorithm take?

11. [**2**] Prove that the following iterative algorithm solves the Towers of Hanoi problem and that it can be implemented so that only a constant amount of computation is done between moves. The algorithm moves disks from peg A to peg C.

    1. Order pegs A,B,C if n even and A,C,B if n odd
    2. **repeat**
    3. Move disk 1 according to order given in step 1.
    4. If all disks on destination peg then stop.
    5. Move smallest allowed disk that is greater than 1.
    6. **until** false.

12. [**2**] Prove that the following two lines of Java code solves the towers of Hanoi problem. Each line of output has the form `0-->2`, meaning move the top disk on peg 0 to peg 2.

```
for (int m=1; m < (1 << n); m++)
   System.out.println( (m&m-1)%3 + "-->" + ((m|m-1)+1)%3 );
```

13. **[2+]** Prove that for $1 \le j \le n$ and $1 \le k \le 2^n$, that the jth letter of the kth word of the BRGC is equal to $\binom{2^n-2^{n-j}-1}{2^n-2^{n-j-1}-k/2}$ modulo 2.

14. **[2]** Develop a CAT algorithm for generating $\mathbf{F}(n)$, the set of all bitstrings of length n with no substring 11, in Gray code order. Successive strings should differ by a single bit flip.

Questions about Combinations

15. **[2]** Prove that the number of calls to `gen` and `neg` in Algorithm 5.8 is

$$1 + 3 \sum_{i=1}^{\lceil k/2 \rceil} \binom{n-2i}{n-k-1}$$

16. **[1+]** Regarding Algorithm 5.8 as generating the elements of $\mathbf{A}(n,k)$, on each iteration some a_i changes. Prove that the sum of all such indices i over the entire algorithm is

$$\binom{n+1}{k} - \binom{k+1}{k},$$

and thus that the average position of the element that changes is $(n+1)/(n-k+1) - (k+1)/\binom{n}{k}$.

17. **[1+]** Show that the following recursion defines a list of combinations that that we have already encountered. This recursion is from the first editon of Nijenhuis and Wilf [289].

$$A(n,k) = A(n-1,k)\cdot 0 \circ A(n-2,k-2)\cdot 11 \circ \overline{A(n-2,k-1)}\cdot 01$$

18. **[1+]** We call this the "piano player's problem". Recall the chord playing motivation at the beginning of Section ??? on generating combinations by homogeneous transpositions. The problem, as stated, is not very realistic since it assumes that the player's fingers can stretch arbitrarily far apart. Suppose that we assume that the chord is being played on one hand and that the fingers on that hand can stretch over p keys. Can all such restricted chords be played so as to satisfy the [H-Trans] condition? To be more precise, prove that there is a listing of the strings of the set $\mathbf{P}(n,k,p) = \{a_1 a_2 \cdots a_k \mid 1 \le a_1 < a_2 < \cdots a_k \le n, \text{ and } a_k - a_1 \le p\}$ so that each successive string differs in only one position. What is $|\mathbf{P}(n,k,p)|$?

19. **[1]** The $\mathbf{R}(n,k,p)$ lists can be defined for $p > 2$. Show that they exist, as long as $n - k \ge p$.

Questions about permutations

20. **[1+]** How many transpositions does Algorithm 5.10 use in generating all permutations of $[n]$?

21. [**2**] Develop a loopless version of Algorithm 5.12 for generating all permutations by adjacent transpositions.

22. [**1**] Show that the parity difference of $\mathbf{A}_n$, the set of alternating permutations of length n, is 1 if n is even and is 0 if n is odd.

23. [***R***] For n odd show that $\mathbf{A}_n$ can be generated by transpositions. The case of n even follows from the results of [365] or [319].

24. [**2**] Show that there is a Hamilton path, call it $\mathbf{A}(n)$, in the hypercube Q_n from 0^n to 1^n if and only if n is odd. Show that there is a Hamilton path, call it $\mathbf{Z}(n)$, in Q_n from 0^n to $1^{n-1}0$ if and only if n is even. Explain why these paths can be generated by a CAT algorithm.

25. [**2+**] Using the results of the previous exercise, develop a Gray code for $\mathbf{M}(n) = \{x \in \mathbf{B}(n) : x \leq x^R\}$, the set of lexicographically smallest bitstrings of the equivalence classes of bitstrings equivalent under reversal. Successive words should differ by a single bit. Show two types of paths: $\mathbf{T}(n)$ from 0^n to 1^n and a cyclic path $\mathbf{N}(n)$. Explain why these paths can be generated by a CAT algorithm.

26. The *Butterfly Network* can be be thought of as a graph with vertices ??? and edges ???. Give a simple algorithm for listing the vertices along a Hamiltonian path in the Butterfly Network.

27. [**1+**] Modify Algorithm 5.10 so that it generates all derangements of $[n]$. The algorithm does not have to be BEST, nor do successive derangements have to differ by single transpositions.

28. [**1+**] Modify the Johnson-Trotter algorithm to develop a CAT algorithm for generating all $(n-1)!/2$ rosary permutations of $[n]$.

29. [**1+**] Give an example of a set of multiset permutations that have a cyclic Gray code under transposition, but which do not have the prefix property (permutations with a common prefix appear consecutively).

30. [***R*−**] Is every bipartite graph obtainable as the transposition graph of some set of multiset permutations?

Questions about numerical partitions

31. [**2+**] Give a direct CAT implementation of the recursive definition of **L** and **M** for generating numerical partitions.

Questions about trees

32. [**2**] Develop a ranking algorithm for the Gray Code that generates binary trees by rotations.

33. **[3]** It is remarkable that sometimes a lexicographic listing is also a Gray code listing; this exercise gives one such naturally occuring example (another example is numerical partitions using the multiplicity representation). Let T be the binary tree which consists of a single path of left children. Any binary tree may be constructed by moving up this path toward the root, performing some number of rotations at the successive nodes on this left path. Let c_i be the number of rotations performed at the node on the left path at height $i-1$. (a) Show that the sequence $c_1, c_2, \ldots, c_n$ uniquely identifies a tree and that these sequences are characterized by the property that $\sum_{i=j}^{n} c_i \leq n-j$ for $1 \leq j \leq n$. (b) Let $\mathbf{C}$ be the set of all such sequences and let them be ordered lexicographically. Show that two successive sequences in $\mathbf{C}$ differ in one or two adjacent positions. (c) Develop a loopless algorithm for generating the sequences of $\mathbf{C}$ in lex order.

34. **[*R*−]** Can well-formed parenthesis strings be generated by homogeneous transpositions? What about 2-apart transpositions?

35. **[2−]** Give a small example showing that there does not exist a Gray code, using "diagonal flips", of the triangulations of an arbitrary polygon.

Questions about Linear Extensions

36. **[1]** Prove that if $\mathcal{P}$ is a poset with a pair of siblings, then $G(\mathcal{P})$ is Hamiltonian. **[[*R*−]]** Under the same assumption, prove the stronger result that $G'(\mathcal{P})$ is Hamiltonian.

37. **[2]** Let F be a forest poset on elements $v_1, v_2, \ldots, v_n$, and let $d(v_i)$ be the number of descendants of v_i in the forest (counting an element as a descendant of itself). Prove that the number of extensions is $e(F) = n!/\prod_{i=1}^{n} d(v_i)$.

Miscellaneous questions

38. **[*R*+]** Consider the graph G_k whose vertex set consists of all bitstrings of length $2k+1$ in which each bitstring has k or $k+1$ ones. The edges of G_k connect those vertices at Hamming distance one. Find a Hamiltonian path in G_k for all $k \geq 0$. [This is a well-known problem, sometimes known as the "middle two levels problem," which, as of this writing, is unresolved.]

39. **[3]** Implement the Eades-Hickey-Read algorithm to run in constant amortized time and $O(n)$ space.

40. **[1+]** Modify Algorithm 5.20 so that it outputs the elements of $\mathbf{B}(n+k-1, n)$ by homogeneous transpositions.

41. **[2]** Consider the algorithm for generating the k-compositions of n by transpositions, modified to list the elements of $\mathbf{B}(n+k-1, n)$ (as in the previous exercise). Let $s(n,k)$ be the number of 0's between the two bits that get transposed, summed over all combinations. Show that $s(n,1) = 0$ and, if $k > 1$,

$$s(n,k) \;=\; s(n-1,k) + s(n,k-1) + \begin{cases} 0 & \text{if } n \text{ odd} \\ k-2 & \text{if } n \text{ even.} \end{cases}$$

Solve this recurrence relation. Show that the corresponding recurrence relation for the Eades-McKay listing is $r(n,1) = r(n,n) = 0$ and, if $1 < k < n$,

$$r(n,k) \;=\; n-k-1+r(n-2,k-2)+r(n-2,k-1)+r(n-1,k).$$

Solve this recurrence relation. Of course, $s(n,k) = r(n+k-1,n)$ in view of Lemma 5.5.

42. [**1+**] Derive a simple recursive formula for the rank of a k-composition of n in the order of the Knuth list $\mathbf{Comp}(n,k)$.

43. [**2+**] Show that permutations can be generated by a tree of transpositions (Slater), and that there is a CAT algorithm (Jiang).

44. [**2+**] Implement the SJT algorithm in a loopless manner using the hint at the end of the subsection describing the SJT algorithm.

45. [***R***] Can the elements of $\mathbf{T}(n)$ be generated so as to satisfy the [2-Trans] closeness criteria: $|i-j| \leq 2$?

46. [***R***] Can the elements of $\mathbf{T}(n)$ be generated so as to satisfy the [H-Trans] closeness criteria: no 0's between b_i and b_j?

47. [***R+***] Does $G(\mathcal{P})$ have a Hamilton path whenever $D(\mathcal{P}) = 0$?

48. [**2+**] Show that multiset permutations of type $0^{n_0}1^{n_1}\cdots t^{n_t}$ can be generated by a CAT algorithm so that successive permutations differ by an adjacent transposition whenever there are at least two odd n_i. (Stachowiak)

Questions about set partitions

49. [***R−***] Can RG sequences be generated so that successive sequences differ in only one position, *including the first and last partitions*?

50. Consider the graphs $G(n)$ and $G(n,k)$ whose vertices are the RG sequences in $\mathbf{S}(n)$ and $\mathbf{S}(n,k)$, and where an edge joins those sequences that differ by ± 1 in exactly one position. [**1+**] Show that this graph is connected and bipartite. [**1+**] Define the *index* of an RG sequence to be the sum of values in the sequence. Find a recurrence relation for $D(n,k)$, the number of RG sequences in $\mathbf{S}(n,k)$ of even index minus the number of odd index. Determine the value of $D(n,k)$. Let $D(n) = \sum_{k=1}^{n} D(n,k)$. Show that

$$D(n) = \begin{cases} -1 & \text{if } n \bmod 6 = 3,4 \\ \;\;0 & \text{if } n \bmod 3 = 2 \\ +1 & \text{if } n \bmod 6 = 0,1 \end{cases}$$

[***R−***] Find an involution proof of the equation above. [***R−***] Find a Hamilton path in $G(n)$ if n mod 12 is not 4,6,7,9. These cases are ruled out since $012\cdots(n-1)$ is a pendant vertex.

51. [**2**] Show that $J(\mathcal{P})^2$ is Hamiltonian for any poset $\mathcal{P}$.

52. [***R***] Develop a CAT algorithm for generating the ideals of an arbitrary poset.

53. [***R***] Let $R = (r_1, r_2, \ldots, r_m)$ and $C = (c_1, c_2, \ldots, c_n)$ be nonnegative integral vectors, where C is montone and $r_i \leq n$ for $i = 1, 2, \ldots, m$. The Gale-Ryser theorem asserts that a m by n (0,1)-matrix with row sums R and column sums C exists if and only if $C \leq R^*$, where R^* indicates the conjugate of R (regarded as an integer partition). Let $G(R, C)$ be the graph whose vertices are all such (0,1)-matrices. Two vertices are adjacent if they differ only by a 2 by 2 sub-matrix, one of which is the identity I_2 and the other of which is obtained by flipping the bits of I_2. Ryser ??? showed that $G(R, C)$ is connected. Investigate the Hamiltonicity properties of $G(R, C)$.

54. [**2**] Refering to the previous exercise, and assuming that $c_1 = c_2 = \cdots = c_n$, explain why $G(R, C)$ is Hamiltonian and that a Hamilton cycle can be generated by a CAT algorithm. HINT: How may solutions are there?

55. [***R***] Let G be a planar graph. A *two-switching* in a planar embedding of G is the "flip" of some subgraph of G (explain this more !!!). Define the *embedding graph* $Em(G)$ to be the graph whose vertices are the different planar embeddings of G, where two embeddings are joined by an edge of $Em(G)$ if and only if they differ by a two-switching. Whitney (reference) proved that $E(G)$ is always connected. Show by example that it is not necessarily bipartite. Explore the Hamiltonicity properties of the embedding graph.

56. [***R***] Consider the set of all perfect matchings on the n by m grid. Any grid with a perfect matchings must contain a 4-cycle that contains two matched edges. By switching the matched and unmatched edges on the 4-cycle another perfect matching is obtained. Define the *switching graph* $Sw(n, m)$ to be the graph whose vertices are the perfect matchings on an n by m grid, and whose edges are join vertices that differ by a "switching". Is $Sw(n, m)$ connected? Investigate the Hamiltonicity properties of the switching graph.

57. [***R−***] The number of labellings of a free tree T with n nodes is $n!$ divided by the size of the automorphism group of T. Show that there is Gray code of such labellings, where successive labellings differ by an exchange of labels on some edge. Can such a Gray code be generated efficiently?

5.15 Bibliographic Remarks

The Binary Reflected Gray Code

United States patent number 2632058 was granted to Frank Gray ("assignor to Bell Telephone Laborartories, Inc.") on March 17, 1953 after having been applied for on November 13, 1947 ([160]). For a history of the Gray code see Heath [175].

Papers about the Binary Reflected Gray Code and its applications: Arazi [11], Chang, Lee, and Du [52], Cohn [66], Er [106], Flajolet and Ramshaw [130], Gilbert [150], Kirschenhofer and Prodinger [210], Richards [343], Sharma and Khanna [384], [382], [383], van Zanten [432], Cummings [75]. Diaconis and Holmes [84] discuss applications of combinatorial Gray codes to statistics.

Games and Puzzles

The classic "Towers of Hanoi" puzzle actually has nothing whatever to do with Hanoi (except, perhaps, that the French were involved in Indochina at the time), but was created by the French mathematician and puzzle designer Edouard Lucas in 1883.

For more on the Towers of Hanoi, the excellent article of Poole [313] is a good starting point. A great many papers have been written about the Towers of Hanoi problem and its extensions; see for example, Lu [257], Minsker [282], Wu and Chen [462], Veerasamy and Page [434], Chapter 51 of Dewdney [83], and Section 7.6 of Parberry [304]. Many other references may be found in [313]. The simple iterative solution given in Exercise 11 is from Buneman and Levy [46].

The SPIN-OUT puzzle is described in Pruhs [323].

Combinations

[Trans] Our approach follows Bitner, Ehrlich, and Reingold [32], but see also Misra [284], Tang and Liu [410], [251]. [H-trans] The first published account of a list satisfying this condition is due to Ehrlich [101]; furthermore, Ehrlich provides a loopless implementation of his algorithm. However, his list is much more complicated to define than the one we presented, which was first published by Eades and McKay [96] , although Chase [57] attributes such a list to Donald E. Knuth and William R. Bauer! [2A-trans] The list formulation here is believed to be new, but has its roots in the papers of Chase [57] and McCarthy and Jenkyns [194]. [2-trans] There are three known proofs of that $\mathbf{B}(n,k)$ can be generated by adjacent transpositions if and only if n is even and k is odd (or $k = 0, 1, n-1, n$), discovered independently by Buck and Wiedemann [44], Eades, Hickey, and Read [95], and Ruskey [364].

Other papers about Gray codes for combinations are Lam and Soicher [237] , Chase [57]. The first algorithm to satisfy the homogeneous transposition condition was discovered by Lathroum and implemented by Chase [56]. Ehrlich [101] gave a loopless algorithm that can start at any string of the form $0^p1^k0^{n-p-k}$; a nice description of this algorithm may be found in the book of Even [115]. **!!!! Check on these !!!!**

Gray codes for Lyndon words are discussed by Cummings [74] (for more on Lyndon words see Chapter ??).

!!!! What does Luneburg [258] discuss??? !!!!

The book of Lüneburg [260] contains much material on Gray codes. In particular, he has developed algorithms for doing arithmetic where integer i is represented by $unrank(i)$.

See Sedgewick [377] for an excellent survey of algorithms for generating permutations by transpositions. The Steinhaus-Johnson-Trotter algorithm is from Steinhaus [405], Johnson [198], and Trotter [418]. The discussion of multiset permutations follows Ko and Ruskey [227]. Other algorithms for generating permutations of a multiset by transpositions may be found in Hu and Tien [185] , Sag [368], Bratley [40] and Chase [55]. Korsh and Lipshutz [230] develop a loopless algorithm for listing permutations of a multiset, but in a non-standard representation.

Set Partitions

A Gray code for set partitions is discussed in Kaye [208]; it is based on a Gray code that is attributed to Donald Knuth by Wilf [451]. The lists of partitions into a fixed number of blocks as defined in Section 5.9 are from Ruskey [353]. Another Gray code for set partitions

into a fixed number of blocks is attributed to Brian Hansche in Fill and Reingold[126]. !!! Ehrlich [101] loopless algorithms ??? Other Gray codes for generalized restricted growth strings are developed in Ruskey and Savage [358].

Binary Trees

The Gray code for well-formed parentheses is from Ruskey and Proskurowski [356] (see also [316]). Another Gray code for well-formed parentheses may be found in Vajnovszki [424]. By counting the sizes of the partite sets in $\mathbf{T}(n,k)$ Ruskey and Miller [355] showed that there is no adjacent transposition (i.e., [A-Trans]) Gray code for $\mathbf{T}(n,k)$ for even n or for odd $n > 5$ and odd k (except when $k = 0, 1, n-1,$ or n).

The Gray code for binary trees (Section 5.5) is from Lucas, Roelants van Baronaigien, and Ruskey [256] (see also Lucas [255]). The solution to Exercise 33 is from Roelants van Baronaigien [427].

The rotation graph was used by Pallo [297], who showed that the directed version obtained by using only left rotations is a lattice. This result was anticipated by Huang and Tamari [183]. Further properties of this lattice are given in Pallo [299]. The problem of determining the diameter of the rotation graph G_n was considered by Sleator, Tarjan, and Thurston [392]. Lucas [255] showed that the rotation graph has the Hamilton path generated in Section 5.5; she also showed the much harder result that the rotation graph is Hamiltonian. Knuth [222] (pp. 150-155) presents an algorithm for generating the rotation lattice (or graph).

Compositions

The Gray code for compositions presented in Section 5.7 is due the Knuth (unpublished) and is discussed by Klingsberg [214]. The relationship between this list and the Eades-McKay list was noted in Ruskey [353].

Partially Ordered Sets

A Gray code for the ideals of forest poset is discussed in Koda and Ruskey [226]. Their algorithm is a generalization of the Binary reflected Gray code and can be implemented as loopless algorithm. Gray codes for the ideals of interval orders, both of all sizes and of a given size, have been given by Habib, Nourine, and Steiner [165].

A loopless implementation of the Pruesse-Ruskey algorithm has been developed by Canfield and Williamson [49]. Extensions of the Pruesse-Ruskey algorithm to basic words of antimatroids is carried out in Pruesse and Ruskey [320].

Carla Savage [369] has developed an algorithm for listing partitions so that each partition differs from its predecessor by increasing one part by one and decreasing another part by one (where parts of size 0 are allowed). These results have been extended to the following classes of partitions: (a) partitions of n into parts of size at most k, (b) partitions of n into distinct parts of size at most k; see Rasmussen, Savage and West [333].

Miscellaneous Gray Codes

An excellent survey of results on combinatorial Gray codes is given by Savage [371]

Prokurowski, Ruskey, and Smith [317].

Rall and Slater [325].

Squire, Savage, and West [397] have developed a Gray code for the acyclic orientations of a graph. Permutatation generation papers (some may belong in lex.tex): Boothroyd [37], [38], [39], Bratley [40], Coveyou and Sullivan [72], Eaves [97], Fike [125], Hall and Knuth [168], Heap [174], Howell [181], [182], Langdon [238], Ord-Smith [291], [293], Peck and Schrack [305], Pleszczyński [312], Rohl [348], Robinson [346], Sag [368], Tompkins [415], Wells [439].

Rosemary permutations are discussed in Harada [169] and Read [334].

Combination generation papers: Akl [6]. Bitner, Ehrlich and Reingold [32]. Chase [56]. Ehrlich [100] Ehrlich [101] Kurtzberg [236] Lehmer [243] Lehmer [244] **!!! some belong in lex chapter? !!!**

loopless Algorithms

Loopless algorithms for generating permutations, set partitions with at most k blocks, set partitions with exactly k blocks, r-subsets of n where $l \leq r \leq u$, compositions and subsets are developed in Ehrlich [101], who seems to have originated the term *loopless*.

Let T be a tree with root r. Koda and Ruskey [226] give a loopless algorithm for listing all subtrees of T that are rooted at r.

Dershowitz [80] gives a loopless algorithm for generating permutations. Fenner and Loizou [121] give loopless algorithms for generating numerical partitions. Canfield and Williamson [49] give a loopless implementation of the Pruesse-Ruskey algorithm for generating linear extensions. Roelants [427] gives a loopless algorithm for generating a certain sequence representation of binary trees. Roelants and Neufeld [429] give a loopless algorithm for generating a certain representation of subsets with a given sum.

Chapter 6

Combinatorial Gray Codes: Graph Theoretic Issues

In several countries, and in particular England, there has developed a systematic methodology for ringing a series of bells, typically mounted in a church tower, but sometimes rung with hand-held bells. This "art" of bell ringing is sometimes called "change-ringing" or "campanology" and has its own peculiar terminology. Suppose that there are n distinct bells; they are arranged in descending order of pitch. Bell 1 is called the *treble*, bell n the *tenor*. A *change* is the ringing of each of the bells in some order, each bell rung exactly once. A *round* is the change obtained by ringing the bells in the order $1, 2, \ldots, n$. An *extent* consists of $n! + 1$ changes subject to the following three inviolable rules.

1. The first and last change are both rounds.
2. No change is repeated. Thus except for rounds each possible change is rung exactly once.
3. From one change to the next, no bell moves more that one position in its order of ringing.

The following rules are viewed as being desirable but not absolutely necessary.

4. No bell occupies in the same position for more than two successive changes.
5. Each bell (except perhaps the treble) does the same amount of "work". The work that a bell does is the number of position changes that it undergoes.
6. The sequence of changes is palindromic in a sense to be made clear later.

From these descriptions, it is clear that an extent consists of all $n!$ permutations of $[n]$ subject to various rules. We have already seen lists of permutations that satisfy the first three rules. The list produced by Johnson-Trotter algorithm does, but does not satisfy rule 4 because a bell may occupy the same position for many changes. Because of rule 3, each transition may be regarded as an involution, that is, as a product of disjoint 2-cycles (transpositions).

The natural setting for examining questions of campanology is within the context of finding Hamilton cycles in certain Cayley graphs. The first four rules are the specification

of a combinatorial Gray code for $\mathbb{S}_n$. The following theorem was proven by Rapaport [332], and shows the existence of an extent satisfying the first 4 rules.

THEOREM 6.1 *The Cayley graph $Cay(\mathbb{S}_n : X)$, where X is the three element generating set given below, is Hamiltonian.*

$$X = \{(1\ 2), (1\ 2)(3\ 4)(5\ 6)\cdots, (2\ 3)(4\ 5)(6\ 7)\cdots\}$$

Campanology is a surprising source of combinatorial Gray codes. Combinatorial Gray codes give rise to many interesting mathematical problems, and most of these problems can be converted into questions about graphs. In this chapter we explore the interplay between combinatorial Gray codes and graph theory. The central concept is that of a Hamilton path or cycle in a graph. The subject of Hamiltonicity is vast and impossible to cover in a single chapter or even a book, so we concentrate on those graphs most relevant to combinatorial Gray codes. These are the graphs whose vertices are combinatorial objects with edges defined in some natural way. These graphs often exhibit a high degree of symmetry and tend to be sparse.

We begin our discussion with the many fascinating properties of the hypercube. Succeeding sections discuss results that don't rely on symmetry, and then those that do rely on symmetry — particularly Cayley graphs.

6.1 The Hypercube

The queen of graphs is the hypercube. In the the previous chapter we showed that it is Hamiltonian and how to quickly generate the vertices along a particular Hamilton cycle. In this section we will explore further some of its other properties, concentrating on those that relate to Hamiltonicity. These results are all constructive, but developing algorithms for generating the corresponding lists remains largely unexplored.

LEMMA 6.1 *The graph Q_n is Hamilton-connected.*

Proof: !!! This proof remains to be written. !!! Does this follow from the result that any Cayley graph over an Abelian group is Hamilton-connected? There is a nice proof of this in Savage, Squire, and West [397]. □

6.1.1 Monotone Gray Codes

In the BRGC sometimes a 0 changes to a 1 and other times a 1 changes to a 0. Let us record a x for a 0 to 1 change and a y for a 1 to 0 change. For $n = 4$ we obtain the sequence

$$\alpha = xxyxxyyxxxyyxyy.$$

Let us say that the *weight*, $w(\alpha)$, of such a sequence α is the number of xy or yx pairs that occur. For our example $w(\alpha) = 7$. Over all possible Hamilton paths on the n-cube, what is the maximum value that w could attain? An upper bound is easy to obtain. Consider the contiguous subsequence β of α induced by the changes between 0^n and 1^n. The number of x's in β minus the number of y's is n. Thus

$$w(\alpha) \leq 2^n - n - 1. \tag{6.1}$$

00000	11000	01010	11110
00001	10000	01011	11100
00011	10001	01001	11101
00010	10101	01101	11001
00110	10100	00101	11011
00100	10110	00111	10011
01100	10010	01111	10111
01000	11010	01110	11111

Figure 6.1: A monotone Gray code for $n = 5$.

It is in fact possible to construct a Gray code where the bound is attained when n is odd. The following list for $n = 5$ arises from the Gray code shown in Figure 6.1; it attains the bound.

$$x|xyxyxyxyx|xyxyxyxyxyx|xyxyxyxyx|x$$

Consider any subsequence of alternating x and y's, as delimited by "—" in the sequence above. The corresponding subsets have k or $k+1$ elements for some k. A *monotone Gray code* is one in which all the k-subsets precede any of the $k+2$ subsets. The existence of monotone Gray codes was proven by Savage and Winkler [372], and we give their proof below.

Let $G_n(i)$ denote the subgraph of Q_n induced by $\mathbf{A}(n,i) \cup \mathbf{A}(n,i+1)$; i.e., by all the i and $i+1$-subsets. If π is a permutation of $[n]$ and $A \subseteq [n]$, then $\pi(A)$ denotes the set $\{\pi(a) \mid a \in A\}$. This notation is extended in the natural way to sequences of subsets: $\pi(A_1, A_2, \ldots, A_m) = \pi(A_1), \pi(A_2), \ldots, \pi(A_m)$.

Thinking of Q_n as a lattice under the subset relation, let us say that a chain $\emptyset = x_0 \prec x_1 \prec \cdots \prec x_n = [n]$ in Q_n is an *antipodal chain*. The number of such chains is $n!$ (see Exercise 7). Furthermore, if $y_0 \prec y_1 \prec \cdots \prec y_n$ is another antipodal chain then there is a permutation π of $[n]$ such that $y_i = \pi(x_i)$ for $i = 0, 1, \ldots, n$, namely the one obtained by taking $\pi(x_i \setminus x_{i-1}) = y_i \setminus y_{i-1}$ for $i = 1, 2, \ldots, n$.

THEOREM 6.2 (SAVAGE-WINKLER) *There is a monotone Gray code in Q_n that can be written*

$$P_0, P_1^{-1}, P_2, P_3^{-1}, \ldots, P_{n-1}^{sign(n-1)}$$

such that each P_i is a path in $G_n(i)$ of the form

$$P_i = y_i, \ldots, x_{i+1},$$

where $\{x_i\}_{i=0}^n$ and $\{y_i\}_{i=0}^n$ are antipodal chains.

PROOF: The proof proceeds by induction on n. If $n = 1$, then the Hamilton path is $P_0 = \emptyset, \{1\} = y_0, x_1$. Assume that the theorem is true for n; we prove it for $n+1$. If $A \subseteq [n]$ we denote by A' the set $A \cup \{n+1\}$ and extend this notation componentwise to sequences of subsets.

Let $P_0, P_1, \ldots, P_{n-1}$ be as defined in the statement of the theorem. For $n+1$ we will show how to construct a Gray code $R = R_0, R_1^{-1}, \ldots, R_n^{sign(n)}$ in Q_{n+1} satisfying the theorem. Define R_i as follows.

$$R_i = \begin{cases} P_0 & \text{if } i = 0 \\ \pi(P'_{i-1})^{-1}, P_i & \text{if } 0 < i < n-1 \\ \pi(P'_{n-1})^{-1} & \text{if } i = n \end{cases}$$

Since R contains the subsets of $P_0, P_1, \ldots, P_{n-1}$ (all subsets not containing $n+1$) together with the subsets of $\pi(P'_0), \pi(P'_1), \ldots, \pi(P'_{n-1})$ (all subsets containing $n+1$) and each subset occurs exactly once, R contains all vertices of Q_{n+1} exactly once. We need to show that the interfaces between the $R_i^{sign(i)}$ and $R_{i+1}^{sign(i+1)}$ lists have Hamming distance one and that each R_i is a path in $G_{n+1}(i)$.

For $0 < i < n$ the interface between $\pi(P'_{i-1})^{-1}$ and P_i is $(\pi(x'_i), y_i)$. Since $\pi(x'_i) = y_i \cup \{n+1\}$ they have Hamming distance one. The other interfaces are between R_i and R_{i+1}^{-1}, or between R_i^{-1} and R_{i+1}. Between R_i and R_{i+1}^{-1} the interface is (x_{i+1}, x_i), which has Hamming distance one. Between R_i^{-1} and R_{i+1} the interface is $(\pi x'_{i+1}, \pi x'_i) = (y'_{i+1}, y'_i)$, which has Hamming distance one.

The initial interface between P_0 and P_1^{-1} is (x_1, x_2), which has Hamming distance one. The final interface is either between P_{n-1} and $\pi(P'_{n-1})$ or between $\pi(P'_{n-2})$ and $\pi(P'_{n-1})^{-1}$. Between P_{n-1} and $\pi(P'_{n-1})$ the interface is $(x_n, \pi(x'_n)) = ([n], [n+1])$. Between $\pi(P'_{n-2})$ and $\pi(P'_{n-1})^{-1}$ the interface is $(\pi(x'_{n-1}), \pi(x'_n)) = (y'_{n-1}, y'_n)$. In either case, the interfaces have Hamming distance one.

For the path R, from the definition of the R_i, the two antipodal chains are

$$x_0, x_1, \ldots, x_n, [n+1]$$

and

$$y_0, \pi(y'_0), \pi(y'_1), \ldots, \pi(y'_n).$$

□

We make now a few observations about the above proof. The number of nodes in the path P_i of Q_n is $2\binom{n-1}{i}$. In the first antipodal chain, the recursive construction implites that $x_i = [i]$. When n is odd, the monotone Gray code ends at $[n+1]$; when n is even, it ends at $\pi(y'_{n-1})$.

Cube free Hamilton cycles in the cube

The first 2^{n-1} vertices of the BRGC for n are a Hamilton path in a Q_{n-1}, as are the last 2^{n-1} vertices. It is natural to ask whether there is a Hamilton cycle in Q^n that does not include a Hamilton path in Q^k for any $1 < k < n$. Perhaps surprisingly, the answer is yes.

As far as we are aware, no one has addressed the problem of developing an algorithm for generating such a path.

A collection of results about the hypercube

To close this section we simply list some interesting results and conjectures about the n-cube.Many of these results involve Gray codes, either directly or indirectly.

1. For n even there is a Hamilton decomposition of the n-cube. A *Hamilton decomposition* is a partitioning of the edges of the cube such that each partite set (term?) is a Hamilton cycle. [See Alspach, Bermond, and Sotteau [8] for more information on this problem.]

2. The n-cube is *pancyclic*; i.e., it possesses cycles of all possible even lengths. The n-cube is Hamilton-connected; i.e., it possesses Hamilton paths between every pair of vertices of opposite parity. This result is due to Saad and Scults [367].

3. For $n \geq 5$, there is a Hamilton cycle in Q_n that does not have any subpath that is a Hamilton path of a Q_r for $2 \leq r \leq n-1$. This result is due to Mills [281]; for a simpler proof see Ramras [329].

4. Imagine the following "game" (one player) played on the n-cube (may be easier to visualize as the Boolean algebra lattice). Place 2^n tokens arbitrarily at the vertices of the cube; a vertex may get more than one token. Let v be a node containing at least two tokens and w a node adjacent to v. A token may be moved from v to w but at a cost of one token, charged to v. There is a way to play the game to that a token occurs at the node corresponding to 0^n. This result is due to Fan Chung. [REFERENCE???]

5. All trees are bipartite. Let us say that a tree is *balanced* if the number of nodes in its two partite sets are equal. Havel [173] conjectures that all balanced trees of maximum degree three and with 2^n nodes are spanning subgraphs of the n-cube. A weaker conjecture of Alan Wagner [REFERENCE???] asserts that all trees of maximum degree three that possess a perfect matching with 2^n nodes are spanning subgraphs of the n-cube.

6. The genus of the n-cube is $1 + (n-4)2^{n-3}$. See Ringel [345].

   The crossing number of the n-cube is UNKNOWN ???.

   The thickness of the n-cube is $1 + \lfloor n/4 \rfloor$. See Kleinert [213].

   The *bandwidth* of a graph G with vertices $1, 2, \ldots, n$ is

   $$b(G) \;=\; \min_{\pi \in \mathbb{S}_n} \max_{[vw] \in E(G)} \{|\pi(v) - \pi(w)|\}$$

   A related quantity is obtained by taking the sum rather than the maximum.

   $$s(G) \;=\; \min_{\pi \in \mathbb{S}_n} \sum_{[vw] \in E(G)} \{|\pi(v) - \pi(w)|\}$$

   The bandwidth of the n-cube is $b(Q_n) = \sum_{k=1}^{n-1} \binom{k}{\lfloor k/2 \rfloor}$. The "minimum sum" of an n-cube is $s(Q_n) = 2^{n-1}(2^{n-1} - 1)$. See Harper [171], [172].

7. A natural question to ask is: How many edges can be removed from a n-cube and still approximately preserve the proximity of vertices? There is a spanning subgraph G of $Q = Q_n$ of maximum degree $\lceil (n+1)/2 \rceil$ (and average degree $(n+1)/2$) such that, for all vertices u and v,

   $$d_G(u, v) \;\leq\; 2 + d_Q(u, v).$$

This result is due to Liestman and Shermer [247].

8. Among all bipartite graphs, n-cubes are characterized by the property that there are $d!$ shortest paths between vertices that are distance d apart. This result is due to Foldes [136]. !!! look up more recent result in J. Graph Theory !!!

9. Let P be a perfect matching in Q_n. I conjecture that there is a Hamilton cycle in Q_n that contains every edge in P. Examples can be found of *maximal matchings* that cannot be extended to Hamilton cycles.

10. Venn diagram problem.

11. The number of spanning trees of Q_n is

$$\tau(Q_n) = \frac{1}{2^n} \prod_{i=1}^{n} (2i)^{\binom{n}{i}}. \tag{6.2}$$

This may be found from the Matrix Tree Theorem. Harary, Hayes and Wu [170] attribute this result to A.J. Schwenk (*Spectrum of a graph*, Ph.D. dissertation, University of Michigan (1973)).

12. The "snake-in-a-box" problem asks what is the length of the longest induced path in Q_n. Klee has proven that the length is asymptotic to $2^n/n$. !!! Check on this... REFERENCE ???

13. Defining $dom(G)$ to be the largest value k for which there exists a partition of V into k blocks such that each block is a dominating set (a *dominating set* is a set of vertices that are adjacent to every other vertex). Mike Fellows in his Ph.D. thesis shows that $dom(Q_n)$ is asymptotic to n.

6.2 Hamiltonicity Results for Graphs

The closeness graphs of many types of combinatorial objects are vertex-transitive and the existence of algorithms can often be inferred from general results on vertex-transitive graphs. We mention several such results.

There is a certain natural sense in which "reasonable" closeness criteria always lead to Gray codes. By reasonable we simply mean that the underlying graph is connected. But if this is the case then we can get a Gray code in which successive objects differ by at most three iterations of the closeness operator. The formal statement is as follows.

THEOREM 6.3 (SEKANNINA) *The cube of any connected graph is Hamiltonian*

PROOF: It is sufficient to prove the theorem for trees. Proceed by induction on the height of the tree, where some node r is chosen arbitrarily to be the root. We actually prove the stronger statement that T^3 has a Hamilton path starting at r and ending at any specified child x of r. The statement is clearly true if the height is 0 or 1. So let T be a tree of height at least 2. Let $T_1, T_2, \ldots, T_p$ be the principle subtrees of T, i.e., the rooted trees that remain when r is removed. Assume, without loss of generality, that x is the root of T_p, and

Figure 6.2: Illustration of the 4-cycle construction.

let P_i be a Hamilton path through T_i^3 that starts at the root r_i of T_i and ends at a child of r_i (or at r_i if T_i is a single node). Then

$$r, P_1^{-1}, P_2^{-1}, \ldots, P_p^{-1}$$

is a Hamilton path in T^3 that starts at r and ends at $x = r_p$. □

That proof was relatively simple. Much harder to prove is the following result of Fleishner, which we state but do not prove.

THEOREM 6.4 (FLEISHNER) *The square of a 2-connected graph is Hamiltonian.*

6.2.1 A useful lemma

If G is an undirected graph, then vertex-disjoint cycles C and C' are said to *share a 4-cycle* if there is a 4-cycle u, v, x, y such that $[u, v] \in C$ and $[x, y] \in C'$. A *cycle-decomposition* of G is a collection of vertex-disjoint cycles in G such that every vertex of G is on some cycle. The following simple lemma is the basis of many proofs of Hamiltonicity. It is most often applied when $p = 2$ (and can be viewed as an iterative application of the $p = 2$ case).

LEMMA 6.2 (THE 4-CYCLE CONSTRUCTION) *If $C_1, C_2, \ldots, C_p$ form a cycle-decomposition of G such that C_1 and C_2 share a 4-cycle, C_{p-1} and C_p share a 4-cycle, and C_{i-1} and C_i share a pair of 4-cycles for $i = 3, \ldots, p-1$, then G is Hamiltonian.*

Proof: This proof is best understood by considering Figure 6.2. Note that two of edges on the pair 4-cycles can be adjacent on the same cycle C_i. □

6.3 Vertex-Transitive Graphs

Not all connected vertex trasitive graphs are Hamiltonian. Four such graphs are known. Three of them are shown in Figure 6.3. The first graph is known as the "Peterson graph", which is well-known to every student of graph theory as being non-Hamiltonian and vertex-transitive. The second graph is obtained from it by replacing each vertex with a triangle. The third graph is known as the "Coxeter graph." The fourth example, which is not shown, is obtained from the Coxeter graph by replacing every vertex by a triangle, in the same manner that the second graph is obtained from the Peterson graph.

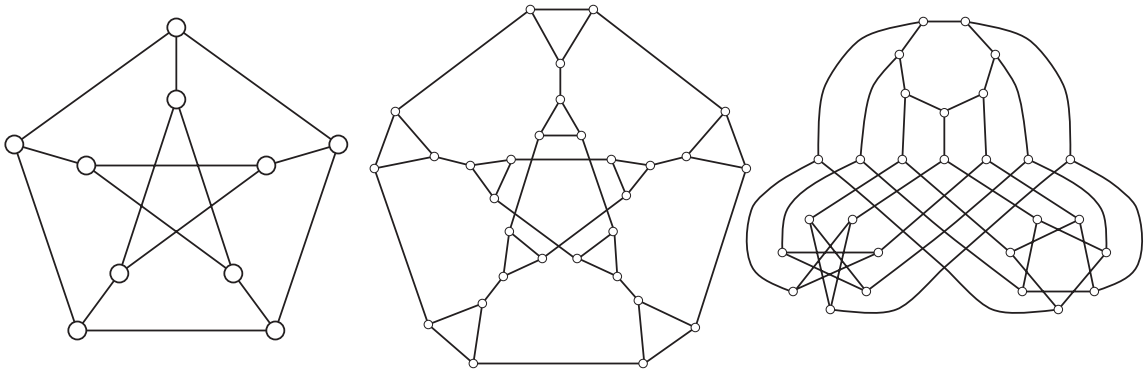


Figure 6.3: Three of the four known non-Hamiltonian vertex-transitive graphs. The second is obtained from the first by replacing each edge with a triangle. The fourth (not shown) is obtained from the third in a similar manner.

QUESTION 6.1 (LOVÁSZ) *Does every vertex-transitive graph have a Hamilton path? Aside from the four known counter-examples, are all other vertex-transitive graphs Hamiltonian?*

Faber-Moore graphs. Arrangement graphs.

6.4 Anti-Gray Codes

In some applications a list of combinatorial objects is required where successive objects differ by a large amount. For example, one might a want a list of all permutations so that successive permutations differ in all positions; i.e., each permutation is a derangement of the previous one. As another example, one might want a list of all length n bitstrings in which successive bitstrings differ in $n-1$ positions. We will simply list some of the known results. Algorithmic issues are usually not a big concern.

6.4.1 Permutations

Below is a list of all 4! permutations of [4] in which successive permutations differ in all positions.

1234	**321**4
4123	4321
2341	2143
3$\underline{4}$12	1$\underline{4}$32
1324	**231**4
4132	4231
3241	3142
2$\underline{4}$13	1$\underline{4}$23
3124	**213**4
4312	4213
1243	1342
2$\underline{4}$31	3$\underline{4}$21

The key to the construction of this table (and its generalization to arbitrary values of n) lies in the permutations of $[n-1] = [3]$ shown in bold. These are the the permutations of $[n-1] = [3]$ that are output by the Steinhaus-Johnson-Trotter algorithm, but any listing of permutations in which successive permutations differ by adjacent transpositions will do. The other permutations of the table are obtained by appending n, repeatedly rotating the permutation to the right, and finally arranging those rotations in an order that leaves n in the second position (the underlined 4's).

Why does this construction work? Clearly, all $n!$ permutations are produced, and those that differ by right rotations differ in all positions. We need only consider what happens before and after every n-th permutation. The interfaces there are of the form

$$\begin{array}{cccccccc} x_{n-1} & n & x_1 & x_2 & \cdots & x_{n-3} & x_{n-2} \\ x'_1 & x'_2 & x'_3 & x'_4 & \cdots & x'_{n-1} & n, \end{array}$$

where $x'_1 x'_2 \cdots x'_{n-1}$ differs from $x_1 x_2 \cdots x_{n-1}$ by an adjacent transposition. For $n \geq 4$, note that x_i ends in a position that is at least two away from i, and thus that it must be in different position than x'_i, even if it participated in the transposition. The construction does not work for $n = 3$, because a derangement listing is impossible if $n = 3$.

The above construction show the existence of a Hamilton cycle in $\vec{Cay}(D : \mathbb{S}_n)$, where D is the set of derangements of $[n]$. In fact, we used the subset of derangements $D = \sigma, \sigma^2, \ldots, \sigma^n, (1\ 2)\sigma^{n-2}, (2\ 3)\sigma^{n-2}, \ldots, (n\text{–}1\ n)\sigma^{n-2}$, where σ is the right rotation $(1\ 2\ \cdots\ n)$.

Savage [370] permutation result.

Odd graphs? Vertices are combinations, edges join those vertices with empty intersection.

6.4.2 Subsets

Culberson and Rawlins result.

6.5 Hamilton Cycles in Cayley Graphs

We begin this section with two easy observations.

LEMMA 6.3 *Every Cayley digraph is vertex-transitive.*

LEMMA 6.4 *Every Cayley digraph is strongly-connected.*

To prove Lemma 6.3, given $a, b \in G$, we wish to show that there is an automorphism θ mapping a to b which preserves adjacencies. Use $\theta(x) = (ba^{-1})x$. To prove Lemma 6.4, in view of Lemma 6.3, it is sufficient to show the existence of a path from the identity, $\mathbf{1}$, to any other group element g. This is easy since there is, by definition of a generating set, a sequence of generators $g_1, g_2, \ldots, g_k$ whose product is g. These lemmata imply that every undirected Cayley graph is vertex-transitive and connected.

There are two ways to specify a Hamilton cycle in a Cayley graph. First, we can list the group elements as they occur along the cycle. Such lists will be delimited by the angle brackets $\langle$ and $\rangle$. Secondly, we could successively list the generators along the cycle (more generally, any path can be specified by a list of generators). Such lists will be delimited by the square brackets [and] as a natural extension of the notation for edges. For example, in the Cayley graph $Cay(\{(1\ 2), (2\ 3)\} : \mathbb{S}_3)$, the Hamilton cycle may be specified either as $\langle 123, 132, 312, 321, 231, 213 \rangle$, or as $[(2\ 3), (1\ 2), (2\ 3), (1\ 2), (2\ 3), (1\ 2)]$.

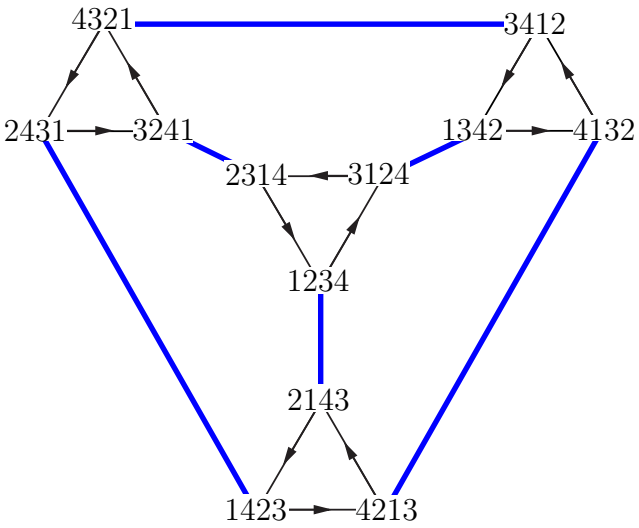


Figure 6.4: The non-Hamiltonian directed Cayley graph $\vec{Cay}(\{(1\ 2)(3\ 4), (1\ 2\ 3)\}:\mathbb{S}_4)$.

CONJECTURE 6.1 *Every connected Cayley graph is Hamiltonian.*

6.5.1 Directed Cayley Graphs

Although it is believed that all undirected connected Cayley graphs are Hamiltonian, there are examples of directed connected Cayley graphs that are non-Hamiltonian, such as $\vec{Cay}(\{(1\ 2)(3\ 4), (1\ 2\ 3)\}:\mathbb{S}_4)$ as illustrated in Figure 6.4. This example can also be specified as $G = \langle a, b \mid a^2 = b^3 = (abab^2)^2 = id\rangle$ and is isomorphic to the direct product $\mathbb{C}_2 \times \mathbb{A}_4$. As another example, Rankin [331] shows that $\vec{Cay}(\{(2\ 4\ 6\ 5\ 3), (1\ 6\ 3)(2\ 4\ 5)\}:\mathbb{A}_6)$ is not Hamiltonian.

THEOREM 6.5 (TROTTER-ERDÖS) *The graph Cayley graph of $\mathbb{C}_n \times \mathbb{C}_m$ is Hamiltonian if and only if there are positive integers p and q such that $\gcd(p, n) = 1 = \gcd(q, m)$ with $\gcd(m, n) = p + q$.*

The Trotter-Erdös [419] result is also said to be implicit in the work of Rankin [331].

Even though an Abelian directed Cayley graph can fail to have a Hamilton cycle, they all possess a Hamilton path. The proof follows the same general reasoning as that of Theorem 6.7, the difference being that we can't use the inverse of generator. In the toroidal spanning grid graph we simply snake back and forth from top to bottom to get the Hamilton path.

LEMMA 6.5 *Every Cayley digraph has a Hamilton path.*

He is a compendium of Hamiltonicity results for Cayley graphs of particular orders. These results are strong in the sense that they apply to any generating set for the group, but are weak in the sense that groups with these orders are rather restricted. For example, it is clear that every Cayley graph over a group of prime order is Hamiltonian, since all such groups are cyclic.

THEOREM 6.6 *Let p and q be prime. Every Cayley digraph on a group of order p ($p > 2$), pq, $4q$ ($q > 3$), p^2q ($2 < p < q$), $2p^2$, $2pq$, $8p$, or $4p^2$ is Hamiltonian.*

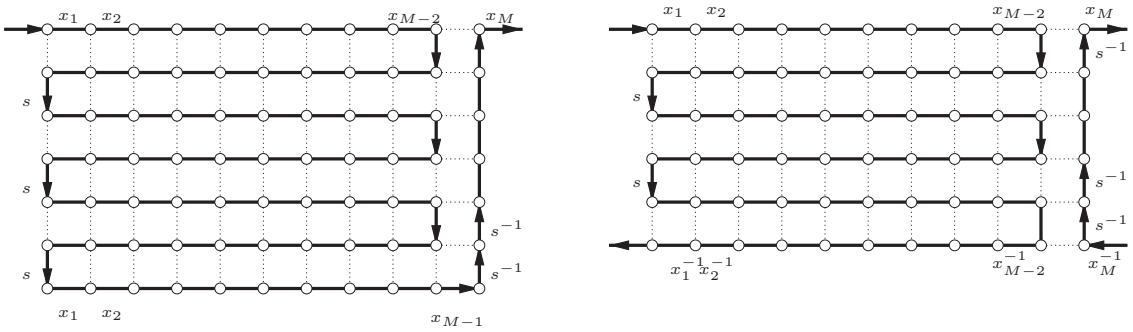


Figure 6.5: Hamilton cycles in Cayley graphs over Abelian groups.

6.5.2 Undirected Cayley Graphs

THEOREM 6.7 *Every Cayley graph over an Abelian group is Hamiltonian.*

PROOF: Let G be an Abelian group and let S be a generating set for G. We proceed by induction on $n = |S|$. If $n = 1$, say with $S = \{s\}$, then

$$(\mathbf{1}, s, s^2, \ldots, s^{m-1} \;=\; [s, s, \ldots, s]$$

is a Hamilton cycle in $Cay(S{:}G)$, where m is the order of s.

If $n > 1$, then let $s \in S$ and $T = S \setminus \{s\}$. Furthermore, let $H = \langle T \rangle$, the subgroup generated by T. Inductively, there is a Hamilton cycle $[x_1, x_2, \ldots, x_M]$ in $Cay(T{:}H)$, where $M = |T|$.

Let $X = [x_1, x_2, \ldots, x_{M-2}]$. If m is odd, then the cycle is

$$[X, s, X^{-1}, s, X, s, X^{-1}, s, X, \ldots, s, X, x_{M-1}, s^{-1}, s^{-1}, \ldots, s^{-1}, x_M].$$

If m is even, then the cycle is

$$[X, s, X^{-1}, s, X, s, X^{-1}, s, X, \ldots, s, X^{-1}, x_M^{-1}, s^{-1}, s^{-1}, \ldots, s^{-1}, x_M].$$

The coset Hs generates the quotient group G/H. Hence, the cosets of H are $H, Hs, Hs^2, \ldots, Hs^m$, where $m = |G|/|H| - 1$. These cycles are illustrated in Figure 6.5, which shows the underlying spanning toroidal grid structure of the Cayley graph. The horizontal direction represents H and the vertical direction G/H. Each row corresponds to a coset of H, the first row to H itself. □

In fact, Cayley graphs over Abelian groups have much stronger Hamiltonicity properties, such as Hamilton-connectedness.

THEOREM 6.8 (CHEN-QUIMPO) *Every connected Cayley graph over an Abelian group is Hamilton-connected.*

6.5.3 Cayley graphs over $\mathbb{S}_n$

In this subsection we consider Cayley graphs whose vertex set consists of all $n!$ permutations of $[n]$ and whose generators are transpositions.

With each set S of transpositions we may associate an undirected graph $G(S)$, obtained by treating each transposition $(x\ y)$ as the edge $[x, y] \in E(G(S))$. Under what conditions on $G(S)$ is it the case that S generates the symmetric group $\mathbb{S}_n$? The answer is given in the following theorem.

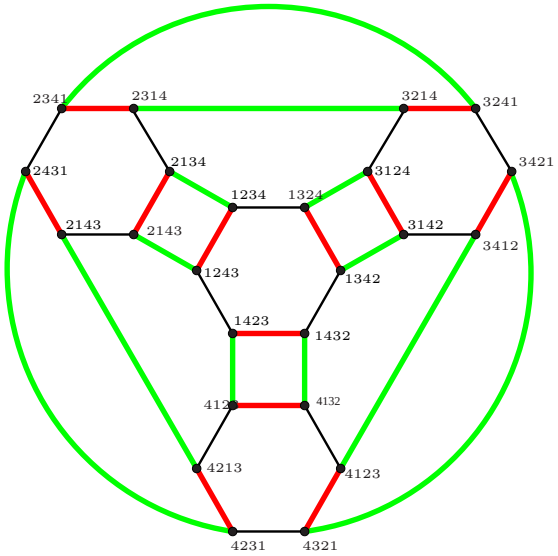


Figure 6.6: The Cayley graph $Cay(\{(1\ 2),(2\ 3),(3\ 4)\}:\mathbb{S}_4)$.

THEOREM 6.9 *A set S of transpositions generates $\mathbb{S}_n$ if and only if $G(S)$ is connected.*

PROOF: If $G = G(S)$ is connected then between every pair of vertices x and y there is a path $x = x_1, x_2, \ldots, x_m = y$ in G. By the SJT (Steinhaus-Johnson-Trotter) algorithm, the transpositions $(x_1\ x_2), (x_2\ x_3), \cdots, (x_{m-1}\ x_m)$ generate all permutations of $\{x_1, x_2, \ldots, x_m\}$. In particular, the transposition $(x\ y)$ is generated and thus the set S generates the transposition of any two elements. Since every permutation is the product of transpositions (Lemma 2.7), every permutation is generated by S.

Conversely, if G is disconnected, say with x and y in different connected components, then (thinking of the permutations in one-line notation and the transpositions acting on the positions) there is no way to transpose an element in position x with an element in position y. Therefore, not all of $\mathbb{S}_n$ can be generated. □

Thus, in considering $\mathbb{S}_n$, as generated by transpositions, we need only consider minimally connected graphs $G(S)$; i.e., we need only consider trees. A *tree of transpositions* is a tree T whose nodes are labelled $1, 2, \ldots, n$, where n is the number of nodes in the tree.

Figure 6.6 shows the Cayley graph on $\mathbb{S}_4$ whose generating set is given by the tree of transpositions that is a path on four vertices, labelled sequentially. Figure 6.7 shows another Cayley graph on $\mathbb{S}_4$, this one with generating set given by the tree of transpositions that is a star. In general, a tree of transpositions always yields a Hamiltonian Cayley graph.

THEOREM 6.10 (SLATER) *Let T be a tree of transpositions with $n \geq 4$ nodes and s and d be elements of $[n]$. Starting at an arbitrary permutation π_1, the graph $Cay(T:\mathbb{S}_n)$ has a Hamiltonian path $\pi_1, \pi_2, \ldots, \pi_{n!}$, such that $\pi_{n!}(d) = s$.*

Proof. The proof is by induction on n. The base cases for $n = 4$ are illustrated in Figure 6.5.3.

If $n > 4$ then let $[p, q]$ be an edge of T, where p is a leaf, and let T' denote T with vertex p removed. We may assume that $d \neq p$ since T has at least two leaves. Now let

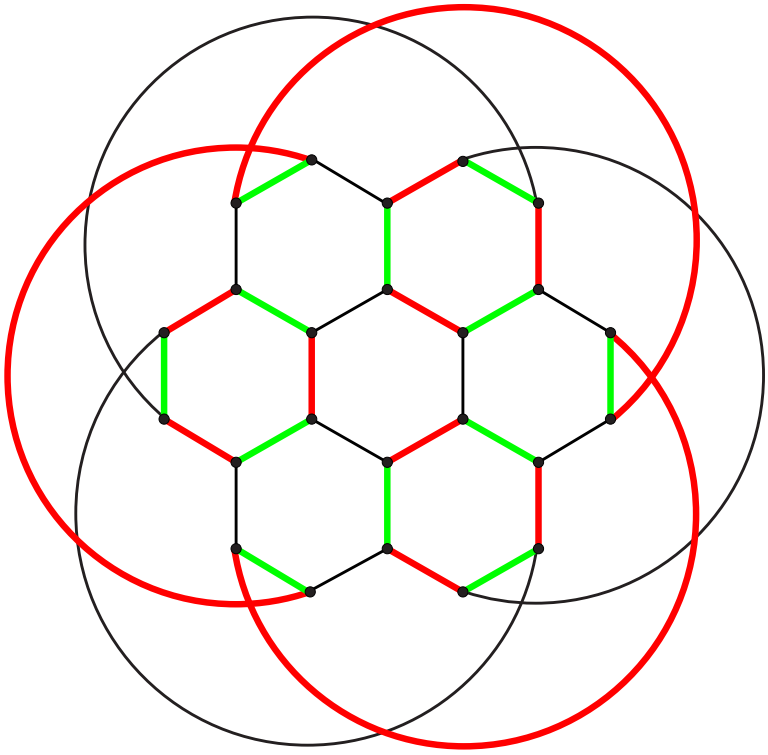


Figure 6.7: The Cayley graph $Cay(\{(1\ 2),(1\ 3),(1\ 4)\}{:}\mathbb{S}_4)$.

$\pi_1(p) = i_1, i_2, \ldots, i_n$ be a sequence of the nodes of the tree (i.e., a permutation of $[n]$) where $i_n \neq s$. Let $\mathbf{L}_z(T', x, y)$ denote a list $\rho_1, \rho_2, \ldots, \rho_{(n-1)!}$ of permutations of $[n]$ in which z always occupies position p, successive permutations differ by a transposition from T', and $\rho_{(n-1)!}(x) = y$. Inductively, such lists L exist with any initial permutation ρ_1 for which $\rho_1(p) = z$ and $y \neq z$ (unless $x = p$).

Then there is a list of the following form that satisfies the conditions of the theorem.

$\mathbf{L}_{i_1}(T', q, i_2)$	$Swap(i_1, i_2)$
$\mathbf{L}_{i_2}(T', q, i_3)$	$Swap(i_2, i_3)$
$\vdots$	
$\mathbf{L}_{i_{n-1}}(T', q, i_{n-1})$	$Swap(i_{n-1}, i_n)$
$\mathbf{L}_{i_n}(T', d, s$	

At the interfaces between the **L** lists the permutations differ only by the transposition $(p\ q)$. □

The proof of Slater is slightly different in that his final condition is $\pi_{n!}(d) = s$, rather than $\pi_{n!}(d) = \pi_1(s)$.

6.6 Exercises

Questions about the hypercube

base case listings go here

Figure 6.8: Base case listings.

1. **[1+]** Show that the subgraph of Q_n induced by those vertices whose values *in binary* are $0, 1, \ldots, k$ has a Hamilton path, for all $0 \leq k \leq 2^n - 1$. Develop a CAT algorithm for generating such a Hamilton path.

2. **[1+]** Show that Q_n is Hamilton laceable.

3. **[1+]** For $n = 2, 3, 4, 5$ show explicitly the monotone Gray codes in Q_n that are produced by the Savage-Winkler proof.

4. **[*R*]** Give a bijective proof of the formula (6.2) for $\tau(Q_n)$, the number of spanning trees of the n-cube.

5. **[2]** The *cube-connected cycle* CCC_n is the graph obtained from Q_n by replacing each vertex with an n-cycle, the ith vertex of the cycle connected to the ith dimension edge of the original vertex. More formally, $V(CCC_n)$ consists of all pairs $(d; \mathbf{x})$ where $\mathbf{x}$ is a length n bitstring, $0 \leq d \leq n-1$, and

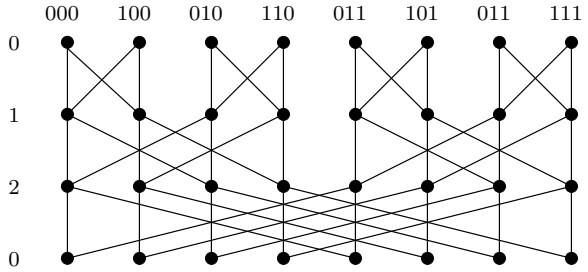
$$\begin{aligned} E(CCC_n) &= \{[(d;\mathbf{x}), (d+1 \pmod n; \mathbf{x})] \mid x \in V(Q_n) \text{ and } 0 \leq d < n\} \\ &\quad \cup \{[(d;\mathbf{x}), (d;\mathbf{y})] \mid [\mathbf{x}, \mathbf{y}] \in E(Q_n)\} \end{aligned}$$

   Show that CCC_n is Hamiltonian. **[2]** Show that CCC_n is a Cayley graph.

6. The d-dimensional *butterfly graph* BG_n^d is a graph whose vertex set consists of all pairs $(r; \mathbf{x})$ where $0 \leq r < n$, $\mathbf{x} \in \Sigma_d^n$, and whose edge set $E(BG_n^d)$ is

$$\{[(r;\mathbf{x}), (r+1 \bmod n; \mathbf{y})] \mid \mathrm{Ham}(\mathbf{x}, \mathbf{y})\}.$$

   The graph BG_3^2 is shown below (vertices with $r = 0$ are repeated for the sake of clarity). The row labels are r, the column labels are $\mathbf{x}$.



   [2] Show that BG_n^d is Hamiltonian. **[2]** Show that BG_n^d is a Cayley graph.

7. **[*R*−]** The transition sequence T_n from the BRGC gives rise to successive pairs of the form $(1, k)$ or $(k, 1)$ which, regarded as the edges of an undirected graph, is a n-star. In this case we call the n-star, the *graph of position changes*. Is there a Hamilton path (or cycle) on Q_n in which the graph of position changes is a path in which successive vertices are labelled $1, 2, ..., n$? What are necessary and sufficient conditions for a graph to be graph of position changes of a Hamilton path on Q_n? **[1]** What is the relationship between this problem and the previous one?

8. **[2]** Every Hamilton path in Q_n gives rise to a transition sequence S obtained by recording the position that changes. Recall that $T_n = 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, \ldots$ was the transition sequence of the BRGC. Prove the following characterization: The sequence $S = s_1, s_2, \ldots, s_N$ is a transition sequence if and only if, for all $1 \leq i \leq j \leq N = 2^n - 1$, the subsequence $s_i, s_{i+1}, \ldots, s_j$ contains at least one digit an odd number of times. (For a Hamilton cycle add the condition that S, s_N contains each symbol an even number of times.)

9. **[2]** Find the 9 non-isomorphic Hamilton cycles in Q_4.

10. **[2]** Show the existence, in Q_5 of a Hamilton cycle none of whose subpaths induce any r-cubes, $r = 2, 3, 4$.

11. **[*R*−]** Find a Gray code on Q_n where the difference in successive strings, regarded as binary numbers, is as small as possible. Results for the maximum difference or the total difference would both be interesting.

12. **[*R*]** Regard the hypercube Q_n with its vertices labelled by the subsets of $[n]$ in the usual manner. Is there a Hamilton path in Q_n with the property that any subset $A \subseteq [n]$ is preceded on the path by all subsets of A, with at most one exception? If there is an exception, then it must occur immediately following A. For example, in Q_4,

    $$\emptyset, 1, 12, \underline{2}, 23, \underline{3}, 34, \underline{4}, 24, 124, \underline{14}, 134, \underline{13}, 123, 1234, \underline{234}$$

    is such a path (the exceptions are underlined).

13. **[1+]** Give an example of a distributive lattice whose Hasse diagram is Hamiltonian but that does not have a monotone Gray code.

14. In the n-disk Towers of Hanoi problem (Exercise ???) record for each disk the peg 1, 2, or 3 on which it is impaled and call the resulting string over [3] a *tower string*. Form a graph H_n whose vertices are the tower strings and whose edges join permutations differing by a valid move. **[1−]** (a) How many tower strings are there as a function of n? **[1+]** (b) Draw H_3. **[2]** (c) Discover a recursive construction of H_n and use it to prove that H_n is Hamiltonian. **[3]** (d) Find a closed form expression for the total sum of lengths of shortest paths between all ordered pairs of vertices in H_n.

15. The following generalization of the Towers of Hanoi problem is due to Bob Jamison. A GTHP (**G**eneralized **T**owers of **H**anoi **P**roblem) is a 4-tuple (P, G, s, t) where P is a poset (the allowed disk placements on a single peg), G is a graph (the pegs), and s and t are vertices of G (the starting and ending peg). The graph G could be directed or undirected; its edges indicate whether a disk can move from one peg to another. In the classic case P is a total order, meaning no disk may be placed on top of a smaller disk, and G is a triangle, meaning that, subject to the constraint imposed by P, a disk can be moved from one peg to any of the other two pegs. **[*R*−]** Investigate GTHP and try to prove some general results.

    The following variants have been discussed in the literature. (a) The graph G is a path of length 3 and s and t are the two pendant vertices. **[1]** Devise a recusive algorithm for this case and determine how many moves it uses. Can you prove that you've used

the least number of moves? (b) The graph G is a directed triangle. Here there are two possiblities for s and t. [**1**] Devise a simple recusive algorithm for this case and determine how many moves it uses. There is a solution that uses $o(4^n)$ moves. (c) The graph G is the complete graph K_n (the case $n = 4$ has been often considered). [**2**] For the case when $n = 4$ devise an algorithm that uses $O(n2^{\sqrt{2n}})$ moves. (d) The poset P is $A_2 \times A_2 \times \cdots \times A_2$; that is, the posets consists of pairs (x, y) such that x and y are incomparable, but are comparable with every other element of the poset. [**1**] Devise an efficient recursive algorithm to solve the problem in this case.

16. [**2**] Show that there is a derangement ordering (anti Gray code) of Σ_k^n if and only if $k > 2$ or $n = 1$.

Questions about Cayley graphs

17. [**1−**] Prove that every Cayley graph is 2-connected.

18. [**1+**] Show that the graph of Figure 6.4 is not only non-Hamiltonian, but that it does not even possess a Hamilton path.

19. [**2**] The graph $Cay(\{(1\ 2\ 3\ 4\ 5), (1\ 2)(3\ 4)\} : A_5)$ is known as the "Buckyball." Prove that the Buckyball is Hamiltonian.

20. [***R*−**] Consider the undirected Cayley graph on $\mathbb{S}_7$ with generators $g_1 = (2\ 3)(4\ 5)(6\ 7)$, $g_3 = (1\ 2)(4\ 5)(6\ 7)$, $g_5 = (1\ 2)(3\ 4)(6\ 7)$, and $g_7 = (1\ 2)(3\ 4)(4\ 5)$. Determine whether there is a Hamilton cycle in G of the form

$$a_1 X a_2 Y a_3 X a_4 Y \cdots a_{840} Y$$

where $a_i \in \{g_5, g_7\}$ and $X = g_1 g_3 g_1 g_3 g_1$ and $Y = g_3 g_1 g_3 g_1 g_3$. This is a special form of an extent known as "Stedman Triples".

21. [**4**] Show that the undirected Cayley graph $Cay(\sigma, \tau : \mathbb{S}_n)$ is Hamiltonian. [**2**] A *doubly-adjacent* listing of permutations is a Hamilton cycle in Cayley graph with generators $(i\ i+1)$ for $1 \le i < n$ in which successive edges are either of the form $(j\ j+1), (j+1\ j+2)$ or the form $(j\ j+1), (j-1\ j)$. What is the relationship between the previous problem and that of finding a "doubly-adjacent" listing of $\mathbb{S}_n$?

Miscellaneous Questions

22. [**2**] How many Hamilton paths are there on a k by n grid graph, for $k = 2, 3$? [**3**] What about $k = 4$ (give a recurrence relation for the number)?

23. [**2+**] Show that the cube of any connected graph is Hamilton-*connected*.

24. [**1**] Find a small connected graph G for which G^2 is not Hamiltonian. There is a seven vertex example.

25. **[3]** Let G be an undirected graph. The *acyclic orientation graph*, $AO(G)$, has as its vertices the acyclic orientations of G and edges joining those vertices that differ by the reversal of the orientation of a single edge. Show that the square of $AO(G)$ is Hamiltonian. Show that the prism of $AO(G)$ is Hamiltonian.

26. **[*R*]** Prove or disprove the conjecture of Dillencourt and Smith [85] that aside from the stellated tetrahedron, any simplicial polyhedron (all faces triangles) with vertex degree at most 6, is 1-Hamiltonian. A graph is *1-Hamiltonian* if deletion of any vertex results in a Hamiltonian graph.

27. **[*R*−]** Are there sets $\mathbf{S}_n$ of binary strings of length n whose distance from each other in lex order and in least distance order are not proportional to each other? For example, if $\mathbf{S}_n$ consists of all binary strings of length n, then in lex order we have distance sum $2 \cdot 2^n$ and in Gray code order we have distance sum 2^n, so here the distance sums are proportional.

6.7 Bibliographic Remarks

An excellent survey of results on combinatorial Gray codes is given by Savage [371]. This survey mentions many of the topics that were covered in this chapter.

Hypercubes

A survey of results on hypercubes may be found in Harary, Hayes and Wu [170]. Hamilton path between vertices of opposite parity (Latifi and Zheng [240]). Ranking the symmetries and colorations of an n-cube is done in Fillmore and Williamson [128]. The number of (nonisomorphic) Hamilton cycles in Q_n is known only up to $n = 5$. For $1 \leq n \leq 5$ the numbers are 1,1,1,9,237675. Papers containing results about the number of Hamilton cycles are Douglas[89] and Smith[393], but determining the asymptotic number is far from resolved.

Papers about Gray codes whose graph of transitions are restricted include Bultena and Ruskey [45] and Wilmer and Ernst [456].

Cayley Graphs

The interesting connection between Hamiltonicity of Cayley graphs and campanology is explored in a series of papers by A.T. White [441], [442], [443], [444].

A good survey of Hamiltonicity results on Cayley graphs is given in Witte and Gallian [458] and its update by Curran and Gallian [78]. Other results about the Hamiltonicity of Cayley graphs may be found in Alspach [9], Marušič [267].

Recent results and references in the broader survey of Hamiltonicity results in general graphs may be found in Gould [158].

Kompel'makher and Liskovets [228] and Slater [391] showed that $Cay(T : \mathbb{S}_n)$ is Hamiltonian, where T is a tree of transpositions; Tchuente [411] showed that the graph is Hamilton-laceable.

Papers about the Hamiltonicity properties of Cayley graphs on vertex sets of interest to combinatorists include Conway, Sloane, and Wilks [69],

The Chen-Quimpo theorem is from [58].

Other Objects

Squire [400] determines necessary and sufficient conditions for there to be a Gray code of q-ary strings that have some given forbidden contiguous substring.

Ruskey [366] showed that the linear extensions of disjoint chains could be generated by transpositions; that they could be generated by adjacent transpositions was established by Stachowiak [401].

Gray codes for the acyclic orientations of a graph are discussed in Squire, Savage, and West [397], Squire [398], and in Pruesse and Ruskey [322].

The *basic word graph* of an antimatroid has vertices that are the basic words of an antimatroid and edges between those words that differ by a transposition of adjacent symbols in the words. The *feasible set graph* of an antimatroid has vertices that are the feasible sets of an antimatroid and edges between those sets that differ by a single element. Pruesse and Ruskey [320] show that the square of the basic word graph is Hamiltonian and that the square of the feasible set graph is also Hamiltonian.

The *shuffle-exchange* network SE_n is a cubic graph over Σ_2^n. A generic vertex $b_1 b_2 \cdots b_{n-1} b_n$ in SE_n is adjacent to $b_2 \cdots b_{n-1} b_n b_1$, $b_n b_1 b_2 \cdots b_{n-1}$, and $b_1 b_2 \cdots \overline{b_n}$. Feldman and Myslivietz [119] showed the existence of a Hamilton path in SE_n; Annexstein and Kuchko [10] showed how to rank this path.

Hamiltonicity and its variants

Karaganis [206] shows that the cube of a connected graph is Hamilton-connected. Hendry and Volger [176] show that the square of a connected $S(K_{1,3})$-free graphs with at least three vertices is vertex-pancyclic. Faudree and Schelp [118] showed that the square of a block is panconnected.

Graphs with combinations as vertices

Let T be a graph on vertex set $[n]$. Define $G_T(n,k)$ to be the graph whose vertex set consists of all k-subsets of $[n]$ and whose edges connect subsets that differ by a transposition of elements specified by an edge of T. If P is a path with vertices labeled sequentially, then $G_P(n,k)$ is the adjacent transposition graph considered in Section 5.3, and Section 5.3 gives a Hamilton path in $G_{P^2}(n,k)$. If the case that C is a cycle labelled sequentially, Enns [102], building on partial results of Joichi and White [199], gives the following necessary and sufficient conditions for $G_C(n,k)$ to contain a Hamilton cycle or a Hamilton path. A Hamilton circuit exists in $G_C(n,k)$ if and only if neither n and k are both even, nor $k = 2$ or $n-2$ for $k > 7$; a Hamilton path exists if and only if n and k are not both even. That $G_{K_n}(n,k)$ is Hamiltonian was demonstrated in Section 5.3 That $G_{K_n}(n,k)$ is Hamilton-connected was demonstrated by Jiang and Ruskey [195]

Other objects not explicitly covered in this chapter

Wang and Savage [438] show the existence of Gray codes of binary necklaces of fixed density. Udea [422] also shows the existence of Gray codes of binary fixed density necklaces and primitive necklaces; however, his representation is non-standard, necklaces are not represented by their lexicographically smallest rotation.

Chinburg, Savage, and Wilf [64] show ??? Bhat and Savage [31] show ???

Chapter 7

DeBruijn Cycles and Relatives

Dominoes is a simple game, enjoyed by children and adults alike. There are 28 domino pieces, each of which consists of an unordered pair of numbers taken from the set $\{0, 1, 2, 3, 4, 5, 6\}$ and printed or stamped as dots on a 1-by-2 piece of wood or plastic. Each of two players takes turns alternately placing dominoes, subject to the restriction that the new domino must abut a domino with the same number (there are various other rules that we ignore). A sample game which used all pieces is shown in Figure 7.1. As is customary, dominoes with identical numbers are placed perpendicular to the other dominoes. Now imagine a solitary player who simply wants to create a configuration like that shown in the figure. We consider a couple of questions. First, does it matter which pieces have already been placed? Can you blindly place the pieces and still know that a successful completion is assured? Secondly, note that we have created a kind of Gray code, in the sense that successive dominoes satisfy a natural closeness condition, but trying to model the problem as one of finding a Hamilton path in a graph with 28 vertices is not as straightforward as you might imagine. Give it a try!

Here's a more serious topic. The following circular bitstring has a rather curious property.

$$00011101 \tag{7.1}$$

Consider the set of all of its contiguous substrings of length three. There are eight such substrings and they are all distinct: 000, 001, 011, 111, 110, 101, 010, 100. In other words, they are all 8 bitstrings of length three. In general it is natural to wonder whether there is a circular k-ary string of length k^n with the property that all of its k^n length n contiguous substrings are distinct. Such circular strings do indeed exist and have come to be known as *De Bruijn Cycles*. De Bruijn cycles have found use in coding and communications, as pseudo-random number generators, in the theory of numbers, in digital fault testing, in the

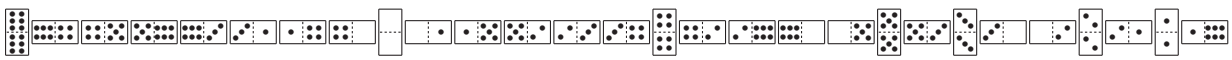


Figure 7.1: Example (maximal) domino game.

design of Sanskrit memory wheels[1], and by illusionists in mind-reading effects.

What ties these two topics together is that they are more naturally modelled as questions about Eulerian cycles in graphs rather than about Hamilton paths in graphs. One of our main aims in this chapter is to show that De Bruijn cycles always exist, and how to generate them quickly. Along the way we encounter other important combinatorial objects, such as necklaces, Lyndon words, and primitive polynomials over finite fields.

7.1 Eulerian Cycles

Certain combinatorial Gray code questions are more naturally posed as Eulerian cycle questions rather than as Hamiltonian cycle questions. Recall that an Eulerian cycle in a (multi)graph is a cycle that includes every edge exactly once. There is a simple characterization of Eulerian graphs, namely as given in Lemma 2.6: a connected (multi)graph is Eulerian if and only if every vertex has even degree.

Now back to the domino problem. Consider the complete graph K_7 with vertices labelled $0, 1, 2, 3, 4, 5, 6$ and with the self-loops $\{i, i\}$ added to each vertex i. There are 28 edges in this graph and each edge corresponds to a domino. An Eulerian cycle corresponds to a maximal domino game like that shown in Figure 7.1. Eulerian cycles clearly exist since each vertex has even degree 8.

7.1.1 Generating an Eulerian cycle

Let G be an directed Eulerian multigraph with n vertices. In this section we develop an algorithm that will generate an Eulerian cycle in in G. Along the way we will discover a nice formula for the number of Eulerian cycles and a CAT algorithm for generating all Eulerian cycles of G.

If G is a directed multigraph then $\overline{G}$ is the directed multigraph obtained by reversing the directions of the edges of G (i.e., for each edge (u, v) of G, there is a corresponding edge (v, u) in $\overline{G}$.

There is a simple algorithm for finding an Eulerian cycle in G given an in-tree rooted at some vertex r. We simply start at r and successively pick edges subject to the restriction that an edge of T is *not* used unless there is no other choice. A simple implementation of this rule is contained in Algorithm 7.1, and an example of its use is to be found in Figure 7.2.

At step (E1) we find a spanning out-tree T of $\overline{G}$ that is rooted at r. In general there may be several spanning out-trees rooted at r and it doesn't matter which one is used; depth-first search is a convenient and efficient way to find such a tree. The tree T is a spanning in-tree of G. We now modify (at line (E2)) the adjacency lists of G so that the unique edge $(u, v) \in T$ on the list for u is at the end of the list. We now say that the adjacency lists are *extreme* with respect to T. The remaining lines simply extract edges from the adjacency lists, destroying the lists in the process.

The running time of this algorithm is $O(m)$ where m is the number of edges in G. Since G is Eulerian, $m \geq n$. The depth-first search at line (E2) runs in time $O(n + m)$. The

[1]The nonsense Indian word yamátárájabhánasalagám is a mnemonic way of remembering the sequence (7.1), where an accented vowel represents a 1 and an unaccented vowel represents a 0. This word was used by medieval Indian poets and musicians as an aid in remembering all possible rythms. (There are 10 bits because the last two have been wrapped around.)

(E1) Use depth-first-search to find a spanning out-tree T of $\overline{G}$ rooted at r.
(E2) Compute adjacency lists `adj` of G that are extreme with respect to T.
(E3) $u := r$;
(E4) **while** $adj[u] \neq$ **null do**
(E5) $v := adj[u].vert$; $adj[u] := adj[u].next$;
(E6) Output((u, v));
(E7) $u := v$;

Algorithm 7.1: Algorithm to find an Eulerian cycle in a directed multigraph.

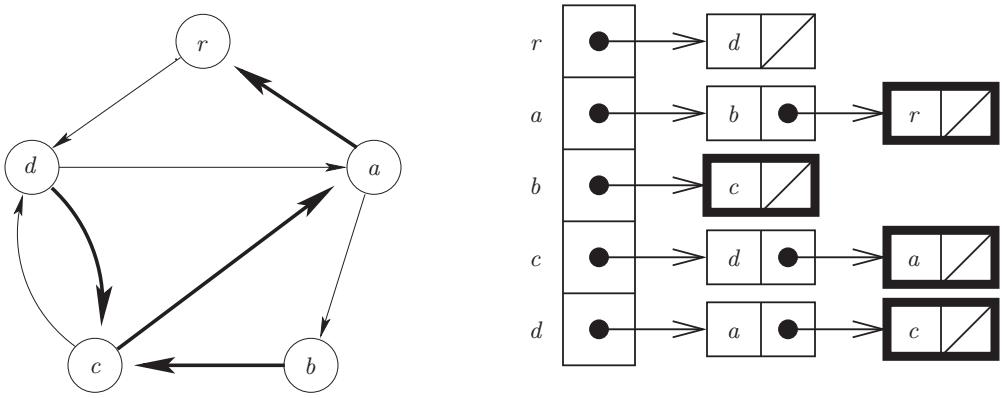


Figure 7.2: Example of finding an Euler cycle in a directed multigraph. (a) A directed multigraph G, with spanning in-tree found by depth-first search of $\overline{G}$ shown in bold, and (b) extreme adjacency lists with respect to T, with edges of T thicker. The cycle produced by this example is $r, d, a, b, c, d, c, a, r$.

computation of $\overline{G}$ takes time $O(m)$ the remaining computation (line (E3-E6)) takes time $O(m)$ since the body of the while loop at line (E3) is executed exactly m times.

THEOREM 7.1 *Given a directed Eulerian multigraph G, Algorithm 7.1 outputs a list of edges along an Eulerian cycle of G.*

PROOF: Since G is Eulerian, the path P produced by the algorithm must end at r. Imagine that there is some edge (v, w) that is not in P. Since the algorithm terminated it must be the case that $v \neq r$. Clearly, any edge on the adjacency list for v that follows (v, w) must also not be in P. Thus, because the edge lists are extreme with respect to T, we may assume that (v, w) is in T. Since $G - P$ is balanced, there is an edge (u, v) also not in P, which again we can take to be in T. Continuing in this manner we obtain a path of edges in $(G - P) \cap T$ that terminates at r. But then, since $G - P$ is balanced, it must contain an edge (r, q), contrary to the terminating condition of the algorithm. □

How many Eulerian does a connected, balanced multigraph G have? In answering this question we regard an Eulerian cycle as being a *circular* list of edges; the edge that starts the list is immaterial. The answer is provided by our algorithm. Clearly, different in-trees T produce different cycles, as do different adjacency lists that are extreme with respect to T. A graph G has $\tau(G)$ different spanning in-trees rooted at a given vertex r and there are $(d^+(v) - 1)!$ ways of arranging the adjacency list of v so that it is extreme with respect to T. Thus it is plausible that the number of Eulerian cycles in G is

$$\tau(G) \prod_{v \in V} (d^-(v) - 1)!. \tag{7.2}$$

To prove (7.2) we must show that we can recover the adjacency lists and tree T from an Eulerian cycle C. Fix an edge (r, s) to be the first on the cycle. Define the adjacency list for vertex v to simply be the edges of the form (v, w) in the order that they are encountered on C. With these adjacency lists lines (E3-E8) will produce the cycle C. To finish the proof we need to show that the the collection of edges

$$S = \{(v, v') \in E \mid v \neq r \text{ and } (v, v') \text{ is the last occurence of } v \text{ on } C\}$$

is an in-tree rooted at r. The set S contains $n - 1$ edges; we must show that it forms no cycles. Assume to the contrary that such a cycle X exists and let (y, z) be the first of its edges that occur on C, and let (x, y) be the previous edge on X. Unless $y = r$, there is some edge (y, z') that follows (x, y) on C, in contradiction to the way (y, z) was chosen. But we cannot have $y = r$ either since then $(y, z) = (r, z)$ would be in S.

How fast can we generate all Eulerian cycles in a graph? We need to generate permutations of edges on adjacency lists. There are many CAT algorithms for generating permutations (e.g., Algorithms ???, ???, and ???). We also need to generate spanning trees of a graph. This topic is taken up in Chapter ???. There are CAT algorithms for generating spanning trees of undirected graphs, but what about spanning in-trees of directed graphs? **!!! Is this a simple reduction or a research problem ???**

7.1.2 An Eulerian Cycle in the Directed n-cube

Given the central role played by hypercubes in the previous chapters, it is fitting that the next problem is one that can be modeled on the n-cube. We must admit, however, that it is

a rather distant relative of De Bruijn cycles. The solution of the following problem of Bate and Miller [22] is useful in circuit testing. Generate a cyclic sequence of $n2^n$ bitstrings of length n with the following three properties: (a) Each distinct bitstring must appear exactly n times; (b) each bitstring must differ by exactly one bit from the previous bitstring; (c) each possible *pair* of successive bitstrings must appear exactly once. An example of such a sequence for $n = 3$ is shown below.

000, 001, 011, 001, 101, 111, 101, 011, 000, 100, 110, 110,

101, 100, 000, 010, 110, 111, 011, 111, 110, 010, 011, 010,

The three occurences of 011 are underlined. Note that the three bitstrings which follow them are all distinct. Let $\vec{Q}_n$ be the *directed n-cube*; every edge of the n-cube Q_n is replaced by two directed edges, one in each direction. The following theorem is obviously true since any digraph for which the in-degree of each vertex is equal to its out-degree is Eulerian. Our interest in this theorem lies in its proof, which shows an explict construction of the Eulerian cycle without building the graph.

THEOREM 7.2 *There is an Eulerian cycle in the directed n-cube $\vec{Q}_n$ for all $n > 0$.*

PROOF: We argue by induction on n. The cycle will be expressed as a list of $n2^n + 1$ vertices, starting and ending at 0^n. The list for $n = 1$ is $0, 1, 0$.

Let $\mathcal{L}$ denote the list of bitstrings of the Eulerian cycle for $n - 1$. Produce from this a list $\mathcal{L}'$ obtained by doing the following steps in order.

1. Append a 0 to each bitstring of $\mathcal{L}$.

2. Replace the *first* occurence of a *non-zero* bitstring $X0$ by the 3 bitstrings $X0, X1, X0$.

3. Concatenate a second copy of $\mathcal{L}$ with a 1 appended to each bitstring.

4. Concatenate to the list a final bitstring of 0's.

It is easy to visualize what this algorithm is doing if $\vec{Q}_n$ is recursively thought of as two copies E_0 and E_1 of $\vec{Q}_{n-1}$, where one copy contains those vertices that end with a 0 and the other with those that end with a 1. The only other edges are the 2-cycles of the form $X0, X1$. It is these 2-cycles that get added to the Eulerian cycle by step 2; call the result E_0'. They are added the first time X is encountered in E_0, except if X is 0. The 2-cycle $\mathbf{0}0, \mathbf{0}1$ is used to join E_0' and E_1. □

For $n = 2$ the cycle produced is

$$00, 10, 11, 10, 00, 01, 11, 01, 00.$$

The table below shows the Eulerian cycle for $n = 3$.

$n=2$	$E_0'0$	$E_1 1$
00	000	001
10	100,101,100	101
11	110,111,110	111
10	100	101
00	000	001
01	010,011,010	011
11	110	111
01	010	011
00	000	001
		000

How can we develop and efficient algorithm based on this proof? The main difficulty comes from deciding which is the first occurence of a non-zero string. A bitstring $\mathbf{b}$ is uniquely identified in the list by the bit that changes, call it c, in obtaining the successor of $\mathbf{b}$, call it $\mathbf{b}'$. If, given $\mathbf{b}$ and c, we can specify c', the bit that changes in obtaining the successor of $\mathbf{b}'$, then we will be able to produce the entire list. Initially $\mathbf{b} = \mathbf{0} = 0^n$ and $c = 1$; these can also be used for the terminating conditions. Below we give rules for obtaining c' from $\mathbf{b}$ and c. These rules will be justified in the paragraphs to follow. In these rules x denotes a single bit (a "don't care"), $\mathbf{X}$ is a generic bitstring, and $\mathbf{0}$ is a string of 0's.

A. If $c = n$ and $\mathbf{b} = \mathbf{X}0$ then $c' = 1$ if $\mathbf{X} = \mathbf{0}$ and $c' = n$ if $\mathbf{X} \neq \mathbf{0}$.

B. If $c = n$ and $\mathbf{b} = \mathbf{X}x1$ then $c' = 1$ if $\mathbf{X} = \mathbf{0}$ and $c' = n - 1$ if $\mathbf{X} \neq \mathbf{0}$.

C. If $c = n - 1$ and $\mathbf{b} = \mathbf{0}1x$ then $c' = n$.

D. If $c = n - k$ $(k \geq 1)$ and $\mathbf{b}$ has more than k trailing 0's, then $c' = n$.

E. If none of the rules A-D above apply, then recursively apply them to the string consisting of the first $n - 1$ bits.

The last transition is from $0^{n-1}1$ to 0^n. Thus when the sequence E_0' or E_1 is completed, the next transition is in position n. So if $\mathbf{b} = \mathbf{0}1x$ and $c = n$, then c' should be n. This explains rule C.

Whenever bit n changes from 0 to 1 ($\mathbf{b} = \mathbf{X}0$ and $c = n$), then it changes back to 0 ($c' = n$) unless $\mathbf{X} = \mathbf{0}$. If $\mathbf{X} = \mathbf{0}$, then a new sequence on $n-1$ bits is starting and c' should be 1. This explains rule A.

In expanding E_0 to E_0' and a bitstring is generated for the first time, c' should be n. This happens whenever bit $n-1$ is about to change from a 0 to a 1 ($\mathbf{b} = \mathbf{X}00$ and $c = n-1$), or whenever bit $n-1$ is a 0 and a pattern on the first $n-2$ bits is about to appear for the first time. Recursively, this gives rise to the patterns $\mathbf{b} = \mathbf{X}000$ and $c = n-2$, and more generally to $\mathbf{b} = \mathbf{X}0^{k+1}$ and $c = n - k$. This explains rule D.

We now consider the case in which bit n changes from 1 to 0. This happens at the penultimate bitstring (i.e., when $\mathbf{b} = 0^{n-1}1$ and $c = n$). It also occurs when finishing up a two cycle; that is, when bit n has changed from 0 to 1 and is now immediately changing from 1 to 0. The sequence on $n-1$ bits must now be resumed. This implies that rule D was used, followed by rule A, and that the following bitstrings have just been generated.

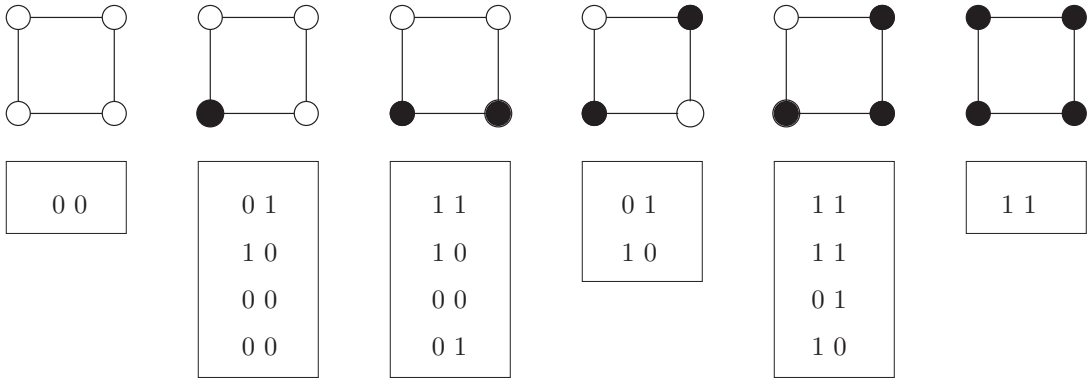


Figure 7.3: The six two-color necklaces with 4 beads. The necklace strings are shown in bold. Each equivalence class of strings under rotation is boxed.

b	c	
$\mathbf{X}0\ 0\ldots0\ 0$	$n-k$	
$\mathbf{X}1\ 0\ldots0\ 0$	n	
$\mathbf{X}1\ 0\ldots0\ 1$	n	current
$\mathbf{X}1\underbrace{0\ldots0}_{k-1}0$	?	$\mathbf{b}', c'$

The interrupted sequence on $n-1$ bits must now be resumed. The two lines where $c = n$ may be ignored, as may bit n. The first bitstring of the four should determin c'. This explains rule B.

Rules A-D cover all cases where $c = n$ or $c' = n$. If bit n is not involved, then we are generating the Eulerian cycle in $\vec{Q}_{n-1}$. This explains rule E.

A straighforward implemenation of these rules overcomes the exponential space obstacle, reducing it to $O(n)$. Time is also clearly $O(n)$ per bitstring produced. Unfortunately, the rules don't lend themselves to a CAT algorithm since about $n^2 2^{n-1}$ applications of the rules (including the recursive applications) are required to generate the Eulerian cycle.

However, a circuit may be designed, based on a refinement of these rules, which produces each bitstring in constant time from its predecessor, by operating in parallel, a similar circuit used for each position.

7.2 Necklaces

Mathematically, a necklace is usually defined as an equivalence class of strings under rotation. This definition is not exactly in accord with our intuition about what constitutes a real necklace, since we expect to be able to pick up a necklace and turn it over. However, we stick with the mathematical tradition and thus regard 001101 as being a different necklace than 001011, even though one may be obtained from the other by scanning backwards (and then rotating). Our principle goal in this section is to develop an efficient algorithm for generating necklaces.

Figure 7.3 shows the six two-color necklaces with 4 beads.

Recall that $\Sigma_k = \{0, 1, \ldots, k-1\}$, that Σ_k^n is the set of all k-ary strings of length n, that

Σ_k^* is the set of all k-ary strings, and that $\Sigma_k^+ = \Sigma_k^* \setminus \{\varepsilon\}$. Define an equivalence relation $\sim$ on Σ_k^* by $\alpha \sim \beta$ if and only if there exist $u, v \in \Sigma_k^+$ such that $\alpha = uv$ and $\beta = vu$. Instead of defining a *necklace* as an equivalence class we choose to define it as the lexicographically least representative of some equivalences class of the relation $\sim$. The set of all necklaces is denoted $\mathbf{N}$ and $\mathbf{N}_k(n)$ denotes the set of all necklaces of length n over a k-ary alphabet.

$$\mathbf{N}_k(n) \stackrel{\text{def}}{=} \{\alpha \in \Sigma_k^n \mid \alpha \leq \beta \text{ for all } \beta \sim \alpha\} \tag{7.3}$$

For example $\mathbf{N}_2(4) = \{0000, 0001, 0011, 0101, 0111, 1111\}$. The cardinality of $\mathbf{N}_k(n)$ is denoted $N_k(n)$.

Recall that a string α is *periodic* if $\alpha = \beta^k$ where β is non-empty and $k > 0$. If β is aperiodic, then it is called the *periodic reduction* of α. An aperiodic necklace is called a *Lyndon word*. The set of all Lyndon words is denoted $\mathbf{L}$ and $\mathbf{L}_k(n)$ denotes the set of all k-ary Lyndon words of length n.

$$\mathbf{L}_k(n) \stackrel{\text{def}}{=} \{\alpha \in \mathbf{N}_k(n) \mid \alpha \text{ is aperiodic}\}$$

For example, $\mathbf{L}_2(4) = \{0001, 0011, 0111\}$. The cardinality of $\mathbf{L}_k(n)$ is denoted $L_k(n)$.

A word α is called a *pre-necklace* if it is the prefix of some necklace. The set of all pre-necklaces is denoted $\mathbf{P}$ and the set of all k-ary pre-necklaces of length n is denoted $\mathbf{P}_k(n)$.

$$\mathbf{P}_k(n) \stackrel{\text{def}}{=} \{\alpha \in \Sigma_k^n \mid \alpha\beta \in \mathbf{N}(n+m, k) \text{ for some } m \geq 0 \text{ and } \beta \in \Sigma_k^m\}$$

For example $\mathbf{P}_2(4) = \mathbf{N}_2(4) \cup \{0010, 0110\}$. The cardinality of $\mathbf{P}_k(n)$ is denoted $P_k(n)$. We also define $W_k(n)$ to be the number of pre-necklaces of length at most n. These numbers will prove useful in analyzing the algorithms developed below. We define

$$W_k(n) \stackrel{\text{def}}{=} 1 + \sum_{i=1}^{n} P_k(i). \tag{7.4}$$

Let $\alpha = a_0 a_1 \cdots a_{n-1}$ be a string that can be written $\alpha = xy = yx$ where neither of x or y is empty. In other words, α is equal to one of its non-trivial circular shifts. Suppose $x = a_0 a_1 \cdots a_{m-1}$. Then $xy = yx$ implies the equation $a_i = a_{i+m}$, where index addition is taken modulo n. Iterating the equation we obtain

$$a_i = a_{i+jm} \quad \text{for all } 0 \leq i, j < n.$$

By Lemma 2.2 on page 16, $jm \pmod n$ for $0 \leq j < n$ gives us the multiples of m modulo n, so that

$$\{jm \pmod n : j = 0, 1, \ldots\} = \{0, d, 2d, \ldots, n-d\},$$

where $d = \gcd(m, n)$. We have now proven the following lemma.

LEMMA 7.1 *If $\alpha = xy = yx$ then*

$$\alpha = (a_0 a_1 \cdots a_{d-1})^{n/d}, \tag{7.5}$$

where $n = |\alpha|$ and $d = \gcd(|x|, n)$.

Note the following corollary.

COROLLARY 7.1 *If $\alpha = xy = yx$ with $x \neq \varepsilon$, $y \neq \varepsilon$, then α is periodic.*

We now count the objects under consideration. In the expressions below ϕ is the Euler totient function and μ is the Möbius function .

THEOREM 7.3 *The following formulae are valid for all $n \geq 1$, $k \geq 1$:*

$$L_k(n) \;=\; \frac{1}{n}\sum_{d\backslash n} \mu\left(\frac{n}{d}\right) k^d, \tag{7.6}$$

$$N_k(n) \;=\; \frac{1}{n}\sum_{j=1}^{n} k^{\gcd(j,n)} \;=\; \frac{1}{n}\sum_{d\backslash n} \phi(d) k^{n/d}, \tag{7.7}$$

$$P_k(n) \;=\; \sum_{i=1}^{n} L_k(i). \tag{7.8}$$

PROOF: Let $A_k(n)$ be the number of k-ary aperiodic strings of length n. Since every string can be expressed as an integral power of some aperiodic string, it follows that

$$k^n \;=\; \sum_{d\backslash n} A_k(d). \tag{7.9}$$

Now apply Möbius inversion (2.11) to obtain

$$A_k(n) \;=\; \sum_{d\backslash n} \mu(n/d) k^d.$$

Since every circular shift of an aperiodic string is distinct, $A_k(n) = n \cdot L_k(n)$, thereby proving (7.6). In the special case where p is a prime (7.6) or (7.7) imply "Fermat's Little Theorem": $k^{p-1} \equiv 1 \pmod{p}$.

To prove (7.7) we use Burnside's lemma . Necklaces are obtained by having the cyclic group $\mathbb{C}_n$ act on the set of k-ary strings. Let σ denote a left rotation by one position. Then the group elements are σ^j for $j = 0, 1, \ldots, n-1$. We need to determine the number of strings α for which $\sigma^j(\alpha) = \alpha$. By Lemma 7.1 this can occur only if $\alpha = \beta^t$ where $\beta \in \mathbf{L}$ and $|\beta| = \gcd(j, n)$. The number of such strings α is $k^{\gcd(j,n)}$. Thus the number of equivalence classes is $k^{\gcd(j,n)}$ summed over all $j = 1, 2, \ldots, n$ (noting that $\sigma^0 = \sigma^n$) divided by $|\mathbb{C}_n| = n$. The second equality follows from equation (2.10).

Equation (7.8) will be proven later. □

It is also worth noting that, since every necklace has the form β^t where $\beta \in \mathbf{L}$,

$$N_k(n) \;=\; \sum_{d\backslash n} L_k(d). \tag{7.10}$$

Equation (7.10) can also be used to prove (7.7); see exercise 5.

Below is a table of these numbers for the most important case, $k = 2$. Note that $2^n/n$ is sandwiched between $L_2(n)$ and $N_2(n)$, a property that holds for all values of n, and more generally, for all $k \geq 2$.

$k = 2$, $n = 6$

000000 N	000110	001101 L	011011 N
000001 L	000111 L	001110	011101
000010	001001 N	001111 L	011110
000011 L	001010	010101 N	011111 L
000100	001011 L	010110	111111 N
000101 L	001100	010111 L	

$k = 3$, $n = 4$

0000 N	0022 L	0122 L	1111 N
0001 L	0101 N	0202 N	1112 L
0002 L	0102 L	0210	1121
0010	0110	0211 L	1122 L
0011 L	0111 L	0212 L	1212 N
0012 L	0112 L	0220	1221
0020	0120	0221 L	1222 L
0021 L	0121 L	0222 L	2222 N

Figure 7.4: Output of the FKM algorithm (read down columns).

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$L_2(n)$	2	1	2	3	6	9	18	30	56	99	186	335	630	1161	2182
$\lceil 2^n/n \rceil$	2	2	3	4	7	11	19	32	57	103	187	342	631	1171	2185
$N_2(n)$	2	3	4	6	8	14	20	36	60	108	188	352	632	1182	2192
$P_2(n)$	2	3	5	8	14	23	41	71	127	226	412	747	1377	2538	4720
$W_2(n)$	3	6	11	19	33	56	97	168	295	521	933	1680	3057	5595	10315

A simple and elegant algorithm was proposed in Fredricksen and Maiorana [140] and Fredricksen and Kessler [139] to generate the sets $\mathbf{P}_k(n)$ and $\mathbf{N}_k(n)$. We will refer to this algorithm as the *FKM algorithm*.

For a given n and k, the FKM algorithm generates a list, $\mathbf{P}_k(n)$, in lexicographic order, where as usual we use the same notation for both the list and the set. The list $\mathbf{P}_k(n)$ begins with the string 0^n and ends with $(k-1)^n$. For a given α in $\mathbf{P}_k(n)$, the successor of α, $succ(\alpha)$, is obtained from $\alpha = a_1 a_2 \cdots a_n$ as follows.

DEFINITION 7.1 *For $\alpha < (k-1)^n$, $\mathrm{succ}(\alpha) = (a_1 a_2 \cdots a_{i-1}(a_i+1))^t a_1 \cdots a_j$, where i is the largest integer $1 \le i \le n$ such that $a_i < k-1$ and t, j are such that $ti + j = n$ and $j < i$.*

We also define the predecessor function *pred*, where if $succ(\alpha) = \beta$, then $pred(\beta) = \alpha$.

It is shown in [140] that the successor function *succ* sequences through the elements of $\mathbf{P}_k(n)$ in lexicographic order and that $succ(\alpha)$ is a necklace if and only if the i of Definition 7.1 is a divisor of n. We will prove these assertions later in this section.

Figure 7.4 shows the output of the FKM algorithm for $n = 6$, $k = 2$, and $n = 4$, $k = 3$. Lyndon words are followed by an "L"; periodic necklaces are followed by a "N" and the periodic reduction of each necklace is underlined. An iterative implementation is given as Algorithm 7.2.

For $\alpha \in \Sigma_k^*$, let $lyn(\alpha)$ be the length of the longest prefix of α that is a Lyndon word.

```
for j := 0 to n do a_j := 0;
Printlt( 1 );
i := n;
repeat
    a_i := a_i + 1;
    for j := 1 to n - i do a[j + i] := a[j];
    Printlt( i );
    i := n;
    while a_i = k - 1 do i := i - 1;
until i = 0;
```

Algorithm 7.2: The original iterative FKM Algorithm (note: a[0] = 0.)

This function is well-defined since $\Sigma_k \subseteq \mathbf{L}$. More formally,

$$lyn(a_1 a_2 \cdots a_n) \stackrel{\text{def}}{=} \max\{1 \le p \le n \mid a_1 a_2 \cdots a_p \in \mathbf{L}_k(n)\}. \tag{7.11}$$

The next theorem provides useful characterizations of necklaces and Lyndon words.

THEOREM 7.4 *The following conditions characterize the sets* $\mathbf{N}$ *and* $\mathbf{L}$*.*

$\alpha = xy \in \mathbf{N}$ *if and only if* $xy \le yx$*, for all* x, y*.*

$\alpha = xy \in \mathbf{L}$ *if and only if* $xy < yx$*, for all non-empty* x, y*.* (7.12)

PROOF: [$\mathbf{N}_k(n)$] This is just a restatement of the definition (7.3).

[$\mathbf{L}_k(n)$] Suppose $\alpha = xy = yx$ with $xy \ge yx$ and $x \ne \varepsilon$ and $y \ne \varepsilon$. If $xy > yx$ then $\alpha \notin \mathbf{N}_k(n)$, so $\alpha \notin \mathbf{L}_k(n)$. If $xy = yx$ then by Corollary 7.1, α is periodic, so $\alpha \notin \mathbf{L}_k(n)$. Conversely, if $\alpha \in \mathbf{N}_k(n)$ is periodic, say $\alpha = \beta^t$ with $t > 1$, then $\alpha = xy = yx$ where $x = \beta$ and $y = \beta^{t-1}$. □

LEMMA 7.2 *If* $\alpha \in \mathbf{N}$*, then* $\alpha^t \in \mathbf{N}$ *for* $t \ge 1$*.*

PROOF: For $t > 1$, if $\alpha^t = xy$, then yx has the form $\gamma\alpha^{t-1}\delta$ where $\alpha = \delta\gamma$. Since α is a necklace $\delta\gamma \le \gamma\delta$. Thus

$$\alpha^t = (\delta\gamma)^t \le (\gamma\delta)^t = \gamma\alpha^{t-1}\delta,$$

and so α^t must also be a necklace. □

LEMMA 7.3 *If* $\alpha \in \mathbf{L}$ *and* $\alpha = \beta\gamma$ *for some* β *and* γ *with* γ *non-empty, then for any* $t \ge 1$

(a) $\alpha^t\beta \in \mathbf{P}$*, and*

(b) $\alpha^t\beta \in \mathbf{N}$ *if and only if* $|\beta| = 0$*.*

PROOF: (a) Since α is a necklace, both α^t and α^{t+1} are necklaces by the Lemma 7.2. Thus $\alpha^t\beta$ is a pre-necklace and is a necklace if $\beta = \varepsilon$. If $\beta \ne \varepsilon$ then $\alpha = \beta\gamma < \gamma\beta$, since $\alpha \in \mathbf{L}$. Therefore,

$$\alpha^t\beta = (\beta\gamma)^t\beta = \beta(\gamma\beta)^t > \beta(\beta\gamma)^t = \beta\alpha^t,$$

so that $\alpha^t\beta$ is not a necklace. □

LEMMA 7.4 *Let $\alpha = a_1 a_2 \cdots a_n$ be a string and $p = lyn(\alpha)$. Then $\alpha \in \mathbf{P}$ if and only if $a_{j-p} = a_j$ for $j = p+1, \ldots, n$.*

PROOF: If $a_{j-p} = a_j$ for $j = p+1, \ldots, n$, then $\alpha = \beta^t \delta$ for some $t \geq 1$ where $\beta = a_1 a_2 \cdots a_p \in \mathbf{L}$ and δ is a prefix of β. Thus $\alpha \in \mathbf{P}$ by Lemma 7.3.

Conversely, assume that $\alpha \in \mathbf{P}$, and let j be the smallest index $j > p$ such that $a_{j-p} \neq a_j$. We will derive separate contradictions, depending whether $a_{j-p} < a_j$ or $a_{j-p} > a_j$. By the definition of j, the string α has the form $\beta^t \delta a_j \cdots a_n$ for some $t \geq 1$ where $\beta = a_1 a_2 \cdots a_p$ and δ is a proper prefix of β. For some γ, $\beta = \delta\gamma$ where the first symbol of γ is a_{j-p}.

Since $\alpha \in \mathbf{P}$ there is a string ω such that $\alpha\omega \in \mathbf{N}$. If $a_{j-p} < a_j$, then

$$\beta^{t-1} \delta a_j \cdots a_n \omega \delta\gamma \ < \ \beta^{t-1} \delta\gamma \delta a_j \cdots a_n \omega,$$

which implies that $\alpha\omega$ is not a necklace, a contradiction.

If $a_{j-p} < a_j$ then consider the string $\rho = \beta^t \delta a_j$; we will use (7.12) to show that $\rho \in \mathbf{L}$, in contradiction to the definition of $p = lyn(\alpha)$. Write $\rho = xy$ with $x \neq \varepsilon$ and $y \neq \varepsilon$. We consider two cases: (a) β^t is a prefix of x and (b) x is a prefix of β^t.

[Case (a)] Here $x = \beta^t u$ and $y = v a_j$ where $uv = \delta$. Since uv is a proper prefix of β there is a $w \neq \varepsilon$ such that $\beta = uvw$. Note that the first symbol of w is a_{j-p}. By (7.12), $uvw < vwu < va_j$. Thus

$$xy = uvw\beta^{t-1}\delta a_j < va_j\beta^t u = yx.$$

[Case (b)] Here $x = \beta^p u$ and $y = v\beta^q \delta a_j$ where $uv = \beta$ and $t = 1 + p + q$. If $u = \varepsilon$ or $v = \varepsilon$, then we may assume without loss of generality that $p \geq 1$ and $u = \varepsilon$. Thus

$$xy = \beta^p \beta^q \delta a_j < \beta^q \delta a_j \beta^p = yx,$$

a contradiction. Thus we may assume that neither of u or v is empty and that $\beta = uv < vu$ since $\beta \in \mathbf{L}$. Now if $q = 0$, then since $uv < vu < va_j$,

$$xy = uv\beta^p \delta a_j < v\delta a_j \beta^p u = yx,$$

again a contradiction. The only remaining case is $q > 0$. But then

$$xy = \beta^p uv\beta^q \delta a_j = u(vu)^p (vu)^q v\delta a_j < v(uv)^q \delta a_j \beta^p u = v\beta^q \delta a_j \beta^p u = yx,$$

a contradiction. □

The following theorem leads to a recursive version of the FKM algorithm. It's proof is inherent in the proof of the previous lemma. This theorem is very useful. We are tempted to call it the "Fundamental Theorem of Necklaces"!

THEOREM 7.5 *Let $\alpha = a_1 a_2 \cdots a_{n-1} \in \mathbf{P}_k(n-1)$ and $p = lyn(\alpha)$. The string $\alpha b \in \mathbf{P}_k(n)$ if and only if $a_{n-p} \leq b \leq k-1$. Furthermore,*

$$lyn(\alpha b) \ = \ \begin{cases} p & \text{if } b = a_{n-p} \\ n & \text{if } a_{n-p} < b \leq k-1. \end{cases}$$

```
procedure gen( t,p : ℕ );
local j : ℕ;
begin
    if t > n then PrintIt( p )
    else
        a[t] := a[t − p]; gen( t + 1, p );
        for j := a[t − p] + 1 to k − 1 do
            a[t] := j; gen( t + 1, t );
end {of gen};
```

Algorithm 7.3: Recursive FKM Algorithm (note: a[0] = 0.)

Algorithm 7.3 is the recursive FKM algorithm. It follows directly from the Theorem 7.5. The initial call is `gen(1,1)`. We assume, as for the iterative FKM algorithm, that $a_0 = 0$. Various types of objects may be produced, depending on `PrintIt(p)`, as shown in the table below.

Sequence type	`PrintIt(p)`
Pre-necklaces ($\mathbf{P}_k(n)$)	Println($a[1..n]$)
Lyndon words ($\mathbf{L}_k(n)$)	**if** $p = n$ **then** Println($a[1..n]$)
Necklaces ($\mathbf{N}_k(n)$)	**if** $n \bmod p = 0$ **then** Println($a[1..n]$)
De Bruijn sequence	**if** $n \bmod p = 0$ **then** Print($a[1..p]$)

The call "Println($a[1..n]$)" prints the array $a[1], a[2], \ldots, a[n]$ on a separate line. Each time `PrintIt` is called a new prenecklace is produced. Since the parameter p to `PrintIt` is the value of $lyn(a[1..n])$, a Lyndon word is produced exactly when $p = n$. By part (b) of Lemma 7.3 a necklace is produced whenever p divides n. De Bruijn cycles will be discussed in the following section.

We can now also understand why Algorithm 7.2, the iterative FKM algorithm, is correct. Consider again the successor function *succ*. It is clear that the rightmost position i that can change is the largest index i for which $a_i < k - 1$. We then increment a_i. What was not so clear was how the remainder of the sequence was to be completed in the lexicographically smallest way. Theorem 7.5 provided the answer.

How fast is the FKM algorithm? We analyze the recursive version; the same conclusions hold for the iterative version. Call the number of nodes in the computation tree $W_k(n)$. From the structure of the algorithm, $W_k(n)$ is equal to the number of prenecklaces of length at most n, as expressed in (7.4).

From the expressions (7.6) and (7.7) we obtain the following bounds.

$$\frac{1}{n}(k^n - (n-1)k^{n/2}) \leq L_k(n) \leq \frac{1}{n}k^n \leq N_k(n) \leq \frac{1}{n}(k^n + (n-1)k^{n/2}) \tag{7.13}$$

We have

$$P_k(n) = \sum_{i=1}^{n} L_k(i) \leq \sum_{i=1}^{n} \frac{1}{i}k^i, \tag{7.14}$$

The equality in (7.14) follows because every prenecklace is obtained as a prefix of β^*, where β is some Lyndon word. Hence

$$W_k(n) - 1 = \sum_{j=1}^{n} P_k(i) \leq \sum_{j=1}^{n} \sum_{i=1}^{j} \frac{1}{i}k^i$$

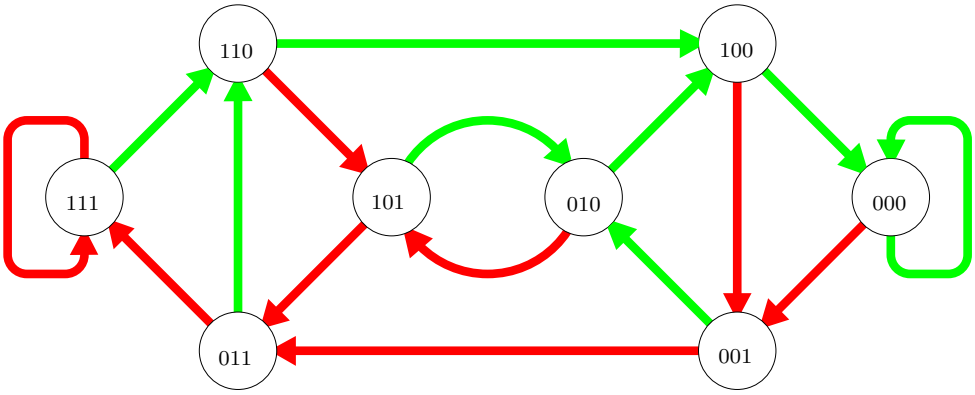


Figure 7.5: The De Bruijn graph for $k = 2$ and $n = 4$.

Thus

$$\frac{W_k(n) - 1}{N_k(n)} \leq \frac{n}{k^n} \sum_{j=1}^{n} \sum_{i=1}^{j} \frac{1}{i} k^i$$

In [357] it is shown that this last expression converges to $(k/(k-1))^2$ as $n \to \infty$. Thus the asymptotic running time per necklace (or per Lyndon word) is constant; the necklace Algorithms 7.2 and 7.3 are both CAT.

7.3 De Bruijn Sequences

An example of a De Bruijn sequence was given at the beginning of this chapter. All De Bruijn sequences arise as Eulerian cycles in a certain graph which we introduce below. The FKM algorithm can be used to generate the lexicographically least Eulerian cycle.

The *De Bruijn graph* , $G_k(n)$ has vertex set consisting of all k-ary strings of length $n-1$; i.e., it is Σ_k^{n-1}. There is a directed edge, labelled b, from $d_1 d_2 \cdots d_{n-1}$ to $d_2 \cdots d_{n-1} b$ for each $b \in \Sigma_k$. Thus the out-degree of each vertex is k and the number of edges is k^n. Figure 7.5 shows $G_2(4)$. An Eulerian cycle in $G_k(n)$ is called a *De Bruijn cycle*. More precisely, the sequence of k^n edge labels is a De Bruijn cycle. A De Bruijn cycle is characterized by the property that each element of Σ_k^n occurs exactly once as a substring on the cycle.

There is a very simple modification of the FKM algorithm that will produce a De Bruijn sequence, as indicated by the appropriate line in the table in the previous section. The idea is to successively concatenate the reduction of each necklace as it is produced by the FKM algorithm. Thus we are concatenating all Lyndon words of length divisible by n in lexicographic order. At first glace it appears most amazing that this algorithm should work. Let's first look at the sequences it produces for the examples given in Figure 7.4. Here's the resulting De Bruijn cycle for $n = 6$ and $k = 2$.

0 000001 000011 000101 000111 001 001011 001101 001111 01 010111 011 011111 1

Here's the resulting cycle for $n = 4$ and $k = 3$.

0 0001 0002 0011 0012 0021 0022 01 0102 0111 0112
0121 0122 02 0211 0212 0221 0222 1 1112 1122 12 1222 2

These are, in fact, the lexicographically smallest De Bruijn cycles.

We now prove that the algorithm is correct. The algorithm is so elegant that one would hope the same of the proof. Unfortunately, it is a rather uninspiring case analysis. To understand the difficulties involved the reader should, before reading the proof, try to figure out the location of a few specific strings in the output of the algorithm. For example, in the list for $k = 3$ and $n = 9$, where does 220121012 appear? For $k = 3$ and $n = 8$, where does 22012201 appear?

THEOREM 7.6 *The list of successive periodic reductions of necklaces as produced by the FKM algorithm forms a De Bruijn cycle.*

PROOF: Let D denote the list of k-ary symbols produced by concatenating the successive periodic reductions of necklaces from the FKM algorithm. Since (7.9) may be written as

$$k^n = \sum_{d \backslash n} d \cdot L_k(d),$$

the list D contains the correct number of digits. We now argue that each k-ary string α of length n occurs as a substring of D, which will finish the proof.

Note that the first two outputs of the algorithms are 0 and $0^{n-1}1$, and the last two outputs are $(k-2)(k-1)^{n-1}$ and $(k-1)$. Thus D has a prefix 0^n and a suffix $(k-1)^n$, from which it follows that all strings of the form $(k-1)^p 0^{n-p}$ (where $0 \leq p \leq n$) occur as substrings in D. All other strings have the form

$$\alpha = (k-1)^p (xy)^t$$

where $0 \leq p < n$, $t \geq 1$, $yx \in \mathbf{L}$, xy contains some non-zero symbol, and the first symbol of x is not $k-1$. We will classify the possibilities for α according to whether $p = 0$, whether $t = 1$, and whether $y = \varepsilon$.

Case 1 [$p = 0$, $t = 1$, $y = \varepsilon$]: Here $\alpha = x$ with $x \in \mathbf{L}$. Trivially x appears as substring in D since x is output by the algorithm.

Case 2 [$p = 0$, $t > 1$, $y = \varepsilon$]: Here $\alpha = x^t$ with $x \in \mathbf{L}$. The string x is output by the algorithm, and the next string output by the algorithm is $x^{t-1}S(x)$, where $S(x)$ is the *necklace* the lexicographically follows x. Thus the string x^t occurs in D since $xx^{t-1}S(x)$ is a substring of D.

Case 3 [$p = 0$, $t = 1$, $y \neq \varepsilon$]: Here $\alpha = xy$ and $yx \in \mathbf{L}$. The string yx is output by the algorithm; what output follows it? Note that $succ(yx) = yz$ for some string z since x is not all $(k-1)$'s (and $y(k-1)^{n-|y|} \in \mathbf{L}$). What is the periodic reduction of yz? We claim that it is a string with prefix y, which will finish this case. Observe from Theorem 7.5 that *if $\beta \in \mathbf{L}$, then β^r is the lexicographically smallest necklace of length $r|\beta|$ with prefix β*. Thus it cannot be that $yz = \beta^r$ and β is a proper substring of y, since yx is a lexicographically smaller necklace.

Case 4 [$p = 0$, $t > 1$, $y \neq \varepsilon$]: Here $\alpha = (xy)^t$ and $yx \in \mathbf{L}$. The string yx is output by the algorithm, and the next string output by the algorithm is $S((yx)^t) = (yx)^{t-1}yS(x)$, which is aperiodic. Thus D contains the substring $yx(yx)^{t-1}yS(x)$; a string which contains α as substring.

Case 5 [$p > 0$, $t = 1$, $y = \varepsilon$]: Here $\alpha = (k-1)^p x$ with $x \in \mathbf{L}$. The string D contains $\beta = pred(x)(k-1)^p$ since $\beta \in \mathbf{L}$. The next string output by the algorithm has prefix x, and so D contains the substring $pred(x)(k-1)^p x$.

Case 6 [$p > 0$, $t > 1$, $y = \varepsilon$]: Here $\alpha = (k-1)^p x^t$ with $x \in \mathbf{L}$. The string D contains $\beta = pred(x)(k-1)^{n-d}$ where $d = |x|$, since $\beta \in \mathbf{L}$. Let $n = md + r$ where $0 \leq r < m$. If $r > 0$ (i.e., d does not divide n), then $succ(\beta) = x^m S(y) \in \mathbf{L}$ where y is the string consisting of the first r symbols of x. Since $m \geq t$, the string α occurs on D. If $r = 0$, then following β the algorithm outputs x, followed by $x^{m-1}S(x)$. Thus D contains the string $pred(x)(k-1)^{n-d}xx^{m-1}S(x)$ which in turn contains α.

Case 7 [$p > 0$, $t = 1$, $y \neq \varepsilon$]: Here $\alpha = (k-1)^p xy$ with $p \geq 1$ and $yx \in \mathbf{L}$. Note that $\gamma = neck(\alpha)$ is either (a) $\gamma = xy(k-1)^p$ or (b) $\gamma = y(k-1)^p x$. In case (a) $\gamma \in \mathbf{L}$. We may now proceed exactly as in Case 5, with xy playing the role of x. That is to say, D will contain the string $pred(xy)(k-1)^p xy$. In case (b), if γ is aperiodic, then γ is output by the algorithm and the following string output is of the form $y(k-1)^p z$, so α appears on D. In case (b) if γ is periodic, then $y(k-1)^p$ must be the periodic reduction, where, say, $\gamma = [y(k-1)^p]^q$. The algorithm outputs $pred(y)(k-1)^{n-|y|}$, followed by $y(k-1)^p$, followed by $[y(k-1)^{q-1}S(y(k-1)^p)$, and thus D contains α.

Case 8 [$p > 0$, $t > 1$, $y \neq \varepsilon$]: Here $\alpha = (k-1)^p (xy)^t$ with $p \geq 1$, $t > 1$, and $yx \in \mathbf{L}$. Note that $\gamma = neck(\alpha)$ must be $\gamma = y(xy)^{t-1}(k-1)^p x \in \mathbf{L}$. The next string output by the algorithm is $y(xy)^{t-1}(k-1)^p S(x)$, so again α occurs as a substring of D. □

From our previous analysis of the FKM algorithm we know that the total amount of work, aside from outputting the symbols, in producing this cycle is $\Theta(k^n/n)$. Thus the time required to produce the De Bruijn cycle is dominated by the time to output the digits; i.e., it is $\Theta(k^n)$ which is best possible.

7.4 Computing the Necklace of a String

Given a string, it is often useful to compute its necklace. Such applications arise in several diverse areas such as graph drawing, where it is used to help determine the symmetries of a graph to be drawn in the plane.

Recall that for any string $\alpha = a_1 a_2 \ldots a_N \in \Sigma_k^N$ its necklace, $neck(\alpha)$ is the lexicographically smallest of its circular shifts. The question naturally arises as how to efficiently compute the necklace given the string. In this section we present an $O(N)$ algorithm for the task.

First we consider the problem of factoring a word as specified in the following theorem.

THEOREM 7.7 (CHEN, FOX, LYNDON) *Any word $\alpha \in \Sigma_k^+$ admits a unique factorization*

$$\alpha = \alpha_1 \alpha_2 \cdots \alpha_m,$$

such that $\alpha_i \in \mathbf{L}$ for $i = 1, 2, \ldots, m$ and

$$\alpha_1 \geq \alpha_2 \geq \cdots \geq \alpha_m.$$

Here are two examples of Lyndon factorizations.

011 011 00111 0 0 and 0102 0102 01 0 0

It is easy to see that such a factorization exists, since each letter is a Lyndon word and any two Lyndon words x and y for which $x < y$ can be concatenated to get another Lyndon word xy. Uniqueness is also not hard to show. See Exercises 12 and 13.

There is also a version of this theorem that deals with factorizations into necklaces.

THEOREM 7.8 *Any word $\alpha \in \Sigma_k^+$ admits a unique factorization*

$$\alpha = \alpha_1 \alpha_2 \cdots \alpha_m,$$

such that $\alpha_i \in \mathbf{N}$ for $i = 1, 2, \ldots, m$ and

$$\alpha_1 > \alpha_2 > \cdots > \alpha_m.$$

Here are two examples of necklace factorizations, using the same words as above.

011011 00111 00 and 01020102 01 00

Duval [92] has developed an elegant, efficient algorithm for factoring a word. Our version of the algorithm is essentially based on Theorem 7.5. The output of the algorithm consists of indices $0 = k_0, k_1, k_2, \ldots, k_m = N$ such that

$$\alpha_i = a_{k_{i-1}+1}, \ldots, a_{k_i}$$

Informally, the idea of the algorithm is to keep extending n and updating p until a value $a_n > a_{n-p}$ is encountered. Then $a_1 \cdots a_p$ is the longest prefix of α that is in $\mathbf{L}$ and $\beta^t \gamma = a_1 \cdots a_{n-1}$ where $pt + |\gamma| = n - 1$ with $|\gamma| < p$. The words

$$\underbrace{\beta, \beta, \ldots, \beta}_{t}, \gamma, \text{ causing output } k_i = i|\beta| \text{ for } i = 1, 2, \ldots, t$$

are the first t factors in the Lyndon factorization of α. Note that γ is the prefix of a Lyndon word that is lexicographically less than β. Now apply the same algorithm to γ and the remainder of α; i.e., to $\gamma a_n a_{n+1} \cdots a_N$. The details may be found in Algorithm 7.4.

```
D       k := 0; a[N + 1] := −1;
        while k < N do
            n := k + 2; p := 1;
            while a[n − p] ≤ a[n] do
                if a[n − p] < a[n] then p := n − k;
                n := n + 1;
            repeat PrintIt( k ); k := k + p;
            until k ≥ n − p;
        PrintIt( N );
```

Algorithm 7.4: Duval algorithm for factoring a string.

By moving the `PrintIt(k)` statement at line `(D8)` outside and just before the repeat loop (i.e., between lines `(D7)` and `(D8)`), the algorithm will produce the necklace factorization.

What is the running time of Duval's algorithm? Note that the comparison $a_{n-p} = a_n$ at line `(D4)` is the most often executed statement. Let us assume that we are doing a necklace factorization and that there are m factors. Let k_i, n_i, p_i be the values of k, n, and p at the end of the ith iteration of the outer while loop. The total number of comparisons done at line `(D4)` on the i-th iteration is $n_i - k_{i-1} - 1$. But by `(D9)`, $n_i - k_i \leq p_i$ so that

$$\begin{aligned} \sum_{i=1}^{m} (n_i - k_{i-1} - 1) &= k_m + \sum_{i=1}^{m} (n_i - k_i - 1) \\ &\leq N - m + \sum_{i=1}^{m} p_i \\ &\leq 2N - m \end{aligned}$$

Thus the total number of comparisons done at line (D4) is at most $2N$ and the running time of the algorithm is therefore $O(N)$.

How to find a necklace

To find the necklace of a word α, use $\alpha\alpha$ as the input to the necklace factorization algorithm (or modify the algorithm to do arithmetic mod n). Suppose that β^t is the necklace of α, where $\beta \in \mathbf{L}$. Then β^t will occur as t factors in the Lyndon factorization of $\alpha\alpha$. We need simply wait until a necklace factor of length $|\alpha|$ appears. When $n - k > N$ (tested after line (D9)), the string $a[k+1..k+N]$ is the necklace of α.

7.5 Universal Cycles

In this section we give a brief introduction to "Universal Cycles". These are a very interesting generalization of DeBruijn cycles introduced by Chung, Diaconis and Graham [65]. Many types of combinatorial objects are represented by strings of fixed length; suppose that there are N total objects and each representation has length n. The problem is to find a (circular) string D of length N such that every representative occurs exactly once as a contiguous substring of D. For De Bruijn cycles we had $N = 2^n$ and the representations were all bitstrings of length n. But what about combinations, permutations, set and numerical partitions, etc?

k-permutations of an n-set

Suppose that we try to extend the De Bruijn cycle idea from subsets (bitstings of length n) to k-permutations of an $[n]$. That is, we would like a circular string of length $(n)_k$ over the alphabet $[n]$ such that each k-permutation occurs exactly once as a substring. It is natural to define a digraph $G(n, k+1)$ whose vertices are $([n])_k$, the k-permutations of $[n]$, and where a generic vertex $a_1a_2\cdots a_k$ has $n-k+1$ outgoing edges whose endpoints are $a_2\cdots a_kb$ for $b \in [n] \setminus \{a_2, \cdots, a_k\}$. For example if $k = 1$ then $G(n, 2)$ is the complete directed graph; i.e., there is an edge between every pair of distinct vertices. Note that $G(n, k)$ is vertex-transitive.

One Eulerian cycle in $G(4, 2)$ is shown below.

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

LEMMA 7.5 *For $1 \le k < n$, the graph $G(n, k)$ is Eulerian.*

PROOF: It is easy to see that the in-degree of each vertex is also $n - k + 1$. We need only show that $G(n, k)$ is strongly connected. Given $\mathbf{a} = a_1a_2\cdots a_k \in ([n])_k$, note that there is a cycle of length k in $G(n, k)$ that contains every circular permutation of $\mathbf{a}$, namely the one obtained by repeatedly using edges of the form $x\alpha \to \alpha x$. We now show that there is a path from $\mathbf{a}$ to $\mathbf{a}$ with any two of its adjacent elements transposed. To do this, we need only show the existence of a path from $\mathbf{a}$ to $a_2a_1a_3\cdots a_k$. Let $x \notin \{a_1, a_2, \ldots, a_k\}$. Then there is a path

$$a_1a_2a_3\cdots a_k \to a_2a_3\cdots a_kx \to a_3\cdots a_kxa_1 \stackrel{*}{\to}$$
$$xa_1a_3\cdots a_k \to a_1a_3\cdots a_ka_2 \stackrel{*}{\to} a_2a_1a_3\cdots a_k$$

as claimed, where $\stackrel{*}{\rightarrow}$ denotes a path of some length. This implies that there is a path from $\mathbf{a}$ to $\mathbf{b} = b_1 b_2 \cdots b_k$ where $b_1 < b_2 < \cdots < b_k$ and $\{b_1, b_2, \ldots, b_k\} = \{a_1, a_2, \ldots, a_k\}$. We now show that there is a path from $\mathbf{b}$ to $12 \cdots k$. Since $G(n,k)$ is vertex-transitive, that will finish the proof. Let i be the smallest value for which $b_i \neq i$ and $b_{i+1} > i+1$; if there is no such value, then we are done. The following path is in $G(n,k)$.

$$b_1 b_2 \cdots b_k \stackrel{*}{\rightarrow} b_{i+1} \cdots b_k 1 \cdots i \rightarrow b_{i+2} \cdots b_k 1 \cdots i(i+1) \stackrel{*}{\rightarrow} 12 \cdots i(i+1) b_{i+2} \cdots b_k.$$

Continuing in this manner we eventually reach $12 \cdots k$. □

It would be interesting to develop a fast algorithm to output an Eulerian cycle whose existence is guaranteed by the lemma.

If $n = k$ then $G(n,k)$ consists of $n!/2$ 2-cycles of the form $\pi_1 \pi_2 \cdots \pi_n \rightleftharpoons \pi_2 \cdots \pi_n \pi_1$, and is thus not Eulerian. Another way to view this is as follows. Consider $n = 3$. The substring 123 must occur, but what symbol follows the 3? It must be 1. And then 2 and then 3. But 123123 does not satisfy our criteria since, for example, 321 does not occur as a substring. The problem occurs because we have insisted on $[n]$ as our alphabet. Consider the string

1 4 5 2 4 3

This contains the substring 431 which we consider to represent the permutation 321 (which we missed before). Below are the six substrings of length 3 and the corresponding permutations of [3].

substring	145	452	524	243	431	314
permutation	123	231	312	132	321	213

More formally, we say that permutations of natural numbers, π and π', both of length n, are *order isomorphic* if $\pi_i < \pi_j$ if and only if $\pi'_i < \pi'_j$ for all $1 \leq i, j \leq n$. Our problem is to find a string of numbers whose substrings are order isomorphic to the $n!$ permutations of $[n]$.

Here's a string that works for $n = 4$.

1 2 3 4 1 2 5 3 4 1 5 3 2 1 4 5 3 2 4 1 3 2 5 4

How could we construct such a string? Do they always exist? How many extra symbols must be used? Taking our clue from De Bruijn sequences we define a graph and look for Eulerian cycles in that graph. **!!! need to put in the rest of the stuff from the CDG paper !!!**.

Let $N(n)$ be the number of extra symbols that are necessary to construct a U-cycle for permutations. Chung, Graham and Diaconis conjecture that $N(n) = n + 1$; that $n + 1 \leq N(n) \leq n + 6$ is known.

Set Partitions

a b c b c c c c d d c d e e f

Murasaki Universal Cycles?

Combinations

Here's a U-cycle for $n = 8$ and $k = 3$, where the underlying set is Σ_8.

024561457123612467036713450346012501356725602347235701 47

LEMMA 7.6 *If there is a U-cycle for* $\mathbf{A}(n, k)$ *then* k *divides* $\binom{n-1}{k-1}$.

Chung, Graham and Diaconis conjecture that this necessary condition is also sufficient as long as n is large enough, but there are also conjectures to the contrary!

7.6 Polynomials over finite fields

Polynomials whose coefficients are the elements of a finite field have many applications in mathematics and engineering. Of particular interest are what are known as irreducible polynomials and primitive polynomials. Somewhat surprisingly, we can use our algorithm for generating Lyndon words as the basis of an algorithm for generating all such polynomials of a given degree. This section will be brief and most proofs will be omitted as the mathematics necessary is non-trivial and would take more room than we have here. Several excellent introductions to finite fields are mentioned in the bibliographic remarks. Nevertheless, the careful reader should have little trouble turning the discussion here into a functioning program, particulary if written in a language for symbolic calculations, such as Maple. There is another connection between polynomials and the material presented earlier in this chapter, namely that a primitive polynomial ca be used to efficiently generate a de Bruijn sequence.

Let $p(x)$ be a monic polynomial over GF(q). Recall that GF(q) refers to the field of integers mod q and that q must be a prime power. The polynomial $p(x)$ is *irreducible* if it cannot be expressed as the product of two polynomials of lower degree.

If β is a root of a degree n polynomial $p(x)$ over GF(q), then the *conjugates* of β are $\beta, \beta^q, \ldots, \beta^{q^{n-1}}$. Each conjugate is also a root of $p(x)$. Irreducible polynomials are characterized in the following theorem.

THEOREM 7.9 *Let* p *be a degree* n *polynomial over GF(q) and* β *a root of* p. *The polynomial* p *is irreducible if and only if the conjugates of* β *are distinct.*

An element α of GF(q) is *primitive* if $k = q^n - 1$ is the smallest non-zero value for which $\alpha^k = 1$. A degree n polynomial $p(x)$ over GF(q) is *primitive* if it is irreducible and contains a primitive element of GF(q^n) as a root. In other words, to test whether an irreducible polynomial $p(x)$ is primitive.....

The number of primitive polynomials over GF(q) is known to be

$$P_n(q) \;=\; \frac{1}{n}\phi(q^n - 1)$$

Here is a small table for $q = 2$.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$P_n(q)$	1	1	2	2	6	6	18	16	48	60	176	144	630	756	1800	2048

The number of irreducible polynomials of degree n over GF(q) is given by the so-called "Witt formula"

$$I_q(n) \;=\; \frac{1}{n}\sum_{d\backslash n}\mu(n/d)q^d$$

This is the same as the number of Lyndon words, $I_k(n) = L_k(n)$. This remarkable coincidence cries out for an explanation and a purpose of this section is to supply one. The key to the correspondence is Theorem 7.9.

If α is a generator of GF(q^n) $-$ {0}, then $\beta = \alpha^k$ for some k. The conjugates of β are thus

$$\alpha^k, (\alpha^k)^q, \ldots, (\alpha^k)^{q^{n-1}}.$$

Which k's will give rise to β's that are the roots of irreducible polynomials? Since the conjugates are all distinct, they are the k's for which $k, qk, q^2k, \ldots, q^{n-1}k$ are all distinct mod $q^n - 1$. Thinking of k as a length n base q number, the conjugates are all the circular shifts of k. For them to be distinct k has to be a q-ary length n Lyndon word.

Lyndon word	k	irreducible polynomial	order e
000001	1	1000011	63
000011	3	1010111	21
000101	5	1100111	63
000111	7	1001001	9
001011	11	1101101	63
001101	13	1011011	63
001111	15	1110101	21
010111	23	1110011	63
011111	31	1100001	63

Given k the order of the polynomial is $(q^n-1)/gcd(q^n-1, k)$. The primitive polynomials are those of order $q^n - 1$.

The polynomial $p(x) = x^6 + x + 1$ is known to be primitive over GF(2). Let β be a root of $p(x)$. Thus $\beta^6 = 1 + \beta$. We now illustrate the correspondence on the Lyndon words 000001 and 001011; corresponding to $\alpha = 1$ and $\alpha = 11$, respectively. We now compute (note that $(1 + \alpha)^2 = 1 + \alpha^2$). First, with $\alpha = 1$.

$$\begin{aligned}
r(x) &= (x+\beta)(x+\beta^2)(x+\beta^4)(x+\beta^8)\,(x+\beta^{16})(x+\beta^{32}) \\
&= (x^2+(\beta+\beta^2)x+\beta^3)(x+\beta^4)(x+\beta^2(1+\beta))\times \\
&\quad (x+\beta^4(1+\beta)^2)(x+\beta^2(1+\beta)^2(1+\beta)^2(1+\beta)) \\
&= (x^2+(\beta+\beta^2)x+\beta^3)(x+\beta^4)(x+\beta^2+\beta^3))\times \\
&\quad (x+\beta^4(1+\beta^2))(x+\beta^2(1+\beta^2)^2(1+\beta)) \\
&= (x+\beta)(x+\beta^2)(x+\beta^4)(x+\beta^2+\beta^3))(x+1+\beta+\beta^4)(x+\beta^2(1+\beta^4)(1+\beta)) \\
&= (x+\beta)(x+\beta^2)(x+\beta^4)(x+\beta^2+\beta^3))(x+1+\beta+\beta^4)(x+1+\beta^3) \\
&= (x+(\beta+\beta^2)x+x^2)(x+\beta^4)(x+\beta^2+\beta^3))(x+1+\beta+\beta^4)(x+1+\beta^3) \\
&= (x+(\beta+\beta^2)x+x^2)(x+\beta^4)(x+\beta^2+\beta^3))(x+1+\beta+\beta^4)(x+1+\beta^3) \\
&= x^6+x+1
\end{aligned}$$

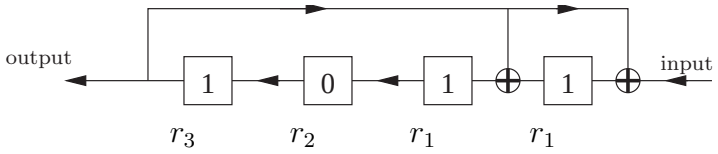


Figure 7.6: The LFSR corresponding to the polynomial $x^4 + x + 1$.

Now with $\alpha = 11$ (note that $\beta^{64} = 1$).

$$\begin{aligned} r(x) &= (x+\beta^{11})(x+\beta^{22})(x+\beta^{44})(x+\beta^{88})(x+\beta^{176})(x+\beta^{352}) \\ &= (x+\beta^{11})(x+\beta^{22})(x+\beta^{44})(x+\beta^{24})(x+\beta^{48})(x+\beta^{32}) \\ &= (x+\beta^5(1+\beta))(x+\beta^4(1+\beta)^3)(x+\beta^2(1+\beta)^7) \times \\ &\quad (x+(1+\beta)^4)(x+(1+\beta)^8)(x+\beta^2(1+\beta)^5) \\ &= x^6+x^5+x^3+x^2+1 \end{aligned}$$

THEOREM 7.10 *Every string of length n, except 0^n occurs in a linear recurring sequence generated by a primitive polynomial over* GF(p^n).

For example, the polynomial $p(x) = x^4 + x + 1$ is primitive over GF(2). It generates a linear recurring sequence defined by the recurrence relation

$$c_n = c_{n-3} + c_{n-4}.$$

Using the initial conditions $c_1, c_2, c_3, c_4 = 1, 0, 0, 0$ we obtain the sequence

1 0 0 0 1 0 0 1 1 0 1 0 1 1 1

Inserting another 0 into the block of $n - 1$ 0's produces a De Bruijn sequence, but not all De Bruijn sequences are obtained in this manner.

7.6.1 Linear Feedback Shift Registers

A *linear feedback shift register*, or LFSR, is a device like that illustrated in Figure 7.6 On each clock cycle the bit in *cell* r_{k-1} is output and fed back into some subset of the cells, depending on the placement of the feedback lines. There is one feedback line corresponding to each non-zero term of its polynomial. If the feedback line is present, then the new value of r_i is $r_{i-1} \oplus r_{k-1}$, otherwise it is just r_{i-1}. The value of r_{-1} is taken to be zero, which means that the exclusive-or gate feeding into r_0 is redundant; it is shown for the sake of consistency.

```
do { if ((x <<= 1) >> k) x = x & ONES ^ PP;
} while (x != INITIAL);
```

Algorithm 7.5: C code for iterating an LFSR.

Algorithm 7.5 generates the De Bruijn sequence in the least significant bit of the unsigned integer x, whose initial value is `INITIAL`. The algorithm is simulating the action of the LFSR.

Unsigned integer `PP` is a bitstring encoding of a primitive polynomial, 0011 in the example above. Constant `ONES` is a bitstring of k 1's in the lower order bits. This code essentially generates each new value of x in constant time. The only problematic operation is the shift right by k positions. On most modern computers this shift should be executed in one or two machine instructions.

7.6.2 Another look at the BRGC

We may identify a bitstring $b_0 b_1 \cdots b_{n-1}$ with a degree $n-1$ polynomial over GF(2) under the bijection

$$b_0 b_1 \cdots b_{n-1} \leftrightarrow \sum_{i=0}^{n-1} b_i x^i.$$

Under this bijection a right shift corresponds to multiplying by x and componentwise exclusive-or to addition mod 2. Thus if $g(x)$ is the polynomial corresponding to the bth word in the BRGC, then

$$g(x) = (1+x)b(x).$$

Since the algebraic inverse of $1+x$ is the polynomial $1 + x + \cdots + x^{n-1}$ (check this!), the procedure for converting from the BRGC to binary corresponds to multiplication by $1 + x + \cdots + x^{n-1}$. That is,

$$b(x) = (1 + x + \cdots + x^{n-1})g(x).$$

7.7 Exercises

1. **[1]** What is the length of a maximal domino game for a general value of n (dots taken from $[n]$)?

2. **[2+]** Show that there is a string $s_0 s_2 \cdots s_{2n!-1}$ of length $2n!$ over the alphabet $[n]$ such that

   $$\{s_i s_{i+1} \cdots s_{i+n!-1} : i = 0, 2, \ldots, 2n!\},$$

   where the index arithmetic is taken mod $2n!$, is the set of all $n!$ permutations of $[n]$.

3. **[*R*−]** How many Eulerian cycles does the directed n-cube $\vec{Q}_n$ have?

4. **[*R*−]** Develop a CAT or, better yet, loopfree algorithm for generating an Eulerian cycle in $\vec{Q}_n$.

5. **[2]** Use (7.10) to prove (7.7).

6. **[2+]** Let $N(n_0, n_1, \ldots, n_t)$ denote the number of necklaces composed of n_i occurrences of the symbol i, for $i = 0, 1, \ldots, t$. Let $n = n_0 + n_1 + \cdots + n_t$. Prove that

   $$N(n_0, n_1, \ldots, n_t) = \frac{1}{n} \sum_{d \backslash \gcd(n_0, \ldots, n_t)} \phi(d) \frac{(n/d)!}{(n_0/d)! \cdots (n_t/d)!}$$

In particular, if $t = 1$, then

$$N(r, n-r) \;=\; \frac{1}{n} \sum_{d \backslash \gcd(r,n-r)} \phi(d) \binom{n/d}{r/d}$$

gives the number of necklaces with r black beads and $n - r$ white beads.

7. [**1+**] Find a simple one-to-one correspondence between length $2n$ necklaces with n black beads and n white beads and rooted plane trees with n edges.

8. [**2+**] Derive a formula for the number of binary Lyndon words of length n and weight r. The *weight* of a string of digits is the sum of those digits. [**2+**] Let $eL_k(n)$ be the number of length n Lyndon words of even weight, $oL_k(n)$ be the number of odd weight, and $dL_k(n)$ be the difference $dL_k(n) = eL_k(n) - oL_k(n)$. Show that $dL_2(n) = n^{-1} \sum \mu(d) 2^{n/d}$ where the sum is over all odd $d \backslash n$. Show that $dL_2(2n) = -oL_2(n)$.

9. [**1**] For odd n, show that $eN_2(n)$, the number of binary necklaces with an even number of 1's, is equal to $oN_2(n)$, the number of binary necklaces with an odd number of 1's.

10. [**1+**] What bitwise operations preserve necklaces (or prenecklaces, or Lyndon words)? I.e., is the intersection of necklaces a necklace? What about union and exclusive-or?

11. [**1+**] Modify both the iterative and the recursive versions of the necklace generating algorithms so that the produce necklaces in relex order, instead of lex order. Which algorithm was easier to modify?

12. [**2**] Prove that $w \in \mathbf{L}$ if and only if $w < v$ for all $uv = w$ where $v \neq \varepsilon$. I.e., a word is a Lyndon word if and only if it is strictly less than all of its proper suffixes.

13. [**2**] Prove that if x and y are both Lyndon words and $x < y$ then xy is also a Lyndon word. Finish the proof of the Chen, Fox, Lyndon Theorem 7.7 by showing that the Lyndon factorization is unique.

14. [**3**] Define an involution τ on $\{0,1\}^n$ by

$$\tau(x_1 \ldots x_n) = x_1 \ldots x_{n-1} \overline{x_n},$$

where $\overline{x_n}$ denotes the complement of the bit x_n, and let $\sigma(x)$ denote the rotation of string x one position left. Show that the calls: Print(0^n); Gen($0^{n-1}1$); generate all necklaces of length n in two colors, where Gen is the procedure shown below.

```
procedure Gen ( x : necklace );
begin
    Print( x );
    x := τ(x);
    while IsNecklace(x) do begin
        Gen( x );
        x := τστ^{-1}(x);
    end;
end {of Gen};
```

15. **[2]** Let $\rho = \tau\sigma$ where τ and σ are as in the previous exercise. In other words,

$$\rho(b_1 b_2 \cdots b_n) = b_2 \cdots b_n \overline{b_1}$$

Define an equivalence relation $\sim$ between bitstrings $\mathbf{x}$ and $\mathbf{y}$ of length n as follows: $\mathbf{x} \sim \mathbf{y}$ if and only if there is a number k such that $\rho^k(\mathbf{x}) = \mathbf{y}$. Show that the number of such equivalence classes is

$$\frac{1}{2n} \sum_{\substack{d \backslash n \\ d \text{ odd}}} 2^{n/d} \phi(d).$$

There is a one-to-one correspondence between such equivalence classes and what are known as "vortex-free" tournaments.

16. **[*R*−]** Let $\mathbf{V}(n)$ denote the set of lexicographically least representatives of length n of the equivalence classes of $\sim$, as defined in the previous exercise. For example $\mathbf{V}(n) = \{00000, 00010, 00100, 01010\}$. Develop an efficient, and preferably CAT, algorithm to generate $\mathbf{V}(n)$.

17. **[2]** How many n-bead necklaces are there, composed of white and black beads, and with no two adjacent black beads? How many such necklaces are aperiodic? Prove the following, a kind of Fermat's Little Theorem for Fibonacci numbers: If p is a prime, then $F_{p+1} + F_{p-1} \equiv 1 \pmod{p}$. **[*R*−]** Given a forbidden pattern P (a bitstring — 00 in the first part of this problem), how many necklaces of length n are there without P as a substring? How fast can you compute the number?

18. **[*R*−]** Develop an efficient ranking algorithm for prenecklaces, necklaces, and Lyndon words in the order that they are generated by the FKM algorithm.

19. **[*R*]** Develop an efficient ranking algorithm for some De Bruijn sequence. If the De Bruijn sequence arising from the FKM algorithm is used then the results of the previous exercise should be useful.

20. **[1+]** Prove that if $\alpha \in \mathbf{P}_k(n)$ then $\alpha(k-1)^n \in \mathbf{N}_k(n)$. This means that every prenecklace α is also the prefix of a Lyndon word, unless α is a string of $(k-1)$s of length greater than 1.

21. **[2]** Prove: $\alpha \in \mathbf{P}$ if and only if $xy \leq yz$ and $x \leq z$ for all x, y, z such that $\alpha = xyz$ with $|x| = |z|$.

22. **[*R*]** Find a Gray code (successive words differ by one bit) of Lyndon words when $k = 2$ or prove that no such code exists.

23. **[3+]** Let $n > 1$ and $p, q > 1$. Show that $L_{q+4}(n) - L_q(n)$ is even and hence that if $p \equiv q \pmod 4$, then $L_q(n) \equiv L_p(n) \pmod 2$. Use this result to determine the parity of $L_q(n)$.

24. **[2]** Prove that $L_{pq}(n) = \sum \gcd(i,j) L_p(i) L_q(j)$ where the sum is taken over all i and j such that $\operatorname{lcm}(i,j) = n$.

25. [**2**] Find a class of words that causes the Duval algorithm to use $2N-o(N)$ comparisons for infinitely many values of N.

26. [***R*—**] For a binary alphabet, what is the maximum number of comparisons that can be used by the Duval algorithm on a string of length N? For a k-ary alphabet?

27. [***R*—**] Develop a CAT algorithm for generating all necklaces where the number of beads of each color is fixed. The number of such necklaces was determined in Exercise 6. Even the two color case is open.

28. [***R*—**] Develop a CAT algorithm for generating the lexicographically least representatives of the equivalence classes of k-ary strings of length n that are equivalent under rotation *or reversal*. In other words, the dihedral group is acting on the strings and not just the cyclic group. Such equivalence classes are sometimes called *bracelets*.

29. [**2**] What about 2-dimensional analogues of De Bruijn cycles? These are sometimes called *De Bruijn torii*. (a) Consider the set of 2 by 2 tiles where each square of the tile is colored black or white. There are 16 such tiles. Can they be placed in a 4 by 4 arrangement so that their borders match, where the border wrap-around in the manner of a torus? (b) Now let each tile be colored with three colors, say white, gray, and black. There are 81 tiles. Can they be placed in a 9 by 9 arangements so that the colors along their borders match?

30. [**2**] For those who know about context-free languages: Use a closure property to prove that **N** and **L** are not context-free languages. Use the "pumping lemma" to prove that **N** and **L** are not context-free languages.

31. [**2**] Let $\mathbf{I}(n,k)$ be the set of permutations $\pi \in \mathbb{S}_n$ such that $\pi^k = \mathbf{1}$, the identity permutation, and $I(n,k) = |\mathbf{I}(n,k)|$. Thus $I(n,2)$ counts the number of involutions in $\mathbb{S}_n$. Show that

$$I(n,k) = \sum_{d \backslash k} \binom{n-1}{d-1} (d-1)! I(n-d,k).$$

[***R*—**] Develop a CAT algorithm for generating the elements of $\mathbf{I}(n,k)$.

32. [**1**] Modify algorithm 7.5 so that it outputs a De Bruijn sequence.

7.8 Bibliographic Remarks

The Eulerian cycle in $\vec{Q}_n$ is from Bate and Miller [22].

An early paper containing an algorithm for constructing a DeBruijn cycle is Martin [266]. More recent papers about generating De Bruijn cycles include Fredricksen and Kessler [142], Fredricksen [141], Ralston [326], Huang [184], and Xie [463]. It is rather remarkable that there is a simple closed form formula for the number of distinct De Bruijn cycles.

$$(k!)^{k^{n-1}}/k^n.$$

This formula was proven by Van Aardenne-Ehrenfest and De Bruijn [426], but was anticipated by Flye-St. Marie [135]. It is even possible to count the number of Eulerian

cycles (directed or undirected) in a graph. See Fleishner's book [132], which contains everything you could want to know about Eulerian cycles. Generating all Eulearian cycles was considered by Fleishner [133].

Interesting material about De Bruijn cycles may be found in Stewart [406].

The algorithm ??? for generating necklaces is from Fredricksen and Maiorana [140] and Fredricksen and Kessler [139]. Independently, Duval [93] developed a version of the algorithm that generates Lyndon words. Duval [92] is our source for the material of the section on finding the necklace of a string.

The formula (7.7) has been attributed by Comtet [68] to the 1892 paper of Jablonski [190]. It is credited to "Colonel Moreau of the French Army" by Metroplis and Rota [279]. Cummings and Mays [76] determine the parity of the Witt formula.

Papers about generating primitive and irreducible polynomials include Lüneburg [259], and Gulliver, Serra and Bhargava [161]. The relationship between irreducible polynomials and Lyndon words is explored in Golomb [155], Lüneburg [259], [260], and Reutenauer [342]. Tables of primitive polynomials may be found there and in Peterson and Weldon [307] and Serra [381].

An analysis of the FKM algorithm is given in Ruskey, Savage, and Wang [357]. Duval's nearly identical algorithm is analyzed in Berstel and Pocchiola [29]. The recursive version of the FKM algorithm (Algorithm 7.3) is believed to be new.

A CAT algorithm for necklaces where the number of 0's is fixed may be found in Ruskey and Sawada [359]. A CAT algorithm for binary bracelets was developed by Sawada [373].

More material on Lyndon words may be found in Lothaire [253].

Universal cycles were introduced in Chung, Diaconis and Graham [65]. Later developments may be found in Jackson [191] and Hurlbert [186], [187].

There is a delightful chapter called "The Autovoracious Ourotorus" about De Bruijn cycles and generalizations in Stewart [406].

Chapter 8

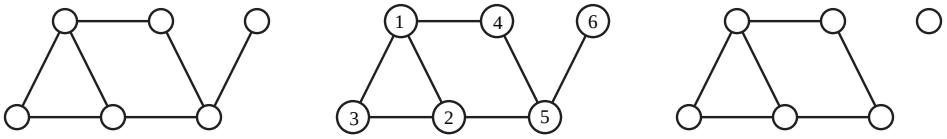
Orderly Algorithms

For complicated objects progress in combinatorial generation can be measured in terms of whether it is more expedient to generate the objects via a program or to read them from storage on an external device. An interesting case in point is the set of unlabelled graphs. Ron Read computed all such graphs in 1960's and made them available on magnetic tape. This tape proved to be quite popular. Now, however, no one uses this tape any longer because there is a program called "`makeg`", developed by Brendan McKay that produces the graphs more quickly and conveniently than they can be read from a tape or any other external storage device. In fact in 1993, he generated all 90,548,672,803 of the 12 vertex graphs on at most $\binom{12}{2}/2$ edges. The computation took 5277 hours of computer time — distributed over a network of computers.

There are three things that drive such progress: (a) better algorithms, (b) faster hardware (including the use of parallelism), and (c) better implementations (i.e., code optimization).

There are many types of combinatorial objects that do not seem to have nice recursive decompositions that translate into efficient listing algorithms, such as those that we encountered in Chapters 4 and 5. Nevertheless there may be enumerative results about the number of such objects. Consider the example of unlabelled graphs. They can be counted by using Polya theory, resulting in a sum over integer partitions with a multiplicative factor of $1/n$. However, this formula seems to be of little use in actually generating the graphs.

In general we may wish to list members of equivalence classes of combinatorial objects. For example graphs under the equivalence relation of isomorphism, as mentioned above. There is a general technique for developing algorithms that list equivalence classes, particularly equivalence classes that arise from symmetry conditions, called orderly algorithms. In orderly algorithms there must be a systematic way of constructing larger objects recursively from smaller objects. For example, graphs can be constructed by adding edges, starting with the empty graph. The main idea in orderly algorithms is to carefully pick the representative of each equivalence class and the order in which larger objects are constructed from smaller objects. Our final aim is a backtracking algorithm to generate the objects. Typically the resulting backtracking algorithms have the property of having successes at all nodes; i.e., they are BEST algorithm. This property is certainly one to be strived for; unfortunately, the amount of work done at a node is often exponential in the size of the individual objects (at least in the worst case) so we don't necessarily obtain CAT algorithms. But enough of this general rambling; let's look at a few specific examples.



G $\langle G \rangle$ $P(G)$

Figure 8.1: An unlabelled graph, it's canonic labelling, and it's parent.

8.1 Undirected Graphs

In this section we develop an algorithm for generating all unlabelled graphs. With each unlabelled graph G we associate a unique labelled graph $\langle G \rangle$, which is said to be *canonic*. If two labelled graphs G and H are isomorphic, then $\langle G \rangle = \langle H \rangle$. The algorithm outputs all the canonic labelled graphs. By ignoring the labels of the output canonic graphs, we obtain all unlabelled graphs.

Orderly algorithms are a particular type of BEST backtracking algorithm. Our strategy is to impose a tree structure on the set of canonically labelled graphs and construct our bactracking so it has this tree as its computation tree.

If G is a graph and e is a nonedge of G, the $G \oplus e$ denotes the graph obtained by adding e to the edge set of G. Similarly, $G \ominus e$ denotes the graph obtained by removing e from the edge set of G.

We begin by trying to associate a single graph, call it the *parent*, $P(G)$, with G such that $G = P(G) \oplus e$ for some edge e. To do this it is necessary to decide which edge to remove from G to get $P(G)$. An unambiguous choice can be made if we do two things.

(A) Fix an ordering of the set of all possible edges of labelled graphs with vertex set $[n]$.

(B) Label the vertices of G in some unique way; i.e., so that isomorphic graphs get the same labelling. This is the canonic labelling refered to above.

With respect to the ordering (A) and labelling (B), the vertex e to be removed is the largest one in G. The most natural choices for these orderings are as follows.

(A') Each edge is a 2-subset of n. Use the lexicographic ordering of $\mathbf{A}(n, 2)$, as in Section 4.3. In other words

$$\{1,2\} < \{1,3\} < \cdots < \{1,n\} < \{2,3\} < \cdots < \{n-1,n\}.$$

(B') For any labelling of the vertices of G, the edges of G can be listed

$$e_1 < e_2 < \cdots < e_q.$$

Among all labellings $\pi \in \mathbb{S}_n$, choose the one that lexicographically minimizes the string $(e_1, e_2, \ldots, e_q)$.

Figure 8.1 shows a canonic labelling and it's parent.

Now that $P(G)$ is well-defined, observe that it imposes a tree structure, T_n, on the set of all graphs with n vertices. See Figure 8.2 for an example with $n = 4$. The labelling of

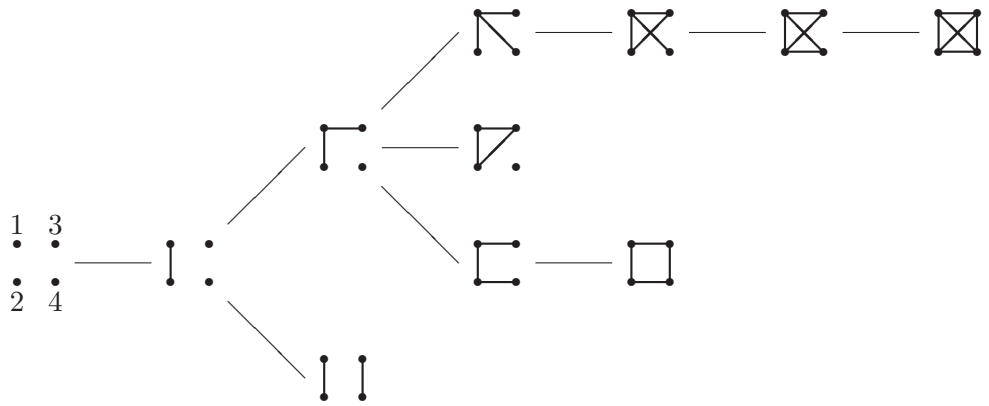


Figure 8.2: The tree of unlabelled graphs on four vertices.

vertices shown at the root is maintained throughout the tree. If the edges are selected in lexicographic order, then this tree is traversed in preorder from the top of the page towards the bottom; e.g.,  is output before  . The reader should study this figure carefully. Our object is to now develop a backtracking algorithm whose computation tree is T_n.

Consider $P(G)$. What are its children? They are obtained by adding an edge e, larger than any other edge of $P(G)$. The resulting graph $G = P(G) \oplus e$ may or may not satisfy condition (B'); i.e., it may or may not be canonic. If it is canonic, then G is a valid graph and can be output. If it is not canonic, then G can be ignored. Clearly every canonic child of $P(G)$ will be produced by this process. Furthermore, no child can be produced more than once because the added edges e are all different. Why is the algorithm correct? The key observation is that *the parent of a canonic graph is canonic*. For other types of combinatorial objects, it's not always easy (or possible) to choose definitions of canonicity and parenthood that have this crucial property.

```
procedure Gen ( G : graph; x : edge );
local y : edge;
begin
    Print( G );
    for all y > x do
        if Canonic( G ⊕ y ) then Gen( G ⊕ y, y );
end {of of Gen};
```

Algorithm 8.1: Orderly algorithm (recursive version).

The incarnation of the preceding discussion is the deceptively simple Algorithm 8.1. The initial call is $Gen(G, e)$ where G is the graph containing the single edge $e = \{1, 2\}$. As is usual in backtracking, it is most expedient to leave the major data structure, which in this case is G, global, as is illustrated in Algorithm 8.2. In both algorithms the most recently added edge x is passed as a parameter.

What are the resource requirements of this algorithm? The height of the backtracking tree T_n is $\binom{n}{2}$ and only a constant amount of space is needed at each node (for edges x and y). Thus space usage is $O(n^2)$. The time required depends on the following three factors.

- The number of graphs on n vertices.

```
procedure Gen ( x : edge );
local y : edge;
begin
    Print;
    for all y > x do
        G := G ⊕ y;
        if Canonic then Gen( y );
        G := G ⊖ y;
end {of Gen};
```

Algorithm 8.2: Orderly algorithm with G global.

- The number of times *Canonic* fails.
- The running time of *Canonic*.

The number of graphs on n vertices is known to be asymptotic to

$$\frac{1}{n!}2^{\binom{n}{2}}.$$

The running time of *Canonic* is essentially the time it takes to test whether two graphs are isomorphic. Naively implemented, its running time is $\Theta(n^2 n!)$. The number of times *Canonic* fails is unknown, but at any node of the tree it can fail at most $\binom{n}{2}$ times, so the total time is $O(\binom{n}{2}\frac{1}{n!}2^{\binom{n}{2}})$. Thus we can say that overall running time of the algorithm is $O(\binom{n}{2}2^{\binom{n}{2}})$, but this is extremely pessimistic. For the `makeg` program mentioned in the first paragraph of this chapter, the canonicity tester is another program `nauty` that is extremely fast on a random graph — so much so, that `makeg` appears to be a CAT algorithm!

Enumerating graphs

Using Polya theory, one can derive a formula for the number of unlabelled graphs. The derivation of this formula is outside the scope of this book, but we can give a little insight. Consider the expression

$$\frac{1}{n!}\sum_{(j)}\frac{n!}{\prod k^{j_k}j_k!}\prod_k s_{2k+1}^{kj_{2k+1}}\prod_k(s_k s_{2k}^{k-1})_{2k}^{j}s_k^{k\binom{j_k}{2}}\prod_{r<t}s_{[r,t]}^{(r,t)j_r j_t}$$

The sum is over (j) which represents all numerical partitions of n, where j_k is the number of occurances of k. We can evaluate the sum by making use of our algorithm for generating numerical partitions with the multiplicity representation. In the formula $s_1, s_2, \ldots$ are indeterminates, (r,t) denotes the greatest common divisor of r and t, and $[r,t]$ denotes their least common multiple.

The number of graphs with n vertices and m edges is the coefficient of x^m in the polynomial $g_n(x)$ given below:

$$g_n(x) = Z(S_n^{(2)}, 1+x)$$

This means to use the substitution $s_k = 1 + x^k$.

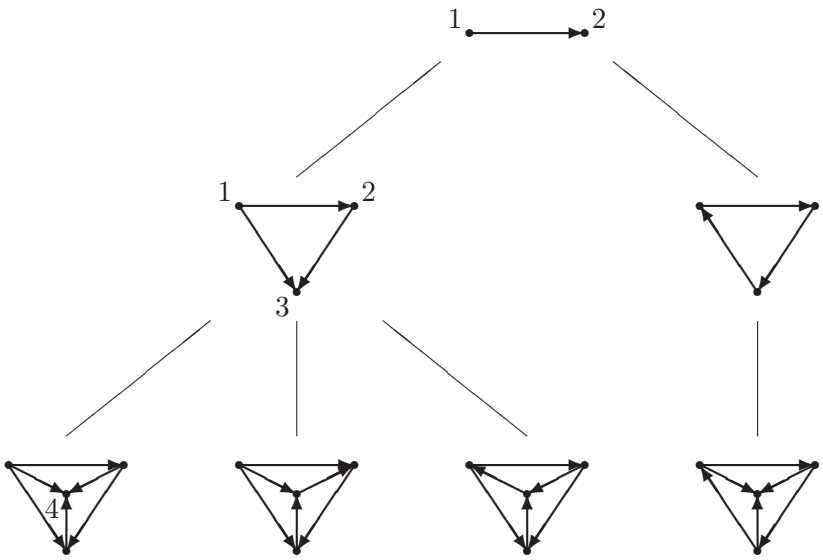


Figure 8.3: Tree of tournaments for $n \leq 4$.

8.2 Tournaments

In this section we develop an orderly algorithm for generating all unlabelled tournaments. We follow the same pattern as laid out in the previous section. It is necessary to define appropriate notions of canonicity and parenthood.

Recall that a labelled tournament is a directed graph that has either the edge $[i, j]$ or has the edge $[j, i]$, but not both, for each $1 \leq i < j \leq n$. We construct tournaments by adding a vertex at a time, rather than an edge at a time as was done for graphs. First we order the possible edges of a labelled tournament. Define a colex ordering $c_1, c_2, \ldots, c_{\binom{n}{2}}$ on those edges $[i, j]$ where $i < j$.

$$[1,2] < [1,3] < [2,3] < [1,4] < \cdots < [1,n] < \cdots < [n-1,n]$$

Define a lex ordering $l_1, l_2, \ldots, l_{\binom{n}{2}}$ on those edges $[i, j]$ where $i > j$.

$$[2,1] < [3,1] < [3,2] < [4,1] < \cdots < [n,1] < \cdots < [n,n-1]$$

A tournament is represented by a string of edges

$$\mathbf{e} = e_1 e_2 \cdots e_{\binom{n}{2}} \text{ where } e_i = c_i \text{ or } e_i = l_i$$

As the canonic representative of each unlabelled tournament we choose that labelling that lexicographically minimizes $\mathbf{e}$. The parent of a canonic tournament T with n vertices is obtained by removing vertex n from T. Figure 8.3 shows the resulting tree of tournaments for $n \leq 4$. Let X_p denote the following cartesian product.

$$X_p = \{[1,p],[p,1]\} \times \{[2,p],[p,2]\} \times \cdots \times \{[p-1,p],[p,p-1]\}$$

Another way of thinking of $\mathbf{e}$ above is as an element of $X_1 \times X_2 \times \cdots \times X_p$.

Our orderly algorithm for tournaments is Algorithm 8.3. The call $Gen(T, p)$, given a $p-1$ vertex canonic tournament T, generates all children of T that have at most n vertices. The statement **for all e** $\in X_p$ is assumed to select strings in X_p in lexicographic order. The initial call is Gen([1, 2],3).

```
procedure Gen ( G : tournament; p : ℕ);
local e : EdgeString;
begin
    if p ≤ n then
        Print( G );
        for e ∈ X_p { in lex order } do
            if Canonic( G ⊕ e ) then Gen( G ⊕ e, p + 1 );
end;
end {of Gen};
```

Algorithm 8.3: Orderly algorithm for tournaments on at most n vertices.

Why does this algorithm work? First, note that if vertex n is removed from a canonic tournament, then a canonic tournament results. This means that every tournament is produced at least once. Secondly, since the parent of a tournament is unique, no tournament can be produced more than once.

8.3 Restricted Classes of Graphs

We have now seen examples of orderly algorithms for generating two types of graphs. In one instance the augmentation operation added an edge, in the other instance the augmentation operation added a vertex. In this section we develop orderly algorithms for generating restricted classes of graphs, using one or the other of those augmentation operations.

8.3.1 Bipartite Graphs

8.3.2 Cubic Graphs

Let C_n denote the number of cubic graphs on n vertices.

n	4	6	8	10	12	14	16	18	20	22	24
C_n	1	2	5	19	85	509	4060	41301	510489	7319447	117940535

8.4 Posets

In this section we discuss an orderly algorithm for generating non-isomorphic posets.

8.4.1 Dedekind's Problem

[!!! Recent results of Wiedemann !!!]

8.5 Coset Enumeration

8.6 Exercises

1. [**1**] What is the canonic labelling of the following graph G? What is it's parent $P(G)$?

2. [**1**] What is the canonic labelling of the following tournament G? What is it's parent $P(G)$?

3. [***R*−**] Develop an orderly algorithm for generating all non-isomorphic unicyclic graphs. An undirected graph is *unicyclic* if it contains a single cycle; such graphs are obtained by adding an edge to a tree.

4. [***R*−**] Use and orderly algorithm to determine the number of non-isomorphic antimatroids with n feasible sets for small values of n. Do the same for greedoids.

8.7 Bibliographic Remarks

Read [335] was the paper that popularized the name and method.

Other papers about orderly algorithms include Read [337], Colbourn and Read [336], Cameron, Colbourn, Read, and Wormald [48], Royle [352], White and Williamson [445], McKay [271]. The basic algorithm for generating unlabeled graphs can be modified so that it uses polynomial space (see Colbourn and Read [67]), and so that polynomial delay occurs between the output of successive graphs (see Go—dberg [151]).

McKay and Royle develop an orderly algorithm that is used to generate all cubic graphs on up to 20 vertices [272]. The best method for generating cubic graphs, and that described in Section ???, is due to Brinkmann [41]. Orderly algorithms for generating k-colorable graphs, in particular bipartite graphs, may be found in Koda [224].

The generation of graphs with transitive automorphism group is consider in McKay and Royle [273] (up to 26 vertices).

The algorithm for generating posets in Section ?? is due to Culberson and Rawlins [73]. Other papers about generating posets are Erne and Stege [113], and ..... Koda [225] discusses orderly algorithms for generating all lattices and finite topologies.

Redelmeier [339] develops an algorithm for generating all non-isomorphic polyominoes.

Walsh [436] develops an orderly algorithm for generating all nonisomorphic maps with a given number of vertices and edges and of a given genus. [Is a *map* simply a graph embedded in a surface?]

The recent thesis of Goldberg [152] disusses many complexity issues associated with orderly algorithms and the types of problems to which they are applied.

Other references about graph generation include Baraev and Faradjev [21], Faradjev [117], McKay [270], Yap [464], Petrenjuk and Petrenjuk [308].

A variety of algorithms for generating combinatorial objects subject to group actions may be found in Laue [241].

Group theorists have long used the Todd-Coxeter algorithm for generating coset representatives of groups presented as generators and defining relations. See for example, Todd and Coxeter [414] and Canon, Dimino, Havas, and Watson [50]. Allison [7] has developed an algorithm for the case where the group is presented in terms of permutations.

Chapter 9

Subgraph Generation

!!!! Put in some sort of nifty introductory example. (E.G., Why do electrical engineers want to generate spanning trees) !!!!

In this brief chapter we consider generation questions where the input is a graph and the output consists of all subgraphs of a specified type. We consider the problem of listing all spanning trees, all cliques, and so on.

In the previous chapter on orderly algorithms we saw the utility of imposing a rooted tree structure on the set of objects to be generated; in fact, this is a theme throughout the book. It will also be a theme of this chapter, but the aim of the tree is somewhat different. In the previous chapter we were concerned with not generating isomorphic instances of already generated objects, here the tree is simply a guide that will hopefully lead us to an efficient algorithm.

9.1 Spanning Trees

Every connected graph has a spanning tree and the selection of a spanning tree satisfying some optimization criteria is an oft occuring problem in discrete optimization. For simple criteria, such as total edge weight, well-known efficient algorithms are known — this is just the minimum spanning tree problem. For other criteria, we must examine all spanning trees and just pick the best.

9.1.1 A naive algorithm

The number of spanning trees is given by the Matrix Tree Theorem (Theorem 2.4). We can generate spanning trees by using the basic Contraction-Deletion idea.

9.1.2 A CAT algorithm

In the remainder of this section we present a CAT algorithm for generating spanning trees that is due to Shioura and Tamura [387]. Let $G = (V, E)$ be a graph and assume that the vertices and edges have been ordered according to a depth-first search (DFS) of the graph as described below. The vertex set is $V = \{v_1, v_2, \ldots, v_n\}$, where $v_1, v_2, \ldots, v_n$ is the order in which vertices are encounted in the DFS. The edge set is $E = \{e_1, e_2, \ldots, e_m\}$, where $e_1, e_2, \ldots, e_{n-1}$ are the tree edges of the graph in the order that they are encountered in DFS; the non-tree edges are also ordered according to when they are encounted in the DFS.

```
procedure Gen ( G : graph; T : tree );
var y : edge;
begin
    if |G| = 1 then PrintTree( T ) else
    if G connected then begin
        e ← some non-loop edge of G;
        Gen( G • e, T ∪ {e} );
        Gen( G − e, T );
    end; end {of Gen};
```

Algorithm 9.1: Contraction-Deletion algorithm for generating spanning trees.

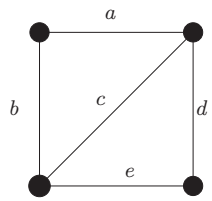


Figure 9.1: The connected graph $K_4 - e$ with edges labelled.

Let T^0 the the depth-first search tree; it is rooted at v_1. Figure **??** shows a graph with a numbering arising from a DFS.

9.1.3 A Spanning Tree Gray Code

Can spanning trees be generated by single edge replacements? This is the most natural Gray code question for spanning trees. The answer is yes. In this subsection we provide a proof of this assertion and give some hints about how to generate this Gray code in a CAT manner.

Given a graph G, the *tree graph* of G, denoted $T(G)$ has as vertices the spanning trees of G, with edges joining those vertices that differ by a single edge replacement. Figure 9.1 shows a graph, with its spanning trees shown in Figure 9.2 and its tree graph shown in Figure 9.3. Note that $t_1, t_2, \ldots, t_8$ is a Hamilton cycle in $T(G)$.

To prove that $T(G)$ is hamiltonian we prove that it has a stronger property, uniform Hamiltonicity. A graph X is *positively Hamiltonian* if every edge of X is included in some Hamilton cycle, and is *negatively Hamiltonian* if every edge of X is avoided by some Hamilton cycle. A graph that is both positively and negatively Hamiltonian is *uniformly Hamiltonian*.

The basis of the algorithm and the proof of uniform Hamiltonicity is the decomposition of the set of spanning trees into two disjoint sets, those that contain some specified edge and those that don't. Let e be an edge of a multigraph $G(V, E)$. Recall that $G - e$ is the graph obtained from G by removing the edge e, and that $G \bullet e$ is the graph obtained from G by contracting the edge e.

$$\tau(G) = \tau(G \bullet e) + \tau(G - e)$$

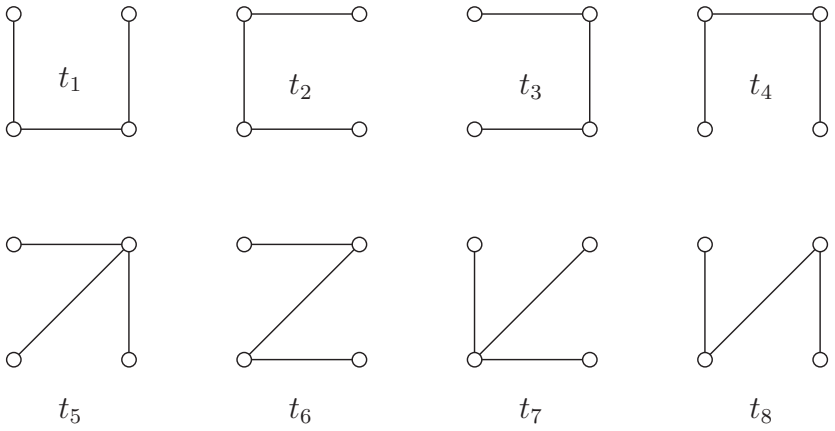


Figure 9.2: The spanning trees of $K_4 - e$.

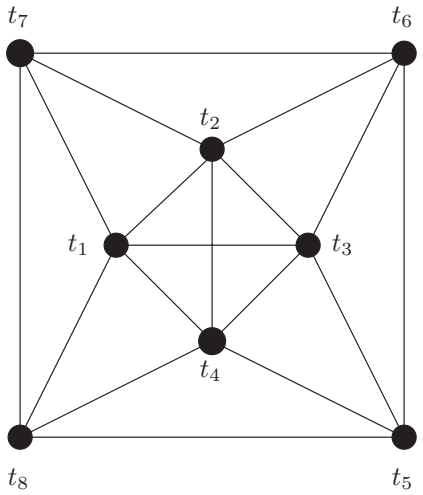


Figure 9.3: The tree graph of $K_4 - e$.

THEOREM 9.1 *The tree graph $T(G)$ of a graph G is uniformly Hamiltonian if $T(G)$ contains at least two cycles.*

PROOF: □

9.2 Reverse Search

9.3 Cycles

9.4 Cliques

9.5 Maximal Independent Sets

9.6 Cutsets

Let $G = (V, E)$ be a graph or a digraph, with $s, t \in V$. A *cut* is a set of edges of the form

$$(X, \overline{X}) \;=\; \{(u, v) \in E \mid u \in X, v \in \overline{X}\},$$

where $X \subseteq V$ and $\overline{X} = V \setminus X$. A cut $(X, \overline{X})$ is an (s, t)-*cut* if $s \in X$ and $t \in \overline{X}$. A (s, t)-*cutset* is an edge minimal (s, t)-cut.

Cuts and (s, t)-cuts have a range of applications, such as switching functions, sensitivity analysis of optimization problems, vertex packing, and network reliability calculations ([318]).

9.7 Chromatic Polynomials

9.8 Convex Polytopes

Generating all vertices and faces.

9.9 Exercises

1. [1] Show that $T(K_3)$ is K_3. Draw $T(K_4)$. Find simple graphs G_n for which $T(G_n)$ is isomorphic to K_n?

9.10 Bibliographic Remarks

Spanning Trees

Read and Tarjan [338] show the spanning trees, cycles, and simple cycles of a graph can all be generated in space $O(n+m)$ and time $O(n+m+mN)$, where n is the number of vertices in the graph, m is the number of edges, and N is the number of spanning trees, cycles, or simple cycles, respectively. According to the 1970 book of Moon [285], "Many papers have been written giving algorithms, of varying degrees of usefulness, for listing the spanning trees of a graph". He mentions the early papers of Feussner [123], [124], and the papers of Wang [437], Duffin [90], [166], Mayeda and Seshu [269], Mukherjee and Sarker [286],

Chen [59], Berger [27], and Char [54]. More recent algorithms may be found in Minty [283], Kamae [202], Kishi and Kajitani [211], Fang [116], Wojciechowski [459], Cherkasskiĭ [61], Winter [457], Hakimi and Green [167], Chen [60], Jayakumar, Thulasiraman, and Swamy [193], and Gabow and Myers [147]. The later paper also considers the problem of generating the spanning trees of directed graphs. Gabow [146] discusses the generating of weighted spanning trees in increasing order.

The best current algorithms for generating spanning trees are those of Kapoor and Ramesh [205], Shioura and Tamura [387], and Smith [394]. Their algorithms run in time $O(N+m+n)$ and space $O(mn)$ for a graph with n vertices m edges and N spanning trees. The Shioura and Tamura algorithm is an instance of reverse search. Smith's algorithm generates spanning trees in a Gray code order; successive trees differ by a single edge replacement.

The proof that tree graphs are uniformly Hamiltonian is from Holzmann and Harary [179]; this proof applies more generally to the bases of a matroid. Earlier papers on this same topic are Cummins [77] and Kishi and Kajitani [212].

Colbourn, Myrvold and Neufeld develop a $O(n^3)$ (arithmetic operations) algorithm for ranking the spanning trees of a graph.

Other stuff

Johnson [197] gives an algorithm for generating all circuits of a graph. Cai and Kong [47] give fast algorithms for generating all cliques of certain perfect graphs.

Avis and Fukuda [17] give an algorithm for generating all vertices of a convex polytope. A generalization of this procedure called "Reverse Search" [18] resulted in fast algorithms for listing other types of combinatorial objects.

Fukuda and Matsui [144] give an algorithm for generating all perfect matchings in a bipartite graph. An algorithm to enumerate all common bases in two matroids is presented in Fukuda and Namiki [143].

Eppstein [103] gives an $O(n)$ algorithm for listing all bipartite subgraphs of a graph of bounded arboricity.

Here we mention some algorithms for the important class of planar graphs. Sysło [409] gives an algorithm for listing all cycles of a planar graph. Listing all triangles (3-cycles) was considered by Itai and Rodeh [188]. Listing of triangles, 4-cycles, and cliques was considered in Chiba and Nishizeki [62] (see also Chapter 8 of Nishizeki and Chibe [63]); their algorithm specialized to planar graphs run in linear amortized time. See also Matula and Beck [268] and Papadimitriou and Yannakakis [303].

The number of triangles, 4-cycles and cliques in planar graphs are all $O(n)$ so that these problems are of a somewhat different character than those we've usually considered, since our typical problem has exponentially many instances in terms of m and n.

Tsukiyama, Ide, Ariyoshi, and Shirakawa [420] generate maximal independent sets as does Loukakis [254]. The maximal independent sets of trees are generated by Chang, Wang, and Lee [53]. The *maximum* independent sets of bipartite graphs are generated by Kashiwabara, Masuda, Nakajima, and Fujisawa [207] in amortized $O(n)$ time per set.

!! paper by Szwarcfiter on perfect graphs? !!

Cutsets

There have quite a few papers written about generating all cutsets. Algorithms for listing all cutsets in undirected graphs may be found in Abel and Bicker [1], Bellmore and Jensen [25], and Tsukiyama, Shirakawa, Ozaki, and Ariyoshi [421] The best of these algorithms is that found in [421], which requires $O(|E|)$ computation per cutset generated.

Algorithms for generating all minimum cardinality cuts have been proposed by Ball and Provan [19], Gardner [149], and Picard and Queyranne [310]. Ramanathan and Colbourn [328] generate what they call "nearly" minimum cardinality cuts. Kanevsky [203] generates vertex cutsets of minimum cardinality in linear time per cutset. According to the abstract, he generates all M minimum size separating sets of a k connected graph in time $\Theta(\min\{\max\{Mnk, knm\min\{k, \sqrt{n}\}\}, \max\{Mn, k^2n^3\}$. A simple unanalyzed algorithm for generating all cutsets of a graph has been given by Ahmad [2]; this paper contains some other references from the reliability literature. A unifying framework for cutset problems is to be found in Provan and Shier [318].

Chapter 10

Random Selection

In cases where exhaustive enumeration is impractical it is often useful to generate a large number of objects at random. Unranking algorithms can be used but they suffer from a number of drawbacks

- They typically involve large integers and thus require the used of exended precision arithmetic packages.
- The bit complexity is usually $\Omega(n^2)$. This is because they require $\Omega(n)$ operations on integers of size exponential in n.

10.1 Permutations

We start with a simple example. There are many applications that require the production of random permutations, from simple programs that shuffle cards to sophisticated simulation routines. Let $rand(1, n)$ be a procedure that produces a integer selected uniformly at random from $\{1, 2, \ldots, n\}$. Here is a simple two line procedure to produce a random permutation of $1, 2, \ldots, n$:

for $i := 1$ **to** n **do** $\pi[i] := i$;
for $i := 2$ **to** n **do** $\pi[i] :=: \pi[rand(1, i)]$;

The first line sets π to be the identity permutation; any other permutation could have been used. Why does this work? We argue by induction on n. Note that the integer n does not move until the nth iteration of the second loop. Thus we may assume that the procedure produces a random permutation $\pi_1, \pi_2, \ldots, \pi_n$ of $1, 2, \ldots, n-1$ in the first $n-2$ iterations; i.e., that each permutation of $1, 2, \ldots, n-1$ has probability $1/(n-1)!$ of occuring. In the final iteration π_n then gets set to a value, say k, with probability $1/n$. Thus, each permutation of $1, 2, \ldots, n$ has probability $1/(n-1)! \cdot 1/n = 1/n!$ of occuring.

Note that stopping the second loop at k instead of n will produce a random k-permutation of an n-set. Also note that the first line could set π to be any permutation of $1, 2, \ldots, n$.

10.2 Combinations

The problem of producing a random subset S of $\{1, 2, \ldots, n\}$ is easily solved by adding each i to S with probability $1/2$ for $i = 1, 2, \ldots, n$. Similarly, it is trivial to produce a random

element of a product space uniformly at random.

The production of a random combination is considerably more subtle. Here's a very elegant algorithm due to Knuth. Why does this algorithm work? **!!! Is this really wothwhile presenting??**

Algorithm 10.1 is another very elegant algorithm, this one due to Floyd.

```
function Sample ( n, k ) : Set;
if k = 0 then return( ∅ )
else begin
      S := Sample(n − 1, k − 1);
      t := rand(1, n);
      if t ∉ S then S := S ∪ {t} else S := S ∪ {n}
      return( S );
end;
```

Algorithm 10.1: Generate a random k-subset of $[n]$.

Why does this algorithm work?

The running time of Floyd's algorithm will depend upon how the set S is maintained. There are a number of options that might be considered:

- One could simply use a Boolean array $b[1..n]$, where $b[i]$ is true if and only if $i \in S$. The running time is $O(n)$ since that is the time it takes to initialize the array.
- If k is small compared with n, then $O(n)$ is an unacceptable running time. There is a tricky technique whereby an array can be used without intializing the array, which would reduce the time to $O(k)$, but the storage requirement is still $O(n)$.
- A "dictionary" data stucture could be used; that is, a data structure that supports the operations of *Insert* and *Member*. Assuming that we also wish to output the items in order, various balanced tree schemes seem appropriate. Then the running time is $O(k \log k)$ and the storage is $O(k)$.

10.3 Permutations of a Multiset

We can adapt the ideas of the previous section to generate the permutation of a multiset of specification $\langle n_0, n_1, \ldots, n_t \rangle$.

10.4 Necklaces

Let us first consider the problem of generating a Lyndon word uniformly at random. The idea is to generate a random word and check it for aperiodicity as in the following probabilistic algorithm:

repeat
    w := random element of Σ_k^n;
until w is aperiodic;

Since the ratio of aperiodic words to all words is asymptotically one, the expected number of iterations is also one. To produce a random necklace we need to weight the divisors of n by ???

10.5 Numerical Partitions

From sci.math.research April 1998:

```
In article <6hksce$upk$1@isn.dac.neu.edu>, meleis@coe.neu.edu (Waleed Meleis) writes:
|> Hello, I'm looking for an efficient way to compute a random partition of a
|> fixed integer n such that every distinct partition is equally likely.

Start by precomputing the number p_k(m) of partitions of m with each term <= k,
for m = 1 .. n and k = 1 .. m.  This can be obtained by the recursion

p_k(m) = sum_j = 0^floor(m/k) p_k-1(m-jk) for k <= m

(noting that p_k(m) = p_m(m) for k > m).  In the process, store the
partial sums s(k,m,l) = sum_j = 0^l p_k-1(m-jk) for k <= m, l <= m/k.
This computation takes O(n^2 ln(n)) arithmetic operations.

Now the following pseudocode procedure produces a random partition of m
with each term <= k, where m <= n.  So to get a random partition of n you
would use randpart(n,n).

function randpart(m,k: integer): partition;
  if k > m then return randpart(m,m)
  else if k = 1 then return m 1's
  else
    choose j in 0, 1, ..., floor(m/k) with probabilities
         P(j) = p_k-1(m-jk)/p_k(m);
    return j k's + randpart(m-jk, k-1)
  end

Note that the step of choosing j can be done as follows: generate a random
number X in [0,1] and use binary search to find the least j so that
s(k,m,j) >= X p_k(m).

Robert Israel                              israel@math.ubc.ca
Department of Mathematics                (604) 822-3629
University of British Columbia               fax 822-6074
Vancouver, BC, Canada V6T 1Z2
```

10.6 Set Partitions

10.7 Trees

10.7.1 Well-formed Parentheses

The Cycle Lemma can be used to generate a random well-formed parenthesis string.

10.8 Graphs

10.9 Exercises

1. [1] Show that if the interchange of the algorithm for generating a random permutation is changed to $\pi[i] :=: \pi[rand(n)]$, then the procedure no longer produces random permutations (for $n \geq 3$).

2. [1+] Show how to compute a random derangement of $[n]$ in $O(n)$ arithmetic operations.

10.10 Bibliographic Remarks

The algorithm for generating a random permutation is due to Durstenfeld [91].

Floyd's algorithm for generating random combinations is described in Bentley [26].

Constant time initialization of arrays is discussed in Lewis and Denenberg [246].

Wilf [450] discusses the random selection of free tress. Tinhofer [412] presents a survey of methods for generating various types of trees and graphs uniformly at random.

The following papers discuss the random generation of binary trees: Arnold and Sleep [12], Remy [341] (in $O(n)$ time),Atkinson and Sack [13], Johnsen [196], Korsh [229], and Martin and Orr [265]. Furnas [145] discusses the random generation of binary *rooted* trees. Atkinson and Sack [15] develop a $O(n^3 h)$ algorithm to generate all binary trees on n nodes of height h. Atkinson and Sack [14] discuss the generation of combinatorial objects in parallel.

Dixon and Wilf [87] generate unlabelled graphs uniformly at random. Wormald [460] discusses the random selection of regular graphs; for moderate degrees a superior algorithm is given in McKay and Wormald [274].

The number of non-singular n-by-n matrices over a finite field of ρ elements is

$$(\rho^n - 1)(\rho^n - \rho) \cdots (\rho^n - \rho^{n-1}).$$

Randall [330] shows how to efficiently select such a matrix at random.

Broder [42] contains an algorithm for selecting a random spanning tree.

Flajolet, Zimmerman and van Cutsem [131] develop a systematic approach to the random generation of labelled combinatorial objects.

Sinclair's book [388] contains much material on generating combinatorial objects at random.

Other useful references include the book of Devroye [82] on generating non-uniform random variables, and Hickey and Cohen [177] on generating random strings in a context free language. Also, Lapointe and Legendre [239].

In *Generating Triangulations at Random* (Proceedings of the 4th Canadian Conference on Computational Geometry, pp.305-321) Epstein and Sack give an $O(n^4)$ algorithm for generating a triangulation of a polygon with n vertices, uniformly at random.

Given as input a regular language L as specified by a *non-deterministic* finite automaton or a regular expression, Kannan, Sweedyk, and Mahaney [204] present a $n^{O(\lg n)}$ expected time algorithm for generating a random string in L of length n.

Bibliography

[1] U. Abel and R. Bicker, *Determination of all minimal cut-sets between a vertex pair in an undirected graph*, IEEE Trans. Reliability **R-31** (1982), 167–171.

[2] S. Hasanuddin Ahmad, *Enumeration of minimal cutsets of an undirected graph*, Microelectronics and Reliability **30** (1990), 23–26.

[3] M. Aigner, *Combinatorial theory*, Springer-Verlag, 1979.

[4] Selim G. Akl, *A new algorithm for generating derangements*, BIT **20** (1980), 2–7.

[5] ———, *A comparison of combination generation methods*, ACM Transactions on Mathematical Software **7** (1981), 42–45.

[6] ———, *The design and analysis of parallel algorithms*, Prentice-Hall, 1989.

[7] L. Allison, *Generating coset representatives for permutation groups*, Journal of Algorithms **2** (1981), 227–244.

[8] B. Alspach, J.-C. Bermond, and D. Sotteau, *Decomposition into cycles I: Hamilton decompositions*, Cycles and Rays, Kluwer Academic Publishers, 1990, pp. 9–18.

[9] Brian Alspach, *The search for long paths and cycles in vertex-transitive graphs and digraphs*, Combinatorial Mathematics VIII, Lecture Notes in Mathematics #884 (K.L. McAvency, ed.), Springer, Berlin, 1981, pp. 14–22.

[10] F.S. Annexstein and E.A. Kuchko, *A ranking algorithm for Hamilton paths in shuffle-exchange graphs*, Proc. 4th Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science, vol. 955, Springer-Verlag, 1995, pp. 263–269.

[11] Benjamin Arazi, *An approach for generating different types of Gray codes*, Information and Control **63** (1984), 1–10.

[12] D.B. Arnold and M.R. Sleep, *Uniform random number generation of n balanced parenthesis strings*, ACM Transactions on Programming Languages and Systems **2** (1980), 122–128.

[13] M.D. Atkinson and J.R. Sack, *Generating binary trees at random*, Information Processing Letters **41** (1992), 21–23.

[14] ———, *Uniform generation of combinatorial objects in parallel*, Journal of Parallel and Distributed Computing **??** (1993), to appear.

[15] ———, *Uniform generation of forests of restricted height*, Information Processing Letters **50** (1994), 323–327.

[16] P. Auger and T.R. Walsh, *Theoretical and experimental comparison of four binary tree generation algorithms*, Congressus Numerantium **93** (1993), 99–109.

[17] D. Avis and K. Fukuda, *A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra*, Discrete and Computational Geometry **8** (1992), 295–313.

[18] ———, *Reverse search for enumeration*, Discrete Applied Mathematics **65** (1992), 21–46.

[19] M.O. Ball and J.S. Provan, *Computing bounds on reachability and connectedness in stochastic networks*, Networks **13** (1983), 253–278.

[20] Vinnakota Bapiraju and V.V. Bapeswara Rao, *Enumeration of binary trees*, Information Processing Letters **51** (1994), 125–127.

[21] A.M. Baraev and I.A. Faradjev, *Construction and study by computer of regular and regular bipartite graphs*, in: Algorithmic studies in combinatorics **(In Russian)** (1978), Moscow.

[22] J.A. Bate and D.M. Miller, *Fault detection in CMOS circuits and an algorithm for generating Eulerian cycles in directed hypercubes*, Congressus Numerantium **47** (1985), 107–117.

[23] B. Bauslaugh and F. Ruskey, *Generating alternating permutations lexicographically*, BIT **30** (1990), 17–26.

[24] Mounir Belbaraka and Ivan Stojmenović, *On generating b-trees with constant average delay and in lexicographic order*, Information Processing Letters **49** (1994), 27–32.

[25] M. Bellmore and P.A. Jensen, *An implicit enumeration scheme for proper cut generation*, Technometrics **12** (1970), 775–788.

[26] Jon Bentley, *More programming pearls*, Addison-Wesley, 1988.

[27] I. Berger, *The enumeration of all trees without duplication*, IEEE Transactions on Circuit Theory **CT-14** (1967), 417–418.

[28] E.R. Berlekamp, J.H. Conway, and R.K. Guy, *Winning ways for your mathematical plays*, Academic Press, New York, 1982.

[29] Jean Berstel and Michel Pocchiola, *Average cost of Duval's algorithm for generating Lyndon words*, Theoretical Computer Science **132** (1994), 415–425.

[30] T. Beyer and S.M. Hedetniemi, *Constant time generation of rooted trees*, SIAM J. Computing **9** (1980), 706–712.

[31] G.S. Bhat and C.D. Savage, *Balanced Gray codes*, Electronic Journal of Combinatorics **3** (1996), #R25, 11pp.

[32] J.R. Bitner, G. Ehrlich, and E.M. Reingold, *Efficient generation of the binary reflected Gray code and its applications*, Communications of the ACM **19** (1976), 517–521.

[33] J.R. Bitner and E.M. Reingold, *Backtrack programming techniques*, Communications of the ACM **18** (1975), 651–656.

[34] A. Björner and G. Ziegler, *Introduction to greedoids*, Matroid Applications (Neil White, ed.), Cambridge, 1992.

[35] J.A. Bondy and U.S.R. Murty, *Graph theory with applications*, North-Holland, 1976.

[36] A. Bonnin and J.M. Pallo, *Sur la génération des arbres binaires bar les B-suites*, Discrete Mathematics **51** (1984), 111–117.

[37] J. Boothroyd, *Perm (Algorithm 6)*, Computer Bulletin **3** (1965), 104.

[38] ———, *Permutation of the elements of a vector (Algorithm 29)*, The Computer Journal **10** (1967), 310–311.

[39] ———, *Fast permutation of the elements of a vector (Algorithm 30)*, The Computer Journal **10** (1970), 311–312.

[40] P. Bratley, *Permutations with repetitions (Algorithm 306)*, Communications of the ACM **10** (1967), 450.

[41] Gunnar Brinkmann, *Loosers are easy to find* or *an easy and efficient method to generate regular graphs and related structures*, Tech. Report ???, University of Bielefeld, 1993.

[42] A.Z. Bröder, *Generating random spanning trees*, 30th Annual Symposium on Foundations of Computer Science **IEEE Press** (1989), 442–453.

[43] H. Brunn, *Uber verkettung*, Sitzungberichte der Bayerischer Akad. Wiss. Math-Phys. **22** (1892), 77–99.

[44] M. Buck and D. Wiedemann, *Gray codes with restricted density*, Discrete Mathematics **48** (1984), 163–171.

[45] A. Bultena and F. Ruskey, *Transition restricted Gray codes*, Electronic Journal of Combinatorics **3** (1996), #R11 (?? pages).

[46] P. Buneman and L. Levy, *The towers of Hanoi problem*, Information Processing Letters **10** (1980), 243–244.

[47] Y. Cai and M.C. Kong, *Generating all maximal cliques and related problems for certain perfect graphs*, Congressus Numerantium **90** (1992), 33–55.

[48] R.D. Cameron, C.J. Colbourn, R.C. Read, and N.C. Wormald, *Cataloguing the graphs on 10 vertices*, Journal of Graph Theory **9** (1985), 551–562.

[49] R. Canfield and S.G. Williamson, *A loop-free algorithm for generating the linear extensions of a poset*, Order **12** (1995), 57–75.

[50] J.J. Canon, L.A. Dimino, G. Havas, and J.M. Watson, *Implemenation and analysis of the Todd-Coxeter algorithm*, Mathematics of Computation **123** (1973), 463–490.

[51] M. Carkeet and P. Eades, *Performance of subset generating algorithms*, Annals of Discrete Math. **26** (1985), 49–58.

[52] C.C. Chang, R.C.T. Lee, and M.W. Du, *Symbolic Gray code as a perfect multiattribute hashing scheme for partial match queries*, IEEE Trans. Software Engineering **8** (1982), 235–249.

[53] Y.H. Chang, J.S. Wang, and R.C.T. Lee, *Generating all maximal independent sets on trees in lexicographic order*, Information Sciences **76** (1994), 279–296.

[54] J.P. Char, *Generation of trees, two-trees, and storage of master forests*, IEEE Transactions on Circuit Theory **CT-15** (1968), 228–238.

[55] P.J. Chase, *Algorithm 383: Permutations of a set with repetitions*, Communications of the ACM **13** (1970), 368–369.

[56] ———, *Combinations of m out of n objects*, Communications of the ACM **13** (1970), 368.

[57] ———, *Combination generation and Graylex ordering*, Congressus Numerantium **69** (1989), 215–242.

[58] C.C. Chen and N.F. Quimpo, *On strongly hamiltonian Abelian groups*, Combinatorial Mathematics VIII (Geelong 1980), Lecture Notes in Mathematics, vol. 884, Springer-Verlag, Berlin, 1981, pp. 23–34.

[59] W.K. Chen, *On directed trees and directed k-trees of a graph and their generation*, SIAM J. Applied Mathematics **14** (1966), 550–560.

[60] ———, *Applied graph theory*, North-Holland, 1976.

[61] B.V. Cherkasskiĭ, *A new algorithm for the generation of spanning trees*, Kibernetica **1** (1987), 85–89, English translation from the Russian: Cybernetics 23 (1987), no.1, 107-113.

[62] N. Chiba and T. Nishizeki, *Arboricity and subgraph listing algorithms*, SIAM J. Computing **14** (1985), 210–223.

[63] ———, *Planar graphs: Theory and algorithms*, North-Holland, 1988, Annals of Discrete Mathematics, vol. 32.

[64] T. Chinburg, C.D. Savage, and H.S. Wilf, *Combinatorial families that are exponentially far from being listable in gray code sequence*, Trans. Amer. Math. Soc. **351** (1999), 379–402.

[65] F.R.K. Chung, P. Diaconis, and R.L. Graham, *Universal cycles for combinatorial structures*, Discrete Mathematics **110** (1992), 43–59.

[66] Martin Cohn, *Walsh functions, sequency, and Gray codes*, SIAM J. Applied Mathematics **21(3)** (1971), 442–447.

[67] C.J. Colbourn and R.C. Read, *Orderly algorithms for graph generation*, Intern. J. Computer Math. **7** (1979), 167–172.

[68] L. Comtet, *Advanced combinatorics*, D. Reidel, 1974.

[69] J.H. Conway, N.J.A. Sloane, and A.R. Wilks, *Gray codes for reflection groups*, Graphs and Combinatorics **5** (1989), 315–325.

[70] Thomas H. Cormen, Charles E. Leiserson, and Thomas H. Cormen, *Introduction to algorithms*, McGraw Hill, 1989.

[71] T.M. Cover, *Enumerative source encoding*, IEEE Trans. Information Theory **19** (1973), 73–77.

[72] R.R. Coveyou and J.G. Sullivan, *Permutation (algorithm 71)*, Communications of the ACM **4** (1961), 497.

[73] J.C. Culberson and G.J.E. Rawlins, *New results from an algorithm for counting posets*, Order **7** (1991), 361–374.

[74] L.J. Cummings, *Gray paths of Lyndon words in the N-cube*, Congressus Numerantium **69** (1989), 199–206.

[75] ———, *Gray codes and strongly square-free strings*, U. Waterloo, 1993.

[76] L.J. Cummings and M.E. Mays, *On the parity of the Witt formula*, Congressus Numerantium **80** (1991), 49–56.

[77] R.L. Cummins, *Hamilton circuits in tree graphs*, IEEE Transactions on Circuit Theory **CT-13** (1966), 82–90.

[78] S.J. Curran and J.A. Gallian, *Hamiltonian cycles and paths in Cayley graphs and digraphs — a survey*, Discrete Mathematics **??** (199?), submitted?

[79] E. Damiani, O. D'Antona, and G. Naldi, *On the connection constants*, Studies in Applied Mathematics **85** (1991), 289–302.

[80] N. Dershowitz, *A simplified loop-free algorithm for generating permutations*, BIT **15** (1975), 158–164.

[81] N. Dershowitz and S. Zaks, *Enumerations of ordered trees*, Discrete Mathematics **31** (1980), 9–28.

[82] L. Devroye, *Non-uniform random variate generation*, Springer Verlag, 1986.

[83] A. Dewdney, *Ying and yang: recursion and iteration, the Tower of Hanoi and the Chinese rings*, Scientific American **251** (1984), 19–28.

[84] P. Diaconis and S. Holmes, *Gray codes for randomization procedures*, Tech. Report No. 444, Department of Statistics, Stanford University, 1994.

[85] Dillencourt and Smith, *???*, Proc. 6th Canadian Conference on Computational Geometry, ACM Press, 1994, pp. ???–???

[86] E.A. Dinits and M.A. Zaitzev, *Algorithm for the generation of nonisomorphic trees*, Avtomat. i Telemekh. (Automat. Remote Control) **4 (38)** (1977), 121–126 (554–558).

[87] J.D. Dixon and Herbert S. Wilf, *The random selection of unlabelled graphs*, Journal of Algorithms **4** (1983), 205–213.

[88] B. Djokić, M. Miyakawa, S. Sekiguchi, I. Semba, and I. Stojmenović, *A fast iterative algorithm for generating set partitions*, The Computer Journal **32** (1989), 281–282.

[89] R.J. Douglas, *Bounds on the number of Hamiltonian circuits in the n-cube*, Discrete Mathematics **17** (1977), 143–146.

[90] R.J. Duffin, *An analysis of the Wang algebra of a network*, Trans. American Math. Society **93** (1959), 114–131.

[91] R. Durstenfeld, *Random permutation (algorithm 235)*, Communications of the ACM **7** (1964), 420.

[92] J.-P. Duval, *Factorizing words over an ordered alphabet*, Journal of Algorithms **4** (1983), 363–381.

[93] ———, *Génération d'une section des classes de conjugaison et abre des mots de Lyndon de longueur bornée*, Theoretical Computer Science **60** (1988), 255–283.

[94] S. Dvořák, *Combinations in lexicographic order*, The Computer Journal **33** (1990), 188.

[95] P. Eades, M. Hickey, and R.C. Read, *Some Hamilton paths and a minimal change algorithm*, Journal of the ACM **31** (1984), 19–29.

[96] P. Eades and Brendan D. McKay, *An algorithm for generating subsets of a fixed size with a strong minimal change property*, Information Processing Letters **19** (1984), 131–133.

[97] B.C. Eaves, *Permute (Algorithm 130)*, Communications of the ACM **11** (1962), 551.

[98] Anthony Edwards, *Pascal's arithmetical triangle*, Oxford University Press, 1987.

[99] A. Ehrenfeucht, J. Haemer, and D. Haussler, *Quasi-monotonic sequences: algorithms and applications*, SIAM Journal on Algebraic and Discrete Methods **8** (1987), 410–429.

[100] Gideon Ehrlich, *Algorithm 466: Four combinatorial algorithms*, Communications of the ACM **16** (1973), 690–691.

[101] ———, *Loopless algorithms for generating permutations, combinations and other combinatorial configurations*, Journal of the ACM **20** (1973), 500–513.

[102] T.C. Enns, *Hamiltonian circuits and paths in subset graphs with circular adjacency*, Discrete Mathematics **122** (1993), 153–165.

[103] David Eppstein, *Arboricity and bipartite subgraph listing algorithms*, Information Processing Letters **51** (1994), 207–211.

[104] P. Erdös and L.A. Szekely, *Application of antilexicographic order I. An enumerative theory of trees*, Advances in Applied Mathematics **10** (1989), 488–496.

[105] M.C. Er, *A note on generating well-formed parenthesis strings lexicographically*, The Computer Journal **26** (1983), 205–207.

[106] ———, *On generating the N-ary reflected Gray codes*, IEEE Transactions on Computers **C-33** (1984), 739–741.

[107] ———, *Enumerating ordered trees lexicographically*, The Computer Journal **28** (1985), 538–542.

[108] ———, *Efficient generation of binary trees from inorder-postorder sequences*, Information Sciences **40** (1986), 175–181.

[109] ———, *Classes of admissible permutations that are generatable by depth-first traversals of ordered trees*, The Computer Journal **32** (1988), 76–??

[110] ———, *A fast algorithm for generating set partitions*, The Computer Journal **31** (1988), 283–284.

[111] ———, *A simple algorithm for generating non-regular trees in lexicographic order*, The Computer Journal **31** (1988), 61–64.

[112] ———, *Efficient generation of k-ary trees in natural order*, The Computer Journal **35** (1992), 306–308.

[113] M. Erne and K. Stege, *Counting finite posets and topologies*, Order **8** (1991), 247–265.

[114] O. Eğecioğlu and J.B. Remmel, *Bijections for Cayley trees, spanning trees, and their q-analogues*, Journal of Combinatorial Theory, Series A **42** (1986), 15–30.

[115] Shimon Even, *Algorithmic combinatorics*, MacMillan, New York, 1973.

[116] Da Zhong Fang, *Enumeration of all spanning trees of an unndirected graph*, J. Tianjin Univ. **4** (1988), 108–116, (In Chinese with English summary).

[117] I.A. Faradjev, *Constructive enumeration of combinatorial objects*, in: Problèmes Combinatoires et Théorie des Graphes Colloque. Internat. **CNRS 260, Paris** (1978), 131–135.

[118] R.J. Faudree and R.H. Schelp, *The square of a block is strongly path connected*, J. of Combinatorial Theory **20** (1976), 47–61.

[119] R. Feldman and P. Mysliwietz, *The shuffle exchange network has a Hamilton path*, Proc. 17th Mathematical Foundations of Computer Science Conference, Lecture Notes in Computer Science, vol. 629, Springer-Verlag, 1992, pp. 246–254.

[120] T.I. Fenner and G. Loizou, *A binary tree representation and related algorithms for generating integer partitions*, The Computer Journal **23** (1980), 332–337.

[121] ———, *An analysis of two related loop-free algorithms for generating integer partitions*, Acta Informatica **16** (1981), 237–252.

[122] ———, *Tree traversal related algorithms for generating integer partitions*, SIAM J. Computing **12** (1983), 551–564.

[123] W. Feussner, *Über stromverzweigung in netzförmigen leitern*, Ann. Physik **9** (1902), 1304–1329.

[124] ———, *Zur berechnung der stromstärke in netzförmigen leitern*, Ann. Physik **15** (1904), 385–394.

[125] C.T. Fike, *A permutation generation method*, The Computer Journal **18** (1975), 21–22.

[126] James Allen Fill and Edward M. Reingold, *Solutions manual for combinatorial algorithms: Theory and practice*, Prentice-Hall, 1977.

[127] J.P. Fillmore and S.G. Williamson, *On backtracking: a combinatorial description of the algorithm*, SIAM J. Computing **3** (1974), 41–55.

[128] ———, *On ranking functions: The symmetries and colorations of the n-cube*, SIAM J. Computing **5** (1975), 297–304.

[129] L.L. Fischer and K.C. Krause, *Lehrbuch der combinationslehre und der arithmetik*, Dresden, 1812.

[130] P. Flajolet and L. Ramshaw, *A note on Gray code and odd-even merge*, SIAM J. Computing **9** (1980), 142–158.

[131] P. Flajolet, P. Zimmerman, and B. Van Cutsem, *A calculus for the random generation of combinatorial structures*, Theoretical Computer Science **132** (1994), no. 1-2, 1–35.

[132] H. Fleishner, *Eulerian graphs*, North-Holland, 19??

[133] ———, *A way of constructing and a way of counting all Eulerian trails in a connected Eulerian graph*, Tech. Report unpublished, ????, 19??

[134] ———, *The square of every two-connected graph is Hamiltonian*, Journal of Combinatorial Theory, Series B **16** (1974), 29–34.

[135] C. Flye-Sainte Marie, *Solution to problem number 58*, l'Intermediare des Mathematiciens **1** (1894), 107–110.

[136] S. Foldes, *A characterization of hypercubes*, Discrete Mathematics **17** (1977), 155–159.

[137] A.S. Fraenkel, *Systems of numeration*, American Mathematical Monthly **92** (1985), 105–114.

[138] ———, *The use and usefulness of numeration systems*, Information and Computation **81** (1989), 46–61.

[139] H. Fredricksen and I. J. Kessler, *An algorithm for generating necklaces of beads in two colors*, Discrete Mathematics **61** (1986), 181–188.

[140] H. Fredricksen and J. Maiorana, *Necklaces of beads in k colors and k-ary de Bruijn sequences*, Discrete Mathematics **23** (1978), 207–210.

[141] H. Fredricksen, *A survey of full length non-linear shift register cycle algorithms*, SIAM Review **24** (1982), 195–221.

[142] H. Fredricksen and I. J. Kessler, *Lexicographic compositions and de Bruijn sequences*, Journal of Combinatorial Theory **22** (1977), 17–30.

[143] K. Fukuda and M. Namiki, *Finding all common bases in two matroids*, Tech. Report Working paper No. 32, University of Tokyo, Komaba, Dept. of Social and International Relations, 1993.

[144] Komei Fukuda and Tomomi Matsui, *Finding all minimum-cost perfect matchings in bipartite graphs*, Networks **22** (1992), 461–468.

[145] G.W. Furnas, *The generation of random, binary unordered trees*, J. Classification **1** (1984), 187–233.

[146] H.N. Gabow, *Two algorithms for generating weighted spanning trees in order*, SIAM J. Computing **6** (1977), 139–150.

[147] H.N. Gabow and E.W. Myers, *Finding all spanning trees of directed and undirected graphs*, SIAM J. Computing **7** (1978), 280–287.

[148] Martin Gardner, *Knotted doughnuts (and other mathematical entertainments)*, W.H. Freeman, 1986.

[149] M.L. Gardner, *An algorithm to aid in the design of large scale networks*, Large Scale Systems **8** (1985), 147–156.

[150] E.N. Gilbert, *Gray codes and paths on the n-cube*, Bell Systems Technical Journal **37** (1958), 815–826.

[151] L.A. Goldberg, *Efficient algorithms for listing unlabeled graphs*, J. Algorithms **13** (1992), 128–143.

[152] ———, *Efficient algorithms for listing combinatorial structures*, Cambridge University Press, 1993.

[153] S.W. Golomb, *Permutations by cutting and shuffling*, SIAM Review **3** (1961), 293–297.

[154] ———, *Polyominoes*, Charles Scribner's Sons, New York, 1965.

[155] ———, *Irreducible polynomials, synchronization codes, primitive necklaces, and the cyclotomic algebra*, Combinatorial Mathematics and its Applications (1969), 358–370.

[156] S.W. Golomb and L.D. Baumert, *Backtrack programming*, Journal of the ACM **4** (1965), 516–524.

[157] I.P. Goulden and D.M. Jackson, *Combinatorial enumeration*, Wiley, 19??

[158] H.W. Gould, *Research bibliography of two number sequences*, Combinatorial Research Institute, Morgantown, W. Va., June 8, 1976.

[159] R.L. Graham, D.E. Knuth, and O. Patashnik, *Concrete mathematics : A foundation for computer science*, Addison-Wesley, 1989.

[160] F. Gray, *Pulse code communication*, U.S. Patent 2,632,058, March 17, 1953.

[161] T.A. Gulliver, M. Serra, and V.K. Bhargava, *The generation of primitive polynomials in GF(q) with independent roots and their applications for power residue codes, VLSI testing, and finite field multipliers using normal basis*, Int. J. Electronics **71** (1991), 559–576.

[162] U.I. Gupta, D.T. Lee, and C.K. Wong, *Ranking and unranking of 2-3 trees*, SIAM J. Computing **11** (1982), 582–590.

[163] ———, *Ranking and unranking of B-trees*, Journal of Algorithms **4** (1983), 51–60.

[164] Dan Gusfield and Robert W. Irving, *The stable marriage problem*, The MIT Press, 1989.

[165] Michel Habib, Lhouari Nourine, and George Steiner, *Gray codes for the ideals of interval orders*, Journal of Algorithms **25** (1997), 52–66.

[166] S.L. Hakimi, *On trees of a graph and their generation*, Journal of the Franklin Institute **272** (1961), 347–359.

[167] S.L. Hakimi and P.G. Green, *???*, IEEE Transactions on Circuit Theory **11** (1964), 247–255.

[168] M. Hall and D.E. Knuth, *Combinatorial analysis and computers*, American Mathematics Monthly **2** (1972), 21–28.

[169] Kazuaki Harada, *Generation of rosary permutations expressed in Hamiltonian circuits*, Communications of the ACM **14** (1971), 373–379.

[170] F. Harary, J. Hayes, and H.-J. Wu, *Survey of hypercubes*, Computer and Mathematics with Applications **15** (1988), 277–289.

[171] L.H. Harper, *Optimal assignments of numbers to vertices*, SIAM J. **12** (1964), 131–135.

[172] ———, *Optimal numberings and isoperimetric problems on graphs*, J. Combinatorial Theory **1** (1966), 385–393.

[173] I. Havel, *On Hamiltonian circuits and spanning trees of hypercubes*, Časopis Pěst. Mat. **109** (1984), 135–152.

[174] B.R. Heap, *Permutations by interchanges*, The Computer Journal **6** (1963), 293–294.

[175] F.G. Heath, *Origins of the binary code*, Scientific American **227** (1972), 76–83.

[176] G. Hendry and W. Volger, *The square of a connected $S(K_{1,3})$-free graph is vertex pancyclic*, J. Graph Theory **9** (1985), 535–537.

[177] T. Hickey and J. Cohen, *Uniform random generation of strings in a context-free language*, SIAM J. Computing **12** (1983), 645–655.

[178] T. Hikita, *Listing and counting subtrees of equal size of a binary tree*, Information Processing Letters **17** (1983), 225–229.

[179] C.A. Holzmann and F. Harary, *On the tree graph of a matroid*, SIAM J. Applied Math. **22** (1972), 187–193.

[180] T. Hough and F. Ruskey, *An efficient implementation of the Eades, Hickey, Read adjacent interchange combination generation algorithm*, J. Combinatorial Mathematics and Combinatorial Computing **4** (1988), 79–86.

[181] J.R. Howell, *Generation of permutations by addition*, Math. Comp. **16** (1962), 243–244.

[182] _____, *Permutation generator (algorithm 87)*, Communications of the ACM **5** (1962), 452–453.

[183] S.S. Huang and D. Tamari, *Problems of associativity: A simple proof for the lattice property of systems ordered by a semi-associative law*, Journal of Combinatorial Theory, Series A **13** (1972), 7–13.

[184] Y. Huang, *A new algorithm for the generation of binary de Bruijn sequences*, Journal of Algorithms **11** (1990), 44–51.

[185] T.C. Hu and B.N. Tien, *Generating permutations with non-distinct items*, American Mathematical Monthly **83b** (1976), 629–631.

[186] G. Hurlbert, *On universal cycles for k-subsets of an n-set*, SIAM J. Discrete Mathematics **7** (1994), 594–604.

[187] _____, *Multicover ucycles*, Discrete Mathematics **137** (1995), 241–249.

[188] A. Itai and M. Rodeh, *Finding a minimum circuit in a graph*, SIAM J. Computing **7** (1978), 413–423.

[189] F.M. Ives, *Permutation enumeration: Four new permutation algorithms*, Communications of the ACM **19** (1976), 68–72.

[190] M.E. Jablonski, *Théorie des permutations et des arrangements circulaires complets*, Journal de Mathématiques pures et appliqueés, fondé par Joseph Liouville **8** (1892), 331–349.

[191] B. Jackson, *Universal cycles for k-subsets and k-permutations*, Discrete Mathematics **117** (1993), 141–150.

[192] K.R. James and W. Riha, *Algorithm for generating graphs of a given partition*, Computing **16** (1976), 153–161.

[193] R. Jayakumar, K. Thulasiraman, and M.N.S. Swamy, *Complexity of computation of a spanning tree enumeration algorithm*, IEEE Trans. Circuits and Systems **31** (1984), 853–860.

[194] T.A. Jenkyns and D. McCarthy, *Generating all k-subsets of* $\{1, 2, \ldots, n\}$ *with minimum changes*, ars **40** (1995), 153–159.

[195] M. Jiang and F. Ruskey, *Determining the Hamilton-connectedness of certain vertex-transitive graphs*, Discrete Mathematics **133** (1994), 159–169.

[196] B. Johnsen, *Generating binary trees with uniform probability*, BIT **31** (1991), 15–31.

[197] Donald B. Johnson, *Finding all the elementary circuits of a directed graph*, SIAM J. Computing **4** (1975), 77–84.

[198] S.M. Johnson, *Generation of permutations by adjacent transpositions*, Math. Computation **17** (1963), 282–285.

[199] J.T. Joichi and D.E. White, *Gray codes in graphs of subsets*, Discrete Mathematics **31** (1980), 29–41.

[200] J.T. Joichi, D.E. White, and S.G. Williamson, *Combinatorial Gray codes*, SIAM J. Computing **9** (1980), 130–141.

[201] A.D. Kalvin and Y.L. Varol, *On the generation of all topological sortings*, Journal of Algorithms **4** (1983), 150–162.

[202] T. Kamae, *The existence of a Hamilton circuit in a tree graph*, IEEE Transactions on Circuit Theory **CT-14** (1967), 279–283.

[203] Arkady Kanevsky, *On the number of minimum size separating vertex sets in a graph and how to find all of them*, Proc. 1st ACM-SIAM Symposium on Discrete Algorithms, ACM Press, 1990, pp. 411–421.

[204] Sampath Kannan, Z. Sweedyk, and Steve Mahaney, *Counting and random generation of strings in regular languages*, Proc. 6th ACM-SIAM Symposium on Discrete Algorithms, ACM Press, 1995, pp. 551–557.

[205] S. Kapoor and H. Ramesh, *Algorithms for generating all spanning trees of undirected, directed and weighted graphs*, Lecture Notes in Computer Science **??** (1992), 461–472.

[206] Jerome J. Karaganis, *On the cube of a graph*, Canadian Mathematical Bulletin **11** (1968), 295–296.

[207] Toshinobu Kashiwabara, Sumio Masuda, Kazuo Nakajima, and Toshio Fujisawa, *Generation of maximum independent sets of a bipartite graph and maximum cliques of a circular-arc graph*, Journal of Algorithms **13** (1992), 161–174.

[208] R.A. Kaye, *A Gray code for set partitions*, Info. Proc. Lett. **5** (1976), 171–173.

[209] Pierre Kelsen, *Constant time generation of B-trees*, unpublished manuscript, 1987.

[210] P. Kirschenhofer and H. Prodinger, *Subblock occurances in positional number systems and Gray code representation*, J. Inform. Optim. Sci. **5** (1984), 29–42.

[211] G. Kishi and Y. Kajitani, *On Hamiltonian circuits in tree graphs*, IEEE Transactions on Circuit Theory **CT-15** (1968), 42–50.

[212] _____, *On the realization of tree graphs*, IEEE Transactions on Circuit Theory **CT-15** (1968), 271–273.

[213] M. Kleinert, *Die dicke des n-dimensionalen Würfel-graphen*, J. Combinatorial Theory **3** (1967), 10–15.

[214] P. Klingsberg, *A Gray code for compositions*, Journal of Algorithms **3** (1982), 41–44.

[215] G. D. Knott, *A numbering system for combinations*, Communications of the ACM **17** (1974), 45–46.

[216] ———, *A numbering system for permutations of combinations*, Communications of the ACM **19** (1976), 45–46.

[217] ———, *A numbering system for binary trees*, Comm. Assoc. Comput. Mach. **20** (1977), 113–115.

[218] D.E. Knuth, *Fundamental algorithms*, Addison-Wesley, 1973.

[219] ———, *Sorting and searching*, Addison-Wesley, 1973.

[220] ———, *Estimating the efficiency of backtracking programs*, Mathematics of Computation **29** (1975), 121–136.

[221] ———, *Lexicographic permutations with restrictions*, Discrete Applied Mathematics **1** (1979), 117–125.

[222] ———, *The stanford graphbase*, Addison-Wesley, 1994.

[223] D.E. Knuth and J. Szwarcfiter, *A structured program to generate all topological sorting arrangements*, Information Processing Letters **2** (1974), 153–157.

[224] Y. Koda, *Orderly algorithms for generating k-colored graphs*, Bulletin of the Institute of Combinatorics and Its Applications **91** (1993), 223–238.

[225] ———, *The numbers of finite lattices and finite topologies*, Congressus Numerantium **10** (1994), 83–89.

[226] Y. Koda and F. Ruskey, *A Gray code for the ideals of a forest poset*, Journal of Algorithms **15** (1993), 324–340.

[227] C.W. Ko and F. Ruskey, *Generating permutations of a bag by interchanges*, Information Processing Letters **41** (1992), 263–269.

[228] V.L. Kompel'makher and V.A. Liskovets, *Sequential generation of arrangements by means of a basis of transpositions*, Kibernetica **3** (1975), 17–21.

[229] J.F. Korsh, *Counting and randomly generating binary trees*, Information Processing Letters **45** (1993), 291–294.

[230] J.F. Korsh and S. Lipschutz, *Generating multiset permutations in constant time*, Journal of Algorithms **25** (1997), 321–335.

[231] Bernard Korte, László Lovász , and Rainer Schrader, *Greedoids*, Springer-Verlag, 1991.

[232] A. V. Kozina, *Coding and generation of nonisomorphic trees*, Cybernetics (Kibernetica) **15 (5)** (1975 (1979)), 645–651 (38–43).

[233] D.L. Kreher and D.R. Stinson, *Combinatorial algorithms: Generation, enumeration, and search*, CRC Press, 1999.

[234] E. Kubicka and G. Kubicki, *Constant time algorithm for generating binary rooted trees*, Congressus Numerantium **90** (1992), 57–64.

[235] Ewa Kubicka, *An efficient algorithm for examining all trees*, unpublished manuscript, 1993.

[236] J. Kurtzberg, *Algorithm 94: Combination.*, Communications of the ACM **5** (1962), 344.

[237] C.W.H. Lam and L.H. Soicher, *Three new combination algorithms with the minimal change property*, Communications of the ACM **25** (1982), 555–559.

[238] G.G. Langdon, Jr., *An algorithm for generating permutations*, Communications of the ACM **10** (1967), 298–299.

[239] F.-J. Lapointe and P. Legendre, *The generation of random ultrametric matrices representing dendograms*, J. Classification **8** (1991), 177–200.

[240] S. Latifi and S.Q. Zheng, *On Hamiltonian paths between two vertices in hypercube graphs*, Congressus Numerantium **89** (1992), 111–117.

[241] R. Laue, *Construction of combinatorial objects – a tutorial*, Bayreuther Math. Schriften **43** (1993), 53–96.

[242] C.C. Lee, D.T. Lee, and C.K. Wong, *Generating binary trees of bounded height*, Acta Informatica **23** (1986), 529–544.

[243] D.H. Lehmer, *Teaching combinatorial tricks to a computer*, Proc. Symp. Applied Mathematics, American Mathematical Society, 1960, pp. 179–193.

[244] ———, *The machine tools of combinatorics*, Applied Combinatorial Mathematics, Wiley, 1964, pp. 5–31.

[245] J.K. Lenstra and A.H.G. Rinnooy-Kan, *A recursive approach to the generation of combinatorial configurations*, Mathematisch Centrum, Amsterdam, 1975.

[246] Harry R. Lewis and Larry Denenberg, *Data structures & their algorithms*, HarperCollins Publishers, Inc., 1991.

[247] A.L. Liestman and T.C. Shermer, *Additive spanners for hypercubes*, Parallel Processing Letters **1** (1991), 35–42.

[248] G. Li and F. Ruskey, *The advantages of forward thinking in generating rooted and free trees*, 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), SIAM-ACM, 1999, pp. S939–940.

[249] Liwu Li, *Ranking and unranking of AVL trees*, SIAM J. Computing **15** (1986), 1025–1035.

[250] W. Lipski, *More on permutation generation methods*, Computing **23** (1979), 357–365.

[251] C.N. Liu and D.T. Tang, *Algorithm 452: Enumerating combinations of m out of n objects*, Communications of the ACM **16** (1973), 485.

[252] Gui Zhen Liu, *An algorithm for generating ordered forests lexicographically*, Chinese J. Operations Research **4** (1985), 65–66.

[253] M. Lothaire, *Combinatorics on words*, Addison-Wesley, Reading, MA, 1983.

[254] E. Loukakis, *A new backtracking algorithm for generating the family of maximal independent sets of a graph*, Computers and Mathematics with Applications **9** (1983), 583–589.

[255] J. Lucas, *The rotation graph of binary trees is Hamiltonian*, Journal of Algorithms **9** (1988), 503–535.

[256] J. Lucas, D. Roelants van Baronaigien, and F. Ruskey, *On rotations and the generation of binary trees*, Journal of Algorithms **15** (1993), 343–366.

[257] X.-M. Lu, *Towers of Hanoi problem with arbitrary $k \geq 3$ pegs*, International Journal of Computer Mathematics **24** (1988), 39–54.

[258] H. Lüneburg, *Gray codes*, Abh. Math. Sem. Univ. Hamburg **52** (1982), 208–227.

[259] ———, *On Dedekind numbers*, Lecture Notes in Mathematics (Combinatorial Theory) **696** (1982), 251–257.

[260] Heinz Lüneburg, *Tools and fundamental constructions of combinatorial mathematics*, BI Wissenschaftsverlag, Mannheim/Wein/Zurich, 1989.

[261] E. Mäkinen, *Left distance binary trees representations*, BIT **27** (1987), 163–169.

[262] ———, *Efficient generation of rotational-admissible codewords for binary trees*, The Computer Journal **34** (1991), 379.

[263] ———, *A survey of binary tree codings*, The Computer Journal **34** (1991), 438–443.

[264] George E. Martin, *Polyominoes: A guide to puzzles and problems in tiling*, MAA Spectrum, 1991.

[265] H.W. Martin and B.J. Orr, *A random binary tree generator*, Computing trends in the 1990's, ACM Seventeenth Computer Science Conference, ACM Press, 1989, Louisville, Kentucky, pp. 33–38.

[266] M.H. Martin, *A problem in arrangements*, Bulletin of the American Mathematical Society **40** (1934), 859–864.

[267] Dragan Marušič, *Hamiltonian circuits in Cayley graphs*, Discrete Mathematics **46** (1983), 49–54.

[268] D.W. Matula and L.L. Beck, *Smallest-last ordering and clustering and graph coloring algorithms*, Journal of the ACM **30** (1983), 417–427.

[269] W. Mayeda and S. Seshu, *Generation of trees without duplications*, IEEE Transactions on Circuit Theory **CT-12** (1965), 181–185.

[270] Brendan D. McKay, *Transitive graphs with fewer than 20 vertices*, Mathematics of Computation **33** (1979), 1101–1121.

[271] ———, *Isormorph-free exhaustive generation*, Journal of Algorithms **26** (1998), 306–324.

[272] Brendan D. McKay and G.F. Royle, *Constructing the cubic graphs on up to 20 vertices*, Ars Combinatoria **21A** (1986), 129–140.

[273] ———, *The transitive graphs with at most 26 vertices*, Ars Combinatoria **30** (1990), 161–176.

[274] Brendan D. McKay and Nicholas C. Wormald, *Uniform generation of random regular graphs of moderate degree*, Journal of Algorithms **11** (1990), 52–67.

[275] J.K.S. McKay, *Algorithm 262: Number of restricted partitions of n*, Communications of the ACM **8** (1965), 493.

[276] ———, *Algorithm 263: Partition generator*, Communications of the ACM **8** (1965), 493.

[277] ———, *Algorithm 264: Map of partitions into integers*, Communications of the ACM **8** (1965), 493.

[278] ———, *Algorithm 371: Partitions in natural order*, Communications of the ACM **13** (1970), 52.

[279] N. Metropolis and G-C. Rota, *Witt vectors and the algebra of necklaces*, Advances in Mathematics **50** (1983), 95–125.

[280] C.J. Mifsud, *Algorithm 154: Combination in lexicographic order*, Communications of the ACM **6** (1963), 103.

[281] W.H. Mills, *Some complete cycles on the n-cube*, Proceedings of the AMS **14** (1963), 640–643.

[282] S. Minsker, *The towers of Antwerpen problem*, Information Processing Letters **38** (1991), 107–111.

[283] G.J. Minty, *A simple algorithm of listing all the trees of a graph*, IEEE Transactions on Circuit Theory **CT-12** (1965), 120–???

[284] J. Misra, *Remark on algorithm 246*, ACM Transactions on Mathematical Software **1** (1975), 285.

[285] J.W. Moon, *Counting labelled trees*, Canadian Mathematical Mongraphs, No. 1, 1970.

[286] T. Mukherjee and P.K. Sarkar, *On connectedness and trees of an n-graph*, International Journal of Control **4** (1966), 465–498.

[287] W. Myrvold and F. Ruskey, *Ranking and unranking permutations in linear time*, Information Processing Letters **?** (2001), ???–???

[288] T.V. Narayana, R.M. Mathsen, and J. Saranji, *An algorithm for generating partitions and its applications*, Journal of Combinatorial Theory, Series A **11** (1971), 54–61.

[289] A. Nijenhuis and H.S. Wilf, *Combinatorial algorithms, 2nd ed.*, Academic Press, 1978.

[290] S.E. Orcutt, *Implementation of permutation functions in Illiac IV-type computers*, IEEE Transactions on Computers **C-25** (1976), 929–936.

[291] R.J. Ord-Smith, *Generation of permutations in pseudo-lexicographic order (algorithm 308)*, Communications of the ACM **10** (1967), 452.

[292] ———, *Generation of permutations in lexicographic order (algorithm 323)*, Communications of the ACM **2** (1968), 117.

[293] ———, *Generation of permutations in lexicographic order (algorithm 323)*, Communications of the ACM **11** (1968), 117.

[294] ———, *Generation of permutation sequences: Part 1*, The Computer Journal **13** (1970), 152–155.

[295] ———, *Generation of permutation sequences: Part 2*, The Computer Journal **14** (1971), 136–139.

[296] E.S. Page and L.B. Wilson, *An introduction to computational combinatorics*, Cambridge, 1979.

[297] J. Pallo, *Enumerating, ranking, and unranking binary trees*, The Computer Journal **29** (1986), 171–175.

[298] ———, *Generating trees with n nodes and m leaves*, International Journal of Computer Mathematics **16** (1987), 133–144.

[299] ———, *Some properties of the rotation lattice of binary trees*, The Computer Journal **31** (1988), 564–565.

[300] ———, *Lexicographic generation of binary unordered trees*, Pattern Recognition Letters **10** (1989), 217–221.

[301] ———, *On the listing and random generation of hybrid binary trees*, International Journal of Computer Mathematics **50** (1994), 135–145.

[302] J. Pallo and R. Racca, *A note on generating binary trees in A-order and B-order*, International Journal of Computer Mathematics **18** (1985), 27–39.

[303] C.H. Papadimitriou and M. Yannakakis, *The clique problem for planar graphs*, Information Processing Letters **13** (1981), 131–133.

[304] Ian Parberry, *Problems on algorithms*, Prentice-Hall, 1995.

[305] J.E.L. Peck and G.F. Schrak, *Permute (algorithm 86)*, Communications of the ACM **5** (1962), 208.

[306] C. Peter-Orth, *All solutions of the Soma cube puzzle*, Discrete Mathematics **57** (1985), 105–121.

[307] W.W. Peterson and E.J. Weldon, *Error-correcting codes*, MIT Press, Cambridge, MA, 1972.

[308] L.P. Petrenjuk and A.N. Petrenjuk, *On constructive enumeration of 12 vertex cubic graphs*, Combinatorial Analysis No. 3 **Moscow** (1974), in Russian.

[309] J.E.L. Phillips, *Permutation of the elements of a vector in lexicographic order (algorithm 28)*, The Computer Journal **10** (1967), 310–311.

[310] J.C. Picard and M. Queyranne, *On the structure of all minimum cuts in a network and applications*, Math. Prog. Study **13** (1980), 8–16.

[311] L. Pitt, *A note on extending Knuth's tree estimator to directed acyclic graphs*, Information Processing Letters **24** (1987), 203–206.

[312] S. Pleszczyński, *On the generation of permutations*, Information Processing Letters **3** (1975), 180–183.

[313] David G. Poole, *The towers and triangles of Professor Claus (or, Pascal knows Hanoi)*, Mathematics Magazine **67** (1994), 323–344.

[314] A. Proskurowski, *On the generation of binary trees*, Journal of the ACM **27** (1980), 1–2.

[315] A. Proskurowski and E. Laiman, *Fast enumeration, ranking, and unranking of binary trees*, Congressus Numerantium **35** (1980), 401–413.

[316] A. Proskurowski and F. Ruskey, *Binary tree Gray codes*, Journal of Algorithms **6** (1985), 225–238.

[317] A. Proskurowski, F. Ruskey, and M. Smith, *Analysis of algorithms for listing equivalence classes of k-ary strings induced by simple group actions*, SIAM J. Discrete Mathematics **11** (1998), 94–109.

[318] J.S. Provan and D.R. Shier, *A paradigm for listing (s,t)-cuts in graphs*, Tech. Report UNC/OR TR91-3, Univ. of North Carolina at Chapel Hill, 1991.

[319] G. Pruesse and F. Ruskey, *Generating the linear extensions of certain posets by transpositions*, SIAM J. Discrete Mathematics **4** (1991), 413–422.

[320] ———, *Gray codes from antimatroids*, Order **10** (1993), 239–252.

[321] ———, *Generating linear extensions fast*, SIAM J. Computing **23** (1994), 373–386.

[322] ———, *The prism of the acyclic orientation graph is hamiltonian*, Electronic Journal of Combinatorics **2** (1995), no. R5, (6pp).

[323] K. Pruhs, *The SPIN-OUT puzzle*, SIGCSE Bulletin **25** (1993), 36–38.

[324] Paul. W. Purdom, *Tree size by partial backtracking*, SIAM J. Computing **7** (1978), 481–491.

[325] D.F. Rall and P.J. Slater, *Generating all permutations by graphical derangements*, unpublished manuscript, < 1990.

[326] A. Ralston, *A new memoryless algorithm for de Bruijn sequences*, Journal of Algorithms **2** (1981), 50–62.

[327] P.V. Ramanan and C.L. Liu, *Permutation representation of k-ary trees*, Theoretical Computer Science **38** (1985), 83–98.

[328] A. Ramanathan and C.J. Colbourn, *Counting almost minimum cutsets with reliability applications*, Math. Prog. **39** (1987), 253–261.

[329] Mark Ramras, *A new method of generating Hamiltonian cycles on the n-cube*, Discrete Mathematics **85** (1990), 329–331.

[330] D. Randall, *Efficient generation of random nonsingular matrices*, Random Structures and Algorithms **4** (1993), 111–118.

[331] R.A. Rankin, *A campanological problem in group theory*, Proc. Cambridge Phil. Society **44** (1948), 17–25.

[332] E.S. Rapaport, *Cayley colour groups and Hamilton lines*, Scripta Math. **24** (1959), 51–58.

[333] D. Rasmussen, C.D. Savage, and D.B. West, *Gray code enumeration of families of integer partitions*, Journal of Combinatorial Theory, Series A **70** (1995), 201–229.

[334] R.C. Read, *A note on the generation of rosemary permutations*, Communications of the ACM **15** (1972), 775.

[335] ———, *Every one a winner or how to avoid isomorphism search when cataloguing combinatorial configurations*, Annals of Discrete Math. **2** (1978), 107–120.

[336] ———, *Orderly algorithms for generating restricted classes of graphs*, Journal of Graph Theory **3** (1979), 187–195.

[337] ———, *A survey of graph generation techniques*, Combinatorial Mathematics, VIII, Lecture Notes in Mathematics, vol. 844, Springer-Verlag, 1981, pp. 77–89.

[338] R.C. Read and R.E. Tarjan, *Bounds on backtracking algorithms for listing cycles, paths, and spanning trees*, Networks **5** (1975), 237–252.

[339] D.H. Redelmeier, *Counting polyominoes: yet another attack*, Discrete Mathematics **36** (1981), 191–203.

[340] E.M. Reingold, J. Nievergelt, and N. Deo, *Combinatorial algorithms: Theory and practice*, Prentice-Hall, 1977.

[341] J.L. Rémy, *Un procédé itératif dénombrement d'arbes binaires et son application à leur génération aléatoire*, R.A.I.R.O. Informatique Théorique **19** (1985), 179–195.

[342] Ch. Reutenauer, *Mots circulaires et polynomes irreductibles*, Tech. Report 87-39, CNRS Université Paris VII, Informatique Theoretique et Programmation, 1987.

[343] D. Richards, *Data compression and Gray-code sorting*, Information Processing Letters **22** (1986), 201–205.

[344] W. Riha and K.R. James, *Efficient algorithms for doubly and multiply restricted partitions*, Computing **16** (1976), 163–168.

[345] G. Ringel, *Über drei kombinatorische probleme am n-dimensionalen Würfel und Würfelgitter*, Abh. Math. Sem. Univ. Hamburg **20** (1955), 10–19.

[346] C.L. Robinson, *Permutation (algorithm 317)*, Communications of the ACM **10** (1967), 729.

[347] Jeffrey Rohl, *Recursion via pascal*, Cambridge University Press, 1984.

[348] J.S. Rohl, *Programming improvements to Fike's algorithm for generating permutations*, The Computer Journal **19** (1976), 156.

[349] ———, *Generating permutations by choosing*, The Computer Journal **21** (1978), 302–305.

[350] D. Rotem and Y. Varol, *Generation of binary trees from ballot sequences*, Journal of the ACM **25** (1978), 396–404.

[351] M.K. Roy, *Permutation generation methods*, The Computer Journal **21** (1978), 196–301.

[352] Gordon Royle, *An orderly algorithm and some applications in finite geometry*, Discrete Mathematics **185** (1999), 105–115.

[353] F. Ruskey, *Simple combinatorial Gray codes constructed by reversing sublist*, Proc. ISAAC Conference, Lecture Notes in Computer Science, vol. 762, Springer-Verlag, 1993, pp. 201–208.

[354] F. Ruskey and T.C. Hu, *Generating binary trees lexicographically*, SIAM J. Computing **6** (1977), 745–758.

[355] F. Ruskey and D.J. Miller, *Adjacent interchange generation of combinations and trees*, Tech. Report DCS-44-IR, U. Victoria, Dept. of Computer Science, 1984.

[356] F. Ruskey and A. Proskurowski, *Generating binary trees by transpositions*, Journal of Algorithms **11** (1990), 68–84.

[357] F. Ruskey, C. Savage, and T.M.Y. Wang, *Generating necklaces*, Journal of Algorithms **13** (1992), 414–430.

[358] F. Ruskey and C.D. Savage, *Gray codes for set partitions and restricted growth tails*, Austalasian J. Combinatorics **10** (1994), 85–96.

[359] F. Ruskey and J. Sawada, *An efficient algorithm for generating necklaces with fixed density*, SIAM J. Computing **29** (1999), 671–684.

[360] F. Ruskey and D. Roelants van Baronaigien, *Fast recursive algorithms for generating combinatorial objects*, Congressus Numerantium (1984), 53–62.

[361] Frank Ruskey, *Generating t-ary trees lexicographically*, SIAM J. Computing **6** (1977), 424–439.

[362] ———, *Algorithmic solution of two combinatorial problems*, Ph.D. thesis, Information and Computer Science, University of California at San Diego, 1978.

[363] ———, *Listing and counting subtrees of a tree*, SIAM J. Computing **10** (1981), 141–150.

[364] ———, *Adjacent interchange generation of combinations*, Journal of Algorithms **9** (1988), 162–180.

[365] ———, *Transposition generation of alternating permutations*, Order **6** (1989), 227–233.

[366] ———, *Generating linear extensions of posets by transpositions*, Journal of Combinatorial Theory, Series B **54** (1992), 77–101.

[367] Y. Saad and M.H. Sculls, *Topological properties of hypercubes*, IEEE Transactions on Computers **22** (1973), 176–180.

[368] T.W. Sag, *Permutations of a set with repetitions (Algorithm 242)*, Communications of the ACM **7** (1964), 585.

[369] C. Savage, *Gray code sequences of partitions*, Journal of Algorithms **10** (1989), 577–595.

[370] ———, *Generating permutations with k-differences*, SIAM J. Discrete Mathematics **3** (1990), 561–573.

[371] ———, *A survey of combinatorial Gray codes*, SIAM Review **39** (1997), no. 4, 605–629.

[372] C.D. Savage and P. Winkler, *Monotone Gray codes and the middle levels problem*, Journal of Combinatorial Theory, Series A **70** (1995), 230–248.

[373] J. Savage, *An efficient algorithm for generating bracelets*, SIAM J. Computing **??** (to appear), ???–???

[374] G.F. Schrak and M. Shimrat, *Permutation in lexicographic order (algorithm 102)*, Communications of the ACM **5** (1962), 346.

[375] F. Schuh, *Master book of mathematical recreations*, Dover, 1968.

[376] H.I. Scions, *Placing trees in lexicographic order*, Machine Intelligence **3** (1969), 43–60.

[377] R. Sedgewick, *Permutation generation methods*, Computing Surveys **9** (1977), 137–164.

[378] M. Sekanina, *On an ordering of the set of vertices of a connected graph*, Publication of the Faculty of Science, University of Brno **412** (1960), 137–142.

[379] I. Semba, *Generation of stack sequences in lexicographic order*, J. Information Processing **15** (1982), 17–20.

[380] ———, *An efficient algorithm for generating all partitions of the set* $\{1, 2, \ldots, n\}$, J. Information Processing **7** (1984), 41–42.

[381] M. Serra, *Tables of irreducible and primitive polynomials for GF(3)*, Tech. Report DCS-??-IR, U. Victoria, Dept. of Computer Science, 1986.

[382] B.D. Sharma and R.K. Khanna, *!!!! unknown !!!!*, J. Combin. Inform. System Sci. **4** (1979), 227–236.

[383] ———, *On level weight studies of binary and m-ary Gray codes*, Information Sciences **21** (1980), 179–186.

[384] Bhu Dev Sharma and Ravinder Kumar Khanna, *On m-ary Gray codes*, Information Sciences **15** (1978), 31–43.

[385] Mok-kong Shen, *On the generation of permutations and combinations*, BIT **2** (1962), 228–231.

[386] ———, *Generation of permutations in lexicographic order*, Communications of the ACM **6** (1963), 517.

[387] A. Shioura and A. Tamura, *Efficiently scanning all spanning trees of an undirected graph*, Tech. Report B-270, Tokyo Institute of Technology, Dept. of Information Sciences, 1993.

[388] Alistair Sinclair, *Algorithms for random generation and counting*, Birkhauser, 1992.

[389] W. Skarbek, *Generating ordered trees*, Theoretical Computer Science **57** (1988), 153–159.

[390] Steven Skiena, *Implementing discrete mathematics*, Addison-Wesley, 1990.

[391] P.J. Slater, *Generating all permutations by graphical transpositions*, Ars Combinatoria **5** (1978), 219–225.

[392] D. Sleator, R.E. Tarjan, and W. Thurston, *Rotation distance, triangulations, and hyperbolic geometry*, J. AMS **1** (1988), 647–682.

[393] D.H. Smith, *Hamiltonian circuits on the n-cube*, Canadian Mathematical Bulletin **17** (1975), 759–761.

[394] M. Smith, *Generating spanning trees*, Masters thesis, Dept. of Computer Science, University of Victoria, 1994.

[395] M. Solomon and R. Finkel, *A note on enumerating binary trees*, J. Assoc. Comput. Mach. **27** (1980), 3–5.

[396] R. Sosic and J. Gu, *A polynomial time algorithm for the n-queens problem*, SIGART Bulletin **1** (1990), 7–11.

[397] M. Squire, C.D. Savage, and D.B. West, *A Gray code for the acyclic orientations of a graph*, Dept. of Computer Science, North Carolina State University, 1993.

[398] Matthew B. Squire, *The square of the acyclic orientation graph is Hamiltonian*, Dept. of Computer Science, North Carolina State University, 1993.

[399] _____, *Generating ideals fast*, Dept. of Computer Science, North Carolina State University, 1994.

[400] _____, *Gray codes for a-free strings*, Electronic Journal of Combinatorics **3** (1996), R17.

[401] G. Stachowiak, *Hamilton paths in graphs of linear extensions for unions of posets*, SIAM J. Discrete Mathematics **5** (1992), 199–206.

[402] Richard P. Stanley, *Enumerative combinatorics, volume 1*, Wadsworth, 1986.

[403] Dennis Stanton and Dennis White, *Constructive combinatorics*, Springer-Verlag, 1986.

[404] G. Steiner, *An algorithm to generate the ideals of a partial order*, Operations Research Letters **5** (1986), 317–320.

[405] H. Steinhaus, *One hundred problems in elementary mathematics*, Basic, 1964.

[406] Ian Stewart, *Game, set, & math*, Basil Blackwell, Inc., 1989.

[407] F. Stockmal, *Generation of partitions in part-count form*, Communications of the ACM **5** (1962), 344.

[408] I. Stojmenović and M. Miyakawa, *Applications of a subset-generating algorithm to base enumeration, knapsack and minimal covering problems*, The Computer Journal **31** (1988), 65–70.

[409] M.M. Sysło, *An efficient cycle vector space algorithm for listing all cycles of a planar graph*, SIAM J. Computing **10** (1981), 797–808.

[410] D.T. Tang and C.N. Liu, *Distance-2 cycle chaining of constant weight codes*, IEEE Transactions on Computers **22** (1973), 176–180.

[411] Maurice Tchuente, *Generation of permutations by graphical exchanges*, Ars Combinatoria **14** (1982), 115–122.

[412] G. Tinhofer, *Generating graphs uniformly at random*, Computing (Supplement) **7** (1990), 235–255.

[413] G. Tinhofer and H. Schreck, *Linear time tree codes*, Computing **33** (1984), 211–225.

[414] J. Todd and H. Coxeter, *A practical method for enumerating cosets of a finite abstract group*, Proc. Edinburgh Math. Soc. **5** (1936), 26–34.

[415] C. Tompkins, *Machine attacks on problems whose variables are permutations*, Proc. Symposium in Applied Mathematics, Numerical Analysis, vol. 6, McGraw-Hill, 1956, N.Y., pp. 195–211.

[416] A.E. Trojanowski, *Ranking and listing algorithms for k-ary trees*, SIAM J. Computing **7** (1978), 492–509.

[417] ———, *Ranking and unranking unordered binary trees*, Trans. Illinois State Academy of Science **1** (1979), 46–56.

[418] H.F. Trotter, *Algorithm 115: Perm*, Communications of the ACM **5** (1962), 434–435.

[419] W.T. Trotter and P. Erdös, *When the Cartesian product of directed cycles is Hamiltonian*, Journal of Graph Theory **2** (1978), 137–142.

[420] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa, *A new algorithm for generating all the maximial independent sets*, SIAM J. Computing **6** (1977), 505–517.

[421] S. Tsukiyama, I. Shirakawa, H. Ozaki, and H. Ariyoshi, *An algorithm to enumerate all cutsets of a graph in linear time per cutset*, Journal of the ACM **27(4)** (1980), 619–632.

[422] T. Ueda, *Gray codes for necklaces*, Discrete Mathematics **219** (2000), 235–248.

[423] Vincent Vajnovszki, *Generating, ranking, and unranking for binary unordered trees*, Bulletin of EATCS **57** (1995), 221–229.

[424] ———, *Constant time algorithm for generating binary tree gray codes*, Studies in Informatics and Control **5** (1996), 15–21.

[425] Vincent Vajnovszki and Jean M. Pallo, *Generating binary trees in a-order from codewords defined on four-letter alphabet*, Journal of Information and Optimization Science **15** (1994), 345–357.

[426] T. van Ardenne-Ehrenfest and N.G. de Bruijn, *Circuits and trees in ordered linear graphs*, Simon Stevin **28** (1951), 203–217.

[427] D. Roelants van Baronaigien , *A loopless algorithm for generating binary tree sequences*, Information Processing Letters **39** (1991), 189–194.

[428] ———, *Constant time generation of involutions*, Congressus Numerantium **90** (1992), 87–96.

[429] D. Roelants van Baronaigien and E.M Neufeld, *Loopless generation of subsets with a given sum*, Congressus Numerantium **100** (1994), 147–152.

[430] D. Roelants van Baronaigien and F. Ruskey, *Generating permutations with given ups and downs*, Discrete Applied Mathematics **36** (1992), 57–65.

[431] ———, *Generating t-ary trees in A-order*, Information Processing Letters **27** (1988), 205–213.

[432] A.J. van Zanten, *Index system and separability of constant weight Gray codes*, IEEE Trans. Information Theory **37** (1991), 1229–1233.

[433] Y.L. Varol and D. Rotem, *An algorithm to generate all topological sorting arrangements*, The Computer Journal **24** (1981), 83–84.

[434] J. Veerasamy and I. Page, *On the towers of Hanoi problem with multiple spare pegs*, International Journal of Computer Mathematics **52** (1994), 17–22.

[435] R.J. Walker, *An enumerative technique for a class of combinatorial problems*, Proc. Symposium in Applied Mathematics, Combinatorial Analysis, vol. 10, American Mathematical Society, 1960, Providence, R.I., p. 91.

[436] T.R. Walsh, *Generating nonisomorphic maps without storing them*, SIAM J. Algebraic and Discrete Methods **4** (1983), 161–178.

[437] K.T. Wang, *On a new method for the analysis of networks*, Memoir 2, National Research Institute of Engineering, Academia Sinica, 1934.

[438] T.M. Wang and C.D. Savage, *A gray code for necklaces of fixed density*, SIAM J. Discrete Mathematics **9** (1996), 654–673.

[439] M.B. Wells, *Generation of permutations by transposition*, Mathematics of Computation **15** (1961), 192–195.

[440] ———, *Elements of combinatorial computing*, Pergamon Press, Elmsford, N.Y., 1971.

[441] A.T. White, *Ringing the changes*, Math. Proc. Camb. Phil. Soc. **94** (1983), 203–215.

[442] ———, *Ringing the changes II*, Ars Combinatoria **20-A** (1985), 65–75.

[443] ———, *Ringing the cosets*, American Mathematical Monthly **94** (1987), 721–746.

[444] ———, *Ringing the cosets II*, Math. Proc. Camb. Phil. Soc. **105** (1989), 53–65.

[445] D.E. White and S.G. Williamson, *Construction of minimal representative systems*, Linear and Multilinear Algebra **9** (1981), 167–180.

[446] J.S. White, *Algorithm 373: Number of doubly restricted partitions*, Communications of the ACM **13** (1970), 120.

[447] ———, *Algorithm 374: Restricted partition generator*, Communications of the ACM **13** (1970), 120.

[448] H.S. Wilf, *A unified setting for sequencing, ranking and selection algorithms for combinatorial objects*, Advances in Math. **24** (1977), 281–291.

[449] ———, *A unified setting for selection algorithms (II)*, Annals of Discrete Math. **2** (1978), 135–148.

[450] ———, *The uniform selection of free trees*, Journal of Algorithms **2** (1981), 204–207.

[451] ———, *Combinatorial algorithms: An update*, SIAM, 1989, CBMS 55.

[452] H.S. Wilf and N.A. Yoshimura, *Ranking rooted trees, and a graceful application*, Perspectives in Computing, Proc. Japan-U.S. Joint Seminar in Discrete Algorithms and Complexity, Academic Press, 1987, pp. 341–350.

[453] S. Gill Williamson, *Combinatorics for computer scientists*, Computer Science Press, 1985.

[454] S.G. Williamson, *Ranking algorithms for lists of partitions*, SIAM J. Computing **5** (1976), 602–617.

[455] ———, *On the ordering, ranking, and random generation of basic combinatorial sets*, Combinatoire et Représentation du Groupe Symétrique, Springer-Verlag, 1977, Lecture Notes in Mathematics, No. 579, pp. 311–339.

[456] E. Wilmer and M.D. Ernst, *Graphs induced by Gray codes*, Discrete Mathematics **??** (2001), to appear.

[457] Pawel Winter, *An algorithm for the enumeration of spanning trees*, BIT **26** (1986), 44–62.

[458] David Witte and Joseph A. Gallian, *A survey: Hamiltonian cycles in Cayley graphs*, Discrete Mathematics **51** (1984), 293–304.

[459] Jacek Wojciechowski, *Generation of trees of a graph with the use of decomposition*, J. Franklin Institute **318** (1984), no. 4, 215–231.

[460] N.C. Wormald, *Generating random regular graphs*, Journal of Algorithms **5** (1984), 247–280.

[461] R.A. Wright, B. Richmond, A. Odlyzko, and Brendan D. McKay, *Constant time generation of free trees*, SIAM J. Computing **15** (1986), 540–548.

[462] J. Wu and R. Chen, *The towers of Hanoi problem with parallel moves*, Information Processing Letters **44** (1992), 241–243.

[463] S. Xie, *Note on de Bruin sequences*, Discrete Applied Mathematics **16** (1987), 157–177.

[464] H.P. Yap, *Point symmetric graphs with $p \leq 13$ points*, Nanta Math. **6** (1973), 8–20.

[465] Nancy Allison Yoshimura, *Ranking and unranking algorithms for trees and other combinatorial objects*, Ph.D. thesis, Computer and Information Science, University of Pennsylvania, 1987.

[466] S. Zaks, *Lexicographic generation of ordered trees*, Theoretical Computer Science **10** (1980), 63–82.

[467] S. Zaks and D. Richards, *Generating trees and other combinatorial objects lexicographically*, SIAM J. Computing **8** (1979), 73–81.

[468] D. Zerling, *Generating binary trees using rotations*, Journal of the ACM **32** (1985), 694–701.

Chapter 11

Useful Tables

In this section we have included some tables of frequently used numbers, such as the binomial coefficients

$k \setminus n$	0	1	2	3	4	5	6	7	8	9
0	1									
1	1	1								
2	1	2	1							
3	1	3	3	1						
4	1	4	6	4	1					
5	1	5	10	10	5	1				
6	1	6	15	20	15	6	1			
7	1	7	21	35	35	21	7	1		
8	1	8	28	56	70	56	28	8	1	
9	1	9	36	84	126	126				
10	1	10	45	120	210	252				
11	1	11	55	165	330	462				

Table 11.1: Binomial coefficients $\binom{n}{k}$ for $0 \le k \le 11$.

$k \setminus n$	1	2	3	4	5	6	7	8	9
1	1								
2	1	1							
3	2	2	1						
4	5	5	3	1					
5	14	14	9	4	1				
6	42	42	28	14	5	1			
7	132	132	90	48	20	6	1		
8	9								

Table 11.2: The numbers $T(n,k) = \frac{k}{2n-k}\binom{2n-k}{k}$ for $1 \le k \le 9$.

$k \setminus n$	1	2	3	4	5	6	7	8	9
1	1								
2	1	1							
3	1	3	1						
4	1	7	6	1					
5	1	15	25	10	1				
6	1	31	90	65	15	1			
7	1	63	301	350	140	21	1		
8	1	127	966	1701	1050	266	28	1	
9	1	255	3025	7770	6951	2646	462	36	1

Table 11.3: Stirling numbers of the second kind $\left\{ {n \atop k} \right\}$ for $1 \le k \le 9$.

$k \setminus n$	1	2	3	4	5	6	7	8	9	10	11	12	13
1	1												
2	1	1											
3	1	1	1										
4	1	2	1	1									
5	1	2	2	1	1								
6	1	3	3	2	1	1							
7	1	3	4	3	2	1	1						
8	1	4	5	5	3	2	1	1					
9	1	4	7	6	5	3	2	1	1				
10	1	5	8	9	7	5	3	2	1	1			
11	1	5	10	11	10	7	5	3	2	1	1		
12	1	6	12	15	13	11	7	5	3	2	1	1	
13	1	6	14	18	18	14	11	7	5	3	2	1	1

Table 11.4: The number $p(n, k)$ of numerical partitions of n into k parts for $1 \leq k \leq 11$.

$k \setminus n$	1	2	3	4	5	6	7	8	9
1	1								
2	1	1							
3	1	4	1						
4	1	11	11	1					
5	1	26	66	26	1				
6	1	57	302	302	57	1			
7	1	120	1191	2416	1191	120	1		
8									
9									

Table 11.5: Eulerian numbers for $1 \leq k \leq 9$.

Index