Tin tan din dan bim bam bom bo—
tan tin din dan bam bim bo bom—
tin tan dan din bim bam bom bo—
tan tin dan din bam bim bo bom—
tan dan tin bam din bo bim bom—
.... Tin tan din dan bim bam bom bo.

— DOROTHY L. SAYERS, *The Nine Tailors* (1934)

A permutation on the ten decimal digits is simply a 10 digit decimal number in which all digits are distinct. Hence all we need to do is to produce all 10 digit numbers and select only those whose digits are distinct. Isn't it wonderful how high speed computing saves us from the drudgery of thinking! We simply program $k + 1 \to k$ *and examine the digits of* k *for undesirable equalities. This gives us the permutations in dictionary order too! On second sober thought ... we do need to think of something else.*

— D. H. LEHMER (1957)

7.2.1.2. Generating all permutations. After n-tuples, the next most important item on nearly everybody's wish list for combinatorial generation is the task of visiting all *permutations* of some given set or multiset. Many different ways have been devised to solve this problem. In fact, almost as many different algorithms have been published for unsorting as for sorting! We will study the most important permutation generators in this section, beginning with a classical method that is both simple and flexible:

Algorithm L (*Lexicographic permutation generation*). Given a sequence of n elements $a_1 a_2 \dots a_n$, initially sorted so that

$$a_1 \le a_2 \le \cdots \le a_n, \tag{1}$$

this algorithm generates all permutations of $\{a_1, a_2, \dots, a_n\}$, visiting them in lexicographic order. (For example, the permutations of $\{1, 2, 2, 3\}$ are

1223, 1232, 1322, 2123, 2132, 2213, 2231, 2312, 2321, 3122, 3212, 3221,

ordered lexicographically.) An auxiliary element a_0 is assumed to be present for convenience; a_0 must be strictly less than the largest element a_n.

L1. [Visit.] Visit the permutation $a_1 a_2 \dots a_n$.

L2. [Find j.] Set $j \leftarrow n-1$. If $a_j \geq a_{j+1}$, decrease j by 1 repeatedly until $a_j < a_{j+1}$. Terminate the algorithm if $j = 0$. (At this point j is the smallest subscript such that we have already visited all permutations beginning with $a_1 \ldots a_j$. Therefore the lexicographically next permutation will increase the value of a_j.)

L3. [Increase a_j.] Set $l \leftarrow n$. If $a_j \geq a_l$, decrease l by 1 repeatedly until $a_j < a_l$. Then interchange $a_j \leftrightarrow a_l$. (Since $a_{j+1} \geq \cdots \geq a_n$, element a_l is the smallest element greater than a_j that can legitimately follow $a_1 \ldots a_{j-1}$ in a permutation. Before the interchange we had $a_{j+1} \geq \cdots \geq a_{l-1} \geq a_l > a_j \geq a_{l+1} \geq \cdots \geq a_n$; after the interchange, we have $a_{j+1} \geq \cdots \geq a_{l-1} \geq a_j > a_l \geq a_{l+1} \geq \cdots \geq a_n$.)

L4. [Reverse $a_{j+1} \ldots a_n$.] Set $k \leftarrow j+1$ and $l \leftarrow n$. Then, if $k < l$, interchange $a_k \leftrightarrow a_l$, set $k \leftarrow k+1$, $l \leftarrow l-1$, and repeat until $k \geq l$. Return to L1. ∎

This algorithm goes back to Nārāyaṇa Paṇḍita in 14th-century India (see Section 7.2.1.7); it also appeared in C. F. Hindenburg's preface to *Specimen Analyticum de Lineis Curvis Secundi Ordinis* by C. F. Rüdiger (Leipzig: 1784), xlvi–xlvii, and it has been frequently rediscovered ever since. The parenthetical remarks in steps L2 and L3 explain why it works.

In general, the lexicographic successor of any combinatorial pattern $a_1 \ldots a_n$ is obtainable by a three-step procedure:

1) Find the largest j such that a_j can be increased.
2) Increase a_j by the smallest feasible amount.
3) Find the lexicographically least way to extend the new $a_1 \ldots a_j$ to a complete pattern.

Algorithm L follows this general procedure in the case of permutation generation, just as Algorithm 7.2.1.1M followed it in the case of n-tuple generation; we will see numerous further instances later, as we consider other kinds of combinatorial patterns. Notice that we have $a_{j+1} \geq \cdots \geq a_n$ at the beginning of step L4. Therefore the first permutation beginning with the current prefix $a_1 \ldots a_j$ is $a_1 \ldots a_j a_n \ldots a_{j+1}$, and step L4 produces it by doing $\lfloor (n-j)/2 \rfloor$ interchanges.

In practice, step L2 finds $j = n-1$ half of the time when the elements are distinct, because exactly $n!/2$ of the $n!$ permutations have $a_{n-1} < a_n$. Therefore Algorithm L can be speeded up by recognizing this special case, without making it significantly more complicated. (See exercise 1.) Similarly, the probability that $j \leq n-t$ is only $1/t!$ when the a's are distinct; hence the loops in steps L2–L4 usually go very fast. Exercise 6 analyzes the running time in general, showing that Algorithm L is reasonably efficient even when equal elements are present, unless some values appear much more often than others do in the multiset $\{a_1, a_2, \ldots, a_n\}$.

Adjacent interchanges. We saw in Section 7.2.1.1 that Gray codes are advantageous for generating n-tuples, and similar considerations apply when we want to generate permutations. The simplest possible change to a permutation is to interchange adjacent elements, and we know from Chapter 5 that any

permutation can be sorted into order if we make a suitable sequence of such interchanges. (For example, Algorithm 5.2.2B works in this way.) Hence we can go backward and obtain any desired permutation, by starting with all elements in order and then exchanging appropriate pairs of adjacent elements.

A natural question now arises: Is it possible to run through *all* permutations of a given multiset in such a way that only two adjacent elements change places at every step? If so, the overall program that is examining all permutations will often be simpler and faster, because it will only need to calculate the effect of an exchange instead of to reprocess an entirely new array $a_1 \dots a_n$ each time.

Alas, when the multiset has repeated elements, we can't always find such a Gray-like sequence. For example, the six permutations of $\{1, 1, 2, 2\}$ are connected to each other in the following way by adjacent interchanges:

$$1122 \text{ —— } 1212 \begin{array}{l} \diagup \\ \diagdown \end{array} \begin{array}{l} 2112 \\ 1221 \end{array} \begin{array}{l} \diagdown \\ \diagup \end{array} 2121 \text{ —— } 2211; \tag{2}$$

this graph has no Hamiltonian path.

But most applications deal with permutations of *distinct* elements, and for this case there is good news: A simple algorithm makes it possible to generate all $n!$ permutations by making just $n! - 1$ adjacent interchanges. Furthermore, another such interchange returns to the starting point, so we have a Hamiltonian cycle analogous to Gray binary code.

The idea is to take such a sequence for $\{1, \dots, n-1\}$ and to insert the number n into each permutation in all ways. For example, if $n = 4$ the sequence $(123, 132, 312, 321, 231, 213)$ leads to the columns of the array

$$\begin{array}{cccccc} 1234 & 1324 & 3124 & 3214 & 2314 & 2134 \\ 1243 & 1342 & 3142 & 3241 & 2341 & 2143 \\ 1423 & 1432 & 3412 & 3421 & 2431 & 2413 \\ 4123 & 4132 & 4312 & 4321 & 4231 & 4213 \end{array} \tag{3}$$

when 4 is inserted in all four possible positions. Now we obtain the desired sequence by reading downwards in the first column, upwards in the second, downwards in the third, ..., upwards in the last: $(1234, 1243, 1423, 4123, 4132, 1432, 1342, 1324, 3124, 3142, \dots, 2143, 2134)$.

In Section 5.1.1 we studied the inversions of a permutation, namely the pairs of elements (not necessarily adjacent) that are out of order. Every interchange of adjacent elements changes the total number of inversions by ± 1. In fact, when we consider the so-called inversion table $c_1 \dots c_n$ of exercise 5.1.1–7, where c_j is the number of elements lying to the right of j that are less than j, we find that the permutations in (3) have the following inversion tables:

$$\begin{array}{cccccc} 0000 & 0010 & 0020 & 0120 & 0110 & 0100 \\ 0001 & 0011 & 0021 & 0121 & 0111 & 0101 \\ 0002 & 0012 & 0022 & 0122 & 0112 & 0102 \\ 0003 & 0013 & 0023 & 0123 & 0113 & 0103 \end{array} \tag{4}$$

And if we read these columns alternately down and up as before, we obtain precisely the reflected Gray code for mixed radices $(1, 2, 3, 4)$, as in Eqs. (46)–(51)

of Section 7.2.1.1. The same property holds for all n, as noticed by E. W. Dijkstra [*Acta Informatica* **6** (1976), 357–359], and it leads us to the following formulation:

Algorithm P (*Plain changes*). Given a sequence $a_1 a_2 \dots a_n$ of n distinct elements, this algorithm generates all of their permutations by repeatedly interchanging adjacent pairs. It uses an auxiliary array $c_1 c_2 \dots c_n$, which represents inversions as described above, running through all sequences of integers such that

$$0 \le c_j < j \qquad \text{for } 1 \le j \le n. \tag{5}$$

Another array $o_1 o_2 \dots o_n$ governs the directions by which the entries c_j change.

P1. [Initialize.] Set $c_j \leftarrow 0$ and $o_j \leftarrow 1$ for $1 \le j \le n$.

P2. [Visit.] Visit the permutation $a_1 a_2 \dots a_n$.

P3. [Prepare for change.] Set $j \leftarrow n$ and $s \leftarrow 0$. (The following steps determine the coordinate j for which c_j is about to change, preserving (5); variable s is the number of indices $k > j$ such that $c_k = k - 1$.)

P4. [Ready to change?] Set $q \leftarrow c_j + o_j$. If $q < 0$, go to P7; if $q = j$, go to P6.

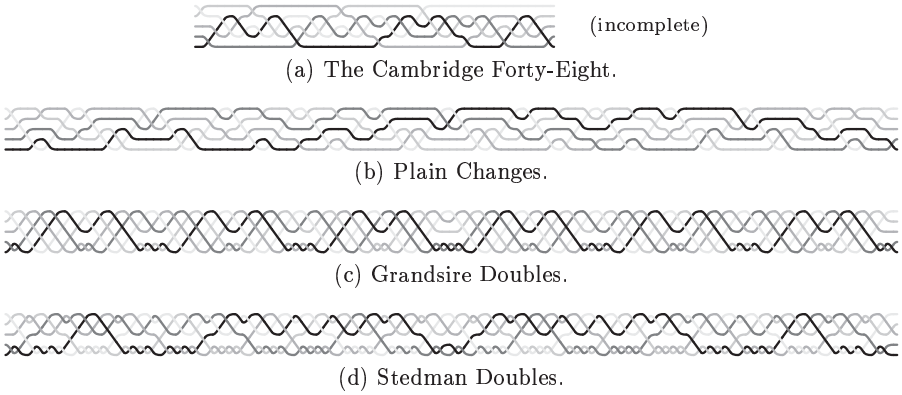
P5. [Change.] Interchange $a_{j-c_j+s} \leftrightarrow a_{j-q+s}$. Then set $c_j \leftarrow q$ and return to P2.

P6. [Increase s.] Terminate if $j = 1$; otherwise set $s \leftarrow s + 1$.

P7. [Switch direction.] Set $o_j \leftarrow -o_j$, $j \leftarrow j - 1$, and go back to P4. ∎

This procedure, which clearly works for all $n \ge 1$, originated in 17th-century England, when bell ringers began the delightful custom of ringing a set of bells in all possible permutations. They called Algorithm P the method of *plain changes*. Figure 18(a) illustrates the "Cambridge Forty-Eight," an irregular and ad hoc sequence of 48 permutations on 5 bells that had been used in the early 1600s, before the plain-change principle revealed how to achieve all $5! = 120$ possibilities. The venerable history of Algorithm P has been traced to a manuscript by Peter Mundy now in the Bodleian Library, written about 1653 and transcribed by Ernest Morris in *The History and Art of Change Ringing* (1931), 29–30. Shortly afterwards, a famous book called *Tintinnalogia*, published anonymously in 1668 but now known to have been written by Richard Duckworth and Fabian Stedman, devoted its first 60 pages to a detailed description of plain changes, working up from $n = 3$ to the case of arbitrarily large n.

> Cambridge Forty-eight, *for many years,*
> *was the greatest* Peal *that was* Rang *or invented; but now,*
> *neither* Forty-eight, *nor a* Hundred, *nor* Seven-hundred and twenty,
> *nor any Number can confine us; for we can* Ring Changes, Ad infinitum.
> ... *On four Bells, there are* Twenty four several Changes,
> *in* Ringing *of which, there is one Bell called the* Hunt,
> *and the other three are* Extream *Bells;*
> *the* Hunt *moves, and* hunts *up and down continually ...;*
> *two of the* Extream *Bells makes a* Change
> *every time the* Hunt *comes before or behind them.*
>
> — DUCKWORTH and STEDMAN, *Tintinnalogia* (1668)



(a) The Cambridge Forty-Eight.

(b) Plain Changes.

(c) Grandsire Doubles.

(d) Stedman Doubles.

Fig. 18. Four patterns used to ring five church-bells in 17th-century England. Pattern (b) corresponds to Algorithm P.

British bellringing enthusiasts soon went on to develop more complicated schemes in which two or more pairs of bells change places simultaneously. For example, they devised the pattern in Fig. 18(c) known as Grandsire Doubles, "the best and most ingenious Peal that ever was composed, to be rang on five bells" [*Tintinnalogia*, page 95]. Such fancier methods are more interesting than Algorithm P from a musical or mathematical standpoint, but they are less useful in computer applications, so we shall not dwell on them here. Interested readers can learn more by reading W. G. Wilson's book, *Change Ringing* (1965); see also A. T. White, *AMM* **103** (1996), 771–778.

H. F. Trotter published the first computer implementation of plain changes in *CACM* **5** (1962), 434–435. The algorithm is quite efficient, especially when it is streamlined as in exercise 16, because $n-1$ out of every n permutations are generated without using steps P6 and P7. By contrast, Algorithm L enjoys its best case only about half of the time.

The fact that Algorithm P does exactly one interchange per visit means that the permutations it generates are alternately even and odd (see exercise 5.1.1–13). Therefore we can generate all the even permutations by simply bypassing the odd ones. In fact, the c and o tables make it easy to keep track of the current total number of inversions, $c_1 + \cdots + c_n$, as we go.

Many programs need to generate the same permutations repeatedly, and in such cases we needn't run through the steps of Algorithm P each time. We can simply prepare a list of suitable transitions, using the following method:

Algorithm T (*Plain change transitions*). This algorithm computes a table $t[1]$, $t[2]$, $\ldots$, $t[n!-1]$ such that the actions of Algorithm P are equivalent to the successive interchanges $a_{t[k]} \leftrightarrow a_{t[k]+1}$ for $1 \le k < n!$. We assume that $n \ge 2$.

T1. [Initialize.] Set $N \leftarrow n!$, $d \leftarrow N/2$, $t[d] \leftarrow 1$, and $m \leftarrow 2$.