TEACHING COMBINATORIAL TRICKS TO A COMPUTER

BY

D. H. LEHMER

The widespread renewed interest in Combinatorial Analysis is largely due to the physical existence of the automatic digital computer which makes possible the actual carrying out of processes hitherto only talked about. Moreover, some of the characteristics of these computers have rendered feasible processes that are entirely beyond the ability of a human being to follow in detail. However, the modern "all purpose" computer is in reality not adept at combinatorial problems, being designed to do rational operations, in its own peculiar arithmetic, operations intended to mimic the corresponding multiplications, additions, divisions, and subtractions in the idealized real number system of the applied mathematician. The fact that a machine has been designed to do a 10×10 multiplication in N microseconds may not be very interesting to a combinatorial analyst with a problem involving no multiplication at all. He may be much more interested in the fact that the machine is capable of some rapid simple logical operation, such as recognition of overflow, which he can apply to his problem. To get the computer to do combinatorial problems efficiently requires a good deal of thoughtful teaching, some of which is done by the computer.

Many steps in the solution of combinatorial problems seem relatively small but nevertheless are not quite straightforward when programmed in the language of the computer's limited repertory of instructions. We give in what follows some suggestions for teaching the machine to handle some of the simpler types of combinatorial operations. It is hoped that some of the ideas set forth may be of use in dealing with the much more elaborate problems of this Symposium.

As a matter of fact, because of the flexibility of the computer control, a surprisingly large percentage of the instructions in an essentially non-combinatorial problem, for example matrix inversion, are actually of a combinatorial nature, although of a quite rudimentary sort such as tallying, comparing, and the other operations usually described as "address arithmetic."

To begin with the simplest problem of the next higher category we consider the problem of set inclusion. Suppose that a set S of n numbers

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

is stored in the machine in n addresses. The machine produces a number x and is asked to discover whether x belongs to the set S or not. To make things easy we may suppose that the n addresses are consecutive or even that the address of a_k is k or, what is the same, the word $w(k)$ at address k

is a_k. Without further information on the a's there can be no other method than a methodical "house to house" search. A simple block diagram for this routine is given in Figure 1.

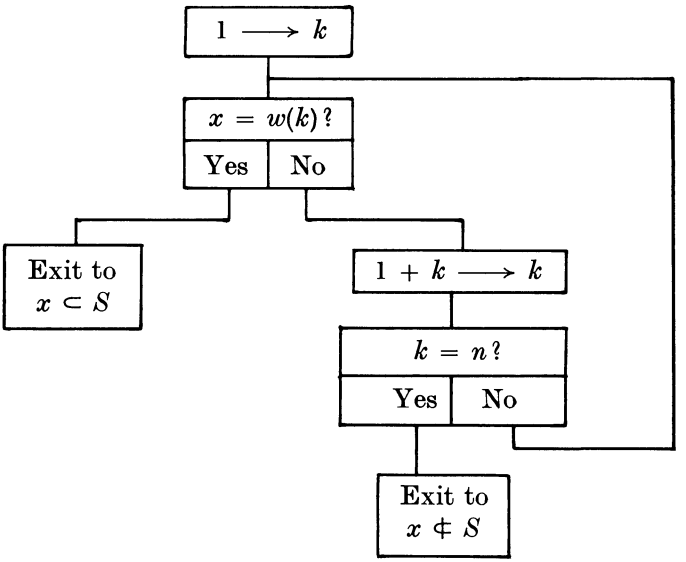

FIGURE 1

This routine may be expected to loop around $n/2$ times on the average. In case the a's in (1) are monotonely increasing, another search method is available in which the effort is proportional to log n instead of n. It proceeds by successive dichotomies as indicated in Figure 2. In the third box $[(a + b)/2]$ denotes the greatest integer not exceeding the average of a and b and is found, of course, by shifting the sum $a + b$ one right.

The routine of Figure 2 is, to be sure, very much faster than the general routine of Figure 1. A still faster process may be used when the members a_k of S (or some function of them) are small positive integers. In this case the machine may ascertain whether x belongs to S by a method in which the time is practically independent of n. In this scheme the set S is represented by its "characteristic binary number" whose rth digit is 1 or 0 according as r is a member of S or not. (In some cases it may be more convenient to interchange the roles of 1 and 0.) For example, the odd prime numbers may be represented by the characteristic binary number

$$11101101101001100101101001\cdots$$

in which the rth digit is 1 or 0 according as $2r + 1$ is prime or composite. Thousands of these digits may be stored in the high speed memory of the machine while millions of others may be stored on magnetic tapes if necessary. Whenever the machine produces a new number x it is instructed to

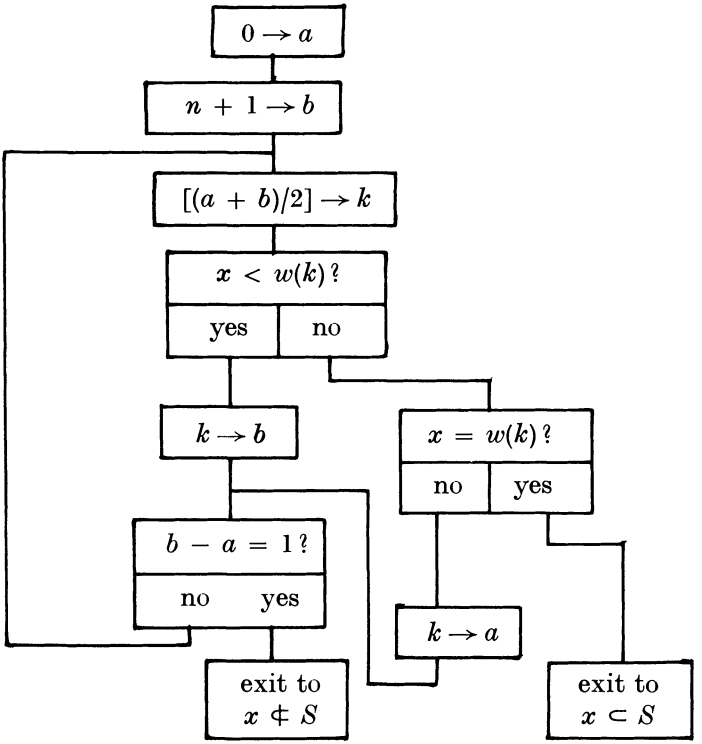

FIGURE 2

"extract" the xth binary digit of the characteristic member and ascertain whether or not it is zero. This will involve a minor amount of address arithmetic in order to choose the appropriate word containing the xth bit and the appropriate extractor to mask out all but the bit in question. This will generally call for a division of x by the number of bits in a word with scrupulous retention of the remainder. However if x runs over consecutive integer values, a simpler trick is available, namely the use of intentional overflow. If we denote the characteristic word by B and if x starts from 1, the simple diagram of Figure 3 illustrates how rapidly the question of set inclusion can be handled in this case.

Actually this routine is oversimplified. Eventually A, which acquires an extra terminal zero at each step, will consist wholly of zeros so that when x finally exceeds the number of bits in the word B it is time either to stop or to rejuvenate the word A and continue as before. One way to keep A alive indefinitely is to insert the command $A + 2^{-m} \rightarrow A$ just after the "yes." This produces a strictly periodic pattern of yes's and no's of period m; a sort of high speed roulette wheel which has a large number of combinatorial uses. For example such a subroutine can be placed after another one so as to steer the control of the program into one of two channels according to a

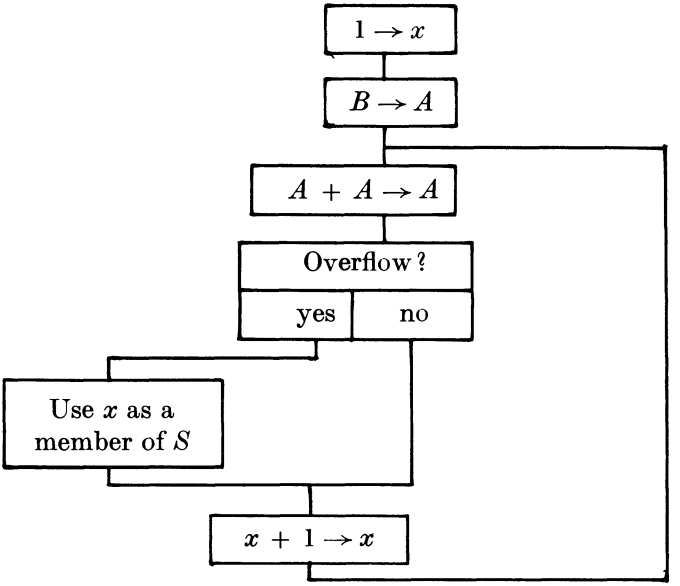

FIGURE 3

prearranged pattern, however complicated. A number of such subroutines can be compounded to produce pseudo-random digits. By using sets of such signals as extractors one produces a high speed parallel "sieve" that can be used to identify the answers to a wide class of diophantine problems [1]. The above brief remarks will serve to indicate what can come out of the simple question: "Does x belong to S?", a question almost never asked by the mathematician who "works" in set theory.

Changing the subject abruptly, there are simple combinatorial problems associated with multiple sums and other operators on functions of many variables. The problem of evaluating $f(x)$ for equally spaced values of x is easily generalized to the case of several variables having equal ranges. With a little address arithmetic one may reduce the problem to the consideration of the m^n lattice points in the n dimensional cube of side m in the "first orthant of E_n", that is, the vectors

$$(k_1, k_2, \cdots, k_n) \quad \begin{cases} k_i = 0(1)m - 1, \\ i = 1(1)n. \end{cases}$$

In dealing with this "unrestricted case" one has only to pretend that the k's are the separate digits of an n-digit integer N written to base m. All such vectors are generated by the simple Peano process $N + 1 \to N$, starting with $N = 0$, observing the ordinary carry rule of base m arithmetic, and halting as soon as $N = m^n$.

Such straightforward programming can be modified to cover the case in which each k has its own upper bound. This involves only a minor change

in the carry rule. A special case of this rectangular, rather than cubical, array of lattice points arises in connection with permutation problems discussed later.

Another variant of the carry rule takes care of the problem of generating vectors whose components are monotone, say,

$$0 \leqq k_1 \leqq k_2 \leqq \cdots \leqq k_n \leqq m - 1,$$

by resetting the overgrown component k_i not by zero but by the newly increased value of k_{i-1}, with only slight complications in case of carry propagation. The case of strict monotoneity is handled by using $1 + k_{i-1}$ as a resetting value.

Suppose that a problem calls for vectors of n non-negative integers

$$(k_1, k_2, \cdots, k_n)$$

subject only to the conditions that their sum

$$k_1 + k_2 + \cdots + k_n = C \tag{2}$$

has a prescribed constant value. Since for each component

$$0 \leqq k_i \leqq C, \tag{3}$$

one could set $m = C + 1$ and use the unrestricted case program to generate each of the n^{C-1} vectors satisfying (3). Then the condition (2) could be imposed to eliminate nearly every candidate as soon as it is generated. This forthright procedure would be very wasteful of machine time producing large quantities of chaff for only a handful of wheat. A more efficient method is the following. We replace n by $n - 1$ and generate vectors

$$(\delta_1, \delta_2, \cdots, \delta_{n-1})$$

for which the δ's are monotone:

$$0 \leqq \delta_1 \leqq \delta_2 \leqq \cdots \leqq \delta_{n-1} \leqq C \tag{4}$$

as described above. Then we set

$$\begin{aligned} k_1 &= \delta_1 - 0 \geqq 0, \\ k_2 &= \delta_2 - \delta_1 \geqq 0, \\ &\cdots \\ k_{n-1} &= \delta_{n-1} - \delta_{n-2} \geqq 0, \\ k_n &= C - \delta_{n-1} \geqq 0. \end{aligned}$$

It is clear that (2) and (3) are satisfied and to every instance of a vector of k's satisfying these conditions there corresponds uniquely a vector of δ's as described by (4).

If a problem requires the k's to be positive, (4) may be replaced by the condition of strict monotoneity. Alternatively one may replace k_i by $h_i + 1$ and C by $C - n$ and solve the original problem for the h's.

In some problems the parameters n, m, or C are so large that the number of vectors becomes unreasonably great. In such cases one may wish to make use of sampling methods. This may be done by replacing the methodical generation procedures by random variable generation in obvious ways. Thus in the preceding problem the δ's are to be selected at random, not the first $n - 1$ k's.

Another rather basic combinatorial problem is that of selecting, in turn, all possible combinations of n objects k at a time. For the machine, these n objects are words stored in the memory in addresses which without loss of generality may be taken to be $1, 2, \cdots, n$. The problem is then to get the machine methodically to select k of these addresses and deliver the corresponding words to k other addresses which we may take to be $n + 1$, $n + 2, \cdots, n + k$. It is worth pointing out that we do not care about the order in which the k selected words are arranged when delivered. To leave this matter to the discretion of the machine would be very unwise since it cannot deliver words in an unspecified manner like a human being drawing a handful of balls from an urn. The simplest way to specify this ordering is to insist on preservation of precedence; that is, if w_1 and w_2 are two selected words and if the address of w_1 in storage is less than that of w_2 the same shall be true on delivery. Thus if the words in storage are numbers in monotone sequence the same will be true of all the selected sets of k. With this additional requirement the subroutine can now be written. A diagram of such a routine is given in Figure 4. It exhibits a typical feature of many combinatorial routines. The sole purpose of the routine is to fetch k words from addresses $1(1)n$ and to deposit them into addresses $n + 1(1)n + k$. Yet there is only one command that does this. All the other commands are needed to process this single fetch and deposit instruction. We note that the arithmetic operations involved consist only of adding and subtracting unity. The number t is a tally that rises and falls between 1 and k and finally becomes zero after the last selection has been made. The numbers A_t are the selected addresses. The routine selects first the k words stored in addresses 1 through k and finally selects the k words in addresses $n - k + 1$ through n. If the n objects are the first n letters of the alphabet, the $C_{n,k}$ k-letter words not only will have the letters of each word in alphabetical order but the words themselves will be in lexicographical order.

Problems involving permutations occur frequently in combinatorial problems. There are, to the writer's knowledge, ten different methods of producing permutations automatically.

A method may be judged on its ability to satisfy one or more of the following requirements:

(a) To generate all permutations,
(b) To generate all favorable permutations,
(c) To generate sample or random permutations,
(d) To generate custom-made permutations.

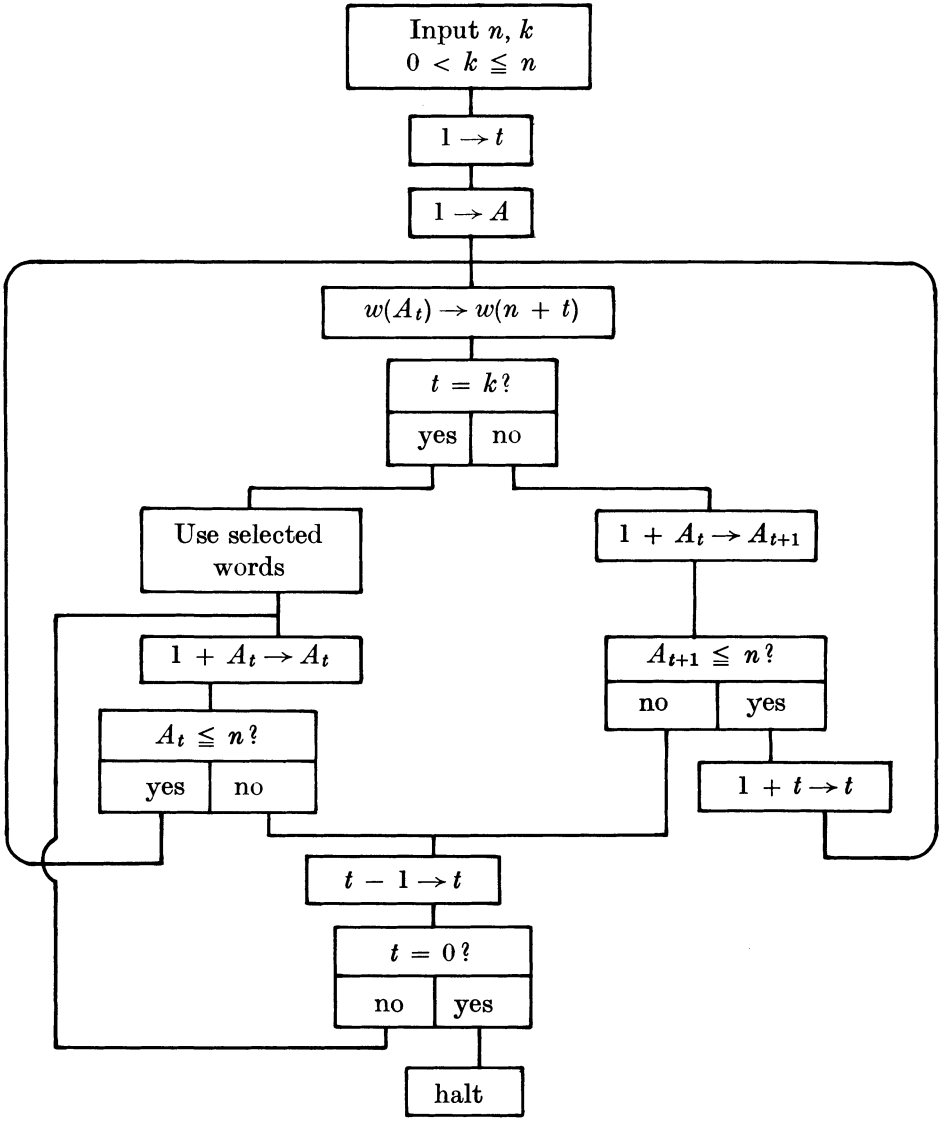

FIGURE 4

The following comments on these requirements may help to clarify the problem.

As far as (a) is concerned, it is well to note that $n!$ is approximately $1.55 \cdot 10^{25}$ for $n = 25$. Hence unless we can generate and utilize more than 10000 permutations per second our problem for $n = 25$ will last more than 10000 times the present age of the earth. For $n = 12$ the time will be about 13 hours. For $n = 6$ the time is negligible.

Because of the preceding, the ability of a method to meet requirement (b) is very important when n is more than 10 or 12. In this case we attempt to skip over millions of unwanted permutations in convenient blocks. For example in the travelling salesman problem with cities in two clusters as shown in Figure 5 the minimum circuit visiting all 10 cities and returning

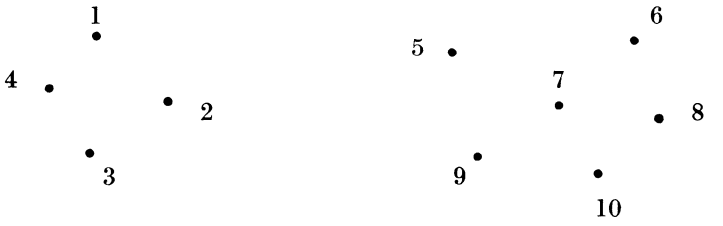

FIGURE 5

home would not be given by any permutation of the form 1 5 2 6 x x x x x x no matter what digits the x's represent. Hence we should skip over all 720 permutations that begin in this way. To get the machine to recognize and take advantage of this opportunity as quickly as a human being can, requires a human programmer with a suitable method.

As for (c), it may be possible in some problems when n is large to use random sampling of the huge population of $n!$ permutations to make shrewd guesses about some function of permutations. Acceptability tests for randomness of permutations have not been generally agreed upon as yet. S. Ulam, in a recent letter, suggests two such tests, one for frequency and one for gaps. In generating random permutations we are generating isolated ones. Thus our program differs greatly from those required by (b) and especially (a) where a recursive formula or algorithm is indicated.

The requirement (d) is often encountered with quite large values of n but with a relatively small number of admissible permutations. It has been possible to deal with $n = 20$ in one example, for instance. All requisite permutations were found in a few minutes by a systematic filling in of addresses by marks according to the very restrictive conditions imposed on the permutations.

A word or two about representing permutations in a computer may be in order. It is clear that the different objects being permuted can be made computer words. A permutation is then merely an assignment of these words or copies of these words to n memory cells. To store all $n!$ permutations in the machine at one time will require a small n and a big memory. Hence in most interesting problems the permutations are made to pass in review timewise. A subroutine inspects each permutation, rejects or gathers information about it, and then consigns it to extinction. However, in some cases one or more good-looking permutations are set aside for output.

Instead of permuting whole words full of information we can make a permanent file of these words in some fixed addresses and then proceed to permute these addresses instead. These addresses may be taken as 0, 1, $2, \cdots, n-1$. Now we are handling such small numbers that most of the arithmetic unit is processing zeros. This suggests that there should be some attempt to parallel the arithmetic by using "fractional precision" methods, that is by storing many marks in one word. In some programs this serves to speed up the generation considerably. To offset this advantage is the fact that the separate marks are less accessible individually without recourse to often fussy extract or shift operations.

In generating or counting permutations a special "factorial" representation of integers is often convenient. Every non-negative integer N less than $n!$ can be uniquely written

$$N = S_1 1! + S_2 2! + S_3 3! + \cdots + S_{n-1}(n-1)!$$

where the "factorial digits", S_k, satisfy

$$0 \leqq S_k \leqq k.$$

Thus, S_1 is a binary digit, S_2 a ternary digit, etc. Following the well-established backwards Arabic way of writing decimal digits we can also write

$$N = S_{n-1}, S_{n-2}, \cdots, S_2, S_1. \tag{5}$$

Whatever the complexities of the arithmetic of this number representation, it cannot be accused of favoring any one particular base. There are two ways of computing the factorial digits of a given number N. To obtain the higher ordered digits first one simply divides N by $(n-1)!$. The quotient is S_{n-1} and the remainder is divided by $(n-2)!$ to obtain S_{n-2}, etc. To get the digits in reverse order, one divides N by 2; the remainder is S_1 and the quotient is divided by 3 to obtain S_2, etc.

To have a method for generating permutations one has only to establish a one-to-one correspondence between the set of all permutations on n marks and the set of all $(n-1)$-digit numbers of the form (5) or what is really the same, the set of all vectors of $n-1$ non-negative integer components, the kth component not exceeding k, or, again the same, the set of lattice points in and on an $(n-1)$-dimensional rectangular parallelepiped of sides $1, 2, \cdots, n-1$.

Since we have already discussed the methodical recursive generation of such vectors by addition, using a simple variant of the usual carry rule, the corresponding permutations are duly generated also. If isolated or random permutations are needed it is a simple matter to generate isolated or random vectors of the right type.

It remains to establish such a one-to-one correspondence. This may be done in more than one way. Three ways are discussed in what follows. Four other entirely different methods will be described later.

The Tompkins-Paige cyclic method makes use of the fact that every permutation on n marks is the product of $n - 1$ cyclic permutations in the following sense. The simple permutation which replaces

$$0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9$$

by

$$0\ 1\ 2\ 6\ 7\ 8\ 9\ 3\ 4\ 5$$

may be said to be of order 7 and degree 3, by which we mean that the last 7 marks of the original permutation have been mounted on a wheel and the wheel rolled forward three spokes. If this result is subjected to permutation of order 6 and degree 0 it remains unchanged; but if the degree is 4 we thus get

$$0\ 1\ 2\ 6\ 4\ 5\ 7\ 8\ 9\ 3.$$

Clearly every permutation is the result of $n - 1$ superposed transformations of order $k + 1$ and degree S_k ($k = 1, 2, \cdots, n - 1$, $0 \leqq S_k \leqq k$), which are uniquely determined either by the permutation of the set of factorial digits S_k. Thus the correspondence is established. For $n = 3$ the correspondences are:

S_2	S_1	permutation
0	0	0 1 2
0	1	0 2 1
1	0	1 2 0
1	1	1 0 2
2	0	2 0 1
2	1	2 1 0

As set up for the machine, the recursive generation of the next permutation from a given one is done as follows. The last two marks of the given permutation are interchanged and 1 is added to the number

$$S_{n-1}, S_{n-2}, \cdots, S_2, S_1.$$

If this produces no carry ($S_1 = 0$) we have our new permutation. If there is a carry to S_2 only, we have an old permutation which is now subjected to a cyclic permutation of order 3 and degree 1, which gives our new permutation. If, however, the carry propagates to S_3 and stops there, a cyclic permutation of order 4 and degree 1 is in order, etc. In this routine all cyclic permutations are of degree 1. This simple method has been coded for the SWAC and the IBM 701 and carefully polished for speed. For the SWAC the time required just to generate and count permutations is close to 2456 microseconds per permutation. The 701 is somewhat slower requiring nearly 3600 microseconds. For this timing, each mark is stored as an individual word. If several marks are stored in a single word, cyclic permutation can be paralleled by an obvious use of the shift command. This should speed the program considerably, though it would also delay

the actual use of the permutation in a practical problem. Of all the methods discussed, the Tompkins-Paige cyclic method is the fastest.

In some problems it is necessary to know whether a permutation is odd or even. The parity of the permutation corresponding to the Tompkins-Paige method is the same as that of the number

$$S_1 + 2S_2 + 3S_3 + \cdots + (n - 1)S_{n-1}.$$

We note that this routine does not generate permutations in lexicographical order since in the above table for $n = 3$ the permutation 120 precedes 102. For some purposes this may be a drawback.

If the first few marks of a permutation are somehow undesirable, the routine can easily be altered to skip over all the permutations that begin in this way simply by adding a unit to the appropriate S_k and performing the corresponding cyclic permutation. Thus the method meets requirement (b). It fails to satisfy requirements (c) and (d).

In an unpublished paper, Tompkins has suggested a different way of realizing the above correspondence which allows it to meet requirement (c). In this method the machine computes the mark located at a given address to produce the permutation directly from its corresponding vector of factorial digits.

We turn now to a second way of making a permutation correspond to its factorial digits. This method was suggested by Marshall Hall and may be called the Method of Derangements. In the previous method the objects being permuted can be any computer words. In the Hall method the objects must be the numbers $0(1)n - 1$. In any such permutation we may, for each mark $k > 0$, ask how many of the k marks less than k actually follow k. Denoting this number by S_k we see at once that

$$S_{n-1}, S_{n-2}, \cdots, S_2, S_1$$

is a set of factorial digits of a number which corresponds to the given permutation and which, conversely, characterizes this permutation. We have for example the following correspondencies when $n = 7$.

S_6	S_5	S_4	S_3	S_2	S_1	permutation
0	0	0	0	0	0	0 1 2 3 4 5 6
3	1	4	1	2	1	4 2 1 6 3 5 0
1	2	2	3	1	1	3 1 4 5 2 6 0
6	5	4	3	2	1	6 5 4 3 2 1 0

The coding of this method is fairly straightforward. The resulting routine is a good deal slower than the Tompkins-Paige method. The parities of successive permutations strictly alternate. The method is well suited to requirement (c).

A third way of setting up a correspondence was, in effect, suggested by

D. N. Lehmer as long ago as 1906. It may be called the lexicographic method since it generates permutations in this order.

If in any permutation

$$a_1, a_2, \cdots, a_n$$

of the numbers $0(1)n - 1$ we strike out a_1 and reduce by unity all the marks which exceed a_1, we get a new permutation

$$\alpha_1, \alpha_2, \cdots, \alpha_{n-1}$$

of the numbers $0(1)n - 2$, which we may denote by

$$M(a_1, a_2, \cdots, a_n).$$

If we now define a rank function $R(a_1, a_2, \cdots, a_n)$ recursively by

$$R(0) = 0,$$

$$R(a_1, a_2, \cdots, a_n) = a_1(n - 1)! + R(M(a_1, a_2, \cdots, a_n))$$

it is seen that R is nothing but the rank or serial number of the permutation $(a_1, a_3, \cdots, a_n)$ in the lexicographical list of all permutations. In fact, in this list the first $(n - 1)!$ permutations have 0 as their first mark, the next $(n - 1)!$ permutations have 1 as their first mark, and so on. Since our permutation has a_1 as its first mark it is preceded by $a_1(n - 1)!$ permutations whose first mark is less than a_1. Among those permutations which begin with a_1, ours has rank $R(M(a_1, a_2, \cdots, a_n))$. If one successively applies the operation M we get a sequence of $n - 1$ permutations whose first elements are the factorial digits of the rank of the original permutation.

Thus for example, for $n = 7$, the permutation 1 4 2 0 5 6 3 gives rise to

$$\begin{array}{ccccccc} 1 & 4 & 2 & 0 & 5 & 6 & 3 \\ & 3 & 1 & 0 & 4 & 5 & 2 \\ & & 1 & 0 & 3 & 4 & 2 \\ & & & 0 & 2 & 3 & 1 \\ & & & & 1 & 2 & 0 \\ & & & & & 1 & 0 \\ & & & & & & 0 \end{array}$$

Hence the rank of 1 4 2 0 5 6 3 is

$$1, 3, 1, 0, 1, 1 = 6! + 3 \cdot 5! + 4! + 2! + 1! = 1107.$$

Conversely given the rank and its factorial digits

$$S_{n-1}, S_{n-2}, \cdots, S_2, S_1$$

we may reconstruct the permutation having this rank. Beginning with 0 we affix S_1 and in case S_1 is 0, we increase the original 0 to 1. Thus we get

$$\begin{array}{ll} 0\ 1 & \text{if } S_1 = 0, \\ 1\ 0 & \text{if } S_1 = 1. \end{array}$$

Inductively having reached a permutation of $0(1)k - 1$ we affix S_k and adjust upward by a unit those elements which exceed S_k. Finally S_{n-1} is attached and a final adjustment, if necessary, completes the permutation. Thus the millionth permutation on 10 marks is found as follows: The factorial digits of a million are

$$10^6 = 2, 6, 6, 2, 5, 1, 2, 2, 0.$$

Hence we write the succession of permutations

```
                    0
                  0 1
                2 0 1
              2 3 0 1
            1 3 4 0 2
          5 1 3 4 0 2
        2 6 1 4 5 0 3
      6 2 7 1 4 5 0 3
    6 7 2 8 1 4 5 0 3
2 7 8 3 9 1 5 6 0 4
```

of which the last is the millionth. Thus the correspondence is established. The parity of the permutation of rank R is that of $[(R + 1)/2]$. The method satisfies requirements (a), (b) and (c).

We mention briefly four quite different methods based on various aspects of permutations.

The Walker Backtrack Method, given elsewhere in this volume in more general form, is described by him as "completely unsophisticated." One regards a permutation of the marks 0, 1, 2, $\cdots$, $(n - 1)$ as simply a vector whose components are taken from the non-negative integers $< n$ but are all distinct. One proceeds to construct such vectors starting from 0, 0, 0, 0, $\cdots$, 0, 0 filling in at each opportunity the least available mark. When a permutation is completed the last two marks are removed and the penultimate address is filled by the next largest mark available. If there is no next largest element available one more mark is removed and replaced by the next larger available mark, etc. The result is a complete set of permutations in lexicographical order. This process was coded by Walker, for a general n, for the SWAC using only twenty-two commands. The program is slower than the Tompkins routine by a factor of two. I believe it could be altered to give random permutation and to skip over blocks of unwanted permutations. A similar program was devised by the writer to meet requirement (d) in 1955. Such programs are difficult to describe except in very general terms. In brief the machine keeps a sort of registry which shows at a glance which marks have been assigned to the permutation under construction and thereby avoids placing two marks in the same place and provides an automatic waiting list of marks as yet unassigned.

We pass on to what may be called the Constant Difference Method. Given a permutation like

2 3 1 5 4 0 7 9 6 8

one can obtain immediately another one by increasing every mark by unity, replacing 9 by 0 rather than 10; thus

3 4 2 6 5 1 8 0 7 9.

In fact, we get in this way 10 permutations all with the same set of differences modulo 10 between consecutive marks, namely

1 8 4 9 6 7 2 5 2.

One may take as representative of these 10 permutation whose first element is zero, namely

0 1 9 3 2 8 5 7 4 6.

Similarly the permutation

1 0 3 2 8 5 7 4 6

in which we have taken the marks modulo 9, is one of 9 represented by

0 8 2 1 7 4 6 3 5.

This continues on down to the case of only two marks 0 1. This suggests the following method exemplified by the case of $n = 5$. We begin with the permutation 0 1 2 3 4. Adding 1 1 1 1 1 modulo 5 five times to return to 0 1 2 3 4. We now subtract 1 1 1 1 and then add it back again, this time modulo 4, obtaining 0 1 2 3 0. Once more we add 1 1 1 1, this time modulo 5, obtaining 0 2 3 4 1. This is our next permutation and there are four others it represents. Continuing we come to 0 4 1 2 3 which, after giving 1 0 2 3 4, 2 1 3 4 0, 3 2 4 0 1, 4 3 0 1 2, 0 4 1 2 3 gives rise in turn to

0 3 0 1 2, 0 0 1 2 3, 0 0 0 1 2, 0 0 1 2 0, 0 0 2 3 1

and finally 0 1 3 4 2, our next permutation. The process finally returns to 0 1 2 3 4.

This process has been coded for the SWAC and for the 701. It is about as fast as the Walker method. If permutations with specified properties of the differences between consecutive marks are required the process is very much faster than any previous one. An example of such a property is the requirement of the differences themselves forming a permutation as in cable splicing and other management problems. The method lends itself to fractional precision representation. For $n = 8$, for example, one permutation can be made from its predecessor in 128 microseconds on the SWAC.

Another method, called the Addition Method, may be explained briefly. Starting with $x = 0$ and programming "$x + 1$ replaces x" we can generate n^n n-digit numbers to the base n. Those numbers whose digits are distinct are the desired $n!$ permutations of $0, 1, \cdots, n - 1$.

Of course for $n = 10$ this method would be very wasteful since only 1 number in $10^{10}/10! = 2756$ has distinct digits. To make this more efficient one should add (when possible) more than 1 to each successive number, in fact, as much as possible. To this effect we can formulate the following rule based on the notion of an "offending digit" of a number to the base n. If the digits of the number are not a permutation, the offending digit is the first digit which is equal to a preceding digit. If the digits are a permutation then the penultimate digit is the offending one. The rule of procedure now becomes: Add 1 to the offending digit (carrying to base n if necessary) and replace all succeeding digits by 0, 1, 2, 3, $\cdots$. Starting with the number $0\ 1\ 2 \cdots (n-1)$ we thus obtain all permutations in lexicographical order. For example for $n = 3$ we have

0 1 2	0 2 0	0 2 1	1 0 0	1 0 1	1 0 2
1 1 0	1 2 0	2 0 0	2 0 1	2 1 0	2 2 0

where the permutations are underlined. The efficiency of this method depends upon the ease with which the machine can locate the offending digit. This problem can be solved by a registry method mentioned before in connection with the Walker method. The total number of numbers generated is $n - 1$ times the number of permutations produced. The method is capable of improvement.

An unpublished method of Ulam which might be called the Random Product Method is designed only to meet requirement (c). It makes use of the fundamental fact that a permutation of a permutation is another permutation, and is used to generate random permutations for n as large as 100 or more. Two permutations P_1 and P_2 are put into the machine. With probability 1/2, P_1 or P_2 is chosen to be applied to P_1 to produce either

$$P_3 = P_1^2 \quad \text{or} \quad P_3 = P_1 P_2,$$

a new permutation. P_3 in turn is multiplied by either P_1 or P_2 depending again on a random event of probability 1/2. The process continues indefinitely. Tests of randomness are applied and new "starters" P_1, P_2 are chosen if the tests show unsatisfactory characteristics.

In conclusion it is worth pointing out that special purpose electronic equipment attached to the arithmetic unit of a fast computer can add an order of magnitude to the speed of permutation problems. For example, a simple set of n ring counters of periods 2, 3, 4, $\cdots$, n running in parallel would be very useful in the fractional precision handling of permutations. Various micro-programming techniques introduced into the design of a computer would make for more efficiency with permutation problems. Whatever improvements are made, however, one has only to increase n a little and obtain a hopeless problem about permutations.

UNIVERSITY OF CALIFORNIA,
BERKELEY, CALIFORNIA