where x is a vector of measurements on the pattern to be recognized and the matrix M is determined from making the same measurements on known representations of the patterns in the alphabet considered. The above equation firmly embeds the BB recognition scheme in the body of statistical literature associated with classification problems in general.[5] The novel aspect of the BB scheme consists in what it measures. In many other recognition schemes based on measurements, well defined and specific measurements are made—be they connectivity, number of branch points, number of intersections with horizontal and vertical grids, curvature, or whatever. In the BB scheme measurement is made of random properties of the pattern, and the numerical values of the measurements are 1 or 0; 1 if the pattern possesses the property and 0 if it doesn't. If special attention is paid to properties represented by n-typles with high transmission rate, it is possible that measurements will be made which are more useful in classifying patterns than topologically oriented ones.

Toward the end of Section II the suggestion was made that a memory matrix M should be sought which would be in some sense optimum. The statistical theory of classification provides a rationale for determining such an M. Work in this area will be reported at another time.

[5] See, for example, C. R. Rao, "Advanced Statistical Methods in Biometric Research," John Wiley and Sons, Inc., New York, N. Y.; 1952.

Correspondence

On the Number of Types of Self-Dual Logical Functions*

When a switching network for a certain logical function f is known, functions, obtained from f by variable transformations (permuting and/or complementing one or more variables), can easily be realized in the same network by relabeling and/or changing the polarity of input leads. Two logical functions are defined to belong to the same *type*, when one of the two can be transformed into another by a variable transformation. It is, then, an interesting problem to enumerate the number of the types in a given set of logical functions.

The number of symmetry types of *Boolean* functions of n variables was obtained by D. Slepian[1] in 1953. Recently B. Elspas[2] has enumerated the number of self-complementary symmetry types of *Boolean* functions.

The logical function, expressed by a network consisting purely of self-dual logical elements, such as parametrons and magnetic cores without bias input, is also self-dual. Conversely any self-dual function can be realized in a network consisting of self-dual majority decision elements.[3–5]

In this letter, the number of self-dual logical functions and the number of their symmetry types are enumerated with a modified Slepian's method.

The *dual function* $\bar{f}$ $(x_1, \cdots, x_n)$ of a logical function f $(x_1, \cdots, x_n)$ is a function defined by the formula obtained from the definition of $f(x_1, \cdots, x_n)$ through exchanging the operations of logical product $\cdot$ and logical sum $+$, namely,

$$\bar{f}(x_1, \cdots, x_n; +, \cdot) = f(x_1, \cdots, x_n; \cdot, +). \quad (1)$$

By DeMorgan's theorem, (1) can be rewritten as

$$\bar{f}(x_1, \cdots, x_n) = f'(x_1', \cdots, x_n'), \quad (2)$$

where y' denotes the complement of y.

Let an assignment of the states of input variables $x_1=\xi_1, x_2=\xi_2, \cdots, x_n=\xi_n$ be denoted by $\Xi=(\xi_1, \xi_2, \cdots, \xi_n)$ and be called an *input vector*. The *complementary vector* Ξ' of an input vector Ξ is defined as a vector which has the complemented components of Ξ. For example, the complementary vector of the vector (0, 0 1) is (1, 1, 0).

Eq. (2) means that the value of a function f for an input vector Ξ and the value of the dual function $\bar{f}$ for the complementary vector Ξ' are *complementary* to each other, where *complementary* means that if one of the two takes the value of 1 (or 0), the other has the value of 0 (or 1).

A *self-dual function* f is a function for which

$$f = \bar{f}. \quad (3)$$

For a self-dual function, the values of f for mutually complementary input vectors are complementary to each other, as indicated by (2) and (3).

The value of f for Ξ', therefore, is automatically determined by the assignment of the value for Ξ. Hence, there exist 2^{2^n-1} of self-dual logical functions of n variables.

Herewith, it is to be noted that the self-duality of a logical function is a type property, namely, that either all or none of a symmetry type is self-dual. Thus the self-dual functions of n variables can be classified with the aid of variable transformations.

The set of possible variable transformations will hereafter be denoted by O_n, which forms a finite group. The number P_n of symmetry types of the self-dual functions of n variables can be enumerated by applying Pólya and Slepian's formula;[1]

$$P_n = \frac{1}{2^n n!} \sum_C n_C x_C,$$

were C stands for a conjugate class in the group O_n, n_C is the number of elements belonging to the class C, x_C is the number of the self-dual functions which are invariant under

* Received June 20, 1961; revised manuscript received, September 11, 1961. This paper is based on an article which was published in Japan by the same author—"On the number of types of self-dual logical functions," *J. Information Processing Soc. Japan*, vol. 2, pp. 17–21; February, 1961. (In Japanese.)

[1] D. Slepian, "On the number of symmetry types of Boolean functions of n variables," *Can. J. Math.*, vol. 5, no. 2, pp. 185–193; 1953.

[2] B. Elspas, "Self-complementary symmetry types of Boolean functions," IRE TRANS. ON ELECTRONIC COMPUTERS, vol. EC-9, pp. 264–266; June, 1960.

[3] H. Takahashi, "Computing Machines," Iwanami Book Co., Tokyo; 1958. (In Japanese).

[4] Z. Kiyasu, "Mathematics for Digital Circuits," Kyoritsushuppan Book Co., Tokyo; 1960. (In Japanese).

[5] S. Muroga, I. Toda, and S. Takasu, "Theory of majority decision elements," *J. Franklin Inst.*, vol. 271, pp. 376–418; May, 1961.

operations of the class C and the summation is over all classes of O_n.

The structure of O_n is well known and *ne* has already been fixed (see Slepian[1]). The class C can be identified by specifying the number of *input cycles* of length j and the number of *even input cycles* of length j for j from 1 to n. The words *input cycle* will be used to distinguish this from the cycle of input vectors which will appear later.

Now following Slepian[1] we study the operation of an element of O_n as it *induces* a permutation among the input vectors. To see the permutation induced by an element of O_n, it is convenient to draw a diagram in the following manner. Put the 2^n input vectors in a column in natural order; operate with the given element of O_n on each of the input vectors; put the transformed vectors in the next column, and repeat this process until the initial column appears again. Using such a diagram the induced permutation among the input vectors can easily be found out. For example, let us investigate the permutation induced by the variable transformation N_{100} (12) (3). $N i \sigma$ is Slepian's notation of a variable transformation (an element of O_n), meaning the operation of first applying the permutation σ to the input variables, followed by complementation of the variables specified by the locations of 1 digits of i expressed in binary form. The above-mentioned diagram in this case is the following:

000	100	110	010	000
001	101	111	011	001
010	000	100	110	010
011	001	101	111	011
100	110	010	000	100
101	111	011	001	101
110	010	000	100	110
111	011	001	101	111

From this the permutation among the input vectors is found to consist of two cycles, ((000), (100), (110), (010)) and ((111), (001), (101)).

As is seen from the above digram, if there is a cycle $(\Xi_1, \Xi_2, \cdots, \Xi_m)$ in the induced permutation, there exists another cycle $(\Xi_1', \Xi_2', \cdots, \Xi_m')$ in that permutation, where Ξ and Ξ' are mutually complementary input vectors. Such cycles will be called *complementary* to each other. In the case that these two cycles coincide, the cycle is called *self-complementary*. For example, the cycle ((00), (01), (11), (10)) is self-complementary.

The element of O_n which induces self-complementary cycles among input vectors will be referred to as an *inadmissible* element and the element which is not inadmissible will be called *admissible*. Obviously, if an element of the class C is admissible (or inadmissible), the othe relements of the class are all admissible (or inadmissible). Hence the classes of O_n are divided into two parts, admissible classes and inadmissible classes.

A logical function remains invariant under the operation of an element of O_n, if and only if, the function is constant in value along every cycle of input vectors induced by the given element.

Any self-dual function should have complementary output values for a pair of mutally complementary input vectors, and thus no self-dual function is invariant under the operation of an inadmissible element of O_n, since the inadmissible element induces a self-complementary cycle which contains mutually complementary input vectors in it. On the other hand for an admissible element the two-valued output can be assigned arbitrarily to one of a pair of mutually complementary cycles. There exist, therefore, $2^{K(C)/2}$ self-dual functions, which are invariant under the operation of the admissible element, where C is the conjugate class to which the given element belongs and $K(C)$ is the number of cycles in the induced permutation, which is the same for all elements in the class C.

Therefore

$$x_C = 2^{K(C)/2} \quad \text{for an admissible class and}$$
$$= 0 \quad \text{for an inadmissible class.} \quad (5)$$

The calculation of $K(C)$ is described in Slepian.[1]

Thus the remaining problem is to decide whether or not a given class is admissible.

A permutation induced by an element of O_n contains a self-complementary cycle, if, and only if, each subvector of some input vector corresponding to each input cycle of the element is eventually transformed into its complementary vector and these subvectors are simultaneously transformed into their complementary vectors when operated upon repeatedly with the given element. For example the variable transformation (1234) (56) is inadmissible since the subvectors (1010) and (01) of the input vector (101001) are transformed simultaneously into their complementary vectors upon an operation with the given variable transformation.

When a subvector is first transformed into the complementary vector after operating P times with an element of O_n, P is defined as the *period* of the input cycle relative to the subvector. The subvector will be transformed into its complementary vector, when operated upon with the element $P, 3P, 5P, \cdots$, times.

The totality of the periods of an input cycle relative to all possible subvectors is called the *period set* of the input cycle.

Thus an element of O_n is inadmissible, if and only if, each input cycle in the element has a nonempty period set and, numbering the input cycles from 1 to j, there exists a set of periods $P_1, P_2 \cdots, P_j$, such that,

$$(2l_1 + 1)P_1 = (2l_2 + 1)P_2 = \cdots = (2l_j + 1)P_j, \quad (6)$$

where P_i is a period in the ith period set and l_i is a non-negative integer.

The period set can be obtained as follows. As is easily shown, the period sets are the same for all elements in a class of O_n, so that it suffices to determine the period sets for a representative element of the class. As a representative, let's take such an element that its even input cycles have no complementation, such as $(123 \cdots n)$ and its odd cycles with a single complementation upon the first position, such as $N_{100\cdots0}$ $(12 \cdots m)$.

Suppose an even input cycle, $(1, 2, \cdots, l)$, has a period P, then, the relevant subvector must be of the form in Fig. 1.

Therefore, we have

$$l = 2mP. \quad (7)$$

Conversely, for any P satisfying (7), we can find a relevant vector of the form in Fig. 1.

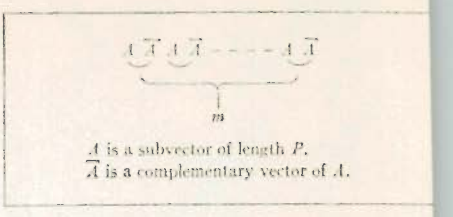

Fig. 1—A relevant subvector of an even cycle of period P.

Thus the period set of the even input cycle of length l is a set of all integers satisfying (7).

An odd input cycle $N_{100\cdots0}(\cdots\cdots l)$ with a period P must have the relevant subvector of the form in Fig. 2, since the vector, obtained by operating P times with the odd input cycle, has as its first P digits just the complementary vector of the last P digits of the original vector and the remaining $(l-P)$ digits are the same as the first $(l-P)$ digits of the original vector shifted rightward by P digits.

Therefore, we have

$$l = (2m + 1)P. \quad (8)$$

Hence the period set of an odd input cycle of length l is obtained by decomposing l into the form of (8) in all possible ways.

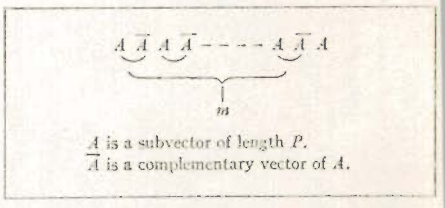

Fig. 2—A relevant subvector of an odd cycle of period P.

From the above discussion, we have the following criteria:

1) An element of O_n containing at least one even input cycle of odd length is admissible since the period set for such a cycle is empty from (7).
2) An element consisting purely of odd input cycles of odd length is inadmissible since any odd cycle of odd length has a period of one.
3) An element consisting purely of odd input cycles of even length may be either admissible or inadmissible so that the period sets of all the cycles must be calculated and examined to determine if there is a set of periods satisfying (6).

   In the case where only odd input cycles occur and where they are all of the equal length l, the element is inadmissible since all cycles share the period of l.
4) An element consisting purely of even input cycles of even length is inadmissible since all cycles have the period of one in common.
5) An element, containing at least a pair of odd input cycles of even length and of odd length, is admissible since the period of one of odd length is odd while the period of the other is even. Thus, (6) can never be satisfied.
6) An element consisting purely of odd input cycles of odd length and of even input cycles of even length is inadmissible

TABLE I
NUMBER OF TYPES OF SELF-DUAL FUNCTIONS

n	Number of Self-Dual Functions	Number of Types of Self-Dual Functions	Number of Types of of Logical Functions*	Number of Self-Complementary Types†
1	2	1	3	1
2	4	1	6	2
3	16	3	22	6
4	256	7	402	42
5	65 536	83	1228 158	4094
6	4 294 967 296	109 958	400 507 806 843 728	—

* These values are quoted from Slepian.[1]
† These values are from Elspas.[2]

since all cycles share the period of one.

7) An element, where only even input cycles and odd input cycles both of even length occur, may be either admissible or inadmissible, and must be carefully investigated.

The above exhaust all possible cases so that once the structure of a class in O_n is given, it can be determined by applying the above criteria whether or not the class is admissible.

It is now possible to determine xc in (5) and so we may calculate P_n from (4) since the other quantities, except xc, are already given by Slepian.[1]

The values of P_n up to $n=6$ are tabulated in Table I with the total number of self-dual functions. In the same entry are shown the numbers of the symmetry types[1] and of the *self-complementary* symmetry types of the logical functions of n variables. Self-complementary symmetry type is Elspas' usage,[2] meaning the symmetry type of logical functions which are of the same type as their binary complements. Thus every self-dual function is of the self-complementary type, but the converse is not true.

From the table it is observed that the numbers of self-dual functions and of their symmetry types are exceedingly small compared with the numbers for the whole of the logical functions.

IWAO TODA
Electrical Communication Lab.
Musashino-shi, Tokyo, Japan

An Electronic Generator of Random Numbers*

The random number generator described here was developed as a part of an experimental sequential detector constructed at the U. S. Navy Electronics Laboratory.[1] A sequence of ones and zeros with a predetermined probability description was needed as an input to the detector. This binomial sequence was generated by first generating uniformly distributed 14-bit random numbers and then comparing these random numbers with a preset 14-bit number.

The operation of the random number generator is based upon the following principle. Suppose a gate operating on a 1-mc clock has a single input consisting of a train of pulses of sufficient amplitude to cause the gate to assume the *one* state and of pulse width greater than 1 μsec. Then for a period equal to the input pulse width, the output of the gate is a packet of 1-mc pulses. If a train of variable width pulses is applied to the gate input, the output of the gate is a succession of packets of pulses, each packet containing a variable number of pulses. If the pulse width of the input is a random variable with a uniform distribution over a finite range the number of pulses in each packet is uniformly distributed. The expected number of packets with an odd number of pulses is the same as the expected number of packets with an even number of pulses. Hence Prob {odd} = Prob {even} = $\frac{1}{2}$.

To determine if the number of pulses in a packet is even or odd, a binary flip-flop (initially at *zero*) is triggered by each pulse in a packet, and the state of the flip-flop is sampled in the interval between packets. The *one* state of the flip-flop will indicate an odd number of pulses, the *zero* state an even number. The symbol 1 is assigned the *one* state of the flip-flop upon sampling, and the symbol 0 is assigned to the *zero* state.

The train of random width pulses mentioned above is obtained from a noise controlled pulse generator. The output of a thyratron noise generator is amplified and used to modulate the pulse width of a pulse generator, with the leading edges of the pulses occurring at fixed intervals. The pulse width varies randomly between fixed limits, the maximum width depending on the maximum modulation applied. Pulse width independence from pulse to pulse is assumed, and statistical tests gave no reason to reject such an assumption.

As mentioned above, the motivation for constructing the random number generator was to obtain a random sequence of ones and zeros with a predetermined probability of a one. This is accomplished in the following manner. Fourteen of the flip-flops previously described are triggered independently and used in parallel with a 14-bit shift register R. The state of each flip-flop is "written" into a stage of R. The contents of R are then shifted through a comparator C together with the contents of a shift register Q which contains a preset (fixed) 14-bit binary number. If the content of R is greater than the content of Q (considering the content of each as a 14-bit binary number), the output of C is a one; otherwise the output is a zero. Assuming that the 14 bits of R (*i.e.*, the 14 units of the generator) are independent, 2^{14} binary numbers are equally possible. Hence, the fixed number in Q can be chosen in such a manner that Prob $\{C=1\}$ takes on any fixed value which is a multiple of 2^{-14} and is between zero and one. C is a sequence of ones and zeros with the predetermined probability of a one.

Many problems too complex analytically can be solved by the Monte Carlo method. Monte Carlo methods evaluate integrals, simulate physical processes, etc., and require a large supply of random numbers. Methods of generating pseudo-random numbers on digital computers are known, but all of these methods are time consuming and occasionally fail to possess the desired randomness properties.

The above described random number generator could conveniently be made to operate with most binary computers either by feeding into an operational register or by using magnetic or paper tape. A 15–20 kc sampling rate can be easily attained, the generated numbers are not cyclic, and statistical tests gave no reason for rejecting the hypothesis that the generated numbers possess the desired randomness properties.

G. M. [illegible]
R. E. [illegible]
U. S. Navy Electronics Laboratory
San Diego, Calif.

Inconsistent Canonical Forms of Switching Functions*

A canonical form for switching functions (Boolean functions) is a form in which each function has a distinct representation. Such forms are useful in enumerating and synthesizing functions and in detecting functional properties.

Reed[1] and Muller[2] observed independently that an arbitrary switching function of n variables could be written in a canonical form consisting of modulo-2 addition and multiplication, denoted here by $+$ and $\cdot$ respectively. This canonical form is shown in (1a)–(1c) for one, two, and three variables.

$$n=1: \quad f(z) = k_0 \cdot 1 + k_1 \cdot z$$

$$n=2: \quad f(y,z) = k_0 \cdot 1 + k_1 \cdot z + k_2 \cdot y + k_3 \cdot yz$$

$$n=3: \quad f(x,y,z) = k_0 \cdot 1 + k_1 \cdot z + k_2 \cdot yz + k_3 \cdot y + k_4 \cdot xy + k_5 \cdot xz + k_6 \cdot xyz + k_7 \cdot x$$

The extension to more than three variables is obvious.

It can be argued from symmetry that the replacement, consistently throughout the form, of any variable by its complement ($x' = 1 + x$) should yield an equally valid canonical form. The coefficients k_i for a given function might, of course, be different for consistent complementations, applied to combinations of the variables, [illegible]

* Received August 10, 1961; revised manuscript received, November 13, 1961.

[1] G. M. Dillard and R. E. Simmons, "An Experimental Sequential Detector," U. S. Navy Electronics Lab., San Diego, Calif., Rept. 999; November, 1960.

* Received December 1, 1961.

[1] I. Reed, "A class of multiple-error-correcting codes and the decoding scheme," IRE TRANS. ON INFORMATION THEORY, vol. IT-4, pp. 38–49; September, 1954.

[2] D. Muller, "Applications of Boolean algebra to switching circuit design and to error detection," IRE TRANS ON INFORMATION THEORY, vol. [illegible] September, 1954.

Generation of permutation sequences: Part 1

1540

R. J. Ord-Smith

Computing Laboratory, University of Bradford, Bradford 7

There has been considerable interest in the last ten years or so in methods of generating sequences of arrangements of n elements in such a way that each of the $n!$ arrangements is generated once, and only once, in the sequence. We call such sequences of arrangements *permutation sequences*. In part 1 we consider several kinds of permutation sequences and describe some of their properties. Part 2 is devoted to a detailed examination of the practically most efficient six published algorithms and a discussion of implementation difficulties and compiler overheads. An Appendix to Part 2 contains an extensive bibliography of related work.

(Received November 1969)

1. Introduction

Much use of permutation algorithms has been made in the last few years in studies in Combinatorial Mathematics. Many conjectures have been proved or have fallen by computer techniques involving systematic searches. Increased efficiency in such searches is due in part to improvements in the speed of permutation algorithms. Timing these algorithms on one computer shows speed increases of a hundred or two hundredfold between the earliest and the latest.

One of the interesting applications has been the search for orthogonal Latin Square pairs of order 10, which Euler had conjectured did not exist. Searches began in earnest in the mid-1950s with the construction of large fast computers, but it was not until 1959 that the existence of the first two such pairs was announced. By 1962 thousands of pairs had been discovered, and the speed of search had increased by a factor of 10^{12}. Some of these developments are nicely described by Gardner (1966). The author (Ord-Smith, 1965) has described an application of Block Design techniques to the problem of constructing redundant fault reducing circuits using majority votetakers. Block Design theory usually regards as isomorphic two designs whose incidence matrices have permuted rows and columns, since they constitute merely re-labelled varieties within re-labelled blocks. But by successively presenting fault carrying information to a fault reducing circuit based on such a design, a particular labelling of input and output wires can produce greater fault reducing efficiency, i.e. the elimination of faults in fewer passes. Thus, two isomorphic designs can differ in this respect. Systematic generation of incidence matrices by permutation has been used to find best fault reducing circuits.

2. Computer permutation sequence algorithms

Apart from a few algorithms describing some specialised purposes involving permutations (see, for example, Hill 1968), the general algorithms all provide a common facility, the systematic generation of $n!$ arrangements of n *marks*. It is usual to provide a procedure which, on successive calls, will carry out permutations on a set of marks so that, after $n!$ calls, each arrangement of the marks has been generated once and once only. In some, one can initialise the process by providing a boolean parameter set **true**. This will be returned **false** and when subsequently given **false** in each call, will return **false** until, after $n!$ calls, it will be returned **true** again.

It is necessary, at each procedure call, to recall the point which has been reached in the sequence of arrangements. If the marks are distinct and numerical then the arrangement can itself be used to give this information. This technique has been described by Mok-Kong Shen (1962 and 1963) and featured in several of the algorithms to be described below. However, if there is to be no such restriction on the marks then the information has to be kept separately and, for this purpose, a *signature* is contained within the procedure.

3. The Tompkins algorithms

The first explicit description of computer algorithms for the generation of permutation sequences seems to have been given by Tompkins (1956). Incidentally, his paper also reviews some of the problems for which permutation algorithms are required.

The basis of the Tompkins algorithms is that the signatures are modified at each call by a process involving mixed radix arithmetic. The simplest version, attributed to M. Hall, Jnr., will show the mechanism.

Hall algorithm

A signature consists of n elements $t_1\, t_2 \ldots t_n$ constituting a mixed radix number in which the element t_i has radix k. t_1 is effectively a dummy element which remains zero.

Successive calls produce modulus arithmetic counting which takes place in the signature from the most significant radix end. For example, with $n = 5$ we would have signatures as shown in the first panel of **Table 1**.

Each signature defines an arrangement of a set of simple marks $1, 2, \ldots, n$ such that the value of t_k in the signature tells how many marks $<k$ are to the right of the mark k in the arrangement. Thus, signatures shown in panel 1 of Table 1 correspond to the arrangements of panel 2. Tompkins was already aware of advantages in generating a sequence of arrangements in 'a convenient order'. He describes a variation to this end attributed to Paige.

Table 1

Generation of Hall sequence

SIGNATURE					ARRANGEMENT				
t_1	t_2	t_3	t_4	t_5					
0	0	0	0	0	1	2	3	4	5
0	0	0	0	1	1	2	3	5	4
0	0	0	0	2	1	2	5	3	4
0	0	0	0	3	1	5	2	3	4
0	0	0	0	4	5	1	2	3	4
0	0	0	1	0	1	2	4	3	5
0	0	0	1	1	1	2	4	5	3
0	0	0	1	2	1	2	5	4	3
.	.	.	.	.	.	.	.	.	.

Tompkins–Paige algorithm

In this algorithm modification of the signature again involves modulus arithmetic counting, though this time from the least significant radix end (see **Table 2**). If t_k

Table 2

Generation of T–P sequence

SIGNATURE						ARRANGEMENT				
t_5	t_4	t_3	t_2	t_1						
0	0	0	0	0		1	2	3	4	5
0	0	0	1	0		1	2	3	5	4
0	0	0	(2)	0	*	1	2	3	4	5
0	0	1	0	0		1	2	5	3	4
0	0	1	1	0		1	2	5	4	3
0	0	1	(2)	0	*	1	2	5	3	4
0	0	2	0	0		1	2	4	5	3
0	0	2	1	0		1	2	4	3	5
0	0	2	(2)	0	*	1	2	4	5	3
0	0	(3)	0	0	*	1	2	3	4	5
0	1	0	0	0		1	5	2	3	4
0	1	0	1	0		1	5	2	4	3
.	.	.	.	.		.	.	.	.	.

is the most significant digit to be modified in a particular call, taking account of a possible carry, then the arrangement of marks suffers a cyclic permutation of the k right most marks. If the next signature obtained by a further step involves a carry, the corresponding arrangement will have occurred before. Tompkins calls these recurring arrangements 'useless starred permutations'. They have to be removed by continuing with the transmission of the carry through several digit positions, if necessary.

This algorithm, organised in reverse order, with counting down in the signature, was the second of the permutation algorithms published in the *Communications of the A.C.M.* as Algorithm 86 (Peck and Schrak, 1962). An important improvement in the rules for constructing the arrangement from the signature is that only the position of the marks is important and not their value. As already mentioned in Section 2, if one can rely on distinct numerical marks then no signature is needed at all. In fact, as we shall see in detail, the fastest algorithm of all, an improved version of Algorithm 28 (Phillips, 1967) exploits this most efficiently. Use of a signature should allow the greater generality of any marks. The early published ACM Algorithm 71 (Coveyou and Sullivan, 1961) suffers the same disadvantage as the Hall algorithm in that the marks are restricted in spite of the use of a signature.

4. Nested cycle methods

Inner–outer method

The Tompkins–Paige algorithm is effectively an example of a nested cycle method. The permutations carried out are all cyclic permutations, though the rules of construction are such as to minimise the length of the cycles and thus reduce the total number of transpositions made. There is, however, the cost of carrying [illegible] updating a signature and that of generating the unwanted starred permutations. This is sometimes called the 'inner–outer' nested cycle method in which least work is done in the innermost cycles.

Outer–inner method

Nested cycle methods have been revived more recently by Langdon (1967). He suggests the use of rotational registers to exploit cyclic generation. Langdon's is an 'outer–inner' nested cycle method in which the length of cycle is maximised whilst the generation of starred permutations is minimised. The simplest way in which to conceive the method is to regard it as identical to the Tompkins–Paige algorithm but using a Hall signature. However, by observing that there are always K successive cycles of order K it is possible to dispense with the use of a complicated signature. This is, therefore, a method requiring neither signature nor imposing restriction on the values of the marks. But the number of transpositions required is large and implementation in a high level language is very inefficient. The method is justified only if fast hardware rotation registers are available. See also references, Langdon (1968), Hill (1968), Ord-Smith (November 1967).

5. The Wells, Johnson and Trotter algorithms

The essential distinction in these and other efficient algorithms lies not in the construction of the signature, which remains virtually the same as that of Tompkins–Paige; it lies in the construction of the corresponding arrangement of marks. The inefficiency of the T–P algorithm depends not only in the production of the starred arrangements but in the use of cyclic permutations. Unless one can exploit these in a special way as Langdon suggests, they involve, in the organisation available to most computer programmers, the successive interchange of a number of pairs of marks. Wells (1961) has shown that each arrangement of a sequence can be generated from its predecessor by just a single such transposition. Although effectively of similar construction to the T–P signature, the signature of Wells uses an element t_k of radix k but with allowed values $1(1)k$ rather than $0(1)k-1$. This facilitates the des-

cription of an arrangement as a function of the corresponding signature (see details of Section 6). Thus the complete generation of $n!$ arrangements involves the generation of just $n!$ transpositions. In the T–P algorithm the number of transpositions tends to $(e-1)n! = 1{\cdot}718n!$ transpositions as n increases. Johnson (1963) has shown further that a sequence of arrangements can be found for any n in which these transpositions are adjacent. There may possibly be certain combinatorial advantages in an adjacent transposition sequence but there is a severe penalty in the complication of the algorithm.

Boothroyd's *Computer Journal* Algorithm 29 is a direct implementation of Wells' method. In his Algorithm 30 he makes use of the fact that, for $n \geqslant 5$, there is a very simple pattern in the successive generation of 23 arrangements of the four least significant marks. This gives a very fast algorithm, (Boothroyd, 1967). Improved versions of these algorithms will be given and discussed in Part 2. The existence of both Wells and Johnson sequences shows that transposition sequences are not unique. For $n = 3$ there are six distinct transposition sequences. A sequence and its exact reverse are regarded as the same. ACM Algorithm 115 (Trotter, 1962) is an efficient transposition algorithm in which the administration of the signature vector is particularly elegant and which produces a bell-ringing sequence. This is included among the six algorithms of Part 2.

6. The lexicographic algorithms

The lexicographic sequence is perhaps the most natural. It is most readily imagined by regarding the marks as labelled A, B, C, . . . then the sequence of arrangements places these in dictionary order. For example, on three marks the sequence is ABC, ACB, BAC, BCA, CAB, CBA. It is at once clear that the sequence involves more than $n!$ transpositions. ACB → BAC requires two in this example.

A succession of ACM Algorithms 87, 106, 130, 202 gradually improved the efficiency by a speed factor of 60, though Algorithm 202 remained more than twice as slow as Algorithm 115. A speed comparison of these algorithms for one computer has been made by the author, Ord-Smith (July, 1967).

Rules for the generation of the lexicographic sequence have been given by Mok-Kong Shen (1962) and are included here with a practical improvement. In the case of a lexicographic sequence there is the further problem of determining the number of transpositions involved in generating the full sequences of $n!$ arrangements. A careful examination of the lexicographic sequence shows that generation can be described as a recursive application of a simple set of rules which can be obtained from a signature of precisely Wells kind. The rules for construction of a Wells sequence of signatures and the formation of corresponding arrangements into a lexicographic sequence read right to left are as follows:

Let t_i with $i = 2(1)n$ be a set of elements of a signature in the usual way. There is no need to use the 'dummy' element t_1 which would remain unity. The elements t_i are initially set 1. At any moment let k' be the smallest k for which $t_k \neq k$.

1. (i) If $t_2 = 1$ then $t_2 := 2$.
   (ii) The permutation P_1P_2 is performed on the arrangement of marks.
2. (i) If $t_2 = 2$ then determine k' by examining t_i with increasing i and all the while setting t_i unity for $i < k'$.
   (ii) The permutation $P_{k'}P_{t_{k'}}$ is performed on the arrangement of marks followed by a reversal of the $k'-1$ least significant position marks.
   (iii) $t_{k'} := t_{k'} + 1$.
3. (i) Generation is completed when $t_n = n$.
   (ii) Reversal of all n marks restores the identity.

Rules 1 are included within 2 but an increased efficiency is gained in a computer program by dealing with the case explicitly. Generation of the first few arrangements in the sequence with $n = 5$ will make the process clear and this is shown in **Table 3**.

Table 3

Generation of lexicographic sequence

SIGNATURE t_2 t_3 t_4 t_5	ARRANGEMENT	RULE NO.	k'	$t_{k'}$
1 1 1 1	1 2 3 4 5	1		
2 1 1 1	2 1 3 4 5	2	3	1
	(3 1 2 4 5)			
1 2 1 1	1 3 2 4 5	1		
2 2 1 1	3 1 2 4 5	2	3	2
	(3 2 1 4 5)			
1 3 1 1	2 3 1 4 5	1		
2 3 1 1	3 2 1 4 5	2	4	1
	(4 2 1 3 5)			
1 1 2 1	1 2 4 3 5			
. . . .	. . . . .			

It can be seen that in the lexicographic sequence there is a complete generation of $(k-1)!$ arrangements of the first $k-1$ marks before the kth mark is involved. Then there is a single transposition followed by a reversal of the first $k-1$ marks. This reversal, if carried out without the single transposition, would have completed generation of the first $(k-1)!$ arrangements of marks and restored the identity. This process is repeated k times, whilst each of the first k marks occupies the kth position, before the $(k+1)$th mark is involved. It follows that, if S_k is the number of transpositions involved in complete generation of the kth subsequence before the $(k+1)$th mark is involved,

$$S_k = k(S_{k-1} + 1).$$

If this is used inductively in the calculation of S_n for given n we must add a further term. In the final regeneration of the identity with n marks, only a complete reversal of marks is involved (see 3(ii) above). This only involves an additional transposition compared with reversal of $n-1$ marks if n is even. Hence, if S_{k-1} is truly the number of transpositions involved in generation of $(k-1)!$ arrangements of $k-1$ marks then $S_k = (S_{k-1} + 1)k - \delta(k)$ where $\delta(k)$ is one or zero

as k is odd or even. In this way we can successively evaluate

$$S_1 = 0$$
$$S_2 = 2$$
$$S_3 = (2 + 1)3 - 1 = 8$$
$$S_4 = ((2 + 1)3 + 1)4 - 1.4 = 36$$

and in general

$$S_n = (\ldots((2 + 1)3 + 1)4 + \ldots + 1)n$$
$$-(\ldots(4.5 + 1)6.7 + \ldots + 1)n \quad \text{for } n \text{ even}$$
$$= n!\left\{1 + \tfrac{1}{2}! + \tfrac{1}{4}! + \ldots + \frac{1}{(n-2)!}\right\}$$

Hence $S_n \to \cosh 1 \times n!$

$$= 1{\cdot}583n! \quad \text{as } n \text{ increases.}$$

An explicit computer algorithm using these rules was published by the author as A.C.M. Algorithm 323 (Ord-Smith, 1968). This algorithm shows that, although involving 1·583 times as many transpositions as the Wells sequence, the rules for generating the sequence are so simple that very little additional time is taken in its construction. A certification and some discussion of this algorithm has subsequently been given by Leitch in Comm. A.C.M. (Leitch, 1969). Phillips has constructed a fast lexicographic algorithm requiring numerical and distinct marks and using no signature. In the case of non-distinct marks Algorithm 28 will produce only those distinct orderings of a higher lexicographical order within the numerical code values of the particular data representation. Algorithm 323 on the other hand will generate, from the initial marks ABCDE, all conventionally accepted dictionary orderings independent of the coded numerical values of the alphabetic characters. Since Algorithm 323 always produces $n!$ orderings some of these will be repeated if the marks are not distinct. Ord-Smith and Phillips algorithms comprise two more of the set of six fastest algorithms discussed in Part 2 (Phillips, 1967).

It is interesting to note that a slightly modified lexicographic sequence preserves most of its properties, including that of preserving the position of the kth element during generation of a $(k-1)$th arrangement of marks, whilst demanding many fewer transpositions in its generation. This is obtained from the lexicographic rules simply by replacing 2(ii) by:

a reversal of the k' least significant position marks.

The recursive formula for S_n now becomes:

$$S_k = (S_{k-1} + 1 - \delta(k))k$$

and the number of transpositions $S_n \to \sinh 1 \times n! = 1{\cdot}178n!$ as n increases. The algorithm was published by the author as A.C.M. Algorithm 308 (Ord-Smith, July 1967) but an improvement to copy exactly the rules 1, 2, 3 given above, with the modification to 2(ii), slightly improves the performance (Ord-Smith, 1969). This algorithm is discussed further in Part 2 of this paper.

8. Conclusions

Evolution of permutation algorithms has led to the production of six which are to date the fastest published. Three of these have appeared in *The Computer Journal* (Boothroyd, 1967, Phillips, 1967) and three in *Communications of A.C.M.* (Trotter, 1962, Ord-Smith, 1967, Ord-Smith, 1968). Three are transposition sequence generators and three are lexicographic.

Each of these, given explicitly in Part 2, has been rewritten in a standard form to ensure that comparisons of essential methods are, so far as is possible, compiler-independent. In the process every opportunity has been taken to implement each algorithm in the most efficient manner and this has led to worthwhile improvements in some cases. The author is indebted to the work of Mr. J. Boothroyd of the Hydro-University Computing Centre, University of Tasmania in this connection and for discussions concerning much of the material of the paper.

References

BOOTHROYD, J. (1967). Algorithms 29, 30, *The Computer Journal*, Vol. 10, p. 310.

COVEYOU, R. R., and SULLIVAN, J. G. (1961). Permutation Algorithm 71, *Comm. ACM*, Vol. 4, p. 497.

GARDNER, M. (1966). *New Mathematical Diversions from Scientific American*, Simon and Schuster, New York.

HILL, G. (1968). *Computing Reviews*, Vol. 9, Review No. 13891.

JOHNSON, S. M. (1963). An Algorithm for Generating Permutations, *Math. Comp.*, Vol. 17, p. 28.

LANGDON, G. G. (1967). Generation of Permutations by Adjacent Transposition, *Comm. ACM*, Vol. 10, p. 298.

LANGDON, G. G. (1968). Letter to Editor, *Comm. ACM*, Vol. 11, p. 392.

LEITCH, I. M. (1969). Certification of Algorithm 323, *Comm. ACM*, Vol. 12, p. 512.

MOK-KONG SHEN (1962). On the generation of Permutations and Combinations, *BIT*, Vol. 2, p. 228.

MOK-KONG SHEN (1963). Generation of Permutations in Lexicographic Order. Algorithm 202, *Comm. ACM*, Vol. 6, p. 517.

ORD-SMITH, R. J. (1965). An extension of block design methods and an application in the construction of redundant fault reducing circuits for computers, *The Computer Journal*, Vol. 8, p. 28.

ORD-SMITH, R. J. (1967). Generation of Permutations in Pseudo-Lexicographic Order, Algorithm 308, *Comm. ACM*, Vol. 10, p. 452.

ORD-SMITH, R. J. (July 1967). Remarks on Algorithms 87, 102, 130, 202, *Comm. ACM*, Vol. 10, p. 453.

ORD-SMITH, R. J. (Nov. 1967). Remarks on Langdon's Algorithm, *Comm. ACM*, Vol. 10, p. 684.

ORD-SMITH, R. J. (1968). Generation of Permutations in Lexicographic Order, Algorithm 323, *Comm. ACM*, Vol. 11, p. 417.

ORD-SMITH, R. J. (1969). Remark on Algorithm 308, *Comm. ACM*, Vol. 12, p. 638.

PECK, J. E. L., and SCHRAK, G. F. (1962). Permute, Algorithm 86, *Comm. ACM*, Vol. 5, p. 208.

PHILLIPS, J. P. N. (1967). Algorithm 28, *Comp. J.*, Vol. 10, p. 311.

RODDEN, B. E. (1968). In defence of Langdon's Algorithm, *Comm. ACM*, Vol. 11, p. 150.

TOMPKINS, C. (1956). Machine attacks on problems whose variables are Permutations, *Sec.* 3, *Proc. 6th Symp. App. Maths. Amer. Maths. Soc.*, McGraw-Hill, p. 198.

TROTTER, H. F. (1962). Perm, Algorithm 115, *Comm. ACM*, Vol. 5, p. 434.

WELLS, M. B. (1961). Generation of Permutations by Transposition, *Math. Comp.*, Vol. 15, p. 192.