Behavior Research Methods, Instruments, & Computers
1992, 24 (1), 72-77

Shuffling arrays: Appearances may be deceiving

N. JOHN CASTELLAN, JR.
Indiana University, Bloomington, Indiana

Random permutations of arrays are widely used in experimentation and simulation, and most arrays are shuffled by means of a computer-based algorithm. In this paper, I show that despite the *appearance* of a random process, a shuffling procedure must be carefully scrutinized to determine whether it actually does produce random shuffles. A general sequence-transformation procedure is developed for evaluating permutation and shuffling schemes. Applying the transformation procedure and using the criteria that all possible sequences must be equally likely and that each object in a shuffled array must be equally likely to occupy each possible position in the shuffled array, we see that one algorithm meets the criteria, while another seemingly adequate algorithm fails to meet the criteria and, in addition, exhibits systematic deviations from randomness.

In many research applications, it is necessary to construct sequences of stimuli by using random permutations. An example would be a set of stimuli that is to be presented in a different random order in each block of trials. Assume that one wishes to permute (or shuffle) N objects, and that there is a function *random*(J) that returns a uniformly distributed random integer between 1 and J.[1] Assume the objects are in an array $X()$. Many algorithms for permuting elements have been proposed, and because the problem seems conceptually straightforward, many researchers—perhaps most—code their own algorithms. Indeed, upon examination of books dealing with experiments and simulations, one frequently finds discussions of random-number generators, but relatively little attention is given to permutations. Instead, one finds comments like "Produce a random permutation of the integers ... using a pseudo-random number routine" (Harrison, 1973, p. 93). Algorithms differ widely in their efficiency and elegance. However, as will be shown, some algorithms that appear to produce random permutations fail to do so. The problem will be illustrated if we examine two algorithms.

One procedure (see, e.g., Hergert, 1987; Nilsson, 1978) is the following:

```
Algorithm 1
FOR $i = 1$ TO $N$
    $k = random(N)$
    SWAP $X(i)$, $X(k)$
NEXT $i$
```

This algorithm appears straightforward. It generates a random integer between 1 and N, swaps that element with the first element, generates a second random integer between 1 and N, swaps it with the second element, and so forth.

Another algorithm has been proposed (see, e.g., Green, 1963, 1977; Knuth, 1981; Lehman, 1977);[2] one coding for it is the following:

```
Algorithm 2
FOR $i = 1$ TO $N - 1$
    $k = random(N - i + 1) + i - 1$
    SWAP $X(i)$, $X(k)$
NEXT $i$
```

It initially generates a random integer between 1 and N, swaps the first element with the generated element, then generates a second random integer between 2 and N, swaps the second element with the second generated element, and so forth. This coding is efficient in that it requires only $N-1$ random integers to permute the array of N elements, and the elements are sampled without replacement. Comparison of Algorithm 2 with Algorithm 1 suggests that Algorithm 1 may actually permute an array faster, but accurate comparisons may require an analysis of how the coding is compiled.[3] Of course, it is easy to modify the algorithm to select a random subset of size t from the entire array by stopping the selection process at the appropriate point and using the t selected objects as the desired permutation.

(Other algorithms have been proposed: e.g., Culp & Nickles [1983] and Deni [1986] recommend sampling repeatedly with replacement from the entire list, selecting items until N different items have been selected. Such an algorithm is extremely inefficient, in that as the sampling continues, increasing numbers of random numbers must be generated. For example, for $N = 10$, the expected number of random numbers required to permute the array is 29, for $N = 20$, the expected number of random

This research was supported in part by Grant 90-0215 from the Air Force Office of Scientific Research, and it is Research Report 38 of the Indiana University Cognitive Science Program. The author would like to thank Stephen Edgell and Bernard Flury for their helpful and insightful comments on an earlier draft of this paper. Requests for reprints should be addressed to the author at the Department of Psychology, Indiana University, Bloomington, IN 47405.

Note—A Consulting Editor was given the responsibility of assigning reviewers and evaluating this manuscript.

numbers required is 72. In fact, there is some [small] probability that the shuffling would never be complete. In this paper, we will be concerned primarily with Algorithms 1 and 2 and will not pursue alternatives in detail.)

At first glance, both algorithms appear to produce random permutations of the array $X()$. However, we shall see that the first algorithm *does not* produce random permutations. The author informally discussed Algorithms 1 and 2 with colleagues who regularly use permuted stimuli in their research—experimenters, computer programmers, statisticians, and probabilists. While the "inefficiency" of Algorithm 1 was sometimes recognized, in most cases the defect in the algorithm was not recognized. In fact, when informed that one algorithm was defective, they sometimes chose Algorithm 2 as the defective algorithm.

Brysbaert (1991) tested Algorithms 1 and 2 by generating a large number of sequences. Using a goodness-of-fit criterion, he found Algorithm 1 to be deficient. In a similar analysis, Algorithm 2 yielded good fit. Brysbaert's analysis was based on simulating the algorithms. The analysis in this paper examines the algorithms directly. In any application, the user generates the number of permutation sequences necessary for the particular task. However, the effectiveness of a shuffling algorithm is best understood by letting the algorithm generate the maximum possible number of sequences that it is capable of generating. Our analysis will be based on those sequences. First we indicate two basic requirements for random permutations: all possible permutations should be equally likely, and, after the array has been shuffled, element $X(j)$ should be equally likely to be in final position 1, 2, ..., N.

All Permutations Must Be Equally Likely[4]

For N objects, there are $N!$ permutations, each of which should be equally likely. Algorithm 2 will generate $N!$ equally likely permutations. However, Algorithm 1 produces N^N sequences. For each permutation to be equally likely when Algorithm 1 is applied, N^N would have to be an integral multiple of $N!$ Except for $N = 2$, this does not hold. For example, for $N = 3$, $3^3 = 27$ is not an integral multiple of $3! = 6$. (Note: The criterion given here is a *necessary* condition; it is not *sufficient.*)

While it is strictly true that Algorithm 1 does not produce equally likely permutations (for $N > 2$), one might argue that the generated permutations might "almost" be equally likely with each of the $N!$ permutations having frequencies of occurrence that differ from INT($N^N/N!$) only by 1 or so. We shall see that in reality the deviations from equally likely frequencies are extreme and systematic.[5]

The Probability of Array Elements by Serial Position Must Be Constant

It is possible to write a series of transformation matrices that show what happens to the elements of the array after each successive random swap. We can use these transformation matrices to show that (1) Algorithm 1 does not produce an appropriate distribution of elements, and (2) Algorithm 2 does produce an appropriate distribution. In addition, we will see that the transformation approach could be used to test other generation procedures.

To begin, we need a base or initial matrix to show the way that elements occupy positions in the original array. Of course, that array is set to the identity matrix. The rows indicate an element number from the array at the original step, and the columns denote the current position:

$$\mathbf{B_0} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & \cdots & 0 & 1 & 0 \\ 0 & \cdots & 0 & 0 & 1 \end{bmatrix}.$$

The base matrix is the same for any permutation scheme.[6]

Algorithm 1. N iterations are necessary to permute the entire array. At Step 1, the (original) $X(1)$ could stay in its original position or could move to any of the other $N-1$ positions; hence the first row contains unities. All of these moves are equally likely. Note also, that at Step 1, elements 2, 3, ..., N can move only to position 1; hence the first column contains unities. Now consider the second element (row 2). It can move from its current position (2) to the first position, or it can remain in the same position $N-1$ ways. A similar argument holds for each of the remaining elements (rows). We could write the transformation matrix for transforming the original array to the array at Step 1 as follows:

$$\mathbf{S_{0\to1}} = \begin{bmatrix} 1 & 1 & 1 & \cdots & \cdots & 1 \\ 1 & N-1 & 0 & \cdots & \cdots & 0 \\ 1 & 0 & N-1 & 0 & \cdots & 0 \\ \vdots & \vdots & & \ddots & & \vdots \\ 1 & 0 & \cdots & 0 & N-1 & 0 \\ 1 & 0 & \cdots & 0 & 0 & N-1 \end{bmatrix}.$$

Thus, $\mathbf{S_{0\to1}}$ shows the number of ways that each initial element (row) can be in various positions after the initial element is randomly selected and swapped with the element in the first position. For the transitions from Step 1 to Step 2, we have

$$\mathbf{S_{1\to2}} = \begin{bmatrix} N-1 & 1 & 0 & 0 & \cdots & 0 \\ 1 & 1 & 1 & 1 & \cdots & 1 \\ 0 & 1 & N-1 & 0 & \cdots & 0 \\ 0 & 1 & 0 & N-1 & 0 & 0 \\ \vdots & \vdots & & & & \vdots \\ 0 & 1 & 0 & \cdots & N-1 & 0 \\ 0 & 1 & 0 & \cdots & 0 & N-1 \end{bmatrix}.$$

Again, each row indicates how the element currently in position i can move to position j. And in general,

$$S_{(t-1)\to t} = \begin{bmatrix} N-1 & 0 & 0 & 0 \cdots & 0 & 1 & 0 & \cdots & 0 \\ 0 & N-1 & 0 & 0 \cdots & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & N-1 & 0 \cdots & 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & & & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \cdots & 0 & N-1 & 1 & 0 \cdots & 0 \\ 1 & 1 & \cdots & \cdots & 1 & 1 & 1 & 1 \cdots & 1 \\ 0 & 0 & \cdots & \cdots & 0 & 0 & 1 & N-1 \cdots & 0 \\ \vdots & \vdots & & & \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & \cdots & \cdots & 0 & 1 & 0 \cdots & N-1 \end{bmatrix}.$$

That is, $S_{(t-1)\to t}$ is an $N \times N$ matrix with $N-1$ on the diagonal (except element t, t), unities in the tth row and tth column, and zeros elsewhere.

To obtain the number of times an element is generated in each position at step t, we take the product of these transformation matrices:

$$S_{0\to t} = B_0 S_{0\to 1} S_{1\to 2} \cdots S_{(t-1)\to t}.$$

The matrix $S_{0\to N}$ is the matrix with the number of times each element from the original array appears in each position in the N^N permutations. If each position is equally likely, the matrix would have equal frequencies of N^{N-1} in each cell.

Table 1 contains the matrices $S_{0\to N}$ for $N = 3$, 4, and 5. The interpretation of the matrices is as follows. For $N = 3$, $3^3 = 27$ sequences can be generated. The second element of the original array will appear in the first position of the permuted array for 10 sequences, in the second position for 8 sequences, and the third position for 9 sequences. Clearly, the second element is not equally likely to appear in each position. Indeed, except for the first element of the (original) array, *none* of the elements is equally likely to appear in each position.

Algorithm 2. $N-1$ iterations are necessary to permute the array. At Step 1, the (original) $X(1)$ could stay in its original position, or could move to any of the other $N-1$ positions. All of these moves are equally likely. Note also that at Step 1, elements 2, 3, ..., N can move only to position 1. We could write the transformation matrix for transforming the original array to the array at Step 1 as follows:

$$Q_{0\to 1} = \begin{bmatrix} 1 & 1 & 1 & \cdots & \cdots & 1 \\ 1 & N-1 & 0 & \cdots & \cdots & 0 \\ 1 & 0 & N-1 & 0 & \cdots & 0 \\ \vdots & \vdots & & \ddots & & \vdots \\ 1 & 0 & \cdots & 0 & N-1 & 0 \\ 1 & 0 & \cdots & 0 & 0 & N-1 \end{bmatrix}.$$

The transition matrix from Step 1 to Step 2 is the following:

$$Q_{1\to 2} = \begin{bmatrix} N-1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 1 & 1 & \cdots & 1 \\ 0 & 1 & N-2 & 0 & \cdots & 0 \\ 0 & 1 & 0 & N-2 & 0 & 0 \\ \vdots & \vdots & \vdots & & & \vdots \\ 0 & 1 & 0 & \cdots & N-2 & 0 \\ 0 & 1 & 0 & \cdots & 0 & N-2 \end{bmatrix}.$$

And in general, the transition matrix from step $t-1$ to step t is the following:

$Q_{(t-1)\to t} =$

$$\begin{bmatrix} N-t+1 & 0 & 0 & 0 \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & N-t+1 & 0 & 0 \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & N-t+1 & 0 \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \cdots\ 0 & N-t+1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & \cdots\ 0 & 0 & 1 & 1 & \cdots & 1 \\ 0 & 0 & \cdots & \cdots\ 0 & 0 & 1 & N-t & \cdots & 0 \\ \vdots & \vdots & & & \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & \cdots\ \cdots & 0 & 1 & 0 & \cdots & N-t \end{bmatrix}.$$

That is, $Q_{(t-1)\to t}$ is an $N \times N$ matrix with $N-t+1$ on the diagonal *before* the tth row, $N-t$ on the diagonal *after* the tth row, unities in the tth row and tth column beginning with the tth element, and zeros elsewhere.

To obtain the number of times an element is generated in each position, we take the product of the transformation matrices:

$$Q_{0\to t} = B_0 Q_{0\to 1} Q_{1\to 2} \cdots Q_{(t-1)\to t}.$$

The matrix $Q_{0\to N}$ is the matrix with the number of times each element from the original array appears in each po-

Table 1
Algorithm 1 Transformation Matrices $S_{0\to N}$ for $N = 3$, 4, and 5

$N = 3$					
	9	9	9		
	10	8	9		
	8	10	9		
$N = 4$					
	64	64	64	64	
	75	57	60	64	
	63	72	57	64	
	54	63	75	64	
$N = 5$					
	625	625	625	625	625
	756	564	580	600	625
	656	720	544	580	625
	576	640	720	564	625
	512	576	656	756	625

Table 2
Algorithm 2 Transformation Matrices $Q_{0\to N}$ for $N = 3$, 4, and 5

$N = 3$					
	2	2	2		
	2	2	2		
	2	2	2		
$N = 4$					
	6	6	6	6	
	6	6	6	6	
	6	6	6	6	
	6	6	6	6	
$N = 5$					
	24	24	24	24	24
	24	24	24	24	24
	24	24	24	24	24
	24	24	24	24	24
	24	24	24	24	24

Table 3
Probability of Element Being in Final Position k Given Initial Position j in $N = 10$ Element Array for Algorithm 1

Initial	Final Position k									
Position	1	2	3	4	5	6	7	8	9	10
1	.1000	.1000	.1000	.1000	.1000	.1000	.1000	.1000	.1000	.1000
2	.1287	.0943	.0948	.0953	.0959	.0966	.0973	.0981	.0990	.1000
3	.1197	.1240	.0901	.0911	.0922	.0935	.0949	.0964	.0981	.1000
4	.1116	.1159	.1207	.0873	.0889	.0907	.0927	.0949	.0973	.1000
5	.1044	.1087	.1134	.1188	.0859	.0882	.0907	.0935	.0966	.1000
6	.0978	.1021	.1069	.1122	.1181	.0859	.0889	.0922	.0959	.1000
7	.0919	.0962	.1010	.1063	.1122	.1188	.0873	.0911	.0953	.1000
8	.0866	.0909	.0957	.1010	.1069	.1134	.1207	.0901	.0948	.1000
9	.0818	.0861	.0909	.0962	.1021	.1087	.1159	.1240	.0943	.1000
10	.0775	.0818	.0866	.0919	.0978	.1044	.1116	.1197	.1287	.1000

sition across the $N!$ permutations. (However, note that $Q_{0\to(N-1)} = Q_{0\to N}$, since the transformation $Q_{(N-1)\to N} = I$, the identity matrix.) If each position is equally likely, the matrix would consist of equal frequencies of $(N-1)!$ in each cell.

Table 2 contains the matrices for $N = 3$, 4, and 5. Clearly, each element of the original array is equally likely to be in each position. The Appendix contains a proof that the random permutations are indeed equally likely, and that each initial element is equally likely to appear in each final position of the permuted array.

Examining patterns in the bias in Algorithm 1 is difficult for the small values of N used in Table 1. In order to see the nature of the bias more fully, the transition matrix for $N = 10$, $S_{0\to10}$, was generated. Since this matrix has very large integer values, the entries were converted into conditional probabilities. The entries are the probabilities, $P_N[k|j]$, that an element is in position i given that it was initially in position j in the original array (assuming an N-element array). Inspection of Table 3 shows that, as we saw earlier, the first element is equally likely to be in each position, and that elements 2, 3, ..., N are not equally likely to be in each position. Indeed, after the shuffle, element j is most likely to be in the $(j-1)$st position. The distribution is "saw-toothed"—for (original) position j, the probability of occupying a final location increases from (final) position 1 to (final) position $j-1$, suddenly decreases at (final) position j, and then slowly increases to $1/N$ for the Nth position. This pattern can be seen more clearly in Figure 1. This systematic pattern could have profound effects on simulations or experiments in which equal probabilities are assumed for the elements. Of course, for Algorithm 2, the plots would be flat at $1/N$ for all positions.

In testing Algorithm 1, Brysbaert (1991) generated 100,000 permutations for $N = 10$ and constructed a table of observed frequencies similar to that in Table 3. He found that the observed distribution deviated significantly from a uniform distribution. When one compares the data from his simulation with the expected probabilities in Table 3, the maximum deviation is .003, and the average absolute deviation for the 100 cells is less than .001.

Permuting subsets. In some situations, a person wishes only to select t elements from the entire array of N elements. The transformation matrices above can give the expected frequencies of elements in each position when a subset is needed. However, a slight change in interpretation is necessary. The matrices $S_{0\to t}$ and $Q_{0\to t}$ describe the arrangement of all elements in all positions. If we are selecting only a subset of t elements, we need only the first t columns of the final matrix. Table 4 gives the probability that each element will appear when a subset of size t is selected. It is seen that for Algorithm 1, the elements are not equally likely (except for a sample of Size 1), whereas for Algorithm 2, the frequencies of occurrence are the same.

The Appendix contains a proof that Algorithm 2 does indeed produce sequences with the desired property for all values of N and for subsets of size t as well as the entire sequence itself.

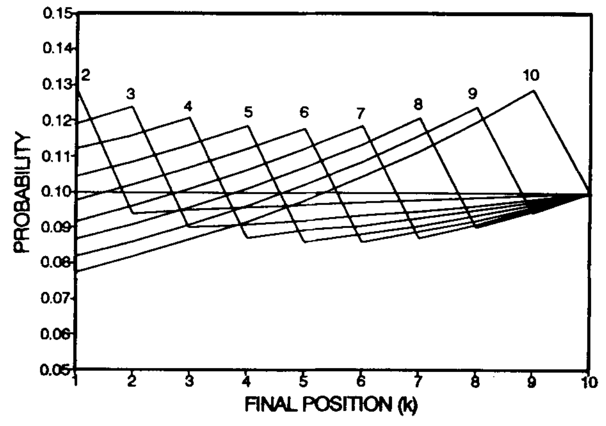

Figure 1. Probability that an element will be in final position k, given starting position j, for a 10-element array using Algorithm 1. Each line represents a different starting position, which is indicated above each line. The line for Starting Position 1 is flat at $p = .10$.

Table 4
Probability That an Element in Various Positions Will Be Included in Samples of Size J From Set of Size $N = 5$ for Algorithm 1

Original	Sample Size J			
Position	1	2	3	4
1	.200	.200	.200	.200
2	.200	.260	.227	.210
3	.200	.180	.248	.218
4	.200	.180	.163	.224
5	.200	.180	.163	.148

Note—For Algorithm 2, the probability is .2 for all initial positions and all sample sizes.

Algorithm misspecification. One early reviewer of this paper asked what would happen if, when implementing Algorithm 2, a programmer misspecified the range in using the random-number generator. It turns out that the consequences can be quite serious. Consider the following algorithm:

```
Algorithm 3
FOR $i = 1$ TO $N - 1$
    $k = random(N - i) + i$
    SWAP $X(i)$, $X(k)$
NEXT $i$
```

In this case, the algorithm generates random integers, but instead of generating random integers between i and N as Algorithm 2 does, the integers vary from $i+1$ to N.[7] In this case, we can write the transformation matrices to examine the effect of the coding. The first thing to notice is that in generating elements for swapping, the ith element *must* be switched with one of the succeeding elements; that is, an element cannot be swapped with itself. For the first transition, we have

$$\boldsymbol{R}_{0\to 1} = \begin{bmatrix} 0 & 1 & 1 & \cdots & \cdots & 1 \\ 1 & N-2 & 0 & \cdots & \cdots & 0 \\ 1 & 0 & N-2 & 0 & \cdots & 0 \\ \vdots & \vdots & & \ddots & & \vdots \\ 1 & 0 & \cdots & 0 & N-2 & 0 \\ 1 & 0 & \cdots & 0 & 0 & N-2 \end{bmatrix}.$$

The transition matrix from Step 1 to Step 2 is the following:

$$\boldsymbol{R}_{1\to 2} = \begin{bmatrix} N-2 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 1 & \cdots & 1 \\ 0 & 1 & N-3 & 0 & \cdots & 0 \\ 0 & 1 & 0 & N-3 & 0 & 0 \\ \vdots & \vdots & \vdots & & & \vdots \\ 0 & 1 & 0 & \cdots & N-3 & 0 \\ 0 & 1 & 0 & \cdots & 0 & N-3 \end{bmatrix}.$$

And, in general, the transition matrix from step $t-1$ to step t is the following:

$$\boldsymbol{R}_{(t-1)\to t} =$$

$$\begin{bmatrix} N-t & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & N-t & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & N-t & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & & & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \cdots & 0 & N-t & 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & \cdots & 0 & 0 & 0 & 1 & \cdots & 1 \\ 0 & 0 & \cdots & \cdots & 0 & 0 & 1 & N-t-1 & \cdots & 0 \\ \vdots & \vdots & & & & \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & \cdots & \cdots & 0 & 1 & 0 & \cdots & N-t-1 \end{bmatrix}.$$

That is, $\boldsymbol{R}_{(t-1)\to t}$ is an $N\times N$ matrix with $N-t$ on the diagonal *before* the tth row, $N-t-1$ on the diagonal *after* the tth row, unities in the tth row and tth column beginning with the $(t+1)$st element, and zeros elsewhere, including 0 for the t,tth cell. It is important to note that the generation process ends with $\boldsymbol{R}_{(N-2)\to(N-1)}$, where the $(N-1)$st and Nth elements are swapped.

To obtain the number of times an element is generated in each position, we take the product of these transformation matrices:

$$\boldsymbol{R}_{0\to t} = \boldsymbol{B}_0\boldsymbol{R}_{0\to 1}\boldsymbol{R}_{1\to 2}\cdots\boldsymbol{R}_{(t-1)\to t}, \qquad t = 1, 2, \ldots, N-1.$$

The effect of applying Algorithm 3 is that after permutation of the entire array, an element cannot be found in its original position, although it is equally likely to be in all other positions. Moreover, it should be noted that even with the constraint that an element cannot remain in its original position, not all of the remaining permutations are possible. For Algorithm 3, only $(N-1)!$ permutations can be generated because, as noted above, the last transformation is deterministic. The transition frequencies $\boldsymbol{R}_{0\to(N-1)}$ are

$$\boldsymbol{R}_{0\to(N-1)} = \begin{bmatrix} 0 & (N-2)! & (N-2)! & \cdots & (N-2)! \\ (N-2)! & 0 & (N-2)! & \cdots & (N-2)! \\ \vdots & & & & \vdots \\ (N-2)! & \cdots & (N-2)! & 0 & (N-2)! \\ (N-2)! & \cdots & (N-2)! & (N-2)! & 0 \end{bmatrix}.$$

Summary

The analyses in this paper show that despite appearances, algorithms for random permutations or shuffling may not produce sequences with desired properties. The consequence of inappropriate shuffling cannot be overstated. Researchers should always take extreme care in choosing algorithms in their research. Only well-documented algorithms should be used. Documentation does not mean that the algorithm has been described in print—even in a reputable journal or book. Appropriate documentation includes an analysis of the properties and behavior of the algorithm. The transformation procedures described in this paper have demonstrated the accuracy of one algorithm and the deficiencies of two others. The transformation technique could be generalized to test other algorithms and situations as well.

REFERENCES

BRYSBAERT, M. (1991). Algorithms for randomness in the behavioral sciences: A tutorial. *Behavior Research Methods, Instruments, & Computers*, **23**, 45-60.

CULP, G., & NICKLES, H. (1983). *An Apple for the teacher: Fundamentals of instructional computing*. Monterey, CA: Brooks/Cole.

DENI, R. (1986). *Programming microcomputers for psychology experiments*. Belmont, CA: Wadsworth.

GREEN, B. F., JR. (1963). *Digital computers in research*. New York: McGraw-Hill.

GREEN, B. F., JR. (1977). FORTRAN subroutines for random sampling without replacement. *Behavior Research Methods & Instrumentation*, **9**, 559.

HARRISON, G. (1973). The computer in psychology experiments. In M. J. Apter & G. Westby (Eds.), *The computer in psychology* (pp. 85-124). New York: Wiley.

HERGERT, D. (1987). *Microsoft QuickBASIC*. Redmond, WA: Microsoft Press.

KNUTH, D. E. (1981). *The art of computer programming: Vol. 2. Seminumerical algorithms* (2nd ed.). Reading, MA: Addison-Wesley.

L'ECUYER, R. (1990). Random numbers for simulation. *Communications of the ACM*, **33**, 85-97.

LEHMAN, R. S. (1977). *Computer simulation and modelling*. Hillsdale, NJ: Erlbaum.

NILSSON, T. H. (1978). Randomization without replacement using replacement without losing your place. *Behavior Research Methods & Instrumentation*, **10**, 419.

RUBENKING, N. J. (1991, August). Creating a random set. *PC Magazine*, **10**(14), 463-464.

NOTES

1. We acknowledge that the function *random(J)* that produces the random numbers is of considerable interest. Indeed, most of the literature on random-number generation focuses on the properties of random-number generators rather than on the *use* of the random-number generator. See Brysbaert (1991) and L'Ecuyer (1990) for discussion of random-number generators. The latter paper is an extremely thorough and up-to-date survey.

It also should be noted that *random(J)* may not be implemented directly in some programming languages; rather, it must be obtained by transforming a random number in the interval (0, 1) to an integer. For example, in QuickBASIC or GWBASIC, the function RND returns a number x, in the interval $0 < x < 1$, and to emulate *random(J)*, the code would be INT(J*RND + 1), while in TurboPascal the function **Random** returns a variable x, where $0 \leq x < 1$. As will be noted later, such transformations to integers must be done with care.

2. It should be noted that in each of the references cited, the algorithm is described as one algorithm. There is no caution about other algorithms. Of course, programmers cannot be provided with an encyclopedic listing of "poor" procedures. Nonetheless, when problems with algorithms go beyond inefficiency and lack of elegance, warnings about conceptual errors would be helpful.

3. Other forms of Algorithm 2 may be more efficient. For example,

```
Algorithm 2A
FOR $i = N$ TO 2 STEP $-1$
    $k = random(i)$
    SWAP $X(i)$, $X(k)$
NEXT $i$
```

permutes the array by moving from element N to 1. Algorithm 2 is presented in the paper because it is the form that is often cited.

4. This criterion for random permutations was pointed out to the author by Stephen Edgell.

5. In a recent article on shuffling arrays, Rubenking (1991) discussed a coding of Algorithm 1 and used the argument given here that N^N is not a multiple of $N!$, but concluded, "and they (the orderings) appear with almost the same frequency." Although he did go on to suggest an implementation of Algorithm 2, the systematic and extreme deviations from equal frequency of occurrence resulting from the use of Algorithm 1 was not recognized.

6. Strictly speaking, the initial matrix $\boldsymbol{B}_0$ is not necessary. It is included only to permit consideration of more general initial configurations.

7. This error could occur fairly easily and might not be detected. If the random-number generator generates a floating-point number, say in the interval (0, 1), the programmer/coder must be very careful to determine that the transformation or rescaling to integer values spans the intended range.

APPENDIX
Proof That All Initial Positions Are Equally Likely in All Final Positions for Algorithm 2

Recall that $\boldsymbol{Q}_{0\to t}$ is the transition matrix of the frequencies with which initial (row) object j appears in position k when t objects have been selected from N. If all initial positions are equally likely to appear in each of the k, $k \leq N$ final positions, the frequencies in the first t rows and columns of $\boldsymbol{Q}_{0\to t}$ must be equal [and should be equal to $(N-1)(N-2)\cdots(N-t+1)$].

We can write the transition matrix $\boldsymbol{Q}_{(t-1)\to t}$ as a partitioned matrix, separating the initial $t-1$ rows and columns from the remaining $N-t+1$ rows and columns. Thus

$$\boldsymbol{Q}_{(t-1)\to t} = \begin{bmatrix} (N-t+1)\boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0} & \begin{matrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & N-t & 0 & \cdots & 0 \\ 1 & 0 & N-t & & 0 \\ \vdots & \vdots & & \ddots & \vdots \\ 1 & 0 & \cdots & 0 & N-t \end{matrix} \end{bmatrix},$$

where $\boldsymbol{I}$ is the identity matrix. After writing a few of the transition products $\boldsymbol{Q}_{0\to 2}$, $\boldsymbol{Q}_{0\to 3}$, and so forth, it appears that the matrix is of the form

$$\boldsymbol{Q}_{0\to(t-1)} = k_{t-2} \begin{bmatrix} \boldsymbol{E}_{(t-1),(t-1)} & \boldsymbol{E}_{(t-1),(N-t+1)} \\ \boldsymbol{E}_{(N-t+1),(t-1)} & (N-t+1)\boldsymbol{I} \end{bmatrix},$$

where $k_{t-2} = (N-1)(N-2)\cdots(N-t+2)$, and $\boldsymbol{E} = (1)$—that is, a matrix whose elements are all equal to 1. This pattern can be checked by induction (by showing it is true for $t = 1$, assume it is true for $t-1$ and prove it is true for t). By definition, it is true for $\boldsymbol{Q}_{0\to 1}$ ($t = 1$). Taking the products, we have

$$\boldsymbol{Q}_{0\to t} = \boldsymbol{Q}_{0\to(t-1)}\boldsymbol{Q}_{(t-1)\to t}$$

$$= k_{t-2} \begin{bmatrix} \boldsymbol{E}_{(t-1),(t-1)} & \boldsymbol{E}_{(t-1),(N-t+1)} \\ \boldsymbol{E}_{(N-t+1),(t-1)} & (N-t+1)\boldsymbol{I} \end{bmatrix}$$

$$\times \begin{bmatrix} (N-t+1)\boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0} & \begin{matrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & N-t & 0 & \cdots & 0 \\ 1 & 0 & N-t & & 0 \\ \vdots & \vdots & & \ddots & \vdots \\ 1 & 0 & \cdots & 0 & N-t \end{matrix} \end{bmatrix}.$$

After multiplying and simplifying, we have

$$\boldsymbol{Q}_{0\to t} = (N-k+1)k_{t-2} \begin{bmatrix} \boldsymbol{E}_{(t),(t)} & \boldsymbol{E}_{(t),(N-t)} \\ \boldsymbol{E}_{(N-t),(t)} & (N-t)\boldsymbol{I} \end{bmatrix},$$

and since $(N-k+1)k_{t-2} = k_{t-1}$, the proof is complete.

(Manuscript received October 23, 1990;
revision accepted for publication February 8, 1991.)