On Random Shuffles & Subsets

T. A. Jenkyns, Department of Mathematics
D. McCarthy, Dept. of Computer Science
Brock University, St. Catharines, Ontario
Canada L2S 3A1

Abstract. Four simple algorithms are presented for producing a random rearrangement of the values in an array X[1], X[2], ..., X[n] using a pseudo-random number generator. These range from one that is naive, inefficient, and biased to one that is elegeant, efficient, and unbiased. By taking the first k entries in X one obtains a random (ordered) subset of size k.

§1. Introduction.

There are many instances where a random permutation of the integers from 1 to n is required. If these integers were put into an array X by

```
for i ← 1 to n do X[i] ← i end
```

we would require a random rearrangement of the values in x. It is this equivalent problem that is addressed, that of shuffling the values in an array X (the opposite of sorting X).

We will assume that we have available a pseudo-random number generator that accepts as input two integers a ≤ b, and that produces an integer value RAND, where

```
a ≤ RAND(a,b) ≤ b
```

and where each of b-a+1 possible values is equally likely. We implicitly assume that the values RAND generates are independent; that is, any particular call of RAND and the value it produces have no effect on the outcome of any other call. Section 3.5 of Knuth [3] is a detailed discussion of the philosophical and practical difficulties of constructing such generators.

Using an intuitive notion of randomness, we develop three different algorithms to produce a random shuffle, and we determine the computational cost of each. The third and more efficient algorithm is demonstrated to be random in a formal sense. A fourth algorithm that appears to use more "randomness" is then proved to be biased. The final section shows that a minor modification of this algorithm may be used to produce a random subset.

The book by Nijenhuis and Wilf [4] is a wonderful catalogue of algorithms for solving combinatorial problems especially those involving sequences and subsets. For further study of the design and analysis of algorithms we also suggest [1] and [2].

§2. Mixing the Values in X

We begin by devising a method to imitate shuffling cards in a deck where the values in array X change their positions in some unpredictable manner and where any value may end up in any position. Suppose that we select two positions, p and q, at random and then interchange the values of X[p] and X[q], and suppose that we repeat this many times. The algorithm so constructed is

Algorithm 1

```
for i ← 1 to M do
    p ← RAND(1,n);
    q ← RAND(1,n);
    T ← X[p];
    X[p] ← X[q];
    X[q] ← T;
end.
```

When M is large, say n^2, the values of X will certainly be thoroughly mixed. Since 2M values are moved, each value moves 2M/n = 2n times on average and may end up in any position. The computational cost of this algorithm is:

```
2M calls of RAND, and M interchanges of elements of X.
```

§3. The Effect of the Mixing Process

Let X′[1], X′[2], ..., X′[n] denote the final values in the array, and let's examine what happens to each input value as it travels to its final position. Taking the input values in order, we see that X[j] must have an equal chance of ending up in any "available" position; i.e., any position except where X[1], X[2], ..., X[j-1] ended. Thus we need some way to mark the "available" positions for the value X[j]. Let's use a second array X′ indexed from 1 to n and initialized to some special value Z which does not occur in X. We could then produce X′ with

Algorithm 2

```
for i ← 1 to n do X'[i] ← Z end;
for j ← 1 to n do
    repeat i ← RAND(1,n) until X'[i] = Z;
    X'[i] ← X[j]
end.
```

It is clear that this algorithm produces a shuffle but it is not clear how many calls of RAND are made. In particular, how hard will it be to find the unique "available"

position in X' for the last value, X[n]? There is the daunting possibility that it might take a great many trials to find that one position for X[n], and this possibility, no matter how improbable, might cause a programmer to abandon Algorithm 2 and to settle for Algorithm 1 despite the inefficiency of that first algorithm.

On the other hand, a programmer who knows a bit about probability (especially the geometric distribution) might make the following analysis. For a given value of j, there are j-1 positions in X' used already, so the probability that a randomly selected position is "available",

$$p_j = \frac{n-(j-1)}{n}$$

and therefore, the expected number of selections required to find the first "available" position is

$$E_j = 1/p_j = \frac{n}{n-(j-1)}.$$

Then, the expected number of calls of RAND for the whole shuffle,

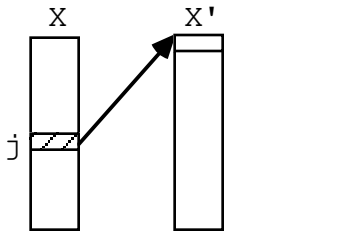
$$E = \sum_{j=1}^{n} E_j = n[\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{2} + 1].$$

With algorithm 2, on average, the cost of a shuffle is about E calls of RAND, and 2n assignments. But because $ln(n) + 0.5 < 1 + 1/2 + \ldots + 1/n \leq ln(n) + 1$ the number E is at most $n + n * ln(n)$ and therefore is much smaller than $2n^2$, so even though Algorithm 2 uses twice the space of Algorithm 1, on average it is much faster.

§4. A Better Way

Let's again look at the effect of the mixing process but this time from the viewpoint of the output. The value finally in X'[1] is one of X[1]..X[n] where each of these "eligible" values has an equal chance of ending up in position X'[1], as if we had done

```
j←RAND(1,n);   X'[1]←X[j]
```



After this assignment, the value X[j] is no longer "eligible" to be placed in X'[2] (or X'[3], ..., X'[n]) but each other value in X is eligible to be placed in X'[2] and must have an equal chance of being selected. To indicate X[j] is an ineligible value for X'[2] we could mark it so by assigning it the value Z or we could set it "aside" by interchanging its value with X[1].

Now we select one of these eligible values of X'[2] at random and interchange it with X[2].

Continuing in this way, we do not need array X', and we obtain our final solution to the shuffling problem.

Algorithm 3

```
for i ← 1 to (n – 1) do
    j ← RAND(i,n);
    T ← X[i];
    X[i] ← X[j];
    X[j] ← T
end.
```

The cost of this third algorithm is exactly n-1 calls of RAND, and n-1 interchanges for each shuffle. It is very simple, and it is more efficient in time and space than the others, and it does not even require finding a value Z not in the input array.

§5. Random Shuffles

There are three notions of randomness related to shuffles:

(1) X is shuffled to produce X' (denoted $X \rightarrow X'$) by some process involving chance or unpredictability.

(2) the process is UNBIASED; i.e.
Prob($X \rightarrow X'$) = α for all possible results X' so $\alpha = 1/n!$.

(3) the process is LOCALLY UNBIASED; i.e.
Prob(X[j] moves to position i) = β for all i and j so $\beta = 1/n$.

Property (2) implies (3) since

Prob(X[j] moves to position i) = Σ {Prob($X \rightarrow X'$): $X[j] \rightarrow X'[i]$} = $(n-1)!*\alpha = 1/n$.

However, (3) does NOT imply (2) since when n = 3, any process that shuffles $X = (x_1, x_2, x_3)$ to

(x_2, x_3, x_1) one third of the time, to
(x_3, x_1, x_2) one third of the time, and to
(x_1, x_2, x_3) one third of the time,

would be locally unbiased but nevertheless biased since $X' = (x_2, x_1, x_3)$ is never produced.

Theorem 1. Algorithm 3 is an unbiased shuffling process.

Proof: Our assumptions about the generator RAND imply that Algorithm 3 generates a sequence of n-1 independent j-values $J=(j_1, j_2, ..., j_{n-1})$ where for each i, $j_i \in \{i, i+1, ..., n\}$ and where each of these n-i+1 possible values for j_i is equally likely. The number of such sequences J is:

(n-1+1)(n-2+1)(n-3+1) ... ((n-[n-1]+1) = n!

and each possible sequence is equally likely to be generated. Each such sequence J produces a permutation of the values in X. If $J_1 = (a_1, ..., a_{n-1}) \neq J_2 = (b_1, ..., b_{n-1})$ then there is a first index k where $a_k \neq b_k$ but $a_i = b_i$ for $1 \leq i < k$; then in the resulting shuffles X_1' and X_2', we get $X_1'[i] = X_2'[i]$ for $1 \leq i < k$ but $X_1'[k] \neq X_2'[k]$. Thus each of the n! permutations of the values in X must be produced by a unique sequence J and therefore each permutation is equally likely to be generated. QED

Algorithm 2 can be shown to be unbiased by a similar argument. On the other hand, unless $n \leq 2$, for any finite value of M, Algorithm 1 is locally biased (and therefore biased). To demonstrate this, let P[i,j,k] denote the probability that X[i] ends in position j after k interchanges. Then

$$P[i,i,1] = \text{Prob}([p=i \ \&\ q=i] \text{ or } [p\neq i \ \&\ q\neq i]) = \frac{1}{n}*\frac{1}{n}+\frac{n-1}{n}*\frac{n-1}{n},$$

and for $i \neq j$,

$$P[i,j,1] = \text{Prob}([p=i \ \&\ q=j] \text{ or } [p=j \ \&\ q=i]) = 2/n^2.$$

Using mathematical induction and the fact that

$$P[i,j,k+1] = \sum_{m=1}^{n} P[i,m,k]*P[m,j,1]$$

one can prove

Theorem 2. When Algorithm 1 is applied, and k=0,1,2,... for $i \neq j$,

$$P[i,j,k] = \frac{1}{n}-\left(\frac{n-2}{n}\right)^k\left(\frac{1}{n}\right)$$

$$P[i,i,k] = \frac{1}{n}+\left(\frac{n-2}{n}\right)^k\left(\frac{n-1}{n}\right).$$

Finally, consider the following algorithm which is tempting because it appears to guarantee a locally unbiased shuffling process.

Algorithm 4

```
for i ← 1 to n do
    j ← RAND(1,n);
    T ← X[i];
    X[i] ← X[j];
    X[j] ← T
end.
```

However, unless $n \le 2$, Algorithm 4 is locally biased. Again, let P[i,j,k] denote the probability that X[i] ends in position j after k interchanges. Let's focus on P[n,j,k] and the movements of the value X[n]. Like Algorithm 3, this algorithm generates a sequence of independent j-values $J=(j_1, j_2, ..., j_n)$. If the value X[n] moves, either it first moves left to position m where j_m is the first entry in J that equals n, or it moves left to position j_n. Any subsequent movement will be to the right when some later entry in J equals m. Thus for $1 \le k < n$,

$$P[n,1,k] = \text{Prob}(j_1=n \ \&\ j_2\neq 1 \ \&\ j_3\neq 1 \ \&\ \ldots \ \&\ j_k\neq 1) = \frac{1}{n}\left(\frac{n-1}{n}\right)^{k-1} < \frac{1}{n}$$

and

$$P[n,1,n] = \text{Prob}([j_1=n \ \&\ j_2\neq 1 \ \&\ j_3\neq 1 \ \&\ \ldots \ \&\ j_n\neq 1 \text{ or } [j_1\neq n \ \&\ j_2\neq n \ \&\ j_3\neq n \ \&\ \ldots \ \&\ j_{n-1}\neq n \ \&\ j_n=1]) = \frac{2}{n}\left(\frac{n-1}{n}\right)^{n-1}.$$

But $\frac{2}{n}\left(\frac{n-1}{n}\right)^{n-1} = \frac{1}{n}$ if and only if $2(n-1)^{n-1} = n^{n-1}$, and, when $n>2$ no prime factors of n-1 divides n, so n-1 cannot divide evenly into n^{n-1}. Thus we have the somewhat paradoxical result that the additional "randomness" afforded by allowing the j-values to range from 1 to n results in a biased shuffling process.

§6. Random Subsets of size k

Suppose that we create Algorithm 5 by modifying Algorithm 3 so that the for-loop terminates when i=L where $1 \le L \le n-1$. Letting X' denote the values in the array after Algorithm 5 is applied, we see that two subsets of values of X result

S = {X'[1], ..., X'[L]} and T = { X'[L+1],..., X'[n]}.

Then a subset of exactly k of the entries in X may be produced by taking L to be the minimum of k and n-k (so $L \le n/2$), applying Algorithm 5, and "returning" either S, the first L entries, or T, the last n-L entries of the partially shuffled array. We will show that this process is unbiased and therefore a random k-subset is generated.

Algorithm 5 generates a sequence of L independent j-values $J = (j_1, j_2,..., j_L)$ where for each i, $i \le j_i \le n$ and where each of the n-i+1 possible values for j_i is equally likely. The

number of such sequences J is N = n(n-1)...(n-L+1), each such sequence is equally likely to be generated, and each such sequence generates exactly one of the N possible ordered L-subsets of elements of X placed in X'[1]..X'[L]. Thus if Q is some fixed L-subset

$$\texttt{Prob(S=Q)} = \sum \{\text{Prob}(X \to X') : X'[1]..X'[L] \text{ is an ordering of Q}\}$$
$$= \texttt{L!*1/N} = \texttt{1/}\binom{n}{L}.$$

Thus all L-subsets Q are equally likely to be produced as S. Furthermore, if R is some fixed (n-L)-subset then

$$\texttt{Prob(T=R)} = \texttt{Prob(S=}\bar{R}\texttt{)}$$
$$= 1/\binom{n}{L} = 1/\binom{n}{n-L}$$

so all (n-L)-subsets R are equally likely to be produced as T. Hence when k is given all k-subsets of entries in X are equally likely to be "returned" by Algorithm 5.

In addition, Algorithm 5 is locally unbiased: i.e.,

$$\texttt{Prob(X[j]}\in\text{S)} = \sum \{\texttt{Prob(S=Q)} : \texttt{X[j]} \in \text{Q}\}$$
$$= \binom{n-1}{L-1} * 1/\binom{n}{L} = \frac{L}{n},$$
$$\texttt{Prob(X[j]}\in\text{T)} = \texttt{Prob(X[j])} \notin \text{S)} = 1\text{-}L/n = (n\text{-}L)/n.$$

Thus Algorithm 5 is a simple, efficient and effective method for generating random subsets.

Finally we note that usually the ordering of the elements of a k-subset of an n-set is irrelevant but in some applications we might want them in increasing order. This can be done without first selecting the k elements and then sorting them, by algorithms which appear in [2], [3], and [4].

References

1. J. Bentley, *Programming Pearls*, Addison-Wesley, Readinf, MA, 1986.

2. S.E. Goodman, S.T.Hedetniemi, *Introduction to the Design and Analysis of Algorithms*, McGraw-Hill, New York, 1977.

3. D.E. Knuth, *The Art of Computer Programming*, *Seminumerical Algorithms*, Vol. 2, Addison-Wesley, Reading, MA, 1973.

4. A. Nijenhuis, H.S. Wilf, *Combinatorial Algorithms*, Academic Press, New York, 1978.