

The topic of primary importance in this section is *random sampling*. That is, we will focus on algorithms that use a random number generator to sample, at random, from a fixed collection (set) of objects. We begin, however, with the related topic of *random shuffling*.

6.5.1 RANDOM SHUFFLING

Definition 6.5.1 Exactly $m!$ different permutations $(a_0, a_1, \dots, a_{m-1})$ can be formed from a finite set \mathcal{A} with $m = |\mathcal{A}|$ distinct elements. A *random shuffle generator* is an algorithm that will produce any one of these $m!$ permutations in such a way that all are equally likely.

Example 6.5.1 If $\mathcal{A} = \{0, 1, 2\}$ then the $3! = 6$ different possible permutations of \mathcal{A} are
 $(0, 1, 2)$ $(0, 2, 1)$ $(1, 0, 2)$ $(1, 2, 0)$ $(2, 0, 1)$ $(2, 1, 0)$.

A random shuffle generator can produce any of these six possible permutations, each with equal probability $1/6$.

Algorithm 6.5.1 The intuitive way to generate a random shuffle of \mathcal{A} is to draw the first element a_0 at random from \mathcal{A} . Then draw the second element a_1 at random from the set $\mathcal{A} - \{a_0\}$ and the third element a_2 at random from the set $\mathcal{A} - \{a_0, a_1\}$, etc. The following *in place* algorithm does that, provided the elements of the set \mathcal{A} are stored (in any order) in the array $a[0], a[1], \dots, a[m-1]$

```

for (i = 0; i < m - 1; i++) {
    j = Equilikely(i, m - 1);
    hold = a[j];
    a[j] = a[i];
    a[i] = hold;
}
/* swap a[i] and a[j] */

```

Algorithm 6.5.1 is an excellent example of the elegance of simplicity. Figure 6.5.1 (corresponding to $m = 9$) illustrates the two indices i and j [j is an *Equilikely*($i, m - 1$) random variate] and the state of the $a[\cdot]$ array (initially filled with the integers 0 through 8 in sorted order, although any order is acceptable) for the first three passes through the for loop.

Figure 6.5.1. prior to $a[0], a[3]$ swap prior to $a[1], a[6]$ swap prior to $a[2], a[4]$ swap

Permutation	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr></table>	0	1	2	3	4	5	6	7	8	<table border="1"><tr><td>3</td><td>1</td><td>2</td><td>0</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr></table>	3	1	2	0	4	5	6	7	8	<table border="1"><tr><td>3</td><td>6</td><td>2</td><td>0</td><td>4</td><td>5</td><td>1</td><td>7</td><td>8</td></tr></table>	3	6	2	0	4	5	1	7	8
0	1	2	3	4	5	6	7	8																						
3	1	2	0	4	5	6	7	8																						
3	6	2	0	4	5	1	7	8																						
generation																														
algorithm.	i j	i j	i j																											

This algorithm is ideal for shuffling a deck of cards ($m = 52$ for a standard deck) — always a useful simulation skill. Moreover, as discussed later in this section, a minor modification to this algorithm makes it suitable for random sampling, without replacement.

6.5.2 RANDOM SAMPLING

We now turn to the topic of primary importance in this section, algorithms for random sampling. To provide a common basis for comparison of the algorithms, the following notation and terminology is used.

- We are given a *population* \mathcal{P} of $m = |\mathcal{P}|$ elements a_0, a_1, \dots, a_{m-1} . Typically m is large, perhaps so large that the population must be stored in secondary memory as a disk file, for example. If m is not large, then the population could be stored in primary memory as an array $a[0], a[1], \dots, a[m-1]$ or a linked list. In either case, whether m is large or not, the population is a *list*. That is, \mathcal{P} is ordered so that there is a first element a_0 , a second element a_1 , etc.
- We wish to obtain a random *sample* \mathcal{S} of $n = |\mathcal{S}|$ elements x_0, x_1, \dots, x_{n-1} from \mathcal{P} . Like \mathcal{P} , the sample \mathcal{S} is also a *list*. Typically n is small relative to m , but not necessarily so. If n is small then the sample can be stored in primary memory as the array $x[0], x[1], \dots, x[n-1]$ with the data type of $x[\cdot]$ the same as the data type of the population elements. For most algorithms, however, the use of an array is not a fundamental restriction; a linked list could be used instead, for example, or the sample could be written to secondary memory as a disk file if n is large.
- The algorithms use two generic functions $\text{Get}(\&z, \mathcal{L}, j)$ and $\text{Put}(z, \mathcal{L}, j)$ where the list \mathcal{L} could be either \mathcal{P} or \mathcal{S} and:

$\text{Get}(\&z, \mathcal{L}, j)$ returns the value of the j^{th} element in the list \mathcal{L} as z ;

$\text{Put}(z, \mathcal{L}, j)$ assigns the value of z to the j^{th} element in the list \mathcal{L} .

Both \mathcal{P} and \mathcal{S} can be accessed by **Get** and **Put**. The use of these generic functions allows the algorithms to be presented in a form that is essentially independent of how the lists \mathcal{P} and \mathcal{S} are actually stored.

- In some random sampling applications it is important to preserve the order of the population in the sample; in other applications *order preservation* is not important.
- A random sampling algorithm may be *sequential*, in which case \mathcal{P} is traversed once, in order, and elements are selected at random to form \mathcal{S} . A sequential algorithm is necessarily used when random access to \mathcal{P} is not possible. In contrast, a *non-sequential* random sampling algorithm is based on the assumption that random access to \mathcal{P} is possible and reasonably efficient. Note that it is access to \mathcal{P} , not \mathcal{S} , that determines whether a random sampling algorithm is sequential or non-sequential.
- A random sampling algorithm may use sampling *with replacement*, in which case the sample can contain multiple instances of the same population element. If so, then n could be larger than m . Instead, if sampling *without replacement* is used, then the sample cannot contain multiple instances of the same population element and so $n \leq m$. Sampling without replacement is the usual case in practice. For the special (trivial) case of sampling without replacement when $n = m$, $\mathcal{P} \equiv \mathcal{S}$.

Relative to random sampling algorithms, the phrase “at random” can be interpreted in at least two different (but closely related) ways: (i) each element of \mathcal{P} is equally likely to be an element of \mathcal{S} or (ii) each possible sample of size n is equally likely to be selected. Because these two interpretations of randomness are not equivalent, it is important to recognize what kind of samples a particular random sampling algorithm actually produces.

Non-Sequential Sampling

Algorithm 6.5.2 This $O(n)$ algorithm provides non-sequential random sampling with replacement. The value of m must be known.

```

for (i = 0; i < n; i++) {
    j = Equilikely(0, m - 1);
    Get(&z, P, j);           /* random access */
    Put(z, S, i);
}

```

Because sampling is with replacement, n can be larger than m in Algorithm 6.5.2. If the elements of \mathcal{P} are distinct, the number of possible samples is m^n and all samples are equally likely to be generated. The order of \mathcal{P} is not preserved in \mathcal{S} . Algorithm 6.5.2 should be compared with Algorithm 6.5.3, which is the *without* replacement analog. For both algorithms, the $O(n)$ complexity is based on the number of random variates generated and ignores, perhaps unrealistically, the complexity of access to \mathcal{P} and \mathcal{S} .

Example 6.5.2 In most non-sequential random sampling applications the population and sample are stored as arrays $a[0], a[1], \dots, a[m-1]$ and $x[0], x[1], \dots, x[n-1]$ respectively, in which case Algorithm 6.5.2 is equivalent to

```

for (i = 0; i < n; i++) {
    j = Equilikely(0, m - 1);
    x[i] = a[j];
}

```

Algorithm 6.5.3 This $O(n)$ algorithm provides non-sequential random sampling without replacement. The value of m must be known a priori and $n \leq m$.

```

for (i = 0; i < n; i++) {
    j = Equilikely(i, m - 1);           /* note, i not 1 */
    Get(&z, P, j);                       /* random access */
    Put(z, S, i);
    Get(&x, P, i);                       /* sequential access */
    Put(z, P, i);                       /* sequential access */
    Put(x, P, j);                       /* random access */
}

```

If the elements of \mathcal{P} are distinct, the number of possible samples is $m(m-1)\dots(m-n+1) = m!/(m-n)!$ and all samples are equally likely to be generated. The order of \mathcal{P} is not preserved in \mathcal{S} . Also, the order that exists in \mathcal{P} is destroyed by this sampling algorithm. If this is undesirable, use a copy of \mathcal{P} instead.

Example 6.5.3 In most non-sequential random sampling applications the population and sample are stored as arrays $a[0], a[1], \dots, a[m-1]$ and $x[0], x[1], \dots, x[n-1]$, respectively. In this case Algorithm 6.5.3 is equivalent to

```

for (i = 0; i < n; i++) {
    j = Equilikely(i, m - 1);
    x[i] = a[j];
    a[j] = a[i];
    a[i] = x[i];
}

```

In this form, it is clear that Algorithm 6.5.3 is a simple extension of Algorithm 6.5.1.

Sequential Sampling

The next three algorithms provide *sequential* sampling. For each algorithm the basic idea is the same — traverse \mathcal{P} once, in order, and select elements to put in \mathcal{S} . The selection or non-selection of elements is random with probability p using the generic statements

```

Get(&z, P, j);
if (Bernoulli(p))
    Put(z, S, i);

```

For Algorithm 6.5.4, p is independent of i and j . For Algorithms 6.5.5 and 6.5.6, however, the probability of selection changes adaptively conditioned on the values of i , j , and the number of elements previously selected.

Algorithm 6.5.4 This $O(m)$ algorithm provides sequential random sampling without replacement. Each element of \mathcal{P} is selected, independently, with probability p .

```

i = 0;                               /* i indexes the sample */
j = 0;                               /* j indexes the population */
while ( more data in P ) {
    Get(&z, P, j);                    /* sequential access */
    j++;
    if (Bernoulli(p)) {
        Put(z, S, i);
        i++;
    }
}

```

Although $p = n/m$ is a logical choice for the probability of selection, Algorithm 6.5.4 does not make use of either m or n explicitly. Note also that Algorithm 6.5.4 is not consistent with the idea of a specified sample size. Although the *expected* size of the sample is mp , the *actual* size is a *Binomial*(m, p) random variable. That is, no matter how p is chosen (with $0 < p < 1$), there is no way to specify the exact size of the sample. It can range from 0 to m . The order of \mathcal{P} is preserved in \mathcal{S} .

Example 6.5.4 In many sequential random sampling applications the population is stored in secondary memory as a sequential file and the sample is stored in primary memory as an array $x[0], x[1], \dots, x[n-1]$ in which case Algorithm 6.5.4 is equivalent to

```

i = 0;
while ( more data in P ) {
    z = GetData();
    if (Bernoulli(p)) {
        x[i] = z;
        i++;
    }
}

```

The fact that Algorithm 6.5.4 does not make use of m can be an advantage in some discrete-event simulation and real-time data acquisition applications. In these applications, the objective may be to sample, at random, say 1% of the population elements, independent of the population size. Algorithm 6.5.4 with $p = 0.01$ would provide this ability, but at the expense of not being able to specify the sample size exactly. In contrast, Algorithm 6.5.5 provides the ability to specify the sample size provided m is known and Algorithm 6.5.6 provides this ability even if m is unknown.

Algorithm 6.5.5 This $O(m)$ algorithm provides sequential random sampling of n sample values from \mathcal{P} without replacement, provided $m = |\mathcal{P}|$ is known a priori.

```

i = 0;                               /* i indexes the sample */
j = 0;                               /* j indexes the population */
while (i < n) {
    Get(&z, P, j);                    /* sequential access */
    p = (n - i) / (m - j);
    j++;
    if (Bernoulli(p)) {
        Put(z, S, i);
        i++;
    }
}

```

The key to the correctness of Algorithm 6.5.5 is that the population element a_j is selected with a probability $(n - i)/(m - j)$ that is conditioned on the number of sample elements left to be selected and the number of population elements left to be considered. Access to the population list is sequential (as is access to the sample list). Because sampling is without replacement, $n \leq m$. If the elements of \mathcal{P} are distinct, then the number of possible samples is the binomial coefficient $m!/(m - n)!n!$ (see Appendix D) and all samples are equally likely to be generated. The order of \mathcal{P} is preserved in \mathcal{S} .

Example 6.5.5 In many sequential random sampling applications the population is stored in secondary memory as a sequential file and the sample is stored in primary memory as an array $x[0], x[1], \dots, x[n - 1]$ in which case Algorithm 6.5.5 is equivalent to

```

i = 0;
j = 0;
while (i < n) {
    z = GetData();
    p = (n - i) / (m - j);
    j++;
    if (Bernoulli(p)) {
        x[i] = z;
        i++;
    }
}

```

Algorithm 6.5.6 This $O(m)$ algorithm provides sequential random sampling of n sample values from \mathcal{P} without replacement, even if $m = |\mathcal{P}|$ is not known a priori

```

for (i = 0; i < n; i++) {
    Get(&z, P, i);
    Put(z, S, i);
}
j = n;
while ( more data in P ) {
    Get(&z, P, j);
    j++;
    p = n / j;
    if (Bernoulli(p)) {
        i = Equilikely(0, n - 1);
        Put(z, S, i);
    }
}

```

Algorithm 6.5.6 is based on initializing the sample with the first n elements of the population. Then, for each of the (unknown number of) additional population elements, with conditional probability $n/(j+1)$ the population element a_j (for $j \geq n$) overwrites an existing randomly selected element in the sample. Access to the population list is sequential. Access to the sample list is *not* sequential, however, and so the order in \mathcal{P} is not preserved in \mathcal{S} . Because sampling is without replacement, $n \leq m$. If the elements of \mathcal{P} are distinct, then the number of possible samples is the binomial coefficient $m!/(m-n)!n!$ (see Appendix D) and all samples are equally likely to be generated. There is an important caveat, however — see Example 6.5.8.

Example 6.5.6 In many sequential random sampling applications the population is stored in secondary memory as a sequential file and the sample is stored in primary memory as an array $x[0], x[1], \dots, x[n-1]$ in which case Algorithm 6.5.6 is equivalent to

```

for (i = 0; i < n; i++) {
    z = GetData();
    x[i] = z;
}
j = n;
while ( more data in P ) {
    z = GetData();
    j++;
    p = n / j;
    if (Bernoulli(p)) {
        i = Equilikely(0, n - 1);
        x[i] = z;
    }
}

```

Algorithm Differences

Although Algorithms 6.5.5 and 6.5.6 have some similarities, there are two important differences.

- Algorithm 6.5.5 requires knowledge of the population size m and Algorithm 6.5.6 does not. This makes Algorithm 6.5.6 valuable in those discrete-event simulation and real-time data acquisition applications where m is not known a priori, particularly if the sample size is sufficiently small so that the sample can be stored in primary memory as an array.
- Beyond this obvious difference, there is a more subtle difference related to the number of possible samples when *order* is considered. Algorithm 6.5.5 preserves the order present in the population, Algorithm 6.5.6 does not. This difference is illustrated by the following two examples.

Example 6.5.7 If Algorithm 6.5.5 is used to select samples of size $n = 3$ from the population list $(0, 1, 2, 3, 4)$, then $m = 5$ and, because the sampling is sequential and the order of the population list is preserved in the samples, we find that exactly

$$\binom{5}{3} = \frac{5!}{2!3!} = 10$$

ordered samples are possible, as illustrated

$(0, 1, 2)$ $(0, 1, 3)$ $(0, 1, 4)$ $(0, 2, 3)$ $(0, 2, 4)$ $(0, 3, 4)$ $(1, 2, 3)$ $(1, 2, 4)$ $(1, 3, 4)$ $(2, 3, 4)$.

Each of these (ordered) samples will occur with equal probability $1/10$.

Example 6.5.8 If Algorithm 6.5.6 is used then 13 samples are possible, as illustrated

$(0, 1, 2)$ $(0, 1, 3)$ $(0, 1, 4)$ $(0, 3, 2)$ $(0, 4, 2)$ $(0, 3, 4)$ $(3, 1, 2)$ $(4, 1, 2)$ $(3, 1, 4)$ $(3, 4, 2)$
 $(0, 4, 3)$ $(4, 1, 3)$ $(4, 3, 2)$

Each of these samples are *not* equally likely, however. Instead, each of the six samples that are pairwise alike except for permutation will occur with probability $1/20$. The other seven samples will occur with probability $1/10$. If permutations are combined (for example, by sorting and then combining like results) then, as desired, each of the resulting 10 samples will have equal probability $1/10$. Because order in \mathcal{P} is not preserved in \mathcal{S} by Algorithm 6.5.6, some ordered samples, like $(1, 3, 4)$, cannot occur except in permuted order, like $(3, 1, 4)$ and $(4, 1, 3)$. For this reason, Algorithm 6.5.6 produces the correct number of equally likely samples only if all alike-but-for-permutation samples are combined. This potential need for post processing is the (small) price paid for not knowing the population size a priori.

6.5.3 URN MODELS

Many discrete stochastic models are based on random sampling. We close this section with four examples. Three of these discrete stochastic models were considered earlier in this chapter, the other is new:

- *Binomial* (n, p) ;
- *Hypergeometric* (n, a, b) ;
- *Geometric* (p) ;
- *Pascal* (n, p) .

All four of these models can be motivated by drawing, at random, from a conceptual urn initially filled with a amber balls and b black balls.

Binomial

Example 6.5.9 An urn contains a amber balls and b black balls. A total of n balls are drawn from the urn, *with* replacement. Let x be the number of amber balls drawn. A Monte Carlo simulation of this random experiment is easily constructed. Indeed, one obvious way to construct this simulation is to use Algorithm 6.5.2, as implemented in Example 6.5.2, applied to a population array of length $m = a + b$ with a 1's (amber balls) and b 0's (black balls). Then x is the number of 1's in a random sample of size n . The use of Algorithm 6.5.2 is overkill in this case, however, because an equivalent simulation can be constructed without using either a population or sample data structure. That is, let $p = a/(a + b)$ and use the $O(n)$ algorithm

```

x = 0;
for (i = 0; i < n; i++)
    x += Bernoulli(p);
return x;

```

The discrete random variate x so generated is *Binomial*(n, p). The associated pdf of X is

$$f(x) = \binom{n}{x} p^x (1-p)^{n-x} \quad x = 0, 1, 2, \dots, n.$$

Hypergeometric

Example 6.5.10 As a variation of Example 6.5.9, suppose that the draw from the urn is *without* replacement (and so $n \leq m$). Although Algorithm 6.5.3 could be used as implemented in Example 6.5.3, it is better to use a properly modified version of the algorithm in Example 6.5.9 instead, as illustrated

```

m = a + b;
x = 0;
for (i = 0; i < n; i++) {
    p = (a - x) / (m - i);
    x += Bernoulli(p);
}
return x;

```

The discrete random variate x so generated is said to be *Hypergeometric*(n, a, b). The associated pdf of X is

$$f(x) = \frac{\binom{a}{x} \binom{b}{n-x}}{\binom{a+b}{n}} \quad x = \max\{0, n-b\}, \dots, \min\{a, n\}.$$

The lower limit in the range of possible values of X accounts for the possibility that the sample size may exceed the number of black balls ($n > b$). If $n > b$ then at least $n - b$ amber balls must be drawn. Similarly, the upper limit accounts for the possibility that the sample size may exceed the number of amber balls ($n > a$). If $n > a$ then at most a amber balls can be drawn. In applications where n is less than or equal to both a and b , the range of possible values is $x = 0, 1, 2, \dots, n$.

Geometric

Example 6.5.11 As another variation of Example 6.5.9, suppose that the draw is from the urn *with* replacement but that we draw only until the first black ball is obtained. Let x be the number of amber balls drawn. With $p = a/(a+b)$ the following algorithm simulates this random experiment

```
x = 0;
while (Bernoulli(p))
    x++;
return x;
```

The discrete random variate x so generated is *Geometric*(p). The associated pdf of X is

$$f(x) = p^x(1 - p) \quad x = 0, 1, 2, \dots$$

This stochastic model is commonly used in reliability studies, in which case p is usually close to 1.0 and x counts the number of successes before the first failure.

This algorithm for generating geometric variates is inferior to the inversion algorithm presented in Example 6.2.8. The algorithm presented here is (1) not synchronized, (2) not monotone, and (3) inefficient. The expected number of passes through the **while** loop is

$$1 + \frac{p}{1 - p}.$$

If p is close to 1, the execution time for this algorithm can be quite high.

Pascal

Example 6.5.12 As an extension of Example 6.5.11, suppose that we draw *with* replacement until the n^{th} black ball is obtained. Let x be the number of amber balls drawn (so that a total of $n + x$ balls are drawn, the last of which is black). With $p = a/(a + b)$ the following algorithm simulates this random experiment.

```
x = 0;
for (i = 0; i < n; i++)
    while (Bernoulli(p))
        x++;
return x;
```

The discrete random variate x so generated is *Pascal*(n, p). The associated pdf is

$$f(x) = \binom{n+x-1}{x} p^x (1-p)^n \quad x = 0, 1, 2, \dots$$

This stochastic model is commonly used in reliability studies, in which case p is usually close to 1.0 and x counts the number of successes before the n^{th} failure.

Further reading on random sampling and shuffling can be found in the textbooks by Nijenhuis and Wilf (1978) and Wilf (1989) or the journal articles by McLeod and Bellhouse (1983) and Vitter (1984).

6.5.4 EXERCISES

Exercise 6.5.1 Two cards are drawn in sequence from a well-shuffled ordinary deck of 52. (a) If the draw is without replacement, use Monte Carlo simulation to verify that the second card will be higher in rank than the first with probability $8/17$. (b) Estimate this probability if the draw is with replacement.

Exercise 6.5.2 Which of the four discrete stochastic models presented in this section characterizes the actual (random variate) sample size when Algorithm 6.5.4 is used?

Exercise 6.5.3 (a) For $a = 30$, $b = 20$, and $n = 10$ use Monte Carlo simulation to verify the correctness of the probability equations in Examples 6.5.9 and 6.5.10. (b) Verify that in both cases the mean of x is $na/(a+b)$. (c) Are the standard deviations the same?

Exercise 6.5.4 Three people toss a fair coin in sequence until the first occurrence of a head. Use Monte Carlo simulation to estimate (a) the probability that the first person to toss will eventually win the game by getting the first head, (b) the second person's probability of winning, and (c) the third person's probability of winning. Also, (d) compare your estimates with the analytic solutions.

Exercise 6.5.5^a How would the algorithm and pdf equation need to be modified if the random draw in Example 6.5.12 is *without* replacement?

Exercise 6.5.6^a An urn is initially filled with one amber ball and one black ball. Each time a black ball is drawn, it is replaced *and* another black ball is also placed in the urn. Let X be the number of random draws required to find the amber ball. (a) What is the pdf of X for $x = 1, 2, 3, \dots$? (b) Use Monte Carlo simulation to estimate the pdf and the expected value of X .

Exercise 6.5.7^a Same as the previous exercise except that there is a reservoir of just eight black balls to add to the urn. Comment.