Programs `ssq1` and `sis1` both require input data from an external source. Because of this, the usefulness of these programs is limited by the amount of available input data. What if more input data is needed? Or, what if the model is changed; can the input data be modified accordingly? Or, what can be done if only a small amount of input data is available, or perhaps, none at all?

In each case the answer is to use a *random number generator*. By convention, each call to the random number generator will produce a real-valued result between 0.0 and 1.0 that becomes the inherent source of randomness for a discrete-event simulation model. As illustrated later in this chapter, consistent with user-defined stochastic models of arrival times, service times, demand amounts, etc., the random number generator's output can be converted to a *random variate* via an appropriate mathematical transformation. The random variates are used to approximate some probabilistic element (e.g., service times) of a real-world system for a discrete-event simulation model.

2.1.1 RANDOM NUMBER GENERATION

Historically three types of random number generators have been advocated for computational applications: (*a*) 1950's-style table look-up generators like, for example, the RAND corporation table of a million random digits; (*b*) hardware generators like, for example, thermal "white noise" devices; and (*c*) algorithmic (software) generators. Of these three types, only algorithmic generators have achieved widespread acceptance. The reason for this is that only algorithmic generators have the potential to satisfy *all* of the following generally well-accepted random number generation criteria. A generator should be:

- *random* — able to produce output that passes all reasonable statistical tests of randomness;
- *controllable* — able to reproduce its output, if desired;
- *portable* — able to produce the same output on a wide variety of computer systems;
- *efficient* — fast, with minimal computer resource requirements;
- *documented* — theoretically analyzed and extensively tested.

In addition, as discussed and illustrated in Chapter 3, it should be easy to partition the generator's output into multiple "streams".

Definition 2.1.1 An *ideal* random number generator is a function, say `Random`, with the property that *each* assignment

```
u = Random();
```

will produce a real-valued (floating-point) number u between 0.0 and 1.0 in such a way that any value in the interval $0.0 < u < 1.0$ is *equally likely* to occur. A *good* random number generator produces results that are statistically indistinguishable, for all practical purposes, from those produced by an ideal generator.

What we will do in this chapter is construct, from first principles, a good random number generator that will satisfy all the criteria listed previously. We begin with the following conceptual model.

- Choose a *large* positive integer m. This defines the set $\mathcal{X}_m = \{1, 2, \ldots, m-1\}$.
- Fill a (conceptual) urn with the elements of $\mathcal{X}_m$.
- Each time a random number u is needed, draw an integer x "at random" from the urn and let $u = x/m$.

Each draw *simulates* a realization (observation, sample) of an independent identically distributed (*iid*) sequence of so-called *Uniform*$(0, 1)$ random variables. Because the possible values of u are $1/m$, $2/m$, ..., $1 - 1/m$, it is important for m to be so large that the possible values of u will be densely distributed between 0.0 and 1.0. Note that the values $u = 0.0$ and $u = 1.0$ are impossible. As discussed in later chapters, excluding these two extreme values is important for avoiding problems associated with certain random variate generation algorithms.

Ideally we would like to draw from the urn independently and *with* replacement. If so then each of the $m - 1$ possible values of u would be equally likely to be selected on each draw. For practical reasons, however, we will use a random number generation algorithm that simulates drawing from the urn *without* replacement. Fortunately, if m is large and the number of draws is small relative to m then the distinction between drawing with and without replacement is largely irrelevant. To turn the conceptual urn model into an specification model we will use a time-tested algorithm suggested by Lehmer (1951).

2.1.2 LEHMER'S ALGORITHM

Definition 2.1.2 *Lehmer's algorithm* for random number generation is defined in terms of two fixed parameters

- *modulus* m, a fixed large *prime* integer
- *multiplier* a, a fixed integer in $\mathcal{X}_m$

and the subsequent generation of the integer sequence x_0, x_1, x_2, ... via the iterative equation

- $x_{i+1} = g(x_i) \qquad i = 0, 1, 2, \ldots$

where the function $g(\cdot)$ is defined for all $x \in \mathcal{X}_m = \{1, 2, \ldots, m-1\}$ as

- $g(x) = ax \bmod m$

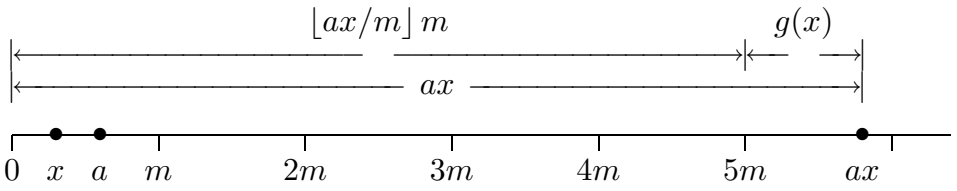
and the *initial seed* x_0 is chosen from the set $\mathcal{X}_m$. The *modulus function* mod gives the remainder when the first argument (ax in this case) is divided by the second argument (the modulus m in this case). It is defined more carefully in Appendix B. A random number generator based on Lehmer's algorithm is called a *Lehmer generator*.

Because of the mod (remainder) operator, the value of $g(x)$ is always an integer between 0 and $m-1$; however, it is important to exclude 0 as a possible value. That is, $g(0) = 0$ and so if 0 ever occurs in the sequence $x_0, x_1, x_2, \ldots$ then *all* the terms in the sequence will be 0 from that point on. Fortunately, if (a, m) is chosen consistent with Definition 2.1.2, it can be shown that $g(x) \neq 0$ for all $x \in \mathcal{X}_m$ so that $g : \mathcal{X}_m \to \mathcal{X}_m$. This guaranteed avoidance of 0 is part of the reason for choosing m to be prime. Because $g : \mathcal{X}_m \to \mathcal{X}_m$, it follows (by induction) that if $x_0 \in \mathcal{X}_m$ then $x_i \in \mathcal{X}_m$ for all $i = 0, 1, 2, \ldots$

Lehmer's algorithm represents a good example of the elegance of simplicity. The genius of Lehmer's algorithm is that *if* the multiplier a and prime modulus m are properly chosen, the resulting sequence $x_0, x_1, x_2, \ldots, x_{m-2}$ will be statistically indistinguishable from a sequence drawn at random, albeit without replacement, from $\mathcal{X}_m$. Indeed, it is *only* in the sense of *simulating* this random draw that the algorithm is random; there is actually *nothing* random about Lehmer's algorithm, except possibly the choice of the initial seed. For this reason Lehmer generators are sometimes called *pseudo*-random.*

An intuitive explanation of why Lehmer's algorithm simulates randomness is based on the observation that if two large integers, a and x, are multiplied and the product divided by another large integer, m, then the remainder, $g(x) = ax \bmod m$, is "likely" to take on any value between 0 and $m-1$, as illustrated in Figure 2.1.1.

Figure 2.1.1. Lehmer generator geometry.



This is somewhat like going to the grocery store with no change, only dollars, and buying many identical items. After you pay for the items, the change you will receive is likely to be any value between 0¢ and 99¢. [That is, a is the price of the item in cents, x is the number of items purchased, and $m = 100$. The analogy is not exact, however, because the change is $m - g(x)$, not $g(x)$].

Modulus and Multiplier Considerations

To construct a Lehmer generator, standard practice is to choose the large prime modulus m first, then choose the multiplier a. The choice of m is dictated, in part, by system considerations. For example, on a computer system that supports 32-bit, 2's complement integer arithmetic, $m = 2^{31} - 1$ is a natural choice because it is the largest possible positive integer and it happens to be prime. (For 16-bit integer arithmetic we are not so fortunate, however, because $2^{15} - 1$ is not prime. See Exercise 2.1.6. Similarly, for 64-bit integer arithmetic $2^{63} - 1$ is not prime. See Exercise 2.1.10.) Given m, the subsequent choice of a must be made with great care. The following example is an illustration.

* Lehmer generators are also known as (take a deep breath) prime modulus multiplicative linear congruential pseudo-random number generators, abbreviated PMMLCG.

Example 2.1.1 As a tiny example, consider the prime modulus $m = 13$.*

- If the multiplier is $a = 6$ and the initial seed is $x_0 = 1$ then the resulting sequence of x's is

  $$1, 6, 10, 8, 9, 2, 12, 7, 3, 5, 4, 11, 1, \ldots$$

  where, as the ellipses indicate, the sequence is actually *periodic* because it begins to cycle (with a *full* period of length $m - 1 = 12$) when the initial seed reappears. The point is that any 12 consecutive terms in this sequence appear to have been drawn at random, without replacement, from the set $\mathcal{X}_{13} = \{1, 2, \ldots, 12\}$.

- If the multiplier is $a = 7$ and the initial seed is $x_0 = 1$ then the resulting full-period sequence of x's is

  $$1, 7, 10, 5, 9, 11, 12, 6, 3, 8, 4, 2, 1, \ldots$$

  Randomness is, like beauty, only in the eye of the beholder. Because of the $12, 6, 3$ and $8, 4, 2, 1$ patterns, however, most people would consider this second sequence to be "less random" than the first.

- If the multiplier is $a = 5$ then either

  $$1, 5, 12, 8, 1, \ldots \quad \text{or} \quad 2, 10, 11, 3, 2, \ldots \quad \text{or} \quad 4, 7, 9, 6, 4, \ldots$$

  will be produced, depending on the initial seed $x_0 = 1, 2$ or 4. This type of less-than-full-period behavior is clearly undesirable because, in terms of our conceptual model of random number generation, this behavior corresponds to first partitioning the set $\mathcal{X}_m$ into several disjoint subsets (urns), then selecting one subset and thereafter drawing exclusively from it.

Example 2.1.1 illustrates two of the three central issues that must be resolved when choosing (a, m).

- Does the function $g(\cdot)$ generate a full-period sequence?
- If a full-period sequence is generated, how random does the sequence appear to be?

The third central issue is implementation.

- Can the $ax \bmod m$ operation be evaluated efficiently and correctly for all $x \in \mathcal{X}_m$?

For Example 2.1.1 the issue of implementation is trivial. However, for realistically large values of a and m, a portable, efficient implementation in a high-level language is a substantive issue because of potential integer overflow associated with the ax product. We will return to this implementation issue in the next section. In the remainder of this section we will concentrate on the full-period issue.

---

* For a hypothetical computer system with 5-bit 2's complement integer arithmetic, the largest positive integer would be $2^4 - 1 = 15$. In this case $m = 13$ would be the largest possible prime and, in that sense, a natural choice for the modulus.

Full Period Considerations

From the definition of the mod function (see Appendix B), if $x_{i+1} = g(x_i)$ then there exists a non-negative integer $c_i = \lfloor ax_i/m \rfloor$ such that

$$x_{i+1} = g(x_i) = ax_i \bmod m = ax_i - mc_i \qquad i = 0, 1, 2, \ldots$$

Therefore (by induction)

$$\begin{aligned}
x_1 &= ax_0 - mc_0 \\
x_2 &= ax_1 - mc_1 = a^2x_0 - m(ac_0 + c_1) \\
x_3 &= ax_2 - mc_2 = a^3x_0 - m(a^2c_0 + ac_1 + c_2) \\
&\vdots \\
x_i &= ax_{i-1} - mc_{i-1} = a^ix_0 - m(a^{i-1}c_0 + a^{i-2}c_1 + \cdots + c_{i-1}).
\end{aligned}$$

Because $x_i \in \mathcal{X}_m$ we have $x_i = x_i \bmod m$. Moreover, $(a^ix_0 - mc) \bmod m = a^ix_0 \bmod m$ independent of the value of the integer $c = a^{i-1}c_0 + a^{i-2}c_1 + \cdots + c_{i-1}$. Therefore, we have proven the following theorem.

Theorem 2.1.1 If the sequence x_0, x_1, x_2, ... is produced by a Lehmer generator with multiplier a and modulus m then

$$x_i = a^ix_0 \bmod m \qquad i = 0, 1, 2, \ldots$$

Although it would be an eminently bad idea to compute x_i by first computing a^i, particularly if i is large, Theorem 2.1.1 has significant theoretical importance due, in part, to the fact that x_i can be written equivalently as

$$x_i = a^ix_0 \bmod m = [(a^i \bmod m)\,(x_0 \bmod m)] \bmod m = [(a^i \bmod m)\,x_0] \bmod m$$

for $i = 0, 1, 2, \ldots$ via Theorem B.2 in Appendix B. In particular, this is true for $i = m - 1$. Therefore, because m is prime and $a \bmod m \neq 0$, from Fermat's little theorem (see Appendix B) it follows that $a^{m-1} \bmod m = 1$ and so $x_{m-1} = x_0$. This observation is the key to proving the following theorem. The details of the proof are left as an exercise.

Theorem 2.1.2 If $x_0 \in \mathcal{X}_m$ and the sequence x_0, x_1, x_2, ... is produced by a Lehmer generator with multiplier a and (prime) modulus m then there is a positive integer p with $p \leq m - 1$ such that $x_0, x_1, x_2, \ldots, x_{p-1}$ are all different and

$$x_{i+p} = x_i \qquad i = 0, 1, 2, \ldots$$

That is, the sequence $x_0, x_1, x_2, \ldots$ is *periodic* with *fundamental period* p. In addition, $(m - 1) \bmod p = 0$.

The significance of Theorem 2.1.2 is profound. If we pick *any* initial seed $x_0 \in \mathcal{X}_m$ and start to generate the sequence $x_0, x_1, x_2, \ldots$ then we are guaranteed that the initial seed will reappear. In addition, the first instance of reappearance is guaranteed to occur at some index p that is either $m-1$ or an integer divisor of $m-1$ and from that index on the sequence will cycle through the same p distinct values as illustrated

$$\underbrace{x_0, x_1, \ldots, x_{p-1}}_{\text{period}}, \underbrace{x_0, x_1, \ldots, x_{p-1}}_{\text{period}}, x_0, \ldots$$

Consistent with Definition 2.1.3, we are interested in choosing multipliers for which the fundamental period is $p = m - 1$.

Full Period Multipliers

Definition 2.1.3 The sequence x_0, x_1, x_2, ... produced by a Lehmer generator with modulus m and multiplier a has a *full period* if and only if the fundamental period p is $m-1$. If the sequence has a full period then a is said to be a *full-period multiplier* relative to m.*

If the sequence x_0, x_1, x_2, ... has a full period then any $m-1$ consecutive terms in the sequence have been drawn without replacement from the set $\mathcal{X}_m$. In effect a full-period Lehmer generator creates a virtual *circular list* with $m-1$ distinct elements. The initial seed provides a starting list element; subsequent calls to the generator traverse the list.

Example 2.1.2 From Example 2.1.1, if $m = 13$ then $a = 6$ and $a = 7$ are full-period multipliers ($p = 12$) and $a = 5$ is not ($p = 4$). The virtual circular lists (with traversal in a clockwise direction) corresponding to these two full-period multipliers are shown in Figure 2.1.2.

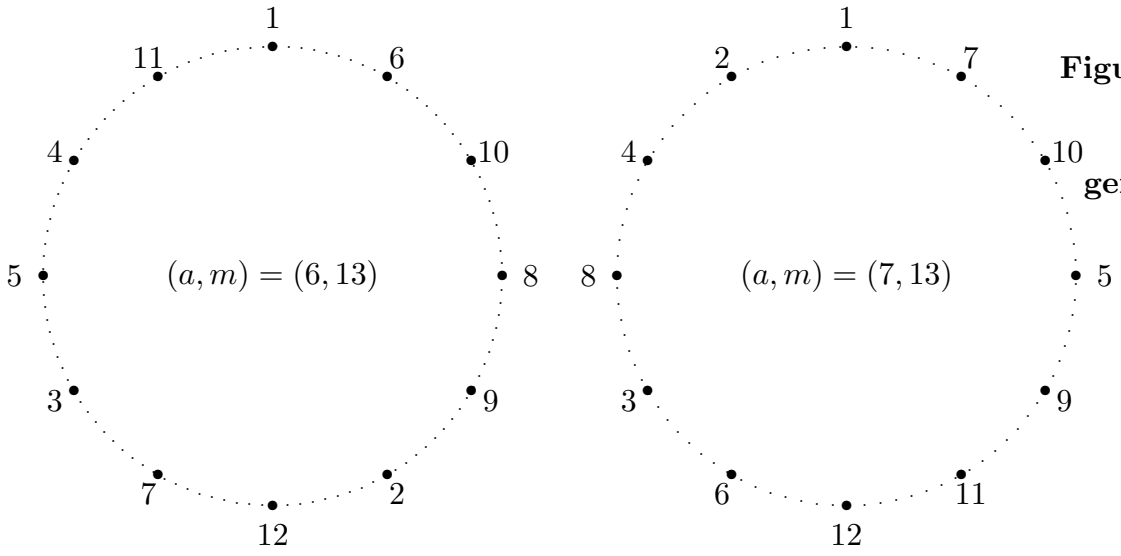


Figure 2.1.2. Two full period generators.

* A full-period multiplier a relative to m is also said to be a *primitive root* of m.

Algorithm 2.1.1 This algorithm, based upon Theorem 2.1.2, can be used to determine if a is a full-period multiplier relative to the prime modulus m.

```
p = 1;
x = a;
while (x != 1) {
   p++;
   x = (a * x) % m;                    /* beware of a * x overflow */
}
if (p == m - 1)
   /* a is a full-period multiplier */
else
   /* a is not a full-period multiplier */
```

If m is not prime Algorithm 2.1.1 may not halt. It provides a slow-but-sure $O(m)$* test for a full-period multiplier. The algorithm uses $x_0 = 1$ as an arbitrary initial seed (note that $x_1 = a$) and the recursive generation of new values of x until the initial seed reappears. Before starting to search for full-period multipliers, however, it would be good to know that they exist and with what frequency. That is, given a prime modulus m, how many corresponding full-period multipliers are there? The following theorem provides the answer to this question.

Theorem 2.1.3 If m is prime and $p_1, p_2, \ldots, p_r$ are the (unique) *prime factors* of $m-1$ then the number of full-period multipliers in $\mathcal{X}_m$ is

$$\frac{(p_1 - 1)(p_2 - 1)\ldots(p_r - 1)}{p_1 p_2 \cdots p_r}\,(m-1).$$

Example 2.1.3 If $m = 13$ then $m - 1 = 2^2 \cdot 3$. From the equation in Theorem 2.1.3 this prime modulus has $\frac{(2-1)(3-1)}{2\cdot 3}(13-1) = 4$ full-period multipliers: $a = 2$, 6, 7, and 11.

Example 2.1.4 The Lehmer generator used in this book has the (Mersenne, i.e., of the form $2^k - 1$, where k is a positive integer) prime modulus $m = 2^{31} - 1 = 2\,147\,483\,647$. Because the prime decomposition of $m - 1$ is

$$m - 1 = 2^{31} - 2 = 2 \cdot 3^2 \cdot 7 \cdot 11 \cdot 31 \cdot 151 \cdot 331$$

from the equation in Theorem 2.1.3 the number of full-period multipliers is

$$\left(\frac{1 \cdot 2 \cdot 6 \cdot 10 \cdot 30 \cdot 150 \cdot 330}{2 \cdot 3 \cdot 7 \cdot 11 \cdot 31 \cdot 151 \cdot 331}\right)(2 \cdot 3^2 \cdot 7 \cdot 11 \cdot 31 \cdot 151 \cdot 331) = 534\,600\,000.$$

Therefore, for this prime modulus approximately 25% of the multipliers between 1 and $m - 1$ are full-period multipliers.

* A function f is called "order m", for example, written $O(m)$, if there exist real positive constants c_1 and c_2 independent of m such that $c_1 m \le f(m) \le c_2 m$.

Using Algorithm 2.1.1 (and *a lot* of computer time) it can be shown that if $m = 2^{31} - 1$ then $a = 2, 3, 4, 5, 6$ are not full-period multipliers but that $a = 7$ is. Remarkably, once *one* full-period multiplier has been found, in this case $a = 7$, then *all* the others can be found using the following $O(m)$ algorithm. This algorithm presumes the availability of a function `gcd` that returns the greatest common divisor of two positive integers (see Appendix B).

Algorithm 2.1.2 Given the prime modulus m and any full-period multiplier a, the following algorithm generates all the full-period multipliers relative to m.*

```
i = 1;
x = a;
while (x != 1) {
   if (gcd(i, m − 1) == 1)
     /* x is a full-period multiplier equal to a^i mod m */
   i++;
   x = (a * x) % m;                          /* beware of a * x overflow */
}
```

Algorithm 2.1.2 is based on Theorem 2.1.4 which establishes a one-to-one correspondence between integers $i \in \mathcal{X}_m$ that are relatively prime to $m-1$ and full-period multipliers $a^i \bmod m \in \mathcal{X}_m$. That is, the equation in Theorem 2.1.3 counts both the number of full-period multipliers *and* the number of integers in $\mathcal{X}_m$ that are relatively prime to $m - 1$. The proof of Theorem 2.1.4 is left as an (advanced) exercise.

Theorem 2.1.4 If a is any full-period multiplier relative to the prime modulus m then each of the integers

$$a^i \bmod m \in \mathcal{X}_m \qquad\qquad i = 1, 2, 3, \ldots, m - 1$$

is also a full-period multiplier relative to m if and only if the integer i has no prime factors in common with the prime factors of $m - 1$, i.e., i and $m - 1$ are relatively prime (see Appendix B).

Example 2.1.5 If $m = 13$ then from Example 2.1.3 there are 4 integers between 1 and 12 that are relatively prime to 12. They are $i = 1, 5, 7, 11$. From Example 2.1.1, $a = 6$ is a full-period multiplier relative to 13; from Theorem 2.1.4 the 4 full-period multipliers relative to 13 are therefore

$$6^1 \bmod 13 = 6, \qquad 6^5 \bmod 13 = 2, \qquad 6^7 \bmod 13 = 7, \qquad 6^{11} \bmod 13 = 11.$$

Equivalently, if we had known that $a = 2$ is a full-period multiplier relative to 13 we could have used Algorithm 2.1.2 with $a = 2$ to determine the full-period multipliers as

$$2^1 \bmod 13 = 2, \qquad 2^5 \bmod 13 = 6, \qquad 2^7 \bmod 13 = 11, \qquad 2^{11} \bmod 13 = 7.$$

* See Exercises 2.1.4 and 2.1.5.

Example 2.1.6 If $m = 2^{31} - 1$ then from Example 2.1.4 there are 534600000 integers between 1 and $m - 1$ that are relatively prime to $m - 1$. The first few of these are $i = 1, 5, 13, 17, 19$. As discussed previously, $a = 7$ is a full-period multiplier relative to this modulus and so from Algorithm 2.1.2

$$7^1 \bmod 2147483647 = 7$$
$$7^5 \bmod 2147483647 = 16807$$
$$7^{13} \bmod 2147483647 = 252246292$$
$$7^{17} \bmod 2147483647 = 52958638$$
$$7^{19} \bmod 2147483647 = 447489615$$

are full-period multipliers relative to $2^{31} - 1 = 2147483647$. Note that the full-period multiplier 16807 is a *logical* choice; 7 is the smallest full-period multiplier and 5 is the smallest integer (other than 1) that is relatively prime to $m - 1$. However, that the multiplier 16807 is a logical choice does not mean that the resulting sequence it generates is necessarily "random". We will have more to say about this in the next section.

Standard Algorithmic Generators

Most of the standard algorithmic generators in use today are one of the following three *linear congruential* types (Law and Kelton, 2000, pages 406–412):

- *mixed*: $g(x) = (ax + c) \bmod m$ with $m = 2^b$ ($b = 31$ typically), $a \bmod 4 = 1$ and $c \bmod 2 = 1$. All integer values of $x \in \{0\} \cup \mathcal{X}_m$ are possible and the period is m.
- *multiplicative* with $m = 2^b$: $g(x) = ax \bmod m$ with $m = 2^b$ ($b = 31$ typically) and $a \bmod 8 = 3$ or $a \bmod 8 = 5$. To achieve the maximum period, which is only $m/4$, x must be restricted to the odd integers in $\mathcal{X}_m$.
- *multiplicative* with m prime: $g(x) = ax \bmod m$ with m prime and a a full-period multiplier. All integer values of $x \in \mathcal{X}_m = \{1, 2, \ldots, m - 1\}$ are possible and the period is $m - 1$.

Of these three, specialists generally consider the third generator, which has been presented in this chapter, to be best. Random number generation remains an area of active research, however, and reasonable people will probably always disagree about the *best* algorithm for random number generation. Section 2.5 contains a more general discussion of random number generators.

For now we will avoid the temptation to be drawn further into the "what is the best possible random number generator" debate. Instead, as discussed in the next section, we will use a Lehmer generator with modulus $m = 2^{31} - 1$ and a corresponding full-period multiplier $a = 48271$ carefully selected to provide generally acceptable randomness *and* facilitate efficient software implementation. With this generator in hand, we can then turn our attention to the main topic of this book — the modeling, simulation and analysis of discrete-event stochastic systems.

2.1.3 EXERCISES

Exercise 2.1.1 For the tiny Lehmer generator defined by $g(x) = ax \bmod 127$, find all the full-period multipliers. (a) How many are there? (b) What is the smallest multiplier?

Exercise 2.1.2 Prove Theorem 2.1.2.

Exercise 2.1.3 Prove that if (a, m) are chosen consistent with Definition 2.1.2, then $g : \mathcal{X}_m \to \mathcal{X}_m$ is a bijection (one-to-one and onto).

Exercise 2.1.4[a] (a) Prove that if (a, m) are chosen consistent with Definition 2.1.2, then $g(x) \neq 0$ for all $x \in \mathcal{X}_m$ and so $g : \mathcal{X}_m \to \mathcal{X}_m$. ($b$) In addition, relative to this definition, prove that there is nothing to be gained by considering integer multipliers outside of $\mathcal{X}_m$. Consider the two cases $a \geq m$ and $a \leq 0$.

Exercise 2.1.5 (a) Except for the special case $m = 2$, prove that $a = 1$ cannot be a full-period multiplier. (b) What about $a = m - 1$?

Exercise 2.1.6 In ANSI C an `int` is guaranteed to hold all integer values between $-(2^{15} - 1)$ and $2^{15} - 1$ inclusive. (a) What is the largest prime modulus in this range? (b) How many corresponding full-period multipliers are there and what is the smallest one?

Exercise 2.1.7[a] Prove Theorem 2.1.4.

Exercise 2.1.8 (a) Evaluate $7^i \bmod 13$ and $11^i \bmod 13$ for $i = 1, 5, 7, 11$. (b) How does this relate to Example 2.1.5?

Exercise 2.1.9 (a) Verify that the list of five full-period multipliers in Example 2.1.6 is correct. (b) What are the next five elements in this list?

Exercise 2.1.10[a] (a) What is the largest prime modulus less than or equal to $2^{63} - 1$? (b) How many corresponding full-period multipliers are there? (c) How does this relate to the use of Lehmer random number generators on computer systems that support 64-bit integer arithmetic?

Exercise 2.1.11 For the first few prime moduli, this table lists the number of full-period multipliers and the smallest full-period multiplier. Add the next 10 rows to this table.

prime modulus m	number of full-period multipliers	smallest full-period multiplier a
2	1	1
3	1	2
5	2	2
7	2	3
11	4	2
13	4	2