*5.2 PSEUDORANDOM NUMBER GENERATORS

Let's look more closely at how a ***pseudorandom number generator*** (PRNG) creates a sequence of numbers that appear to be random. A PRNG algorithm starts with a particular number called a *seed*. If we let $R(t)$ represent the number returned by the PRNG in step t, then we can denote the seed as $R(0)$. We apply a simple, but carefully chosen, arithmetic function to the seed to get the first number $R(1)$ in our pseudorandom sequence. To get the next number in the pseudorandom sequence, we apply the same arithmetic function to $R(1)$, producing $R(2)$. This process continues for $R(3), R(4), \ldots$, for as long as we need "random" numbers, as illustrated in Figure 1.

Reflection 1 *Notice that we are computing $R(t)$ from $R(t-1)$ at every step. Does this look familiar?*

If you read Section 4.4, you may recognize this as another example of a difference equation. A difference equation is a function that computes its next value based on its previous value.

A simple PRNG algorithm, known as a *Lehmer pseudorandom number generator*, is named after the late mathematician Derrick Lehmer. Dr. Lehmer taught at the University of California, Berkeley, and was one of the first people to run programs on the ENIAC, the first electronic computer. A Lehmer PRNG uses the following difference equation:

$$R(t) = a \cdot R(t-1) \bmod m$$

where m is a prime number and a is an integer between 1 and $m-1$. For example, suppose $m = 13$, $a = 5$, and the seed $R(0) = 1$. Then

$$R(1) = 5 \cdot R(0) \bmod 13 = 5 \cdot 1 \bmod 13 = 5 \bmod 13 = 5$$

$$R(2) = 5 \cdot R(1) \bmod 13 = 5 \cdot 5 \bmod 13 = 25 \bmod 13 = 12$$

$$R(3) = 5 \cdot R(2) \bmod 13 = 5 \cdot 12 \bmod 13 = 60 \bmod 13 = 8$$

So this pseudorandom sequence begins $5, 12, 8, \ldots$.

Reflection 2 *Compute the next four values in the sequence.*

The next four values are $R(4) = 1$, $R(5) = 5$, $R(6) = 12$, and $R(7) = 8$. So the sequence is now $5, 12, 8, 1, 5, 12, 8, \ldots$.

Reflection 3 *Does this sequence look random?*

Unfortunately, our chosen parameters produce a sequence that endlessly repeats the subsequence $1, 5, 12, 8$. This is obviously not very random, but we can fix it by choosing m and a more carefully. We will revisit this in a moment.

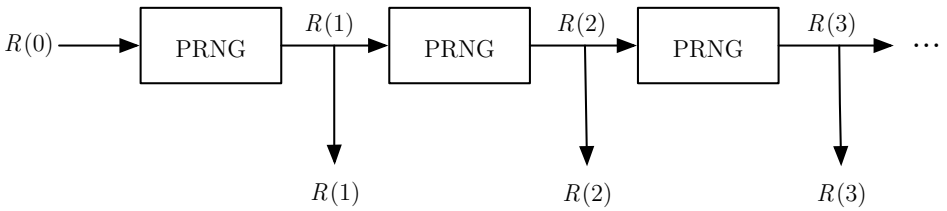

Figure 1 An illustration of the operation of a simple PRNG.

Implementation

But first, let's implement the PRNG "black box" in Figure 1, based on the Lehmer PRNG difference equation. It is a very simple function!

```
def lehmer(r, m, a):
    """Compute the next pseudorandom number using a Lehmer PRNG.

    Parameters:
        r: the seed or previous pseudorandom number
        m: a prime number
        a: an integer between 1 and m - 1

    Return value: the next pseudorandom number
    """

    return (a * r) % m
```

The parameter `r` represents $R(t-1)$ and the value returned by the function is the single value $R(t)$.

Now, for our function to generate good pseudorandom numbers, we need some better values for m and a. Some particularly good ones were suggested by Keith Miller and Steve Park:

$$m = 2^{31} - 1 = 2{,}147{,}483{,}647 \text{ and } a = 16{,}807.$$

A Lehmer generator with these parameters is often called a *Park-Miller psuedorandom number generator*. To create a Park-Miller PRNG, we can call the `lehmer` function repeatedly, each time passing in the previously generated value and the Park-Miller values of m and a, as follows.

```
def randomSequence(length, seed):
    """Returns a list of Park-Miller pseudorandom numbers.

    Parameters:
        length: the number of pseudorandom numbers to generate
        seed:   the initial seed

    Return value: a list of Park-Miller pseudorandom numbers
                  with the given length
    """

    r = seed
    m = 2**31 - 1
    a = 16807
    randList = [ ]
    for index in range(length):
        r = lehmer(r, m, a)
        randList.append(r)
    return randList
```

In each iteration, the function appends a new pseudorandom number to the end of

the list named `randList`. (This is another list accumulator.) The `randomSequence` function then returns this list of `length` pseudorandom numbers. Try printing the result of `randomSequence(100, 1)`.

Because all of the returned values of `lehmer` are modulo m, they are all in the interval $[0..m-1]$. Since the value of m is somewhat arbitrary, random numbers are usually returned instead as values from the interval [0,1), as Python's `random` function does. This is accomplished by simply dividing each pseudorandom number by m. To modify our `randomSequence` function to returns a list of numbers in [0,1) instead of in $[0..m-1]$, we can simply append `r / m` to `randList` instead of `r`.

Reflection 4 *Make this change to the* `randomSequence` *function and call the function again with seeds 3, 4, and 5. Would you expect the results to look similar with such similar seeds? Do they?*

As an aside, you can set the seed used by Python's `random` function by calling the `seed` function. The default seed is based on the current time.

The ability to generate an apparently random, but reproducible, sequence by setting the seed has quite a few practical applications. For example, it is often useful in a Monte Carlo simulation to be able to reproduce an especially interesting run by simply using the same seed. Pseudorandom sequences are also used in car door locks. Each time you press the button on your dongle to unlock the door, it is sending a different random code. But since the dongle and the car are both using the same PRNG with the same seed, the car is able to recognize whether the code is legitimate.

Testing randomness

How can we tell how random a sequence of numbers really is? What does it even mean to be truly "random?" If a sequence is random, then there must be no way to predict or reproduce it, which means that there must be no shorter way to describe the sequence than the sequence itself. Obviously then, a PRNG is not really random at all because it is entirely reproducible and can be described by simple formula. However, for practical purposes, we can ask whether, if we did not know the formula used to produce the numbers, could we predict them, or any patterns in them?

One simple test is to generate a histogram from the list of values.

Reflection 5 *Suppose you create a histogram for a list of one million random numbers in* [0,1)*. If the histogram contains 100 buckets, each representing an interval with length 0.01, about how many numbers should be assigned to each bucket?*

If the list is random, each bucket should contain about 1%, or 10,000, of the numbers. The following function generates such a histogram.

```
import matplotlib.pyplot as pyplot

def histRandom(length):
    """Displays a histogram of numbers generated by the Park-Miller PRNG.

    Parameter:
        length: the number of pseudorandom numbers to generate

    Return value: None
    """

    samples = randomSequence(length, 6)
    pyplot.hist(samples, 100)
    pyplot.show()
```

Reflection 6 *Call* `histRandom(100000)`*. Does the* `randomSequence` *function look like it generates random numbers? Does this test guarantee that the numbers are actually random?*

There are several other ways to test for randomness, many of which are more complicated than we can describe here (we leave a few simpler tests as exercises). These tests reveal that the Lehmer PRNG that we described exhibits some patterns that are undesirable in situations requiring higher-quality random numbers. However, the Python `random` function uses one of the best PRNG algorithms known, called the *Mersenne twister*, so you can use it with confidence.

Exercises

5.2.1* We can visually test the quality of a PRNG by using it to plot random points on the screen. If the PRNG is truly random, then the points should be uniformly distributed without producing any noticeable patterns. Write a function

`testRandom(n)`

that uses turtle graphics and `random.random` to plot `n` random points with x and y each in [0,1). Here is a "skeleton" of the function with some turtle graphics set up for you. Calling the `setworldcoordinates` function redefines the coordinate system so that the lower left corner is (0,0) and the upper right corner is (1,1). Use the turtle graphics functions `goto` and `dot` to move the turtle to each point and draw a dot there.

```
def testRandom(n):
    """ your docstring here """

    tortoise = turtle.Turtle()
    screen = tortoise.getscreen()
    screen.setworldcoordinates(0, 0, 1, 1)
    screen.tracer(100)    # only draw every 100 updates
    tortoise.up()
    tortoise.speed(0)

    # draw the points here
```

```
screen.update()       # ensure all updates are drawn
screen.exitonclick()
```

5.2.2. Repeat Exercise 5.2.1, but use the `lehmer` function instead of `random.random`. Remember that `lehmer` returns an integer in $[0\,..\,m-1]$, which you will need to convert to a number in [0,1). Also, you will need to call `lehmer` twice in each iteration of the loop, so be careful about how you are passing in values of `r`.

5.2.3. Another simple test of a PRNG is to produce many pseudorandom numbers in [0,1) and make sure their average is 0.5. To test this, write a function

```
avgTest(n)
```

that returns the average of `n` pseudorandom numbers produced by the `random` function.

Selected Exercise Solutions

5.2.1

```
import random, turtle

def testRandom(n):
    tortoise = turtle.Turtle()
    screen = tortoise.getscreen()
    screen.setworldcoordinates(0, 0, 1, 1)
    screen.tracer(100)
    tortoise.up()
    tortoise.speed(0)

    for point in range(n):
        x = random.random()
        y = random.random()

        tortoise.goto(x, y)
        tortoise.dot()

    screen.update()
    screen.exitonclick()
```