CHAPMAN & HALL/CRC
TEXTBOOKS IN COMPUTING

EXPLORATIONS IN COMPUTING

An Introduction to Computer Science

John S. Conery

With Illustrations
by Phil Foglio

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group an **informa** business
A CHAPMAN & HALL BOOK


THINK OF A TOTALLY RANDOM NUMBER
RUBY
3
3
3
3
3
©Pi Foglio 10

Chapter 9

Now for Something Completely Different

An algorithm for generating random numbers

Most popular games involve some element of chance. Players roll dice, shuffle a deck of cards, spin a wheel, or use some other method for making a random selection in the game. Computer-based versions of these games are played the same way, but instead of using real dice or using a real deck of cards a computer program manages the game. Somewhere inside the application is an algorithm that generates the next roll of the dice or shuffles the deck of cards.

As you might imagine, a computer-based game would not be very popular if the algorithm that generates moves is not realistic. If you're playing a board game like backgammon, you expect rolls of the dice to be similar to rolls of real dice, and you would start to become suspicious if your opponent rolls doubles far more often than you do. Some people like to practice playing poker against a computer to prepare for tournaments. If the program they train with doesn't deal the same kinds of hands that will be seen in the tournament, the software is not going to be very useful.

The word that best describes our expectations for the computer-based games is **random**. When we play a board game we want the computer to generate a pair of numbers that is just as random as rolling a pair of real dice, and when we play cards we want the computer to generate an ordering for a deck of cards that is as random as what we would get if we carefully shuffle a real deck.

The natural question, of course, is how to define what we mean by "random." Colloquially, random often means "unusual" or "unexpected." But in games, and in other situations where random values are required, something is random if it is **unpredictable**. To be more precise,

what we are looking for is an algorithm that generates a sequence of values in which there is no apparent pattern or relationship between one value and the next. If the algorithm is used to simulate rolls of a six-sided die, each new number should be independent of the previous number, and if it is used to deal cards, each new hand should be unrelated to the previous hand. In other words, to use the phrase in the title of this chapter, each value produced by the algorithm should be "completely different" from the previous value.[1]

The goal for this chapter is to explore techniques for using a computer to generate random values. We will take a closer look at the idea that a sequence of values is random if successive values are independent of one another, and explore various ways of trying to determine whether the values are, in fact, random, or whether there are some unexpected connections between them.

9.1 Pseudorandom Numbers

Before you read this section, here is a simple experiment to try. Start your text editor and create a new file. Type 50 numbers between 1 and 6, putting one number on each line. Try to write the numbers as if you were rolling a die; in other words, each number should be unrelated to the one on the previous line.

As you were thinking of numbers to write, were you able to concentrate on each new number, forgetting about what you had written before? Or did you find yourself thinking something like "I haven't written a 3 in a while, I'd better write one now" or "hmmm, that's two 6s in a row, I'd better write something else." If you gave in to the temptation to think about previous values you were starting to add a bias to your results. It is very hard for people to generate a truly random sequence of values.

Engineers, statisticians, and other professionals have used random sequences for many years. Before the algorithms described in this chapter were available, people who needed a random sequence would look in a book of random numbers. A well-known reference book, published in 1955 by the RAND Corporation, was *A Million Random Digits*, a 400-page book with 2500 random digits between 0 and 9 on each page. The authors used what they called an "electronic roulette wheel" to generate random electronic pulses. The electronic circuit was connected to a computer, and a device that measured the pulses converted them to digital form. Another way to generate random signals is to use a white noise generator, an audio device that produces something that sounds like static.

To play a game with a computer we do not need to connect to a roulette wheel, white noise generator, or other physical device that behaves randomly. Instead, applications use an algorithm that produces a different value each time it is called. For example, there is a method built into Ruby called `rand`. If we pass a number n to `rand`, it will return a value between 0 and $n-1$:

```
>> rand(100)
=> 54
>> rand(100)
=> 39
```

[1] The title is a catch-phrase from the BBC comedy series, *Monte Python's Flying Circus*.

There is an interesting paradox here. According to the definition given in Section 1.3, an algorithm has a well-defined set of inputs, and a precise and unambiguous sequence of steps that leads to the output. If that's the case, how can an algorithm produce a different result each time it is run? Or, in terms of Ruby programs, how is it possible for a method to return a different value each time we call it?

The answer to these questions is the method *appears* to be random, but in fact it follows a predefined set of rules, just like any method. The general idea is to produce a long sequence of numbers, and if we look at a small set of numbers in the middle of the sequence, they will appear random. Even though they are not truly random, many applications, such as computer games, can use them in place of actual random values. Because the numbers are created by an algorithm, and not an external source of random data like the RAND corporation's roulette wheel, we refer to them as **pseudorandom**, and the algorithm that produces the sequence is a **pseudorandom number generator**, or PRNG.

As an introduction to how a PRNG might work, imagine a situation where an event needs to be scheduled at regular intervals throughout a day. Perhaps a nurse in a hospital needs to periodically administer medications to a patient, or a researcher needs to collect data from an experiment, and it is our job to schedule these events. If the events occur every eight hours the schedule is simple and easy to remember. One plan would be to schedule the first three events at midnight, 8 A.M., and 4 P.M. The next event would be at midnight again, and the schedule would repeat. Since the schedule is the same every day it is easy to remember, and we can simply tell people what the schedule is (Figure 9.1).

However, if the events need to occur every seven hours the schedule is more complicated. If the first event is at midnight, the next two events would be 7 A.M. and 2 P.M. But now the fourth event will be at 9 P.M., and the second day would not be like the first.

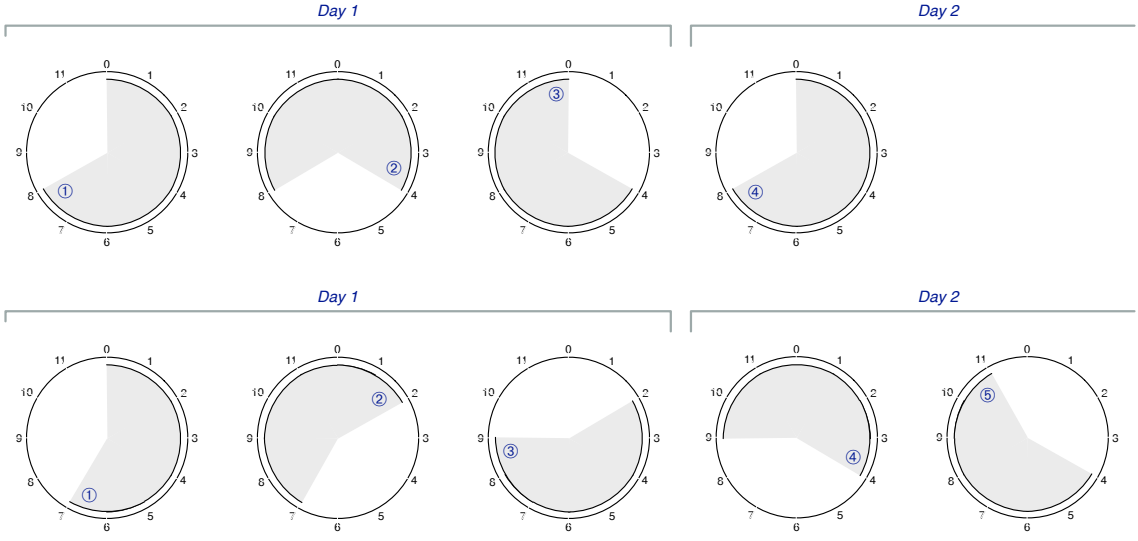

Figure 9.1: *If an event occurs every eight hours (top row), a schedule is very predictable, and events occur at the same time every day. However, if events are seven hours apart (bottom row) each day is different, and it is more difficult to keep track of when events are scheduled to occur.*

It's very straightforward to use Ruby to figure out the schedule. Start by making a list named `t` and initializing it so it has the time for the first event:

```
>> t = [0]
=> [0]
```

For now we will use a 12-hour clock. Let's assume we can tell by context whether "0" is midnight or noon, and whether a "4" means 4 A.M. or 4 P.M. (one of the exercises at the end of the chapter will be to modify the schedule to use a 24-hour clock).

To add the time for the next event, the list needs to be extended with a time that is seven hours later than the one currently at the end of the list. Since we're using a 12-hour clock, the expression is

```
>> t << (t.last + 7) % 12
=> [0, 7]
```

Recall from previous chapters that an expression of the form `a << x` means "attach `x` to the end of array `a`," and that the `%` symbol is Ruby's mod operator, *i.e.*, the expression `x % 12` means "the remainder after dividing `x` by 12."

If the above operation is repeated several more times, this is what the list would look like:

```
[0, 7, 2, 9, 4, 11, 6, 1, 8, 3, 10, 5, 0, 7, 2]
```

That's a pretty difficult schedule to remember, but it matches the specification, and we could either put it on a poster, or ask people to store it in the calendar on their cell phone.

To get back to the subject of generating random numbers, at first it looks like the list above might be random. If we gave the list to a person who did not know the numbers were times from a 12-hour clock, they would probably have a hard time guessing the list was generated by a simple rule, or what that rule was. On closer inspection, however, we can see some regularity in the list. For one thing, every hour appears exactly once in the first 12 places in the list (assuming we interpret 0 as 12 o'clock). When we are rolling a 6-sided die, it would be very rare to see a sequence of rolls in which every number between 1 and 6 showed up before any number appeared again. In fact, it's not at all uncommon to have the same number appear twice in a row. The fact that we used a rule that forces every number to occur once before the first number appears again means the list is not truly random.

If we add a few more items to the list, using the same rule, we will see conclusive evidence that the list is not random. As soon as the first number is generated again, the pattern will start to repeat itself. In other words, as soon as the rule attaches a 0 to the end of the list, all the values that followed 0 the first time will occur again, and in exactly the same order. If we want to know what follows 2 in the list shown above, we can either apply the rule again and evaluate $(2+7) \bmod 12$, or we can find the 2 earlier in the list and look at the number came after it.

The rule of adding 7 and taking the remainder mod 12 does not create a very useful list of pseudorandom numbers, but it is a good starting point. A more general formula for adding a new value x_{i+1} to the end of a list is

$$x_{i+1} = (a \times x_i + c) \bmod m$$

where a, c, and m are all constants and x_i is the previous item in the list. The "add seven to the current time" rule follows this general pattern, since it has $a = 1$, $c = 7$, and $m = 12$.

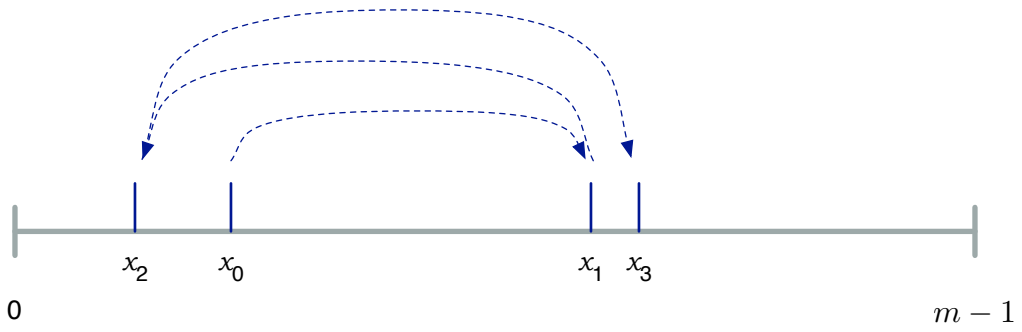

Figure 9.2: *A pseudorandom sequence defined by the rule* $x_{i+1} = (a \times x_i + c) \bmod m$. *If values of* a, c, *and* m *are chosen carefully, every value between 0 and* $m - 1$ *will appear in the first* m *numbers in the sequence and the number line will be completely filled in.*

If a and c are defined properly, this rule will place every value from 0 to $m - 1$ in the list before it repeats (Figure 9.2). As we saw previously, with $a = 1$ and $c = 7$, the rule makes a list of all 12 values between 0 and 11. But if we use $c = 8$ (which is what we did originally, when the schedule called for events every eight hours) the list is much shorter:

```
[0, 8, 4, 0, 8...]
```

The number of items in the list before it starts to repeat is called the **period**. The period when $c = 8$ is 3, because the list only has 3 items before it starts repeating. The period when $c = 7$ is 12 because all 12 numbers are in the list before it repeats.

If we use a large value of m, and values of a and c that work well for that m, we can make a very long list of unique numbers. Furthermore, if we look at small portions of the list, it will be very difficult to figure out what rule is used to generate the numbers. In practical terms, we will have a list of random numbers. Even though they were produced by a pseudorandom number generator, and will not be truly random, for many applications they might be "random enough."

The formula shown above was used in some of the earliest pseudorandom number generators, and is still widely used because it is very easy to implement and does a reasonable job for games and other casual applications. In current implementations, m is typically 2^{32}, or roughly 4×10^9, so this technique will generate a list of over four billion numbers before it repeats. The PRNG built into Ruby is based on a newer and more sophisticated algorithm that has a period of 2^{19937}.

In the next section, we will see how to put this formula to use in making random numbers for games and other applications. Later in the chapter we will come back to the claim that the sequence is "random enough" by looking at different ways to measure randomness and using these techniques to assess the randomness in the sequence of values produced by a PRNG.

Tutorial Project

The first exercises will explore the formula for making pseudorandom numbers. Start an IRB session and load the module that will be used in this chapter:

```
>> include RandomLab
=> Object
```

T1. Make the schedule for events that occur every eight hours. Initialize the schedule with the time for the first event:

```
>> t = [0]
=> [0]
```

T2. Apply the rule that adds a new event that will occur eight hours after the previous event:

```
>> t << (t.last + 8) % 12
=> [0, 8]
```

T3. Apply the rule three more times. Enclose the previous expression in braces and call the `times` method to make a schedule that repeats the basic pattern of 0, 8, 4:

```
>> 3.times { t << (t.last + 8) % 12 }
=> 3
>> t
=> [0, 8, 4, 0, 8]
```

See the sidebar below for an explanation of the `times` method.

T4. Start a new schedule by typing the first expression again:

```
>> t = [0]
=> [0]
```

T5. Repeat the expression that adds new events, but make 16 events scheduled 7 hours apart:

```
>> 16.times {t << (t.last + 7) % 12}
=> 16
>> t
=> [0, 7, 2, 9, 4, 11, 6, 1, 8, 3, 10, 5, 0, 7, 2, 9, 4]
```

Can you see how the period for this new schedule is 12, *i.e.*, how all 12 numbers between 0 and 11 appear in this list before it starts to repeat?

The `times` Method

The projects in this chapter use a Ruby method named `times`. This method is used to repeat an operation a specified number of times. The first example right tells IRB to print a string three times.

```
>> 3.times { puts "hello" }
hello
hello
hello
=> 3
```

We can also put a variable name to the left of the method, as shown in the second example, which appends 10 random numbers to an array. The value returned by `times` is just the number of times the operation was executed.

```
>> n = 10
=> 10
>> n.times { a << rand(100) }
=> 10
```

The RandomLab module defines a method named `prng_sequence` that implements the general form of the equation that creates a list of pseudorandom numbers. The three parameters to the method are the values of a, c, and m to plug into the equation. The list returned by this method will have m numbers, and with the right combination of a and c all values from 0 to $m - 1$ will be in the list.

T6. As a first test, use the `prng_sequence` method to make the schedule of events that occur every eight hours:

```
>> sched8 = prng_sequence(1, 8, 12)
=> [0, 8, 4, 0, 8, 4, 0, 8, 4, 0, 8, 4]
```

As expected, this sequence is not very random.

T7. Ruby arrays have a method named `uniq` that will return a list of all the unique values in an array, *i.e.*, the method returns a copy of the array with all the duplicates removed. Apply this method to the schedule you just made:

```
>> sched8.uniq
=> [0, 8, 4]
```

Do you see how the `uniq` method confirms the fact that only `0`, `8`, and `4` are used in this array?

T8. Now make a second schedule for events that occur every seven hours:

```
>> sched7 = prng_sequence(1, 7, 12)
=> [0, 7, 2, 9, 4, 11, 6, 1, 8, 3, 10, 5]
```

T9. Get a list of unique numbers in this list:

```
>> sched7.uniq
=> [0, 7, 2, 9, 4, 11, 6, 1, 8, 3, 10, 5]
```

T10. One way to see whether every number between 0 and $m - 1$ is in the list is to count the number of unique values. If an array has no duplicates, the length of the list returned by `uniq` will be the same as the length of the original list:

```
>> sched8.uniq.length
=> 3
>> sched7.uniq.length
=> 12
```

T11. This combination of parameters will generate a list of numbers between 0 and 999:

```
>> seq = prng_sequence(3, 337, 1000)
=> [0, 337, 348, 381, 480, ... 97, 628, 221, 0]
```

Look at the list of 1000 numbers in your terminal emulator window. Does it look "random" to you?

T12. Let's see if this combination of a, c, and m made a list with a period of m:

```
>> seq.uniq.length
=> 100
```

So this combination of a, c, and m yields a list that has the same 100 numbers repeated 10 times—not very random at all.

T13. Here is a better combination. Change the first argument to 81 instead of 3:

```
>> seq = prng_sequence(81, 337, 1000)
=> [0, 337, 634, 691, ... 749, 6, 823]
```

T14. How many unique numbers are in this list?

```
>> seq.uniq.length
=> 1000
```

Just what we were looking for. Does this list look random?

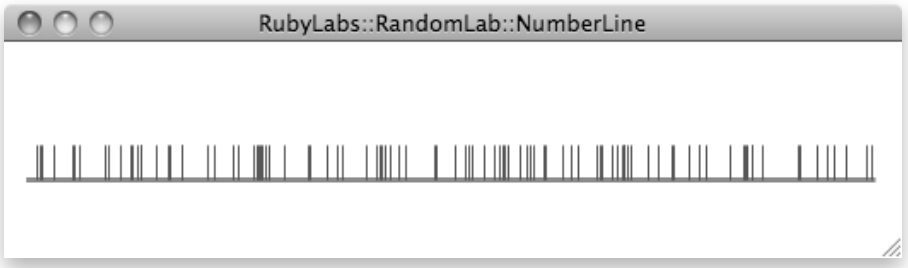

Figure 9.3: *The first 100 values produced by a PRNG with* $a = 81$*,* $c = 337$*, and* $m = 1000$*.*

The RandomLab module includes methods that will draw pictures based on the values produced by a pseudorandom number generator. The next set of exercises will plot the values returned by a PRNG on a number line, so you can get a sense of how the values are scattered over a specified range (Figure 9.3).

T15. Create a window with a number line for values between 0 and 999:

```
>> view_numberline(1000)
=> true
```

You should see a canvas with a line running through the middle of it.

T16. A method named `tick_mark` will display a mark at a specified point on the line. Type this expression to display a tick mark in the middle of the line:

```
>> tick_mark(500)
=> nil
```

T17. This command will draw 100 tick marks, at locations 0 through 99:

```
>> 100.times { |i| tick_mark(i) }
=> 100
```

T18. Reinitialize the display by calling `view_numberline` again (Exercise T15).

T19. Type this expression to plot the points from the pseudorandom sequence that contained only 100 different numbers:

```
>> prng_sequence(3, 337, 1000).each { |i| tick_mark(i) }
=> [0, 337, 348, 381, ... 97, 628, 221]
```

Can you see how only about 1/10 of all the points are filled in? So even though the call to `prng_sequence` made a list of 1000 numbers, the list has 100 different values repeated 10 times.

T20. Reinitialize the drawing, and repeat the call to `prng_sequence`, but change the 3 to 81 so you get the sequence with all 1000 numbers:

```
>> view_numberline(1000)
=> true
>> prng_sequence(81, 337, 1000).each { |i| tick_mark(i) }
=> [0, 337, 634, 691, 308, 285, ... 749, 6, 823]
```

Now your number line should be completely filled in. There are 1000 numbers in the list, and each value from 0 to 999 will occur exactly once.

The expression in Exercise T20 filled in each point in the number line. But we could have done that by repeating Exercise T17 and telling it to draw 1000 tick marks. When you were watching the display, were the numbers created by `prng_sequence` added in a random order?

9.2 Numbers on Demand

Although the sequences produced in the exercises for the previous section may appear to be random, appearances can be deceiving. Later in the chapter we'll look at some techniques for evaluating sequences to test randomness, but first we'll see how the random number generator can be implemented in a method intended for games and other applications that need random numbers.

As a practical matter, applications do not generate a list of all numbers in a pseudorandom sequence. A game may need only a few hundred rolls of the dice, and it would be a big waste of time and space to generate the full list of billions of pseudorandom numbers defined by the best random number generators. Instead, games and other applications use a programming technique that creates random numbers "on demand."

As an analogy for how this might work, imagine a scenario where a statistician has a lab assistant who is in charge of random numbers. The situation where the full sequence is generated ahead of time would correspond to the lab assistant carrying around the RAND book of random digits. Each time the statistician needs a random value, the assistant would look up the current digit in the book, and then move his bookmark one place to the right. However, when the numbers are generated by an algorithm, the assistant just needs to keep track of one number on a small piece of paper. When the statistician needs a random number, the assistant plugs the number into the equation to compute the next value in the sequence and then erases the old number and replaces it with the new value.

The RandomLab module has the definition of a new type of object called a PRNG that uses the one-number-at-a-time strategy to implement a pseudorandom number generator. To make a PRNG object, pass the a, c, and m parameters to the method that makes a new object. For example, in the last section we made a pseudorandom sequence of $m = 1000$ numbers using $a = 81$ and $c = 337$:

```
>> seq = prng_sequence(81, 337, 1000)
=> [0, 337, 634, 691, ... 749, 6, 823]
```

To make a PRNG object based on this sequence, we pass these same values to `PRNG.new`:

```
>> p = PRNG.new(81, 337, 1000)
=> #<RandomLab::PRNG a: 81 c: 337 m: 1000>
```

A method named `advance` moves to the next pseudorandom number in the sequence. If we call `advance` a couple of times, it should return the first two values in the sequence:

```
>> p.advance
=> 337
>> p.advance
=> 634
```

Each time `advance` is called it acts like the assistant with the piece of scrap paper. It sets the value of x to the current value in the sequence. It then evaluates $(a \times x + c) \bmod m$. The value of this expression is saved as the new state of the sequence, and it's also returned as the value of the call to `advance`.

Tutorial Project

T21. If you started a new IRB session since you worked on the project in the previous section, type this expression again to make a pseudorandom sequence:

```
>> seq = prng_sequence(81, 337, 1000)
=> [0, 337, 634, 691, ... 749, 6, 823]
```

T22. Print the first 10 numbers in the sequence:

```
>> seq[0..9]
=> [0, 337, 634, 691, 308, 285, 422, 519, 376, 793]
```

T23. Make a PRNG object using the same values of a, c, and m:

```
>> p = PRNG.new(81, 337, 1000)
=> #<RandomLab::PRNG a: 81 c: 337 m: 1000>
```

T24. A method named `state` will show us the current number in the sequence (the number written on the "scrap paper"):

```
>> p.state
=> 0
```

So the PRNG object starts out with the same value as the array made by `prng_sequence`.

T25. If you call the `advance` method you should get back the next value in the sequence:

```
>> p.advance
=> 337
```

T26. Call `advance` a few more times. Do you get the same numbers you see at the front of the list named `seq`?

T27. Call `p.state` to see what the current number is, and then call `p.advance` again. Do you see how each call to `advance` simply computes the next value in the pseudorandom sequence?

9.3 Games with Random Numbers

The previous two sections showed how it is possible to create a list of numbers that appear to be random. We looked at the equation that defines how each element in the list is derived from the previous one, and then saw how we might implement this technique in a way that lets us get new numbers "on demand" instead of creating the entire list at once. The goal for this section is to show how a pseudorandom sequence can be put to work by an application that uses random numbers.

If the three parameters (named a, c, and m) that define the relationship between successive numbers are chosen carefully, the pseudorandom sequence will have values from 0 up to $m - 1$. However, if we are writing a program to play a game like backgammon, which is based on rolling a pair of dice, we want a series of numbers between 1 and 6.

A simple approach to simulating random rolls of a six-sided die is to get a value from the pseudorandom sequence, find its remainder after dividing by 6 (which will yield a number between 0 and 5), and then add 1. Here is a Ruby expression that transforms a number from a PRNG object named `p` into a number between 1 and 6:

```
>> (p.advance % 6) + 1
=> 2
```

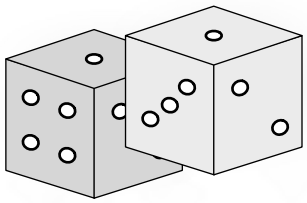

Figure 9.4: *If pseudorandom numbers are used by a program that plays a game with dice, the values in the pseudorandom sequence are mapped to numbers between 1 and 6.*

If we evaluate this expression several more times and save the results in a list, we can see something interesting:

```
>> rolls = []
=> []
>> 10.times { rolls << (p.advance % 6 + 1) }
=> 10
>> rolls
=> [2, 5, 2, 3, 4, 3, 4, 5, 2, 1]
```

As expected, all the numbers in the list are between 1 and 6. But notice the numbers that follow the 2s in this list: the first 2 is followed by 5, the second by 3, and the third by 1.

The fact that a number can be followed by different values adds to the illusion that this sequence is truly random, and not just pseudorandom. Because 1000 was passed as the value of m when the PRNG was created, the pseudorandom sequence generated by `p` will have every number between 0 and 999. But when we divide these numbers by 6, the remainders will be between 0 and 5. Each remainder will show up hundreds of times, and the value that follows a remainder can be any one of the 6 numbers between 0 and 5.

PRNG objects have a method named `random` that does this conversion for us. This method takes two parameters, `min` and `max`. The method gets the next value from the pseudorandom sequence, and then it uses the mod operator to turn that value into an integer between `min` and `max`. For example, a program that wants to use the PRNG object to simulate rolls of a die would call `p.random(1,6)` to get numbers from 1 to 6:

```
>> p.random(1,6)
=> 1
>> p.random(1,6)
=> 2
```

There is one more issue that needs to be addressed if we want to use a PRNG in an application, and that is how to initialize the sequence. Each time a new PRNG object is created, the initial state is set to 0. Since the application alway uses the same values of a, c, and m to create the PRNG, it will always get the same sequence of numbers at the start of each game. Our backgammon program will always start off with the same rolls of the dice: 2, 5, 2, 3, 4, 3, 4, 5, 2, *etc.* Players will soon recognize that every game starts the same way, and they will lose interest.

One way to address this problem is to set the state of the PRNG to a particular value. We can create the new object, letting the `new` method set the state to 0, but then we can change the state to any value we choose. This process of changing the state of a PRNG is called "seeding." If we want to seed one of our PRNG objects we just have to call a method named `seed`, *e.g.*,

```
>> p.seed(226)
=> 226
```

The next call to `advance` will get the number that follows 226 in the underlying pseudo-random sequence.

But now the question becomes, which value do we use for a seed? If we're using the simple PRNG that has $m = 1000$ we can choose any number from 0 to 999. But if we just choose our favorite number, and write that number into the program, we will have the same problem. The game will start out with a different set of rolls than if the sequence started with 0, but since the PRNG always starts in the same state players are again going to see the same set of rolls at the start of each game.

A common solution is to use the system clock. Ruby has a module named `Time` that has a collection of methods for dealing with dates and times. Calling `Time.now` will return the current date:

```
>> Time.now
=> Mon Dec 28 09:30:34 -0800 2009
```

It's also possible to turn a date into an integer by calling the `to_i` method:

```
>> Time.now.to_i
=> 1262021437
```

This value is the number of seconds since January 1, 1970 (the date of the "Big Bang" when time started in the Unix world).

To initialize each new game, an application typically gets the current time, converts it to an integer, and uses that value to seed the random number generator. Although the time value is increasing in a predictable way, the values that follow in the pseudorandom sequence will be very different, and the games will start out with a different sequence of rolls each time.

Tutorial Project

T28. Make a PRNG object with the same values of a, c, and m used previously:

```
>> p1 = PRNG.new(81, 337, 1000)
=> #<RandomLab::PRNG a: 81 c: 337 m: 1000>
```

T29. Get 10 rolls of a die by calling the `random` method and asking it for values between 1 and 6:

```
>> 10.times { puts p1.random(1,6) }
2
5
2
3
...
```

T30. Make a second pseudorandom number generator using the same parameter values:

```
>> p2 = PRNG.new(81, 337, 1000)
=> #<RandomLab::PRNG a: 81 c: 337 m: 1000>
```

T31. Now print the first 10 rolls of the die produced by this PRNG:

```
>> 10.times { puts p2.random(1,6) }
2
5
2
...
```

Do you see why they are the same rolls? Each new PRNG object is initialized to start at the same place in the pseudorandom sequence, so each will create the same sequence of values.

T32. Call the method that gets the current date and time:

```
>> Time.now
=> Mon Dec 28 10:33:57 -0800 2009
```

Obviously the result you get will be different, but you should see a string that contains a date formatted according to the conventions you set for your operating system (this string is in the format seen by users in the United States).

T33. Type the expression again after adding `to_i` to convert the time into an integer:

```
>> Time.now.to_i
=> 1262025240
```

T34. If you want to use the current time to set the seed for the PRNG, convert the time to a number between 0 and $m - 1$:

```
>> rs = Time.now.to_i % 1000
=> 574
```

T35. Create a third PRNG, and then set its seed using the current time:

```
>> p3 = PRNG.new(81, 337, 1000)
=> #<RandomLab::PRNG a: 81 c: 337 m: 1000>
>> p3.seed(rs)
=> 574
```

T36. Now get 10 rolls of the die from this new PRNG:

```
>> 10.times { puts p3.random(1,6) }
4
1
4
...
```

T37. Set the seed for `p3` again, but this time use a value that is one greater than the previous seed, and then get 10 rolls of the die:

```
>> p3.seed(rs+1)
=> 575
>> 10.times { puts p3.random(1,6) }
1
6
3
...
```

After doing these last two exercises, do you see how a slight change in the seed leads to a big change in the pseudorandom sequence? Just a slight difference in the time used to set the seed will result in a completely different sequence of random numbers at the start of each game.

9.4 Random Shuffles

If we want to write a program to play bridge, poker, or some other card game we are going to need a method that makes random collections of cards. Simulating the roll of six-sided die just involved computing a number between 1 and 6, but here the problem is slightly different. It would be easy enough to label each card in the deck with a number between 0 and 51, and then use a pseudorandom number generator to compute a number between 0 and 51 as a way of choosing a card at random. But if we want to deal a hand there is a chance that we could end up with two copies of the same card. In bridge, each hand has 13 cards, and if we make a hand by just calling `random(0,51)` 13 times, odds are we will have a collection where the same number appears twice.

One way to solve this problem is to use a method that "shuffles" a deck of cards. We will start with a collection of all 52 cards, and then each time we play a game we can shuffle the collection by rearranging the items in a random order. The goal for this section will be to develop a method that rearranges objects in an array. The method will use values from a pseudorandom number generator to produce a random ordering.

To make the project more realistic, the RandomLab module defines a type of object named Card. A card object will have a rank (ace, king, queen, *etc.*) and a suit. We will represent a deck of cards as an array of 52 objects that each represent a different card from a standard deck. As with other kinds of objects, an expression with the name of the type followed by the word `new` will create an object of that type. If you don't pass an argument to `new` you will get back a random card:

```
>> Card.new
=> 10♥
>> Card.new
=> 2♣
```

When we make a deck of cards, we don't want 52 random cards, but instead we want one of each possible card. You can pass an integer between 0 and 51 to the `new` method to tell it which card to make:

```
>> Card.new(0)
=> A♠
>> Card.new(1)
=> K♠
>> Card.new(50)
=> 3♣
>> Card.new(51)
=> 2♣
```

As a convenience, we can get a complete deck by calling `new_deck`, which makes each of the 52 card objects for us:

```
>> d = new_deck
=> [A♠, K♠, Q♠, ... 3♣, 2♣]
```

Mathematicians refer to an ordering of items as a **permutation**. The goal for the project in this section is to define a method named `permute!` that we can use to make a new random ordering of the items in an array. For example, after making a full deck of cards as shown above, we can call `permute!` to shuffle the deck. Each time we call `permute!` it will make a new random permutation:

```
>> permute!(d)
=> [9♠, A♣, 4♣, Q♣, 7♠, J♣, 4♥, ... ]
>> permute!(d)
=> [10♣, Q♣, 7♥, A♦, 6♣, 2♥, 8♣, ... ]
```

Unlike the sorting methods we developed in Chapters 4 and 5, which returned sorted copies of the input arrays, each call to `permute!` changes the order of items in the array it is passed.

A program that plays poker would probably "deal" the cards the same way we would in a real game. After calling `permute!` to shuffle the deck, it would then give `d[0]` to the first player, `d[1]` to the next player, and so on. For our experiments with poker hands, however, we'll just need one hand, and we can make this hand by using the first five cards in the deck. This Ruby expression shuffles the deck and makes an array containing the first five cards in the new deck:

```
>> permute!(d).first(5)
=> [2♣, A♠, 5♦, 10♣, 7♣]
```

We've seen the method named `first` before: the expression `a.first` returns the item at the front of the array `a`. This time we're passing the number 5 as an argument so it returns an array of the first 5 items.

One straightforward algorithm for permuting the items in an array is based on an iteration that exchanges two items at each step. Begin by picking up the first item and exchanging it with a random item somewhere to the right. Then exchange the second item with a random item somewhere to its right, then exchange the third item with a random item somewhere to its right, and so on until you reach the end of the array. This algorithm is reminiscent of the insertion sort algorithm: on each iteration the array has two regions, where items in the left part of the array have been exchanged and items to the right are waiting to be moved.

It's easy to exchange the values of two items in Ruby by using a construct known as **parallel assignment**. A normal assignment statement has a single variable on the left side of the assignment operator, *e.g.*,

```
>> x = 7
```

A parallel assignment has two or more variable names on the left, as in

```
>> x, y = 3, 4
```

The variables on the left are assigned the corresponding values from the right, so in this example `x` is set to 3 and `y` is set to 4. It's called a "parallel" assignment because we can think of the two assignments happening at the same time.

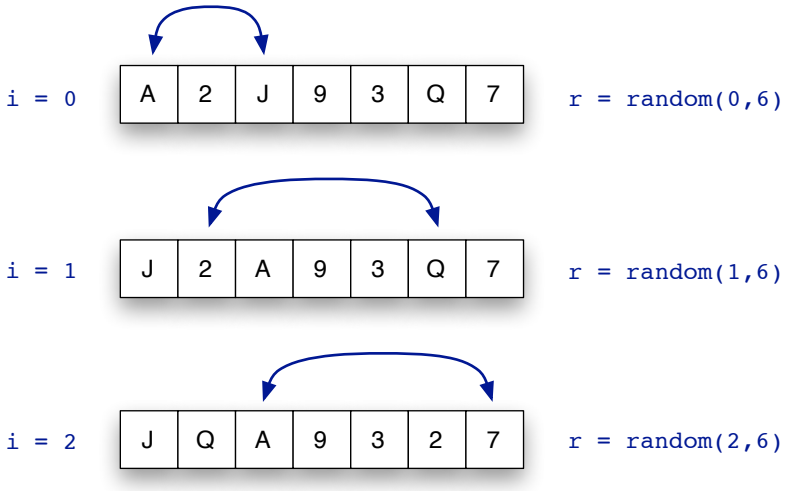

Figure 9.5: *On each iteration, the item at location* `i` *changes places with an item at a random location to the right.*

To use a parallel assignment to exchange the value of two variables, write the names of the variables in different orders on the left and right sides of the assignment operator:

```
>> x, y = y, x
```

Ruby executes this statement by fetching the current values of `y` and `x` and then writing them to the specified locations. It might help to think of a juggler picking up two balls and tossing them to different hands in the same motion: while the value of `y` is being tossed to `x` the value of `x` is being tossed to `y`.

Parallel assignment also works for locations in an array. For example, to exchange the values in the first two locations of an array `a` we can just write

```
>> a[0], a[1] = a[1], a[0]
```

Now that we know how to exchange two items in an array, writing the method is straightforward. The algorithm is an iteration in which a variable named `i` takes on values from 0 to $n-2$, where n is the length of the array. At each step, we just need to set a variable `r` to a random value between `i` and $n-1$, which is the last location in the array. We then exchange the items at locations `i` and `r` using a parallel assignment. One small detail to notice here: it's possible that `i` and `r` will have the same value, in which case the exchange operation has no effect.

An example showing the progress of the `permute!` method as it scrambles an array of 7 items is shown in Figure 9.5. On the first iteration, `i` is 0, and the program sets `r` to a random value between 0 and 6. After assigning `r` the value 2, the program exchanges `r[0]` and `r[2]`. In the second iteration, `r` is a random value between 1 and 6, and this time the program exchanges `r[1]` and `r[5]`. Note that it is possible for an item to be moved several times. The "A" that originally started out in location 0 is going be moved again on the third iteration (unless the call to `random(2,6)` returns 2).

Figure 9.6: *A Ruby method that makes a random permutation of the items in* `x` *(which can be a string or an array).*

```
# Rearrange the items in x in a random order

1:  def permute!(x)
2:    for i in 0..x.length-2
3:      r = random(i, x.length-1)
4:      x[i], x[r] = x[r], x[i]
5:    end
6:    return x
7:  end
```

In the exercises below we will use the `trace` method to monitor the progress of a permutation and see how successive items from an array are exchanged with random items. The listing of the `permute!` method is shown in Figure 9.6. The parallel assignment is on line 4, so if we attach a probe here we will be able to see the state of the array just before the current item is exchanged with another one to its right.

Tutorial Project

The Ruby code for the `permute!` method is shown in Figure 9.6. If you would like to print a version in your IRB session you can call the `listing` method:

```
>> Source.listing("permute!")
```

You can also "check out" a copy if you want to view it in your text editor:

```
>> Source.checkout("permute!")
```

T38. Make a small array of strings:

```
>> a = TestArray.new(5, :colors)
=> ["plum", "thistle", "khaki", "chocolate", "hot pink"]
```

T39. Type a parallel assignment expression that exchanges the values in the first and third locations, and then print the array again:

```
>> a[0], a[2] = a[2], a[0]
=> ["khaki", "plum"]

>> a
=> ["khaki", "thistle", "plum", "chocolate", "hot pink"]
```

Can you see how Ruby exchanged the strings at `a[0]` and `a[2]`?

T40. Make an array of numbers to use when tracing the `permute!` method:

```
>> a = Array(0..9)
=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

T41. A method named `brackets` (similar to the one used in the searching and sorting labs) will print the contents of an array. Attach a probe to the `permute!` method so `brackets` is called just before line 4 is executed:

```
>> Source.probe( "permute!", 4, "puts brackets(x,i,r)" )
=> true
```

T42. Trace the execution of a call to `permute!` as it scrambles your test array:

```
>> trace { permute!(a) }
4: [0  1  2  3  4  5  6  7  8  9]
2:  4 [1  2  3  0  5  6  7  8  9]
6:  4  2 [1  3  0  5  6  7  8  9]
3:  4  2  6 [3  0  5  1  7  8  9]
...
=> [4, 2, 6, 3, 8, 9, 7, 1, 5, 0]
```

Each line above shows the current state of the array. The number at the front of the line, before the colon, is the value of `r`, which is the location where an item will be moved. The left bracket is printed just in front of `a[i]`, which means the item to the right of the bracket is the one that will be moved. The first line shows that the 0 in `a[0]` is going to be swapped with the item in `a[4]`. You can see the effect of this change in the array printed on the second line.

The actual result you get will be different, since the location used for the exchange is chosen at random. But you should be able to follow the steps of the algorithm by looking at the front of each line to see which location was chosen, and then noticing on the following line how the item next to the bracket was swapped with the item at the chosen location.

T43. Call `Card.new` a couple of times to make some random cards:

```
>> Card.new
=> 8♥
>> Card.new
=> 3♦
```

T44. Type this expression to make an array named `a` with 13 random cards, and then print the array after sorting it:

```
>> a = []; 13.times { a << Card.new }; a.sort
=> [K♠, J♠, J♣, 4♠, ... ]
```

There is a chance that by simply calling `Card.new` 13 times to deal a hand we will get a duplicate card. Does your array have any duplicates? Repeat the expression a few times. How often do you get duplicate cards?

T45. Make a full deck of cards containing each of the 52 cards:

```
>> d = new_deck
=> [A♠, K♠, Q♠, J♠, ... 3♣, 2♣]
```

T46. Call the `permute!` method to see if it shuffles the deck:

```
>> permute!(d)
=> [2♥, Q♠, 9♦, K♣, 4♥, 4♣, 2♠, ... ]
```

Repeat the expression a few times. Does it look like a random shuffling after each call?

T47. Shuffle the deck, and save the first five cards in an array named `h`:

```
>> h = permute!(d).first(5)
=> [9♦, 3♦, A♣, 5♠, 7♠]
```

Repeat this expression a few times. Do you get a random poker hand each time?

T48. The following expression is like the one above, except it deals a bridge hand (13 cards) and it sorts the hand before it is displayed:

```
>> h = permute!(d).first(13).sort
=> [J♠, 10♠, 9♠, 5♠, 3♠, 2♠, K♥, 10♥, 9♥, 6♥, 9♦, 10♣, 9♣]
```

Repeat the expression a few times. Since the hand is a random shuffle of a full deck you should never see any duplicates.

9.5 Tests of Randomness

So far we've been relying on our intuition that the output from our pseudorandom number generators looked random. In this section we'll perform some tests on the sequence of numbers produced by a PRNG to investigate the question of whether the sequences are random or not.

The two techniques we will use are both informal tests that use graphical displays. Visualization is a very powerful method for seeing whether there are any biases or hidden patterns in the data. These informal tests will not give us a definitive answer, of the form "yes, this sequence is random" or "no, this sequence is not random," but the tests are simple to do. If a sequence of numbers is not random, the nonrandomness often shows up in the form of patterns in the display.

The first type of display is a **histogram** (often called a "bar chart"). To test a sequence of simulated rolls of a die, we can just count the number of times each number from 1 to 6 occurs in the sequence. The histogram will have one bar for 1s, another bar for 2s, and so on, where the height of a bar indicates how often that number showed up in the random sequence (Figure 9.7).

The algorithm we have been using for making pseudorandom sequences should create what is known as a **uniform distribution**. In an experiment with 1000 rolls of the die, we expect each number will occur roughly $1000 \div 6 = 167$ times. But suppose there is a problem with the PRNG, and it gives us 300 6s. If all the other rolls occur equally often, the bar for 6s will be over twice as high as the others, and we can tell at a glance that the sequence is biased. But if all six bars are roughly the same height we could then do some further statistical tests if we wanted to know for certain whether the sequence is random.

The RAND corporation used a similar approach to test the numbers in their book of random digits. One of their tests was called the "poker test." If we want to apply this test to our random number generator, we can use the PRNG to deal 1000 poker hands. According to the laws of probability we should see 420 hands that have two cards with the same rank (*i.e.*, a pair), 21 hands with three cards of the same rank (three of a kind), and so on. If we plot the results with a bar chart, and see a lot more straights and full houses than pairs, then something is clearly wrong.

To draw a histogram on the RandomLab canvas, we first need to create a set of "bins," with one bin for each result in the experiment. For example, we need six bins for the

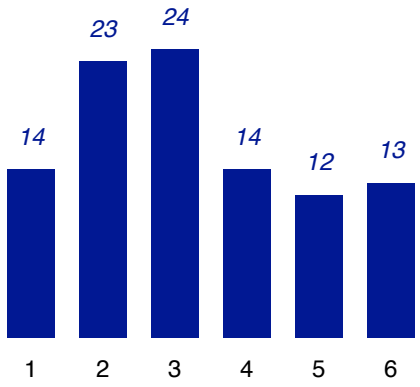

Figure 9.7: *A histogram (bar chart) showing the frequency of random rolls of a die based on the first 100 numbers from the pseudorandom sequence.*

experiment with simulated dice, one for each of the numbers from 1 to 6, so the command to initialize the histogram is

```
>> view_histogram([1,2,3,4,5,6])
=> true
```

To add one to the count in a bin, call a method named `update_bin`, passing it the ID of the bin to increment. This expression adds one to the count for bin 6:

```
>> update_bin(6)
=> true
```

When Ruby evaluates the expression you should see the bar for bin 6 grow slightly.

Another approach to test whether the sequence of values generated by a PRNG is random or not is to look for patterns or correlations between successive numbers in the sequence. It turns out there is a pattern in the sequence produced by the PRNG we used earlier in this chapter. Here is the expression we typed to make a list of 1000 numbers:

```
>> seq = prng_sequence(81, 337, 1000)
```

and these are the first 10 numbers in the sequence:

```
>> seq[0..9]
=> [0, 337, 634, 691, 308, 285, 422, 519, 376, 793]
```

From this short sample it's not clear what the pattern is, but it starts to become more apparent if we convert each of these numbers into a value between 1 and 6, which is what we would do if we're using the numbers in a game based on rolling dice. If you want to try to figure out the pattern yourself, cover up the paragraph below the IRB output shown below. Here are the first 20 numbers in the sequence, converted to values between 1 and 6. Do you see the pattern?

```
>> seq[0..19].map { |x| (x % 6) + 1 }
=> [1, 2, 5, 2, 3, 4, 3, 4, 5, 2, 1, 4, 3, 2, 3, 4, 5, 2, 3, 2]
```

Based on this short list it looks like the numbers alternate between even and odd. If the computer used this pseudorandom sequence to play a game, every time it rolled an even number it would follow with an odd number, and *vice versa*. It wouldn't take long for a person playing a game based on this PRNG to suspect something was wrong. In a game with two die, the computer would never roll "doubles."

A visual display that makes this type of pattern easy to see is known as a **dot-plot**. As the name implies, the drawing will be a set of dots on a canvas. To make a dot-plot on the RubyLabs canvas, call a method named `view_dotplot`:

```
>> view_dotplot(500)
=> true
```

The argument is the number of pixels for the height and width of the plot.

To display a dot on the canvas call `plot_point`, passing it the x and y coordinates of the dot. For example, this expression will draw a dot in the middle of the canvas:

```
>> plot_point(250,250)
=> nil
```

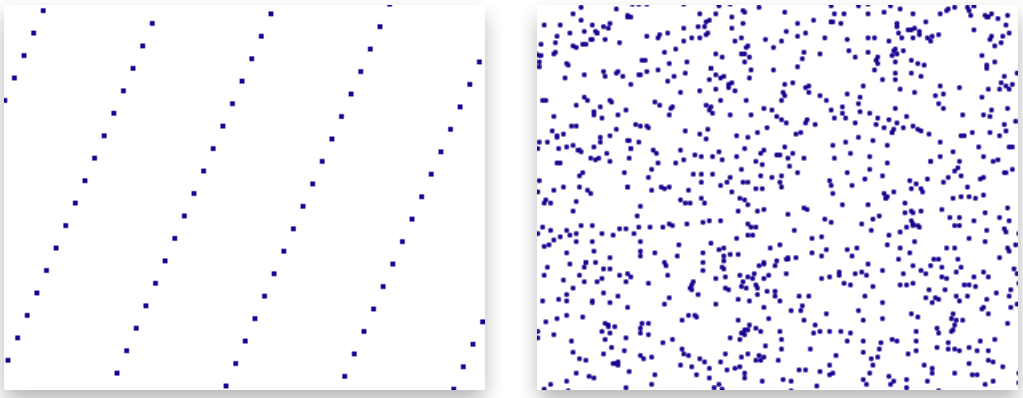

Figure 9.8: *In these images the x and y coordinates of a dot are determined by two successive numbers from a pseudorandom number generator (PRNG). The image on the left is from a PRNG that alternates between even and odd numbers. The image on the right is based on Ruby's own* `rand` *method.*

One way to look for patterns in values from a pseudorandom sequence is to plot a set of points in which the x and y coordinates of a dot are determined by getting two successive numbers from the PRNG. If we make a PRNG named `p`, this loop will paint 1000 points using `p` to set the x and y coordinates:

```
>> 1000.times {
       x = p.random(0,499);
       y = p.random(0,499);
       plot_point(x,y)
   }
```

If there is no correlation between successive numbers, the points will be scattered at random all over the canvas. But if there are any hidden patterns in the data, even if they are very subtle, the dots will line up or form some other distinctive shapes.

The dot-plot for our pseudorandom number generator with $a = 81$, $c = 337$, and $m = 1000$ is shown on the left in Figure 9.8. It's obvious from this plot that the sequence of numbers is not at all random. Even if we had not noticed that the sequence alternated between even and odd numbers, this plot would have told us there was some sort of pattern, and we would not use this set of parameters in an application that requires random numbers.

As was the case with histograms, the visualization is convenient for getting an initial impression. When there are hidden patterns in the data they often jump out when the data is displayed with a dot-plot. But to truly check to see whether the data is random it would be necessary to perform a detailed statistical analysis of the x and y coordinates. For this chapter, however, we will simply do informal tests using pictures.

Tutorial Project

The first set of projects will plot a histogram for rolls of a die produced by the pseudorandom number generator we used in previous projects.

T49. Type this expression to clear the canvas and draw an empty histogram to count the rolls of a die:

```
>> view_histogram( [1,2,3,4,5,6] )
=> true
```

You should see six lines in the middle of the screen, each representing a bin with zero items.

T50. Type this command to increment bin number 5:

```
>> update_bin(5)
=> nil
```

Do you see how the rectangle for this bin got slightly taller?

T51. Repeat the previous command a few times, using different bin numbers, until you're confident you know how this expression updates the histogram.

T52. Make a new pseudorandom number generator using the same parameters from earlier in the chapter:

```
>> p = PRNG.new(81, 337, 1000)
=> #<RandomLab::PRNG a: 81 c: 337 m: 1000>
```

T53. This command will get a random number between 1 and 6 from the PRNG and then pass it to `update_bin` to increment the count for that number:

```
>> x = p.random(1,6); update_bin(x)
=> nil
```

T54. Repeat the command several more times. Do you see how a different bin is updated each time the expression is evaluated?

T55. Clear the histogram by entering the expression from problem T49 again:

```
>> view_histogram( [1,2,3,4,5,6] )
=> true
```

T56. Put the command that draws a random number and then plots it inside a call to `times` so it is executed 100 times:

```
>> 100.times {  x = p.random(1,6); update_bin(x) }
=> 100
```

What does your histogram look like? Are the numbers fairly evenly distributed?

If any of the bins reaches the top of the canvas the drawing methods will automatically rescale the histogram to make it smaller. So if you see the histogram suddenly shrink, don't worry. It just means the drawing methods are "zooming out" and making room for more data.

T57. In an earlier exercise we saw how this PRNG generates every number between 0 and 999. That means there should be the same number of 1s, 2s, 3s, *etc.*, when those numbers are converted to values between 1 and 6. To verify this claim, repeat the previous loop 900 more times, so a total of 1000 numbers are in the histogram:

```
>> 900.times {  x = p.random(1,6); update_bin(x) }
=> 900
```

What did you see? Are the results what you expected?

The next set of exercises make another histogram, this time using poker hands.

T58. The classification of a poker hand is often called its "ranking." To get an array of names of poker hands, call a method named `poker_rankings`:

```
>> poker_rankings
=> [:high_card, :pair, ... :four_of_a_kind, :straight_flush]
```

Note that the rankings are ordered, with the most common hands on the left.

T59. Pass the array of names to the method that creates a new histogram:

```
>> view_histogram(poker_rankings)
=> nil
```

Since there are nine names in the list, you should see nine empty bins.

T60. Make a new deck of cards:

```
>> d = new_deck
=> [A♠, K♠, Q♠, ... 4♣, 3♣, 2♣]
```

T61. This expression will shuffle the deck and copy the first five cards to an array named `h`:

```
>> h = permute!(d).first(5)
=> [Q♥, K♦, 5♠, J♥, K♣]
```

It's OK to cheat if you want to repeat that expression until you get an interesting hand.

T62. A method named `poker_rank` will figure out what sort of hand is in the array:

```
>> poker_rank(h)
=> :pair
```

T63. Since the result from `poker_rank` is one of the histogram labels, we can increment the count for that type of hand by passing the label to `update_bin`:

```
>> update_bin(poker_rank(h))
=> true
```

T64. The next command combines all the operations into a single expression: it shuffles the deck, makes a hand from the first five cards, figures out what sort of hand it is, and updates the corresponding bin in the histogram:

```
>> h = permute!(d).first(5); update_bin(poker_rank(h))
=> nil
```

T65. Repeat that command several times. Are the bins updating? Are they mostly on the left, for the most common hands?

T66. If you want to repeat the experiment for a larger number of hands, put the expression in the body of a loop:

```
>> 1000.times {h = permute!(d).first(5); update_bin(poker_rank(h))}
=> 1000
```

♦ The histogram methods keep track of the number of each type of item. If you want to see the final counts, call a method named `get_counts`:

```
>> get_counts
=> {:straight_flush=>0, :high_card=>524, :two_pair=>41,
   :three_of_a_kind=>29, :full_house=>1, :straight=>5,
   :four_of_a_kind=>1, :flush=>4, :pair=>395}
```

This output shows there were no straight flushes, 524 hands classified as "high card," 41 with two pair, and so on. Find a table of poker probabilities on the Internet, and compare your results with the expected frequency for each type of hand. Do you think there is any bias in these hands, or does it seem like `permute!` did a good job of shuffling the deck?

9.6 Summary

Making lists of random values is a subtle problem. It's hard for humans to do, and it turns out it's not quite so easy for machines, either.

In this chapter we saw how to make a long list of numbers where short stretches appear to be random. Given the right combination of parameters, the equation that defines how to add a new value to the list as a function of the previous value will make sure every number between 0 and $n - 1$ eventually appears in a list of n numbers.

But even though the list has all n numbers, and they appear to be in a random order, there can be some hidden patterns. For example, when we took a closer look at one of the sequences we studied, we saw that it alternated between even and odd numbers. Even more subtle patterns might be hidden in the data. We explored two methods for testing randomness by drawing pictures based on values produced by the random number generator. These informal methods can help us tell, at a glance, whether there is a hidden bias or pattern.

Random number generators are among the most important and widely used algorithms in computer science. They are used in games, of course, but there are many other application areas where random values play a key role. We used random sequences in Chapters 4 and 5 to test searching and sorting algorithms. Many scientific algorithms also use random numbers. For example, some algorithms that reconstruct the evolutionary history of a set of genes use a method based on taking a random sample of all possible family trees to find the one that is most likely given the similarities between the genes. Later in this book, Chapter 12 describes a type of problem known as optimization, where the goal is to find the optimal solution to a problem, and one common approach also uses random samples. E-commerce, Internet banking, and other network traffic depends on encryption algorithms to turn a piece of text into what appears to be a random sequence of letters; many of the important concepts used to design effective random number generators are also used in encryption algorithms.

Concepts and Terminology Introduced in This Chapter

random	A sequence of items is random if the values are independent of one another
pseudorandom	A pseudorandom sequence is one produced by an algorithm; pseudorandom sequences are not truly random, but small subsequences may appear to be random
PRNG	Pseudorandom number generator, an algorithm that creates a sequence of pseudorandom values
permutation	A reordering of the items in a list or array
uniform distribution	The result of generating random values where each value is equally likely
histogram	Also known as a bar chart; a visual display that shows the number of times various events occur

Exercises

1. The schedule in Section 9.1 for events that occur every seven hours had a period of 12, *i.e.*, every number between 0 and 11 appears once in the schedule before it repeats. Can you find other intervals besides 7 hours that also lead to a period of 12?
2. Suppose the schedule for events that occur every seven hours starts on a Monday at 12 A.M., so the next events are Monday at 7 A.M., 2 P.M., and 9 P.M. Will this pattern ever repeat? That is, will there ever be another Monday with this same schedule? If so, can you determine how many days will pass before this same schedule is used?
3. Redo the schedule for events that occur every seven hours using a 24-hour clock. Is the period still 12? Or is it now 24?
4. What are some intervals that lead to schedules with a period of less than 24 when a 24-hour clock is used?
5. What are some intervals that lead to a full schedule with 24 times and every time between 0 and 23 occurring exactly once?
6. What do the answers to the last problem have in common? Is there a common attribute for intervals that lead to schedules with a period of 24, *vs.* those the lead to a period of less than 24?
7. The first day of each month usually falls on a different day of the week. Is the pattern of weekdays random? Pseudorandom?
8. Use the Ruby expressions starting on page 241 to create a dot-plot on the RubyLabs canvas. Then make a plot of 3000 points drawn at random from Ruby's own random number generator (the built-in method named `rand`) instead of numbers from a pseudorandom sequence. Do the points appear to be spread uniformly all over the canvas?
9. The Ruby expression given in Exercise T14 used a call to `uniq` to see if all of the numbers between 0 and `m-1` were in a pseudorandom sequence. Another way to check the sequence would be to sort it so you can compare it to an array that has every number from 0 to `m-1`. Can you write a Ruby expression that will do this test?
10. If you took the challenge at the beginning of Section 9.1 and entered a series of random digits in a file, make a histogram from your numbers. One way to make a histogram is to load the numbers into a spreadsheet application and use its "chart" command. Were you able to make a uniform distribution? Or are some numbers a lot more frequent than others?
11. ♦ Here are some values of a, c, and m that should generate much better pseudorandom sequences than the ones we used in this chapter:[2]

a	c	m
1255	6173	29282
171	11213	53125
421	54773	259200

    Repeat some of the experiments in this chapter to evaluate the quality of the pseudorandom sequences. Make PRNG objects, and use values from the objects to make histograms (not just of values from 1 to 6, but for other ranges, too) and dot plots.

[2] From *Numerical Recipes in C*, by W. H. Press, *et al.*