Shuffling in Java, Part 1: Lotteries, Random Numbers, and Pseudo-random Numbers

Nicholas Bennett
nick@nickbenn.com

January 2019

Last modified: 30 January 2019.

Preface

Objective

This document is intended primarily as curricular material in the Deep Dive Coding Java + Android Bootcamp [1]. While this document and the accompanying presentation aim to introduce some important programming and algorithmic concepts through their application to some well-known problems, they're only an introduction: this is neither a broad programming tutorial nor an in-depth lesson on algorithms.

Audience

This lesson should not be a student's first hands-on experience with programming, algorithms, or pseudocode. Minimally, students should be comfortable writing assignment, conditional, and iteration statements, as well as simple classes and methods in Java. Previous completion of first-year algebra is assumed, and previous completion of—or current enrollment in—second-year algebra (including elementary analytic geometry) is highly recommended.

Implementation Language

The concepts addressed in this document aren't tied to any single programming language, or family of languages. However, the implementations shown are written in Java [2].

The Java code included in and accompanying this document was tested successfully with several JDK versions from 1.8 update 191 to 11.0.2. However, the code doesn't depend on any language features added to the language after version 1.6; thus, it should compile and run with all JDK 1.6, 1.7, 1.8, 9, 10, and 11 releases.

This page intentionally left blank.

Shuffling: The flip side of sorting

Introduction

The shuffling problem is easily stated: How can we rearrange a list of items so that the order is random and fair?

Fortunately, this isn't a hard problem to solve—but it's also easy to get it wrong [3],[4]. There are 2 widely used, effective approaches to shuffling in computer programs: sorting on a random number, and the Fisher-Yates shuffle.[5],[6]

Shuffling algorithms

Sorting on a random number

1. Assign a randomly generated value to each item to be shuffled.
2. Sort the items, in order of the assigned random numbers.
3. Stop. The list is now shuffled.

Though the need for sorting makes this method less efficient than the one that follows, it's very easy to implement in some languages and environments—for example, this can be a good approach for retrieving and presenting database records in a random order.

Durstenfeld's version of the Fisher-Yates shuffle (aka Knuth shuffle)

1. Begin with the sequence of items X, containing the N items $x_0, x_1, \ldots, x_{N-1}$.
2. For each integer value i, starting at $N-1$ and counting down to 1, do the following:
    a. Generate a random value j, which can be any one of $\{0, 1, \ldots, i\}$; each of the values must be equally likely to be selected.
    b. Exchange the positions of x_i and x_j in the sequence. Note that it's possible that $i = j$; obviously, no exchange is needed when that's the case.
3. Stop. The items in the sequence are now shuffled.

As we repeat steps 2a–b (i.e. *iterate*) with values of i from $N-1$ to 1, each randomly selected item is shuffled by exchanging it with the (initially) unshuffled item at index position i. In effect, the list is divided into two parts, one unshuffled and one shuffled, with the latter growing as the former shrinks, until no unshuffled items remain.

Exercise 1: Shuffling a list of 6 items by rolls of a die

Using a six-sided die to generate random numbers, let's shuffle the words “apple”, “banana”, “chile”, “donut”, “egg”, and “flan” into a random order. To help us keep track of the items as we shuffle them, we'll use tables 1-3. Also, since dice are usually numbered starting at 1, we’ll count down from 6 to 2 (instead of 5 to 1) in this case.

			Items (N = Number of Items = 6)					
			Position					
Iteration	***i***	**Roll (*j*)**	**1**	**2**	**3**	**4**	**5**	**6**
Initial			apple	banana	chile	donut	egg	flan
1	6							
2	5							
3	4							
4	3							
5	2							

Table 1: Example of Fisher-Yates shuffle, initial state

We start at row 1—i.e. the row where 1 appears in the **Iteration** column—and roll the die to get a value of j. For example, assume we roll a 4. We write this value in the **Roll (*j*)** column. Next, we copy the six words in our list from the previous row to the current row. As we do so, we exchange the item in column 4 (since that was our roll) with the item in column i, (column 6, in this case); thus we exchange “donut” with “flan”; “donut” is now in its shuffled position. The result appears in table 2.

			Items (N = Number of Items = 6)					
			Position					
Iteration	***i***	**Roll (*j*)**	**1**	**2**	**3**	**4**	**5**	**6**
Initial			apple	banana	chile	donut	egg	flan
1	6	4	apple	banana	chile	flan	egg	donut
2	5							
3	4							
4	3							
5	2							

Table 2: Example of Fisher-Yates shuffle, after 1 iteration

In each successive *iteration* (a repeated set of steps in an algorithm), we move down one row (decreasing the value of i by 1) and roll the die. If the roll is greater than the value in the ***i*** column, we keep rolling until we get a value less than or equal to i; then, we write our roll in the **Roll (*j*)** column. Finally, we copy the items from the previous row to the current row, exchanging the items in columns i and j.

Assume that in iteration 2, we roll a 6. However, because 6 is greater than the current value of i, we roll again; this time, we get a 3, so we write that value in the **Roll (*j*)** column. Then, we copy our six items from the previous row to the current row, exchanging the item in column j (or 3), which is "chile", with the item in column i (or 5), which is "egg". Our randomly selected item, "chile", is now shuffled.

			Items (*N* = Number of Items = 6)					
			Position					
Iteration	***i***	**Roll (*j*)**	**1**	**2**	**3**	**4**	**5**	**6**
Initial			apple	banana	chile	donut	egg	flan
1	6	4	apple	banana	chile	flan	egg	donut
2	5	3	apple	banana	egg	flan	chile	donut
3	4							
4	3							
5	2							

Table 3: Example of Fisher-Yates shuffle, after 2 iterations

Complete this shuffle on your own, by performing iterations 3-5.

Exercise 2: Additional questions and tasks

I. What is a fair shuffle? Does random always imply fair?

II. If each iteration of steps 2a–b of the Fisher-Yates algorithm shuffles one item in the list, how is it that we're able to shuffle 6 items in 5 iterations? More generally, how are we able to shuffle all N items in $N-1$ iterations?

III. In any given iteration of steps 2a–b of the Fisher-Yates shuffle, it's possible that item i will trade places with itself. Would eliminating this possibility, by limiting our random selection to the range from 0 to $i-1$, still give us a complete, fair shuffle?

IV. Is there any value in performing a Fisher-Yates shuffle multiple times in a row, similar to the way we would manually shuffle a deck of cards several times?

This page intentionally left blank.

Lotteries

Introduction

Lotteries have existed—legally and illegally, and with many variations—for over 2000 years. In the U.S.A., lotteries were illegal for most of the 20th century, but by the 1960s, some states were modifying their constitutions to allow state lotteries. 43 states now run (or participate in) lotteries, using them as a source of state revenues. All of these lotteries set the payoffs so that the total amount paid to the winners is always much less than the total amount paid in, giving the state an unbeatable advantage.

In most of the state lotteries, the drawing of numbers is done with actual balls in a container. However, some lotteries are conducted electronically; also, state lotteries usually offer some form of "quick pick" option, letting players buy lottery tickets with the numbers selected at random by computer. So we'll write a Java program that generates our own lottery draws or quick picks.

The next three exercises don't assume anything about your development environment; they can be completed with virtually any Java IDE, or just by using a text editor and the command-line tools for compiling and running Java programs.

Exercise 3: Writing a lottery class in Java

One of the first things we need to do, when writing any Java program, is decide what *classes* we need. In Java, classes are units of Java code, which generally perform one or more of three primary roles:

- A set of related *methods* (named procedures that perform specific tasks) that operate on variables of the basic Java data types, or object types defined by other classes.
- A set of methods to manage the execution (and termination) of Java programs, applets, etc.
- An encapsulation of the attributes and behaviors of a type of object, often corresponding to a physical or logical object related to the problem we're working on. For example, a traffic simulation program might have **`Vehicle`** and **`TrafficSignal`** classes that define variables and methods for managing the speed, location, and direction of vehicles, and the states of traffic signals.

In this case, we need a class that encapsulates the data for a lottery—i.e. the set of numbers that can be selected—with one very important behavior:

- Select a subset of the numbers at random, *without replacement*—i.e. without putting each number back after it's selected. In most lotteries, the order in which the numbers are selected is irrelevant—in fact, they're generally sorted in ascending order before they're announced. So our class will do the same.

We'll begin by creating a **`Lottery`** class, with an *array* of integers to hold the lottery numbers, and a *constructor* that will create and fill the array.

Create a new class file, named `Lottery.java`, in the `edu/cnm/deepdive` source directory (i.e. the **`edu.cnm.deepdive`** package), and write the following code (the line numbers are for reference only; don't write them in your code):

```
 1 package edu.cnm.deepdive;
 2
 3 public class Lottery {
 4
 5   private int[] numbers;
 6
 7   public Lottery(int maximum) {
 8     numbers = new int[maximum];
 9     for (int i = 0; i < numbers.length; i++) {
10       numbers[i] = i + 1;
11     }
12   }
13
14 }
```

Let's review the most important elements of the code:

- Our **`class`** is in the **`edu.cnm.deepdive`** package (line 1), is named **`Lottery`**, and has **`public`** visibility (line 3).

- Our class includes the **`private`** field **`numbers`** (line 5), which is a reference to an **`int[]`**, or an *array* of integers.

- A **`public`** *constructor* begins on line 7. A constructor is a class member whose job is to initialize the data used by an instance of the class. A constructor always has the same name as the class, and no return type is specified.

- Inside the parentheses that follow the name of our constructor, we see that when this constructor is called, some additional information must be provided: an **`int`**, which the constructor refers to as **`maximum`**. Later, when we're using our **`Lottery`** class type, we'll need to provide this value as an *argument* (information passed inside parentheses when invoking a method or constructor) when we call the constructor.

- Inside the curly braces of the constructor (lines 7 and 12), we have the body of the constructor—i.e. the code that initializes our lottery numbers and our random number generator. In this body, our code does the following:

  - First we use the **`new`** keyword to allocate space for the **`numbers`** array, with enough elements for the range of numbers from 1 to **`maximum`** (line 11).

  - Next, we need to put our lottery numbers into the array. We use the **`for`** statement (starting in line 12) to iterate over the elements in the array, using the **`int`** variable **`i`** as an iteration counter, with a starting value of 0, and continuing as long as **`i`** is less than **`maximum`**, After each iteration, we increment the value of **`i`** (using **`i++`**). In each iteration, we execute the code between the curly braces on lines 12 and 14; in that code, we place into element **`i`** of the array a value equal to one more than **`i`**; for example, we put the number 1 into element 0, 2 into element 1, etc.—all the way up to the value of **`maximum`**, which we put into element **`maximum - 1`**.

Before we go any further, make sure you've saved your work. Since the class is named **`Lottery`**, the file name must be `Lottery.java`.—i.e. with the exact same spelling and capitalization as the class name, and with the `.java` extension.

Next, compile the code you've written—making sure to fix any errors that are reported along the way, until your code compiles without error.

So far, we've taken care of the first part of the required functionality. Now, we'll write code to select a subset of the numbers at random. We'll use shuffling to mix up the numbers, but we're going to change the algorithm just a bit. Remember that in order to shuffle N items, we perform $N-1$ iterations of selecting an unshuffled item at random and swapping it with the last unshuffled item. But in this case, we don't really need to shuffle all of the numbers; we only need to shuffle enough for our subset.

In writing the code that follows, the code you already wrote is grayed-out. *Don't re-write the grayed-out code; simply add the new code to it.*

```
 1 package edu.cnm.deepdive;
 2
 3 import java.util.Arrays;
 4 import java.util.Random;
 5
 6 public class Lottery {
 7
 8   private int[] numbers;
 9
10   public Lottery(int maximum) {
11     numbers = new int[maximum];
12     for (int i = 0; i < numbers.length; i++) {
13       numbers[i] = i + 1;
14     }
15   }
16
17   private void mix(int iterations, Random rng) {
18     for (int i = 0; i < iterations; i++) {
19       int destination = numbers.length - i - 1;
20       int source = rng.nextInt(destination + 1);
21       int temp = numbers[source];
22       numbers[source] = numbers[destination];
23       numbers[destination] = temp;
24     }
25   }
26
27   public int[] pick(int count, Random rng) {
28     int[] selection;
29     mix(count, rng);
30     selection = Arrays.copyOfRange(
31         numbers, numbers.length - count, numbers.length);
32     Arrays.sort(selection);
33     return selection;
34   }
35
36 }
```

Let's review the additions, starting at the top:

- We added `import` statements (lines 3–4) that tell the Java compiler that our code will use the `Arrays` and `Random` classes from the `java.util` package.

- Lines 17–25 contain the declaration and body of the `private` (not visible outside the class) method `mix`.

    - When this method is called, it expects to receive an **`int`** (referred to by the method as **`iterations`**) and an instance of **`Random`** (referred to as **`rng`**). **`iterations`** is the number of iterations of steps 2a–b in the Fisher-Yates shuffle that this method will execute; since each iteration shuffles one element in the **`numbers`** array, this is also the number of lottery numbers shuffled by this method. **`rng`** is the source of randomness (pseudo-randomness, actually) used for shuffling.

    - We use the **`for`** statement (starting in line 18), with the counter variable **`i`**, to repeat the code between the curly braces on lines 18 and 24 a total of **`iterations`** times.

    - In lines 19–20, *local variables* (variables existing only within the current block of statements) named **`destination`** and **`source`** are declared: the first is used to refer to the **`i`**th-to-last element of the array; the second is given a random value value (generated by **`rng`**) between 0 (inclusive) and **`destination + 1`** (exclusive).

    - Now, we use another local variable, **`temp`**, to help us exchange the values in 2 elements of the **`numbers`** array: First, the value of the lottery number in the **`source`** element of **`numbers`** is assigned to **`temp`**. Then, we take the lottery number in the **`destination`** element of **`numbers`** and put it in the **`source`** element. Finally, we take the lottery number we stored in **`temp`**, and put it into the **`destination`** element of the **`numbers`** array. This has the effect of swapping a randomly selected unshuffled item with the last unshuffled item in the **`numbers`** array; this randomly selected item is now in its shuffled position.

- We've declared the **`public`** method **`pick`** (lines 27–34), which returns an **`int[]`**, or array of integers.

    - When this method is called, 2 additional pieces of data must be provided: an **`int`**, which this method refers to as **`count`**, and an instance of **`Random`** (referred to as **`rng`**), used as a source of randomness.

    - In the body of the **`pick`** method, we declare a local variable named **`selection`** (line 28). Like the **`numbers`** variable, **`selection`** refers to an array of integers.

- Next, we call the `mix` method, passing `count` and `rng` as arguments (line 29). The `mix` method shuffles `count` lottery numbers to the end of the `numbers` array, using `rng` as the source of randomness.

- After `mix` does the shuffling, we copy the shuffled numbers from the `numbers` array to the `selection` array, using the `copyOfRange` method in the `Arrays` class (lines 30–31). In this call, we specify the array we're copying data from (`numbers`), and the range of elements to copy.

- We use another method of the `Arrays` class, `sort`, to sort the elements of the `selection` array in ascending order (line 32).

- Finally, we return the `selection` array as the result of this method (line 33).

Save and compile the `Lottery` class. If any error messages appear, fix the reported problems, then save and compile again, until you can compile without any errors.

Exercise 4: Testing the `Lottery` class interactively

Our `Lottery` class is complete for our purposes today, but it isn't a Java program; it doesn’t have the required `main` method. Fortunately, tools such as JShell, Eclipse scrapbook pages, IntelliJ IDEA scratch files, or the DrJava interactions pane can be used to explore a class like this interactively, without having to create a running Java program. Use of these tools is beyond the scope of this document. (In any event, the reader should already be familiar with one or more of them.) In general terms, any of the above tools will allow you to do the following:

- Create an instance of `java.util.Random` (or of a subclass of `Random`, such as `java.security.SecureRandom`).

- Create an instance of `Lottery`, passing a maximum number for the lottery selection and the `Random` instance as constructor arguments.

- Invoke the `pick` method of the Lottery instance, specifying the number of values to be picked.

Exercise 5: Writing a Java program that uses the `Lottery` class

For this exercise, we'll write a program to generate several picks for the NM Lottery Roadrunner Cash game, in which the player picks 5 separate numbers from 1 to 37 [7].

For a Java program, we start (again) with a class. Create a new class, **`Roadrunner`**, in the same package as before (**`edu.cnm.deepdive`**), and type this code:

```
 1 package edu.cnm.deepdive;
 2 
 3 public class Roadrunner {
 4 
 5   private static final int MAXIMUM_NUMBER = 37;
 6   private static final int NUMBER_TO_PICK = 5;
 7   private static final int NUMBER_OF_TICKETS = 10;
 8   private static final String PICK_FORMAT =
 9       "Ticket #%d: %s%n";
10 
11   public static void main(String[] args) {
12 
13   }
14 
15 }
```

Let's take a look at the code in more detail:

- As before, we declare our class with the **`class`** keyword, the name of the class, and a set of curly braces (lines 3 and 16) to hold the implementation of the class. We also set the visibility of the class to **`public`** (line 3), so that it can be seen from other classes—and most importantly, by the Java launcher.

- Just inside the class (lines 5–9), we have 4 fields with an important combination of keywords.

  - A variable that's declared **`static final`** is actually a *constant*: once we assign an initial value to a constant, the Java compiler won't let a different value be assigned. We've assigned initial values to all of these constants, so we can be certain that those values won't change.

  - In these constants, we've stored the upper limit of the range of numbers for this lottery (line 5), how many numbers must be selected in each pick (line 6), and how many separate sets of numbers we'll pick (line 7), and the format string that we'll use when printing out each pick (lines 8–9).

- Starting in line 11, we see the application *entry point*. The most important thing to remember about this is that when the Java launcher tries to run a class as a Java application, it looks in that class for a **`public static`** method called **`main`**, which doesn't return any data when it's done (that what **`void`** return type means). Finally, inside the parentheses of the **`main`** method declaration (line 11), we see that the method expects to receive an array of **`String`** objects; this is also a requirement for the **`main`** method of a Java program. (The arguments are values that can be specified on the command line when a Java program is launched—but in this case, we're not going to use any such additional information passed in this way.)

If the Java launcher finds a method with the **`public static void main(String[])`** signature, it calls it; if it doesn't find it, it reports an error and terminates execution.

- As with our earlier methods, we see that the **`main`** method has a set of curly braces (lines 11 and 13); we'll write the body of the method—i.e. the top-level steps of our Roadrunner program—between these braces.

Save the file. Be sure to save it as **`Roadrunner.java`**, in the same directory (corresponding to the **`edu.cnm.deepdive`** package) as your `Lottery.java` file. (Remember that the file name must match the class name in spelling and capitalization, with the addition of the `.java` file extension.)

Compile the file. If you get error messages, read them carefully to try to figure out what's wrong, and try to fix the problems. Save and compile the file again after making any such changes.

Some of the last few lines of code we'll add should look familiar to you, if you already exercised the **`Lottery`** class interactively. Remember to type only the new code (shown in bold blue type), and not the code we wrote before (shown in gray).

```
 1 package edu.cnm.deepdive;
 2 
 3 import java.util.Arrays;
 4 import java.util.Random;
 5 import java.security.SecureRandom;
 6 
 7 public class Roadrunner {
 8 
 9   private static final int MAXIMUM_NUMBER = 37;
10   private static final int NUMBER_TO_PICK = 5;
11   private static final int NUMBER_OF_TICKETS = 10;
12   private static final String PICK_FORMAT =
13       "Ticket #%d: %s%n";
14 
15   public static void main(String[] args) {
16     Lottery lotto = new Lottery(MAXIMUM_NUMBER);
17     Random rng = new SecureRandom();
18     for (int i = 0; i < NUMBER_OF_TICKETS; i++) {
19       int[] pick = lotto.pick(NUMBER_TO_PICK, rng);
20       String prettyPick = Arrays.toString(pick);
21       System.out.printf(PICK_FORMAT, i + 1, prettyPick);
22     }
23   }
24 
25 }
```

There are a few things to notice this time:

- This time, we're importing the `java.util.Arrays`, `java.util.Random`, and `java.security.SecureRandom` classes (lines 3–5). As you probably recall, the first is a class that includes some useful methods for working with arrays, and the second is a class that implements a pseudo-random number generator (PRNG). The third class imported (`SecureRandom`) is actually a subclass of the second (`Random`), and implements a much higher-quality PRNG.

- We're using the `for` statement to iterate (starting in line 18, with curly braces in lines 18 and 22). This time, we're using it to count up to the desired number of "tickets"—i.e. the separate lottery picks the program will draw for us.

- For each ticket, numbers are drawn using the `pick` method (line 19), and the result is assigned to the local variable `pick`.

- In line 20, we use the `Arrays.toString` method to convert the `int[]` with the selected subset of numbers to a `String`; we then assign this to the `prettyPick` local variable.

- We use `System.out.printf` (line 21), to print the number of the current ticket, along with the numbers picked, in a formatted string. With this method, we first specify a format string (in this case, the format string is the `PICK_FORMAT` constant), which may contain static text along with one or more placeholders (you can spot these placeholders easily: they start with the percent sign); then, we pass the data to be displayed, as additional parameters in the method call. Before the text is written out, the placeholders in the format string are replaced by the data values.

  In this case, the format string is: "`Ticket #%d: %s%n`". The placeholders are `%d` (for an integer value displayed as a decimal number), `%s` (for a string value), and %n (for a platform-specific new line). The first placeholder is replaced by the value of `i + 1`, and the second placeholder is replaced by the value of `prettyPick`.

Save, compile, and run your program. Is the result what you expected to see?

Exercise 6: Additional questions and tasks

I. In all, how many different Roadrunner picks are possible? In other words, how many *combinations* (distinct subsets, without regard to the order of items in a subset) of 5 numbers from the set {1, 2, …, 37} are possible?

II. Can you think of a way to modify your `Roadrunner` class, so that you could use it to check whether the method we're using generates all possible combinations, in approximately equal proportions?

III. Create a new `Keno` class with a `main` method, that can be executed as a Java program, so that 20 numbers in the range from 1 to 80 are drawn (using the `Lottery` class), for an automated keno game.

IV. Create a new Java class with a `main` method that generates quick pick numbers for PowerBall, where 5 numbers from 1 to 59 are selected without replacement, and then a 6th number from 1 to 39 is selected.

Random and Pseudo-random Numbers

The need for randomness

In previous exercises, we used 2 of Java's random number generator classes (**`Random`** and **`SecureRandom`**) to generate random values that we used to shuffle lottery numbers. Many different kinds of computer programs benefit from random numbers; when we're working on computational science projects, and building scientific simulations, the need for random numbers becomes even more critical.

But there's a wrinkle: contrary to what many might think, very little that happens in most computers—at least in an easily observed fashion—is truly random. Fortunately for us, we often don't need true randomness: numbers that appear random, even though they're not, are good enough in many cases. Numbers of this type are called *pseudo-random numbers*.

The appearance of randomness

What can we observe in this portion of a sequence of numbers?

$$S = 0,\ 5,\ 6,\ 3,\ 12,\ 1,\ 2,\ 15,\ 8,\ 13,\ 14,\ 11,\ 4, 9,\ 10,\ 7, \ldots$$

At first, we might see some apparently random numbers, all less than 16. We might notice that none of the numbers is repeated—but we could chalk that up to having such a short portion of the sequence (or, we might speculate that the numbers are being selected by sampling without replacement). Looking closer, we might suspect some patterns in the differences between successive numbers; maybe that's just a coincidence. Finally, we might notice that the sequence alternates between odd and even values; that seems very unlikely to be a coincidence.

Let's look at more of the sequence:

$$S = 0,5,6,3,12,1,2,15,8,13,14,11,4,9,10,7,0,5,6,3,12,1,2,15,8,13,14,11,4,9,10,7,\ldots$$

Now we see a complete duplication: the sequence seems to restart with the 17th number and repeat identically. Certainly, the sequence is looking less and less likely to be random. But for some very simple purposes (certainly not involving scientific simulations), it might appear sufficiently unpredictable (at least, to the casual observer, who's probably less picky about these things than we are) to be of use.

In fact, this sequence is an example of a *pseudo-random number sequence*—a sequence of numbers generated by mathematical formulæ (called a *pseudo-random number generator*, or *PRNG*), where each number in the sequence is computed from one or more previous numbers in the sequence, and where the overall result is intended to appear random.

The PRNG that generated the sequence is quite simple:

$$\begin{aligned} s_0 &= 0 \\ s_{i+1} &= (13 s_i + 5) \bmod 16 \end{aligned}$$

The sequence has a *seed* (initial value—note that we usually call the initial value in a sequence s_0, instead of s_1) of 0, and each term in the sequence is generated by multiplying the previous term by 13, then adding 5, and finally taking the remainder after dividing by 16.

We can keep computing the terms of this sequence indefinitely, and the pattern keeps repeating, as shown in Figure 1. The number of iterations required before the sequence repeats is the *period* of a PRNG; here, the period is 16.

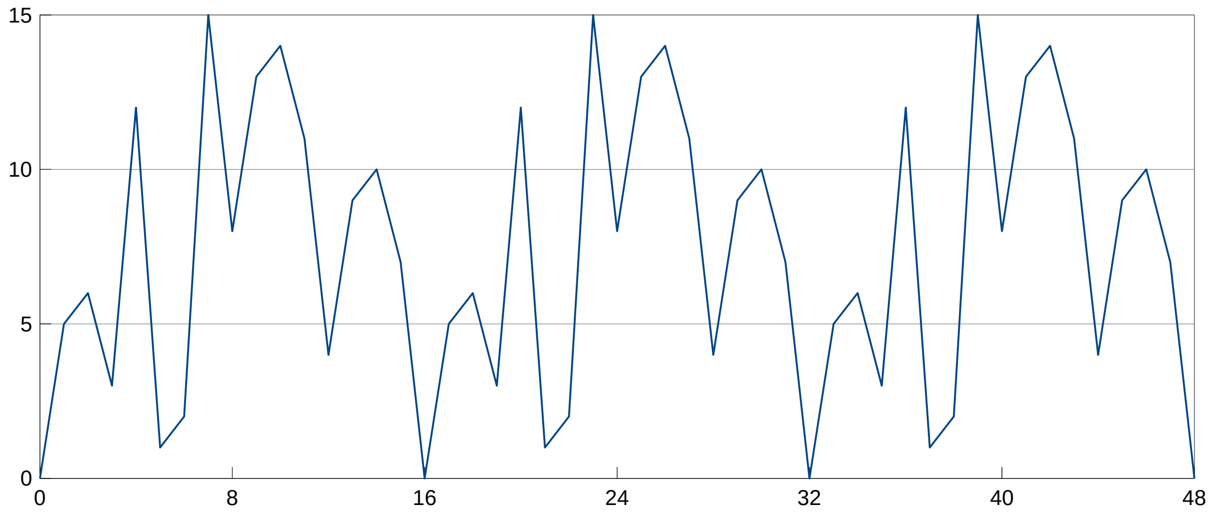

Figure 1: LCG with $s_{i+1} = (13s_i + 5) \bmod 16$, $s_0 = 0$

We can also use a different value for s_0 (i.e. a different seed); we'll get the same repeating sequence, but starting at a different point. For example, with $s_0 = 2$, we have:

$$S = 2, 15, 8, 13, 14, 11, 4, 9, 10, 7, 0, 5, 6, 3, 12, 1, 2, 15, 8, 13, 14, 11, 4, 9, 10, 7, 0, 5, 6, 3, 12, 1, \ldots$$

We might conclude that this isn't a good way to generate pseudo-random numbers, for the simple reason that the results aren't unpredictable enough. But what if our sequence formula used values other than 13, 5, and 16? It turns out that with careful selection of these values, we can get much better results.

Linear congruential generators

The PRNG we just looked at is an example of a *linear congruential generator*, or *LCG*. An LCG generates a sequence of pseudo-random numbers with the equation:

$$s_{i+1} = (a\,s_i + c) \bmod m$$

LCGs are included in the standard libraries of many different programming languages. In most cases, only a portion of each s_i is returned as the pseudo-random value. For example, the standard Java library generates pseudo-random 32-bit integers as follows (note that S is the sequence maintained internally; X is the sequence of values returned) [8]:

$$s_{i+1} = (25214903917\,s_i + 11) \bmod 2^{48}$$
$$x_i = \left\lfloor \frac{s_i}{2^{16}} \right\rfloor$$

With the division by 2^{16}, the most obviously non-random aspects of the sequence (e.g. the alternating odd/even pattern, characteristic of LCGs with even moduli) are minimized. However, there are other issues with LCGs, primarily having to do with the correlation between successive LCG values. For example, in Figure 2 we see what happens when we group the terms of our original sequence in pairs, and treat them as a sequence of points in two dimensions:

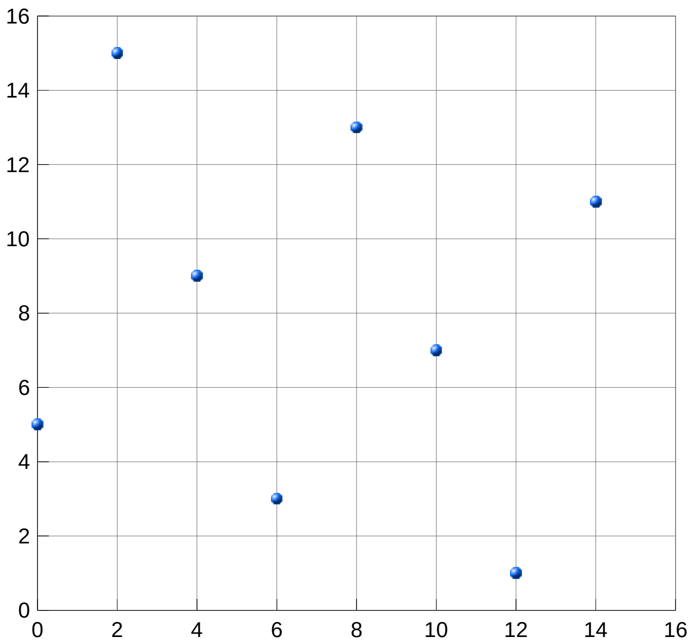

Figure 2: Values from simple LCG plotted in 2 dimensions.

$$\boldsymbol{S} = (0,5),\ (6,3),\ (12,1),\ (2,15),\ (8,13),\ (14,11),\ (4,9),\ (10,7),\ \ldots$$

Not only are the points very regularly spaced, but when we view the graph as a torus (an LCG "wraps around" at the modulus value—in this case, 16), we see that we can draw a single line from (15, 0), through the point at (12, 1), and continuing (wrapping around as appropriate) through all of the other points!

With a well chosen LCG, with a long period, this effect isn't nearly as visible (especially with a small number of dimensions), but it's still present.

Exercise 7: Another LCG

Given the LCG:

$$s_0 = \text{day of the month you were born} \in \{1, 2, \ldots, 31\}$$
$$s_{i+1} = (5 s_i + 13) \bmod 32$$

Compute the terms $s_1, , s_2, \ldots, s_{10}$.

Remember that $a \bmod b$ is the remainder when a is divided by b. For example:

$$11 \bmod 4 = 3$$
$$37 \bmod 5 = 2$$
$$89 \bmod 32 = 25$$

Based on your calculations, and on the parameters of this LCG:

- How random does the result appear?
- What patterns do you see?
- How many distinct values of s_i are possible?
- What is the period of the sequence S?

The Mersenne Twister and other PRNGs

The *Mersenne Twister* is a PRNG that uses a matrix linear recurrence to generate its pseudo-random sequence [9]. It's named for the fact that key parameters of the generating function form the exponent of a *Mersenne prime* (a prime number of the form $2^p - 1$).

Over the last decade, the Mersenne Twister has become a popular alternative to the LCGs provided in many standard libraries. It's now included as the standard PRNG in Python, Ruby, R, MATLAB, and Maple. Its key advantages are a very long period (equal to the Mersenne prime $2^{19937}-1$) and a lack of significant serial correlation.

There are many other PRNGs as well, each with its strengths and weaknesses. Statistical tests can help us assess the quality of the generated sequences, as an aid in choosing between alternative PRNGs; other factors might include ease of implementation and runtime efficiency. But in the end, the selection of a PRNG has a lot to do with context: What's the intended use? How much apparent randomness is required? What's the cost (in time, long-term code maintenance, etc.) of using a PRNG different from the one included with the standard library?

When pseudo-random isn't good enough

Sometimes, there's no acceptable substitute for truly random numbers—at least as seed values for PRNGs. As noted above, most computers aren't very good at producing genuine randomness, but there are any number of real-world processes that have truly random behavior. Even better, some of these processes, while random, have well-understood statistical behavior. If we can measure one or more of these processes over time, that may give us a good source of useful random numbers.

In fact, processes such as the radioactive decay of certain elements, photon emissions by semiconductors, atmospheric or thermal noise—even human users' mouse movements and keyboard activity—serve these purposes very well. Capabilities for observing such sources of randomness, in order to seed pseudo-random number generators with randomly chosen values, are part of most current operating systems. In addition, there are web services that provide truly random values over the Internet.

This page intentionally left blank.

Acknowledgments

Early development of this material was funded in part by the New Mexico Supercomputing Challenge, and was written with the invaluable review and editing assistance of Roger Critchlow and Janet Penevolpe.

References

[1] Deep Dive Coding Java + Android Bootcamp, 2018. [Online]. Available: https://deepdivecoding.com/java-android/. [Accessed: Jan. 30, 2019].

[2] Oracle Technology Network for Java Developers, Jan. 17, 2019. [Online]. Available: https://www.oracle.com/technetwork/java/. [Accessed: Jan. 30, 2019].

[3] B. Arkin, et al, "How We Learned to Cheat at Online Poker: A Study in Software Security," Developer.com, Jun. 7, 2001. [Online]. Available: https://www.developer.com/tech/article.php/616221/How-We-Learned-to-Cheat-at-Online-Poker-A-Study-in-Software-Security.htm [Accessed: Jan. 29, 2019].

[4] Rob Weir, "Doing the Microsoft Shuffle: Algorithm Fail in Browser Ballot," Mar. 6, 2010. [Online]. Available: http://www.robweir.com/blog/2010/02/microsoft-random-browser-ballot.html. [Accessed: Jan. 29, 2019].

[5] "Shuffling: Shuffling algorithms", Wikipedia, Jan. 29, 2019. [Online]. Available: http://en.wikipedia.org/wiki/Shuffling#Shuffling_algorithms. [Accessed: Jan. 29, 2019].

[6] "Fisher-Yates shuffle", Wikipedia, Jan. 5, 2019. [Online]. Available: http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle. [Accessed: Jan. 29, 2019].

[7] "How to Play Roadrunner Cash", New Mexico Lottery, 2016. [Online]. Available: http://www.nmlottery.com/how-to-play-roadrunner-cash.aspx. [Accessed: Jan. 29, 2019].

[8] "Random (Java SE 11 & JDK 11)", Java® Platform, Standard Edition & Java Development Kit, Version 11 API Specification, Sep. 7, 2018. [Online]. Available: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Random.html. [Accessed: Jan. 29, 2019].

[9] "Mersenne twister", Wikipedia, Jan. 19, 2019. [Online]. Available: http://en.wikipedia.org/wiki/Mersenne_twister. [Accessed: Jan. 29, 2019].

This page intentionally left blank.

Appendix A: Fisher-Yates Shuffle

Durstenfeld's version of Fisher-Yates shuffle (aka Knuth shuffle)

1. Begin with the sequence of items X, containing the N items $x_0, x_1, \ldots, x_{N-1}$.
2. For each integer value i, starting at $N-1$ and counting down to 1, do the following:
    a. Generate a random value j, which can be any one of $\{0, 1, \ldots, i\}$; each of the values must be equally likely to be selected.
    b. Exchange the positions of x_i and x_j in the sequence. Note that it's possible that $i = j$; obviously, no exchange is needed when that's the case.
3. Stop. The items in the sequence are now shuffled.

This page intentionally left blank.

Appendix B: Fisher-Yates Shuffle for Six Items and a Six-Sided Die

Algorithm

1. Write the names of the six items to be shuffled in columns 1 through the 6, under **Position**, in the *Initial* row.

2. Start on row 1 (i.e. the row where the number 1 appears in the **Iteration** column).

3. Do the following for rows 1 through 5, in order:

    a. Throw a single die to get a random value j, repeating as necessary to get a value between 1 and i, (the value in the ***i*** column of the current row) inclusive.

    b. Write the j value (the value rolled on the die) in the **Roll (*j*)** column of the current row.

    c. Copy the items from the previous row to the current row, exchanging the items in position i and position j.

4. Stop. Row 5 contains the fully shuffled items.

Working Table

			Items (*N* = Number of Items = 6)					
			Position					
Iteration	***i***	**Roll (*j*)**	**1**	**2**	**3**	**4**	**5**	**6**
Initial								
1	6							
2	5							
3	4							
4	3							
5	2							

This page intentionally left blank.

Appendix C: Lottery Class (Lottery.java)

```java
package edu.cnm.deepdive;

import java.util.Arrays;
import java.util.Random;

public class Lottery {

  private int[] numbers;

  public Lottery(int maximum) {
    numbers = new int[maximum];
    for (int i = 0; i < numbers.length; i++) {
      numbers[i] = i + 1;
    }
  }

  private void mix(int iterations, Random rng) {
    for (int i = 0; i < iterations; i++) {
      int destination = numbers.length - i - 1;
      int source = rng.nextInt(destination + 1);
      int temp = numbers[source];
      numbers[source] = numbers[destination];
      numbers[destination] = temp;
    }
  }

  public int[] pick(int count, Random rng) {
    int[] selection;
    mix(count, rng);
    selection = Arrays.copyOfRange(
        numbers, numbers.length - count, numbers.length);
    Arrays.sort(selection);
    return selection;
  }

}
```

This page intentionally left blank.

Appendix D: Roadrunner Lottery Program (Roadrunner.java)

```java
package edu.cnm.deepdive;

import java.util.Arrays;
import java.util.Random;
import java.security.SecureRandom;

public class Roadrunner {

  private static final int MAXIMUM_NUMBER = 37;
  private static final int NUMBER_TO_PICK = 5;
  private static final int NUMBER_OF_TICKETS = 10;
  private static final String PICK_FORMAT =
      "Ticket #%d: %s%n";

  public static void main(String[] args) {
    Lottery lotto = new Lottery(MAXIMUM_NUMBER);
    Random rng = new SecureRandom();
    for (int i = 0; i < NUMBER_OF_TICKETS; i++) {
      int[] pick = lotto.pick(NUMBER_TO_PICK, rng);
      String prettyPick = Arrays.toString(pick);
      System.out.printf(PICK_FORMAT, i + 1, prettyPick);
    }
  }

}
```