Random Numbers

We have performed a number of simulation experiments using computer-generated "random" numbers.

Random Numbers

We have performed a number of simulation experiments using computer-generated "random" numbers.

As noted earlier, these are more correctly called "psuedorandom" numbers, because they are not actually random.

Because they are used extensively, it is worth spending some time to understand how they are generated.

Random Numbers

We have performed a number of simulation experiments using computer-generated "random" numbers.

As noted earlier, these are more correctly called "psuedorandom" numbers, because they are not actually random.

Because they are used extensively, it is worth spending some time to understand how they are generated.

Most algorithms for generating psuedorandom numbers work by computing a sequence of numbers *recursively*, meaning that we always compute the next number in the sequence from its predecessor(s).

Random Numbers

We might represent a general recursive algorithm by the *recursion formula* or *difference equation*

$$x_{n+1} = f(x_n), \quad n = 1, 2, 3, \ldots$$

Random Numbers

We might represent a general recursive algorithm by the *recursion formula* or *difference equation*

$$x_{n+1} = f(x_n), \quad n = 1, 2, 3, \ldots$$

The "solution" of such an equation is a sequence for which each pair of consecutive terms satisfies the recursion formula:

$$\begin{array}{rcl} x_2 & = & f(x_1) \\ x_3 & = & f(x_2) \\ x_4 & = & f(x_3) \\ \vdots & & \vdots \end{array}$$

Random Numbers

To determine the solution sequence, we need to start with one of the terms.

Then we can determine the sequence from that point on by successively applying the recursion formula.

Random Numbers

To determine the solution sequence, we need to start with one of the terms.

Then we can determine the sequence from that point on by successively applying the recursion formula.

Usually we start with a given *initial value* labelled x_1

Random Numbers

To determine the solution sequence, we need to start with one of the terms.

Then we can determine the sequence from that point on by successively applying the recursion formula.

Usually we start with a given *initial value* labelled x_1

The recursion formula together with an initial value x_1 is called an *initial value problem*

$$x_{n+1} = f(x_n), \quad n = 1, 2, 3, \ldots \quad x_1 = 1$$

Random Numbers

To determine the solution sequence, we need to start with one of the terms.

Then we can determine the sequence from that point on by successively applying the recursion formula.

Usually we start with a given *initial value* labelled x_1

The recursion formula together with an initial value x_1 is called an *initial value problem*

$$x_{n+1} = f(x_n), \quad n = 1, 2, 3, \ldots \quad x_1 = 1$$

Every initial value problem has a unique solution, which is a sequence $x_1, x_2, x_3, \ldots$

Random Numbers

One of the common types of psuedorandom number generators uses the following recursion formula:

$$x_{n+1} = b \cdot x_n \text{mod } a, \quad n = 1, 2, 3, \ldots$$

where $b \cdot x_n$mod a means "the remainder when $b \cdot x_n$ is divided by a"

Random Numbers

One of the common types of psuedorandom number generators uses the following recursion formula:

$$x_{n+1} = b \cdot x_n \text{mod } a, \quad n = 1, 2, 3, \ldots$$

where $b \cdot x_n \text{mod } a$ means "the remainder when $b \cdot x_n$ is divided by a"

Integers that produce the same remainder on division by a are said to be "congruent modulo a". As a result, this class of psuedorandom number generators is called *congruental*.

Random Numbers

Congruential generators are popular because they are easy to program, inexpensive to run, and produce "good" results for certain values of a and b.

Random Numbers

Congruential generators are popular because they are easy to program, inexpensive to run, and produce "good" results for certain values of a and b.

For a psuedorandom number generator, "good" results means that the sequence produced behaves like a truly random sequence, even though it is completely deterministic.

Random Numbers

Congruential generators are popular because they are easy to program, inexpensive to run, and produce "good" results for certain values of a and b.

For a psuedorandom number generator, "good" results means that the sequence produced behaves like a truly random sequence, even though it is completely deterministic.

As an example of how easy they are to program, we will now write a congruental generator in R.

Random Numbers

First start R and define the values of a and b.

We'll use $a = 23$ and $b = 5$. Enter:

```
a<-23
b<-5
```

Random Numbers

First start R and define the values of a and b.

We'll use $a = 23$ and $b = 5$. Enter:

```
a<-23
b<-5
```

Now allocate an array called x to hold the sequence. We'll start with $10,000$ elements:

```
x<-rep(0,10000)
```

Random Numbers

First start R and define the values of a and b.

We'll use $a = 23$ and $b = 5$. Enter:

```
a<-23
b<-5
```

Now allocate an array called x to hold the sequence. We'll start with $10,000$ elements:

```
x<-rep(0,10000)
```

The recursion formula requires a starting value x_1, which we'll assign to the first element of the x array in R.

Always pick an integer greater than 0 and less than a. Enter:

```
x[1]<-13
```

Random Numbers

Now write the R statement that implements the recursion formula

$$x_{n+1} = b \cdot x_n \text{mod } a, \quad n = 1, 2, 3, \ldots \quad n_1 = 13$$

Random Numbers

Now write the R statement that implements the recursion formula

$$x_{n+1} = b \cdot x_n \text{mod } a, \quad n = 1, 2, 3, \ldots \quad n_1 = 13$$

The MOD operator in R is a double percent sign $\%\%$, and the multiplication operator is an asterisk $*$.

Random Numbers

Now write the R statement that implements the recursion formula

$$x_{n+1} = b \cdot x_n \text{mod } a, \quad n = 1, 2, 3, \ldots \quad n_1 = 13$$

The MOD operator in R is a double percent sign $\%\%$, and the multiplication operator is an asterisk $*$.

We designate the n^{th} element of x by $x[n]$. In the R language, the recursion formula for x_{n+1} is:

```
x[n+1] <- (b*x[n]) %% a
```

Random Numbers

We will also make use of a construct called FOR that enables us to execute a block of statements a certain number of times, in this case 9,999.

Random Numbers

We will also make use of a construct called FOR that enables us to execute a block of statements a certain number of times, in this case 9,999.

In the R language, the correct syntax for this is:

for(n in 1:9999) {*some statement to be executed*}

Random Numbers

We will also make use of a construct called FOR that enables us to execute a block of statements a certain number of times, in this case 9,999.

In the R language, the correct syntax for this is:

for(n in 1:9999) {*some statement to be executed*}

This construct will execute the statements in curly brackets $\{\}$ 9,999 times, with n set to:

- 1 the first time
- 2 the second time
- 3 the third time

and so on.

Random Numbers

Now we are ready to write the full R statement by combining the FOR construct with the recursion formula from the previous slide. The result is:

```
for(n in 1:9999) {x[n+1] <- (b*x[n]) %% a}
```

Random Numbers

Now we are ready to write the full R statement by combining the FOR construct with the recursion formula from the previous slide. The result is:

```
for(n in 1:9999) {x[n+1] <- (b*x[n]) %% a}
```

If we typed the statement correctly, it should apply the recursion formula to compute $x[2]$ through $x[10000]$.

Random Numbers

Now we are ready to write the full R statement by combining the FOR construct with the recursion formula from the previous slide. The result is:

```
for(n in 1:9999) {x[n+1] <- (b*x[n]) %% a}
```

If we typed the statement correctly, it should apply the recursion formula to compute $x[2]$ through $x[10000]$.

To view the first 100 values of the result, type:

```
x[1:00]
```

Random Numbers

Now we are ready to write the full R statement by combining the FOR construct with the recursion formula from the previous slide. The result is:

```
for(n in 1:9999) {x[n+1] <- (b*x[n]) %% a}
```

If we typed the statement correctly, it should apply the recursion formula to compute $x[2]$ through $x[10000]$.

To view the first 100 values of the result, type:

```
x[1:00]
```

Notice that the sequence consists of 22 positive integers repeated over and over in this sequence:

13,19,3,15,6,7,12,14,1,5,2,10,4,20,8,17,16,11,9,22,18,21

Random Numbers

Now generate a frequency table of the values in x by entering:

table(x)

Random Numbers

Now generate a frequency table of the values in x by entering:

table(x)

This should produce a list of the unique values among the $10,000$ elements of the x array, with a count of how many times each of them occurs.

Random Numbers

Now generate a frequency table of the values in x by entering:

table(x)

This should produce a list of the unique values among the $10,000$ elements of the x array, with a count of how many times each of them occurs.

Notice that the counts differ by at most one, because the same sequence repeats over and over.

Random Numbers

Now generate a frequency table of the values in x by entering:

table(x)

This should produce a list of the unique values among the $10,000$ elements of the x array, with a count of how many times each of them occurs.

Notice that the counts differ by at most one, because the same sequence repeats over and over.

For certain choices of a and b, number theory guarantees that this will happen:

The sequence will contain the positive integers from 1 to $a-1$ in some shuffled order that repeats over and over.

Random Numbers

One last step remains. We would like our psuedorandom number generator to produce numbers between zero and one, not integers from 1 to 22.

Random Numbers

One last step remains. We would like our psuedorandom number generator to produce numbers between zero and one, not integers from 1 to 22.

We can accomplish this by dividing each element of the sequence by the value of a, 23 in this case. Enter:

```
z<-x/23
z[1:100]
```

Random Numbers

One last step remains. We would like our psuedorandom number generator to produce numbers between zero and one, not integers from 1 to 22.

We can accomplish this by dividing each element of the sequence by the value of a, 23 in this case. Enter:

```
z<-x/23
z[1:100]
```

Notice that the contents of the z array resemble the output of runif().

Random Numbers

In fact, the only thing that keeps this from being a decent random number generator is the fact that it only generates 22 values.

Random Numbers

In fact, the only thing that keeps this from being a decent random number generator is the fact that it only generates 22 values.

The only difference between our congruential generator and the ones actually implemented in many statistical packages is that they have a much larger value of a.

(as well as a corresponding b that produces all integers from 1 to $a - 1$).

Random Numbers

It should be clear why psuedorandom numbers generated in this way are *not* random.

Random Numbers

It should be clear why psuedorandom numbers generated in this way are *not* random.

If you pick any positive integer less than $a-1$, you can set the initial value x_1 to this number and the rest of the sequence is entirely determined.

Random Numbers

It should be clear why psuedorandom numbers generated in this way are *not* random.

If you pick any positive integer less than $a - 1$, you can set the initial value x_1 to this number and the rest of the sequence is entirely determined.

Some implementations allow you to specify a starting value, usually called the *seed*.

If you specify the seed, the same sequence of "random" numbers will be generated every time.

Random Numbers

It should be clear why psuedorandom numbers generated in this way are *not* random.

If you pick any positive integer less than $a - 1$, you can set the initial value x_1 to this number and the rest of the sequence is entirely determined.

Some implementations allow you to specify a starting value, usually called the *seed*.

If you specify the seed, the same sequence of "random" numbers will be generated every time.

This is helpful for things like debugging simulation programs.

Random Numbers

Implementations that do not allow you to specify the seed usually determine it internally in a way that makes it somewhat random.

One way to do this is to use the low order digits of a high resolution clock, which most systems have, at the instant the "enter" key is pressed.