R for 1.5-1.11 and 3.4

A.S. Wagaman, CC BY-NC

R for 1.5-1.9, 1.11, and 3.4 - Counting Methods and Problem-Solving Strategies

```
# NA
```

R for 1.10 - A First Look at Simulation

This section of the text introduces the concept of Monte Carlo simulation, along with some R commands using two examples (which are provided as scripts below).

```
# Book commands - pg. 23
n <- 100 #set for example
sample(0:1, n, replace = TRUE)
```

```
##   [1] 1 1 1 1 0 0 0 0 1 0 1 0 0 1 1 0 1 0 1 0 0 0 1 1 1 0 1 1 0 0 1 1 0 0 0
##  [36] 0 0 1 0 1 0 1 1 1 0 0 0 0 0 1 0 1 1 0 0 0 0 0 0 1 1 1 1 1 0 1 1 0 0 1
##  [71] 0 0 0 1 0 0 0 1 0 1 0 0 0 1 1 0 1 0 0 1 1 0 0 0 1 1 1 0 0 1
```

```
mean(sample(0:1, n, replace = TRUE))
```

```
## [1] 0.51
```

No seed is set here so you can see what the commands do and repeat them as the book requests. Be sure you understand what the sample and mean commands do. Note that because the sample generated was NOT saved, and no seed was set, the sample generated in the second line is not the same as what is fed to the mean command. You could easily fix this with:

```
n <- 100 #set for example
set.seed(589)
tempsample <- sample(0:1, n, replace = TRUE)
mean(tempsample)
```

```
## [1] 0.53
```

However, that's not the purpose of the script below. Here is the next bit of code.

```
# Book commands - pg. 24
trial <- sample(0:1, 3, replace = TRUE)
if(sum(trial) == 3) 1 else 0
```

```
## [1] 0
```

Before we explain the code, we note that the commands are put together into an R script - basically an R file that contains JUST R code. Let's take a look at our first script from the text, and see how the commands are used together. We can copy the entire script into our .Rmd file as long as it stays within an R chunk.

SCRIPT - CoinFlip.R - pg. 24

```
### Coin flip 1.R
###
### What's the probability of getting heads in 3 coin flips?
### Random experiment: Flip 3 coins
### Event of interest: All heads

### The trial
trial <- sample(0:1, 3, replace = TRUE)

### Success?
if (sum(trial) == 3) 1 else 0
```

```
## [1] 0
```

```
### Repeat
n <- 10000   ### Number of iterations
simlist <- replicate(n, 0) ## Initialize list with 0's
for (i in 1:n){ #start of for loop
    trial <- sample(0:1, 3, replace = TRUE)
    success <- if (sum(trial) == 3) 1 else 0
    simlist[i] <- success
} #end of for loop

### Simulated result
mean(simlist)
```

```
## [1] 0.1327
```

Let's look at what this script is doing. Simlist is a vector that contains the list of simulated values. In this case, it records a 1 for a success and a 0 for each failure in the simlist vector, based on what value success takes as a result of the ifelse statement. You could call the vector "results" or any other name that you like. The text reserves the name simlist for this object.

This script introduces several important items: an initialization of simlist based on setting n, a *for* loop, and an ifelse statement.

The initialization is useful for memory allocation. We know how many replications we want to do, so we create a vector of that length filled with zeros. We will then change the 0 on each run to the result of that trial. The text uses replicate for this, but *rep* would work just as well.

The ifelse statement checks to see if the sum of the trial vector is 3. If it is 3, then success is set to 1. Otherwise, success is set to 0. For more complicated operations based on the criteria set, use curly brackets.

Look at the *for* loop next. The for loop has several key pieces. First, the contents contained in the parentheses are the parameters determining how the loop runs. Once n is set, the loop runs based on index i from 1 to n, increasing the loop counter i by 1 each time until *n* runs are achieved. Each of the n runs will run the three lines of code contained in the brackets. For loops can be used for many applications, but become tricky if you are trying to nest several within each other. We will see alternatives to for loops, including the *replicate* function, as shown in the next script, or in our modification of this example, shown below.

```
# Alternative code to CoinFlip.R

n <- 10000   ### Number of iterations
set.seed(54)
simlist <- replicate(n, if (sum(sample(0:1, 3, replace = TRUE)) == 3) 1 else 0)

### Simulated result
mean(simlist)
```

```
## [1] 0.1247
```

The modification provided has a seed set for reproducibility, and is much more condensed than the book's script. Both simulations WORK! This is a good reminder that because R is a programming language, there are often many ways to do desired tasks.

The next script accompanies the second Monte Carlo simulation example.

SCRIPT - Divisible356.R - pg. 25

```
### Divisible356.R
###
### What's the probability that a random integer between 1 and 1,000
###   is divisible by 3, 5, or 6?

##########################################################################
### The function simdivis() simulates one trial, returning a 1 if the event occurs, and 0 otherwise

simdivis <- function()
{
num <- sample(1:1000, 1)   ### Pick a random integer from 1 to 1,000
if (num %% 3 == 0 || num %% 5 == 0 || num %% 6 == 0) 1 else 0
    ### Return 1 if the num is divisible by 3, 5, or 6; 0, otherwise
}
###########################################################################

### Now repeat many times (say, 1000) and take the proportion of 1's as the simulated probability

simlist <- replicate(10000, simdivis() )
mean(simlist)
```

```
## [1] 0.4627
```

This code is set up with a function. It also has a more complicated ifelse statement and uses *replicate* instead of a for loop. The syntax in the ifelse statement with the bars is an "or" and the double %% is for mod operations. Hence num %% 3 ==0 is asking if num is divisible by 3 with remainder 0.

Once the function is created, we see that it outputs just a 1 or 0 each time. Thus, we replicate it 10000 times (you can change this, or make it a variable n which is set), which fills simlist with that number of 1s and 0s based on the output of that many trials.

We can add reproducibility and still use the simdivis function (even though it appears in a different code chunk) as follows.

```
#for reproducibility and to set number of reps in advance
set.seed(500)
n <- 10000
simlist <- replicate(n, simdivis() )
mean(simlist)
```

```
## [1] 0.4689
```

Supplemental R Commands

This section contains supplemental R commands that are not in your text, but are useful for these sections.

R can help with permutation and combination calculations. Sometimes the numbers get so big the computations fail, though that is unlikely for our class. If that happens, then you can swap to looking at the natural log or log base 10 of the value of interest instead. All these commands can be run just in R. On a graphing calculator, the combination function is usually found under the MATH then PRB menu option as nCr.

- choose(n,k) - computes combination of choosing k objects from n.
- prod(n:m) - computes the product of values from n to m. To do a permutation, you want to use prod(n:n-r+1). (For whatever reason, this doesn't work if you assign n and r first, just put in the numbers).
- factorial(x) - computes x!
- lfactorial(x) - computes the natural logarithm of the factorial of x, for when x is large and the factorial computation fails
- lchoose(n,k) - computes the natural logarithm of choose(n,k), for use when the choose computation is failing

For multinomial coefficients, do a succession of choose arguments. For example, if I want to take 9 people and split them into a group of 2, 3, and 4, I could do choose(9,2) times choose(7,3) times choose(4,4). The last one isn't really needed because it will be 1.

```r
# Examples of supplemental commands
choose(n = 10, k = 2)
```

```
## [1] 45
```

```r
prod(8:6)
```

```
## [1] 336
```

```r
factorial(x = 4)
```

```
## [1] 24
```

```r
lfactorial(x = 10)
```

```
## [1] 15.10441
```

```r
lchoose(n = 100, k = 20)
```

```
## [1] 47.73063
```

Textbook Citation

Dobrow, R. P. (2013). Probability: with applications and R. John Wiley & Sons.