R Tutorial #4 – Econ 103

In this tutorial you'll learn about a powerful technique called Monte Carlo Simulation that allows us to use R to calculate probabilities rather than using the various rules and formulas from class. Eight exercises are scattered throughout this tutorial.

Introduction

Roughly speaking, Monte Carlo Simulation means using a computer to repeatedly carry out a random experiment and keeping track of the outcomes. Remember: our definition of probability is "long-run relative frequency." If we repeat an experiment (like flipping a coin) a large number of times and tabulate the outcomes, the relative frequencies will "converge," in a sense that will be made clearer later in the semester, to the probabilities of each outcome. Here's what Wikipedia (https://en.wikipedia.org/wiki/Monte_Carlo_metho has to say on the matter:

> Monte Carlo methods (or Monte Carlo experiments) are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results; i.e., by running simulations many times over in order to calculate those same probabilities heuristically just like actually playing and recording your results in a real casino situation: hence the name.

But where did the name come from? Monte Carlo is a region in the tiny principality of Monaco (https://en.wikipedia.org/wiki/Monaco) famed for its casino. Again from Wikipedia (https://en.wikipedia.org/wiki/Monte_Carlo_metho

> The modern version of the… Monte Carlo method was invented in the late 1940s by Stanislaw Ulam, while he was working on nuclear weapon projects at the Los Alamos National Laboratory. A colleague of… Ulam, Nicholas Metropolis, suggested using the name Monte Carlo, which refers to the Monte Carlo Casino in Monaco where Ulam's uncle would borrow money from relatives to gamble.

Flipping a coin one million times by hand would be tedious in the extreme. But *simulating* a million coin flips in R takes less than a second. This is why Monte Carlo Simulation is such a valuable tool.

Monte Carlo in R

Any Monte Carlo Simulation can be broken into two parts. First, we need code to carry out the random experiment we're interested in on the computer. Depending on the problem, the details of this step will vary. For this tutorial we'll work with a simple but flexible function called `sample` that allows us to simulate discrete experiments like rolling dice, drawing from an urn, or flipping a coin. In later tutorials you'll learn some other functions for constructing more complicated random experiments. Second, we need code to repeat something over and over. This step will always be the same, regardless of the details of the first step: we'll use the function `replicate`.

`sample`

The R command `sample` simulates drawing marbles from a bowl. It turns out that there are many random experiments that can be *reduced* to thinking about a bowl containing different kinds of marbles, so `sample` is *ipso facto* a fairly general command. The function `sample` takes three arguments: `x` is a vector containing the "marbles" in our hypothetical bowl, `size` tells R how many marbles we want to draw, and `replace` is set to `TRUE` or `FALSE` depending

on whether we want to draw marbles from the bowl *with replacement*, which means putting the marble back after each draw, or *without replacement*, which means keeping each marble after we draw it and not returning it to the bowl. Of these three arguments `x` is the most mysterious. What do I mean by a "vector containing marbles"? You can use *any vector at all* as the argument `x`: it simply plays the role of the *label* we give to each of our hypothetical marbles.

Let's look at an example in which I simulate drawing two marbles from a bowl containing one red, one blue and one green marble, *without* replacement:

```
marbles = c('red', 'blue', 'gree
n')
sample(x = marbles, size = 2, re
place = FALSE)
```

```
## [1] "red"  "blue"
```

Notice that we didn't get any repeats since I set `replace = FALSE`.

IMPORTANT: You may get a different result than I did when you run the previous command. This is because the command `sample` simulates a *random* draw, meaning the result won't be the same each time you use the command. (For those of you who know a bit more about computer science, technically they're only pseudo-random draws, but this will suffice for Monte Carlo Simulations.)

In the preceding example, `marbles` was a character vector, but we can also use `sample` with a numeric vector. This example draws 5 numbers between 1 and 10 without replacement:

```
sample(x = 1:10, size = 5, repla
ce = FALSE)
```

```
## [1] 9 8 5 2 6
```

If I instead set `replace = TRUE` I can get repeats of the same number. To make repeats particularly likely, I'll change `size` to 20:

```
numbers = 1:10
sample(x = numbers, size = 20, r
eplace = TRUE)
```

```
##  [1]  6  2 10 10  7  5  4  5
1  7  2  7  5  9  1  6  6  4  5
1
```

Important: If you re-run the preceding command you will, in general, get a *different result*. This is because `sample` carries out a *random experiment*:

```
sample(x = numbers, size = 20, r
eplace = TRUE)
```

```
##  [1] 10  3 10  2  9  5  1  5
3  9 10  8  4  1  2 10  5 10  2
2
```

```
sample(x = numbers, size = 20, r
eplace = TRUE)
```

```
##  [1]  9  1 10  2  5  4  4  1
5  7  6  7  8  9  9  6  7  9  5
7
```

Exercise #1

I'm running a prize drawing and need to select two *different students* from Econ 103 at random. Suppose for simplicity that I only have five students and they're named Alice, Bob, Charlotte, Dan, and Emily. How can I use `sample` to make my drawing?

```
students = c("Alice", "Bob", "Ch
arlotte", "Dan", "Emily")
sample(x = students, size = 2, r
eplace = FALSE)
```

```
## [1] "Bob"       "Charlotte"
```

Exercise #2

Jim is bored and loves to solve math problems for fun. (He's a strange guy.) To pass the time, he labels ten cards with the numbers 1 through 10 and puts them in a bowl. He then draws five of these cards *with replacement* and calculates their sum. Use R to replicate one of Jim's "random sums."

```
random.numbers = sample(x = 1:10
, size = 5, replace = TRUE)
sum(random.numbers)
```

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

Dice Rolls with `sample`

Probability theory was initially developed in the 16th and 17th centuries to solve problems involving gambling games. Many of these problems involved rolling some number of fair, six-sided dice. We can simulate one such die roll in R as follows:

```
sample(1:6, size = 1, replace =
TRUE)
```

```
## [1] 5
```

Quick Quiz: would it have made a difference if you had set `replace = FALSE` in the preceding command?

Answer: No, because `size = 1`. If you only draw once, it doesn't matter if you replace since the experiment is over!

Rolling one die is a pretty boring example, but we can use it to build up more interesting ones. What if we wanted to roll two fair, six-sided dice and compute their sum?

Here's one way to do it:

```
sample(1:6, size = 1, replace = TRUE) + sample(1:6, size = 1, replace = TRUE)
```

```
## [1] 6
```

Again, since each of the `sample` commands here involves only one draw, it doesn't matter whether we set `replace = TRUE` or `FALSE`. But it turns out that there's a much easier way:

```
dice.roll <- sample(1:6, size = 2, replace = TRUE)
dice.roll
```

```
## [1] 4 5
```

```
sum(dice.roll)
```

```
## [1] 9
```

or, in a single command:

```
sum(sample(1:6, size = 2, replac
e = TRUE))
```

```
## [1] 8
```

In each case, what we've done is to draw *two* numbers from the range 1 through 6 *with replacement*. This is the same as drawing *one* number with replacement twice. Note that in this case it *would* make a difference whether we set `replace` to `TRUE` instead of `FALSE`. This is because drawing twice *without* replacement would *not* be the same as making two individual draws.

Exercise #3

Write R code to roll ten fair, six-sided dice and calculate their sum

```
sum(sample(x = 1:6, size = 10, r
eplace = TRUE))
```

```
## [1] 30
```

`replicate`

We now know how to use `sample` to carry out various random experiments. The question is, how can we *repeat* these experiments? In some situations, merely using `sample` is enough. For example, I could repeat the experiment of rolling a single fair die 20 times as follows:

```
die.rolls <- sample(x = 1:6, siz
e = 20, replace = TRUE)
die.rolls
```

```
##  [1] 6 4 4 5 4 1 4 5 3 2 5 2
6 2 4 5 5 6 1 6
```

But if I wanted to repeat the experiment involving the *sum of two dice* there wouldn't be an easy way to do this using `sample`. Instead we'll turn to a function called `replicate` whose sole purpose is to repeat some *other* R command over and over and store the results in a convenient format.

For those of you who have taken some computer science courses, `replicate` is essentially a wrapper to a family of functions called `*apply` that implement common tasks involving `for` loops without explicitly using looping syntax. Using `replicate` for common tasks rather than explicit `for` loops makes your code more readable and makes it easier to get the output you want in the format you want. See this (http://stackoverflow.com/questions/3505701/r-grouping-functions-sapply-vs-lapply-vs-apply-vs-tapply-vs-by-vs-aggrega) excellent post on Stack Overflow for more on the `*apply` family.

The easiest way to use `replicate` for Monte Carlo Simulation is as follows: 1. Write a function that does the simulation once. 2. Repeat the experiment using the command `replicate` and store the result.

Let's take this step-by-step and use rolling two dice as our example.

Step 1: Create a Function

Using what we know about creating functions from R Tutorial #3 (Rtutorial3), we can make a function to roll two fair, six-sided dice and return their sum as follows:

```
two.dice <- function(){
  dice <- sample(1:6, size = 2,
replace = TRUE)
  return(sum(dice))
}
```

Note that this particular function *doesn't take any arguments* but we *still need the parentheses* when creating the function. It's not uncommon to encounter functions that don't take any arguments in code for Monte Carlo Simulations.

IMPORTANT: To call a function that doesn't take any arguments, we *still need the parentheses*. Here's an example to make this clear:

```
#This command actually runs two.
dice
two.dice()
```

```
## [1] 5
```

```
#While this one merely lists the function code!
two.dice
```

```
## function(){
##   dice <- sample(1:6, size = 2, replace = TRUE)
##   return(sum(dice))
## }
```

Step 2: Call The Function Repeatedly Using `replicate`

To use replicate, we need to specify two arguments: `n` tells R *how many times* we want to repeat something and `expr` is the R command we want to repeat. For example

```
replicate(n = 20, expr = two.dice())
```

```
##  [1]  8  5 10 11  8  4  6  9  6 11  7  9  7  9  4  5  8 12  4  4
```

As with all R commands, you don't need the put in the names of the arguments, provided you put everything in the correct order:

```
replicate(20, two.dice())
```

```
##  [1]  8  3  7  7  3  4  6  5
3  3  7 11  4  8  9 11  4  4  8
7
```

As mentioned above, notice that you will, in general, get a different result each time you run this. That's because R is simulating a *random experiment*.

Note: if the function we use as the argument `expr` returns a scalar (i.e. a single value), then `replicate` will return a vector. If our function returns a vector (the same length every replication), `replicate` will return a *matrix* – one *column* for each replication, and rows equal to the length of the output of each experiment.

```
replicate(10, sample(1:10, 1, re
place = FALSE))
```

```
##  [1]  1  2 10  9  4  4  4  1
10  7
```

```
replicate(10, sample(1:10, 5, re
place = FALSE))
```

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

Now let's try writing a slightly more general version of the `two.dice` function so we can see how to use `replicate` with a function that *takes its own arguments*. The function `dice.sum` takes one argument `n.dice` that specifies *how many* six-sided dice we will roll and sum:

```
dice.sum <- function(n.dice){
  dice <- sample(1:6, size = n.d
ice, replace = TRUE)
  return(sum(dice))
}
```

Using `replicate`, we can roll *three* six-sided dice and compute the sum fifty times as follows:

```
replicate(50, dice.sum(3))
```

```
##  [1] 13  8 14  5  6  9  8 14
9  7 14 13  8 10 17 11 12  9 12
15  9  8 14
## [24] 13  7  9 12 16 16  9  6
9 14 10 13 12  6 14 14 11 13 14
8 14 12  6
## [47] 11 11 10 13
```

Exercise #4

Write an *even more general* version of the function `two.dice` called `my.dice.sum` that takes two arguments: `n.sides` tells how many sides each die has and `n.dice` tells how many dice we roll. For example if `n.sides = 4` and `n.dice = 3`, we're rolling three four-sided dice, i.e., dice with sides numbered 1-4. Use `replicate` to simulate the sum of five four-sided dice a total of 100 times.

```
my.dice.sum <- function(n.dice,
n.sides){
  dice <- sample(1:n.sides, size
= n.dice, replace = TRUE)
  return(sum(dice))
}
replicate(100, my.dice.sum(5,4))
```

```
##   [1] 10 15 14 10 16 12 11 15
13 11  8 11 12 10 12 16  9 14 13
16  9 13  5
##  [24] 11 12 17 12 13 14 11 14
10 13 14 11 17 13 10 17 16  9 12
10 12 15 13
##  [47] 15 15 15 11 11 13 11 10
17 17 16 16 12 11 12 13  9  9 13
13 15 14 11
##  [70] 16 15 14 12 13  9 15 13
9  9  9 16 14 14  9 13 14  8 17
12 13 13 14
##  [93] 11 11 14 13 15 14 14 15
```

IMPORTANT!

Something that students often find confusing is the difference between a *function* whose output is random, and the *result* of running such a function. Here's an example of what I mean. Suppose I run the command `sample(1:10, 10, FALSE)` and store the result in a vector called `sim`:

```
sim <- sample(1:10, 10, FALSE)
```

Let's see what's inside `sim`

```
sim
```

```
##  [1]  1  2 10  3  4  8  9  5
7  6
```

Now suppose I enter `sim` again:

```
sim
```

```
##  [1]  1  2 10  3  4  8  9  5
7  6
```

Notice that I got the *same result*. This is because `sim` is *not* random: it's just an ordinary vector. As it happens, the way we *got* `sim` was to perform a random experiment using the function `sample`, but `sim` itself is just an ordinary vector. In contrast if we were to re-run `sample` we would *indeed* generally get a different result:

```
sample(1:10, 10, FALSE)
```

```
##  [1]  5  9  3  1  6  8  7  2
4 10
```

If all this sounds obvious to you, that's great. If not, try to think carefully about what's going on here. The distinction is important when we carry out simulation studies since we need to make sure that we're indeed generating *new* random draws each time we run a random experiment. In other words, we should **do this**

```
replicate(50, sample(1:5, 1, TRU
E))
```

```
##  [1] 5 5 1 2 4 4 5 5 1 4 5 3
2 5 2 3 1 1 3 2 1 5 2 2 4 1 2 4
5 5 3 4 4 4 5
## [36] 3 2 5 2 5 1 4 5 5 3 3 3
2 1 5
```

not this

```
foo <- sample(1:5, 1, TRUE)
replicate(50, foo)
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1
## [36] 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1
```

Make sure you understand the difference.

Approximating Probabilities

Now, let's use R to roll two dice a large number of times. We'll start with 100:

```
sims <- replicate(100, two.dice(
))
```

Notice that I stored the result in a vector called `sims`. When calculating probabilities, we're not interested in *each* of the outcomes, but their

relative frequencies. Using the `table` function introduced in R Tutorial #2 (Rtutorial2), we can summarize the result as follows:

```
table(sims)
```

```
## sims
##  2  3  4  5  6  7  8  9 10 11
12
##  3 10  8 16 10 20  9  5  9  8
2
```

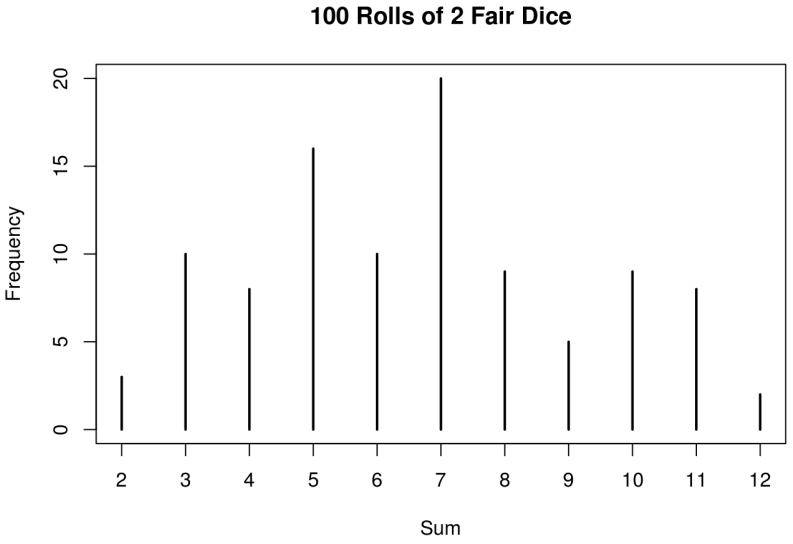
This gives us the *frequency* of every outcome. To convert this to relative frequencies, we need to divide by the number of times we carried out the experiment

```
table(sims)/length(sims)
```

```
## sims
##    2    3    4    5    6    7
8    9   10   11   12
## 0.03 0.10 0.08 0.16 0.10 0.20
0.09 0.05 0.09 0.08 0.02
```

The function `plot` has many special features, one of which is that it knows what to do if you feed it a table as an input. Here's a plot of our result:

```
plot(table(sims), xlab = 'Sum',
ylab = 'Frequency', main = '100
 Rolls of 2 Fair Dice')
```



Or expressed in relative frequencies:

```
plot(table(sims)/length(sims), x
lab = 'Sum', ylab = 'Relative Fr
equency', main = '100 Rolls of 2
Fair Dice')
```

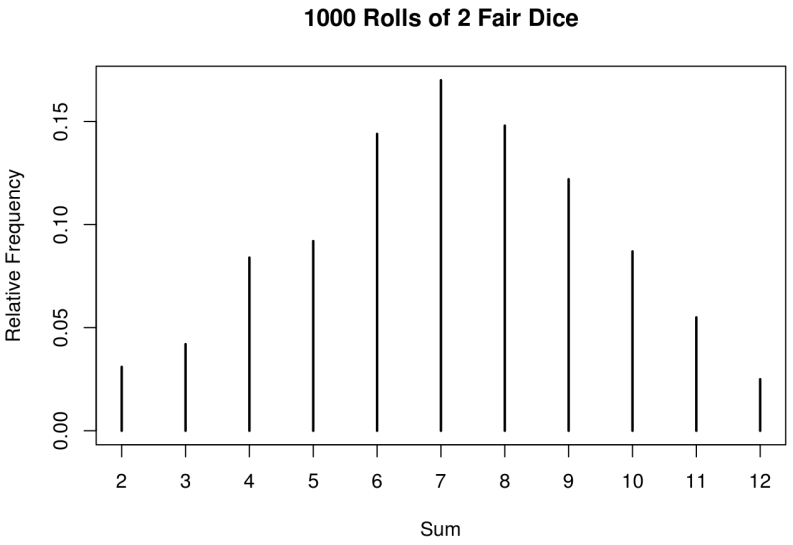
Don't worry if your graphs don't look exactly like these – remember that the results will be random!

IMPORTANT: You might get a table where one of the possible sums doesn't appear at all or where some of the relative frequencies don't agree with the calculated probabilities that we know to be true for real dice. This is because probability is defined as long-run relative frequency, and 100 is *not enough trials* to count as the "long-run." Let's try 1000 rolls:

```
more.sims <- replicate(1000, two.dice())
table(more.sims)/length(more.sims)
```

```
## more.sims
##     2     3     4     5     6     7     8     9    10    11    12
## 0.031 0.042 0.084 0.092 0.144 0.170 0.148 0.122 0.087 0.055 0.025
```

```
plot(table(more.sims)/length(more.sims),
     xlab = 'Sum', ylab = 'Relative Frequency', main = '1000 Rolls of 2 Fair Dice')
```

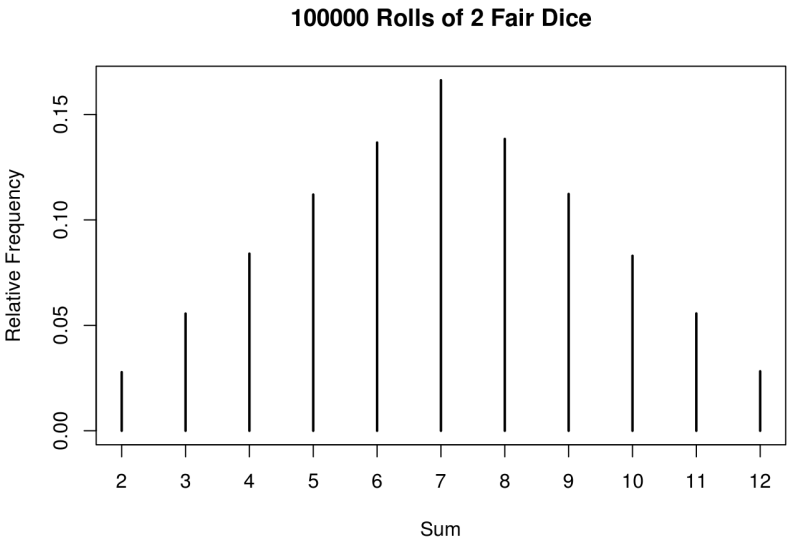


And 100000 rolls

```
even.more.sims <- replicate(100000, two.dice())
table(even.more.sims)/length(even.more.sims)
```

```
## even.more.sims
##       2       3       4       5       6       7       8       9      10
## 0.02780 0.05558 0.08401 0.11204 0.13671 0.16625 0.13843 0.11232 0.08300
##      11      12
## 0.05564 0.02822
```

```
plot(table(even.more.sims)/length(even.more.sims),
     xlab = 'Sum', ylab = 'Relative Frequency', main = '100000 Rolls of 2 Fair Dice')
```



You may have noticed that the previous command took a couple of seconds on your computer. If it didn't, try it with 1 million and it will definitely take longer! As we increased `n` the results got closer to what we know are the true probabilities for rolling dice. Each time we call the function `two.dice` this is called one *simulation replication*. Increasing the number of simulation replications takes us closer to the idea of "long-run" and hence gives more accurate results, but also takes more time on the computer (e.g., doing 100,000 rolls takes roughly 100 times as long as doing 1,000 rolls). How many simulation replications are "enough" depends on the problem at hand, but for examples in this class 10,000 will usually suffice.

More Complicated Probabilities

Suppose I wanted to know the probability of getting a 9 or higher when rolling two dice. How could I use the simulation results from above to calculate this?

The answer is logical conditions. Here's an example to refresh your memory:

```
z <- c(12, 6, 3, 7, 10, 9, 3)
z >= 9
```

```
## [1]  TRUE FALSE FALSE FALSE
TRUE  TRUE FALSE
```

We saw similar expressions in R Tutorials 1 (Rtutorial1) and 2 (Rtutorial2). Notice that R has returned a vector of `TRUE` and `FALSE` where `TRUE` indicates that the corresponding value is 9 or above.

Here's the neat trick: for doing arithmetic, R treats `TRUE` like 1 and `FALSE` like 0. For example:

```
TRUE + TRUE
```

```
## [1] 2
```

```
FALSE * 6
```

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

Using this idea, we can count up the number of elements in the vector `z` that are at least 9 as follows:

```
sum(z >= 9)
```

```
## [1] 3
```

To turn this into a proportion, just divide by the length of `z`

```
sum(z >= 9)/length(z)
```

```
## [1] 0.4285714
```

Exercise #5

Using similar commands, calculate: 1. The proportion of elements in `z` that equal 3. 2. The proportion of elements in `z` that are less than 7.

```
sum(z == 3)/length(z)
```

```
## [1] 0.2857143
```

```
sum(z < 7)/length(z)
```

```
## [1] 0.4285714
```

So, how can we calculate the probability of getting at least a 9 when rolling two dice? We already have the results of a very large number of random dice rolls stored in the vector `even.more.sims`:

```
head(even.more.sims)
```

```
## [1] 2 4 7 6 6 6
```

So all we need to do is *count up* the number of elements of this vector that satisfy the condition and divide by the total number of elements:

```
sum(even.more.sims >= 9)/length(
even.more.sims)
```

```
## [1] 0.27918
```

Exercise #6

Use the same idea to calculate the probability of getting at most 4 when rolling two fair, six-sided dice.

```
sum(even.more.sims <= 4)/length(
even.more.sims)
```

```
## [1] 0.16739
```

To calculate probabilities of more complicated conditions, we can combine the ideas from above with the R commands for AND, namely `&`, and OR, namely `|`.

Here's a simple example using the vector `z` from above. The following expression will be `TRUE` for each element of `z` that is between 7 and 10 *inclusive*:

```
(7 <= z) & (z <= 10)
```

```
## [1] FALSE FALSE FALSE  TRUE
TRUE  TRUE FALSE
```

and we can calculate the proprotion of elements between 7 and 10 as follows:

```
sum((7 <= z) & (z <= 10))/length
(z)
```

```
## [1] 0.4285714
```

To calculate the proportion of elements in `z` that are *either* greater than 10 *or* less than 7, we use `|` as follows:

```
sum((z > 10) | (z < 7))/length(z
)
```

```
## [1] 0.5714286
```

Using this idea, we can calculate the probability of getting a sum between 6 and 8 inclusive

```
sum((6 <= even.more.sims) & (even.more.sims <= 8))/length(even.more.sims)
```

```
## [1] 0.44139
```

and sum below 6 *or* above 8, *exclusive* as follows

```
sum((even.more.sims < 6) | (even.more.sims > 8))/length(even.more.sims)
```

```
## [1] 0.55861
```

Bonus code

The above could also be handled more concisely by being "smarter" about specifying the logic of the conditions.

The key to the is to recall that

$$U < x < L$$

is mathematically equivalent to:

$$|x - m| < \frac{L - U}{2}$$

Where m is the midpoint of the endpoints, $m = \frac{U+L}{2}$. Using this:

```
#same as between 6 & 8 inclusive
mean(abs(even.more.sims - 7) <= 1)
```

```
## [1] 0.44139
```

```
#same as below 6 or above 8 -- the logical opposite of between 6 & 8 inclusive:
mean(!abs(even.more.sims - 7) <= 1)
```

```
## [1] 0.55861
```

Applications

You may be wondering why we bothered with Monte Carlo Simulation above. After all, it's easy to calculate the probabilities directly for all the examples I've shown you so far. As it happens there are a great many important problems for which it is either difficult or indeed impossible to get analytical results for the probabilities of interest. In this section we'll look at some more interesting problems in which carrying out the calculations by hand is less straightforward.

Exercise #7

There is an old Italian gambling game called *Passadieci*, in which the goal is to get at least 11 when three fair, six-sided dice are thrown. The game was famously studied by Galileo at the behest of the Grand Duke of Tuscany, making it one of the earliest examples of the rigorous study of probability theory. Using your function `my.dice.sum` from above and `replicate`, simulate 100,000 replications of this game and store them in a vector called `passadieci`. Use it to answer the following questions: 1. What is the probability of winning the game? 2. Which is more likely when throwing three dice: an 11 or a 12? 3. What is the probability of getting a sum no greater than 7 *or* no less then 15 when throwing three dice? 4. Make a plot of the simulated probabilities of each possible sum when throwing three fair, six-sided dice.

```
passadieci <- replicate(100000,
my.dice.sum(n.dice = 3, n.sides
= 6))
sum(passadieci >= 11)/length(pas
sadieci)
```

```
## [1] 0.49953
```

```
sum(passadieci == 11)/length(pas
sadieci)
```

```
## [1] 0.12391
```

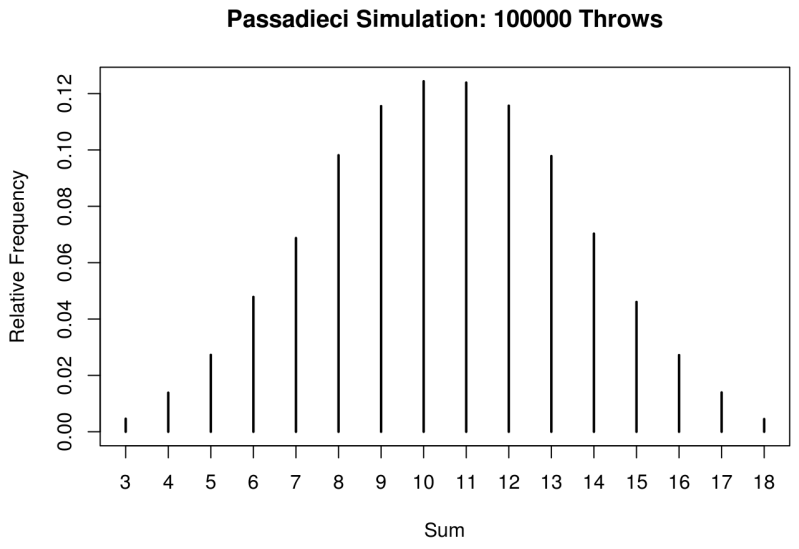
```
sum(passadieci == 12)/length(passadieci)
```

```
## [1] 0.11571
```

```
sum((passadieci <= 7) | (passadieci >= 15))/length(passadieci)
```

```
## [1] 0.25415
```

```
plot(table(passadieci)/length(passadieci), xlab = 'Sum',
     ylab = 'Relative Frequency', main = 'Passadieci Simulation: 100000 Throws')
```



Exercise #8

This problem was initially posed by the famous 17th century gambler Antoine Gombaud, more commonly known as the Chevalier de Méré. Fermat and Pascal discussed its solution in their legendary correspondance that began the study of probability as we know it. Here's the Chevalier's question:

> Which is more likely: (A) getting at least one 6 when rolling a *single* fair six-sided die 4 times or (B) getting at least one *pair* of sixes when *two* fair, six-sided dice are thrown 24 times.

Answer the Chevalier's question using Monte Carlo Simulation

Hint: For each of (A) and (B) begin by writing a function that both carries out the experiment *once* and checks whether the relevant condition is satisfied. Your function should return `TRUE` or `FALSE`. The key will be to come up with a way to write the desired condition using logical operations and the `sum` function, bearing in mind that R treats `TRUE` like 1 and `FALSE` like 0 when doing arithmetic. Finally, use `replicate` to get the answer.

```
experimentA <- function(){
  rolls <- sample(1:6, size = 4,
replace = TRUE)
  condition <- sum(rolls == 6) >
0
  return(condition)
}

experimentB <- function(){
  first.die <- sample(1:6, size
= 24, replace = TRUE)
  second.die <- sample(1:6, size
= 24, replace = TRUE)
  condition <- sum((first.die ==
second.die) & (first.die == 6))
> 0
  return(condition)
}

simsA <- replicate(100000, exper
imentA())
sum(simsA)/length(simsA)
```

```
## [1] 0.5172
```

```
simsB <- replicate(100000, exper
imentB())
sum(simsB)/length(simsB)
```

```
## [1] 0.49179
```