# Lab 1: R Basics

Name: Rufus Petrie

```
## For reproducibility --- don't change this!
set.seed(09262019)
```

**This week's agenda**: manipulating data objects; using built-in functions, doing numerical calculations, and basic plots; reinforcing core probabilistic ideas.

## The binomial distribution

The binomial distribution $\text{Bin}(m, p)$ is defined by the number of successes in $m$ independent trials, each have probability $p$ of success. Think of flipping a coin $m$ times, where the coin is weighted to have probability $p$ of landing on heads.

The R function `rbinom()` generates random variables with a binomial distribution. E.g.,

```
rbinom(n=20, size=10, prob=0.5)
```

produces 20 observations from $\text{Bin}(10, 0.5)$.

## Some simple manipulations

- **1a.** Generate 300 random values from the $\text{Bin}(15, 0.5)$ distribution, and store them in a vector called `bin.draws.0.5`. Extract and display the first 25 elements. Extract and display all but the first 275 elements.

```
bin.draws.0.5 <- rbinom(n=300, 15, 0.5)
bin.draws.0.5[1:25]
```

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

```
bin.draws.0.5[276:300]
```

```
##  [1]  9  8  5  7  8  9  8  8  5  3  6  8  7 11  7  6 12  7  5  9  7  7  4 10  7
```

- **1b.** Add the first element of `bin.draws.0.5` to the sixth. Compare the second element to the fifth, which is larger? A bit more tricky: print the indices of the elements of `bin.draws.0.5` that are equal to 3. How many such elements are there? Theoretically, how many such elements would you expect there to be? Hint: it would be helpful to look at the help file for the `rbinom()` function.

```
bin.draws.0.5[1] + bin.draws.0.5[6]
```

```
## [1] 19
```

```
bin.draws.0.5[2] > bin.draws.0.5[5]
```

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

```
which(bin.draws.0.5 == 3)
```

```
## [1] 154 190 285
```

The first plus the sixth element equals 14. The second element is smaller than the fifth. There are 3 elements that equal 3, which is slightly lower than the 4.16 we would expect.

- **1c.** Find the mean and standard deviation of `bin.draws.0.5`. Is the mean close what you'd expect? The standard deviation?

```
mean(bin.draws.0.5)
```

```
## [1] 7.57
```

```
sd(bin.draws.0.5)
```

```
## [1] 1.89307
```

We expect the mean to equal $np = 15(0.5) = 7.5$ and the standard deviation to equal $\sqrt{np(1-p)} = \sqrt{15(0.5)(0.5) = 1.93}$, so these numbers are fairly reasonable.

- **1d.** Call `summary()` on `bin.draws.0.5` and describe the result.

```
summary(bin.draws.0.5)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    2.00    6.00    8.00    7.57    9.00   12.00
```

The median and mean are reasonable (7.5 is the expected value, and 7/8 makes sense for the integer valued median). We did not see too many extreme values in this sample.

- **1e.** Find the data type of the elements in `bin.draws.0.5` using `typeof()`. Then convert `bin.draws.0.5` to a vector of characters, storing the result as `bin.draws.0.5.char`, and use `typeof()` again to verify that you've done the conversion correctly. Call `summary()` on `bin.draws.0.5.char`. Is the result formatted differently from what you saw above? Why?

```
typeof(bin.draws.0.5)
```

```
## [1] "integer"
```

```
bin.draws.0.5.char <- as.character(bin.draws.0.5)
typeof(bin.draws.0.5.char)
```

```
## [1] "character"
```

```
summary(bin.draws.0.5.char)
```

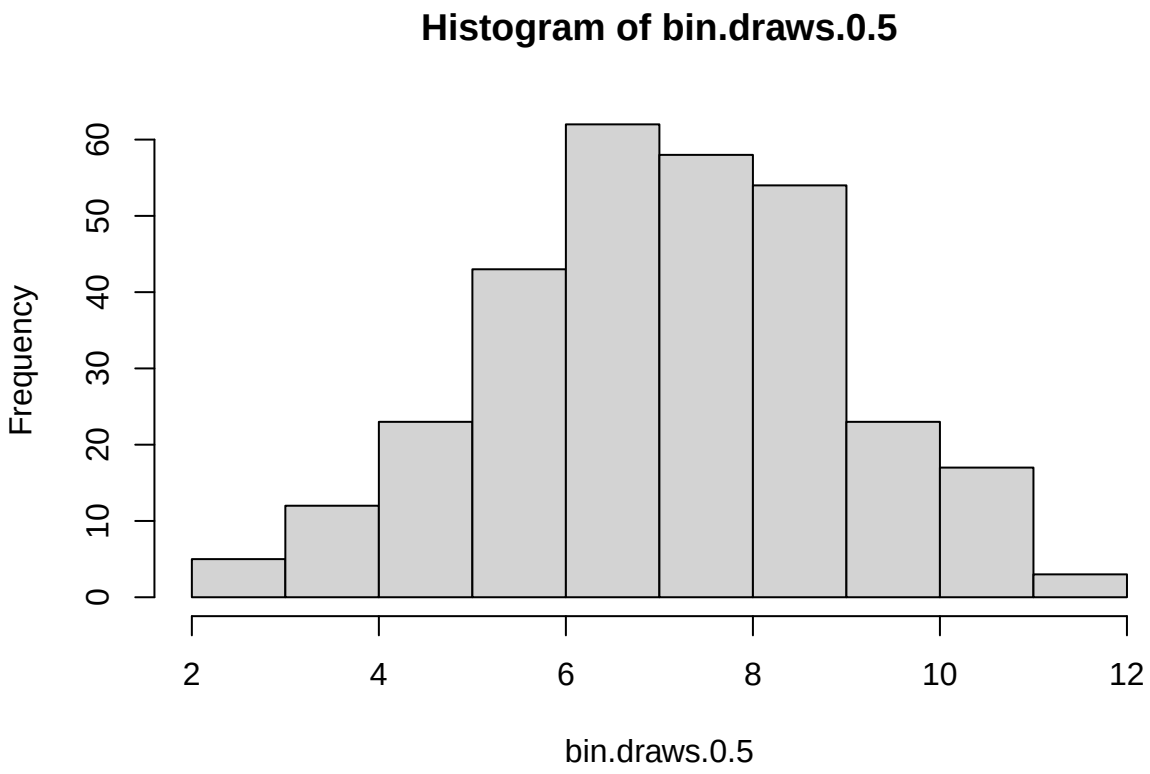
```
##    Length     Class      Mode
##       300 character character
```

The result is formatted differently because numerical measurements no longer apply to the data.

## Some simple plots

- **2a.** The function `plot()` is a generic function in R for the visual display of data. The function `hist()` specifically produces a histogram display. Use `hist()` to produce a histogram of your random draws from the binomial distribution, stored in `bin.draws.0.5`.

```
hist(bin.draws.0.5)
```




- **2b.** Call `tabulate()` on `bin.draws.0.5`. What is being shown? Does it roughly match the histogram you produced in the last question?

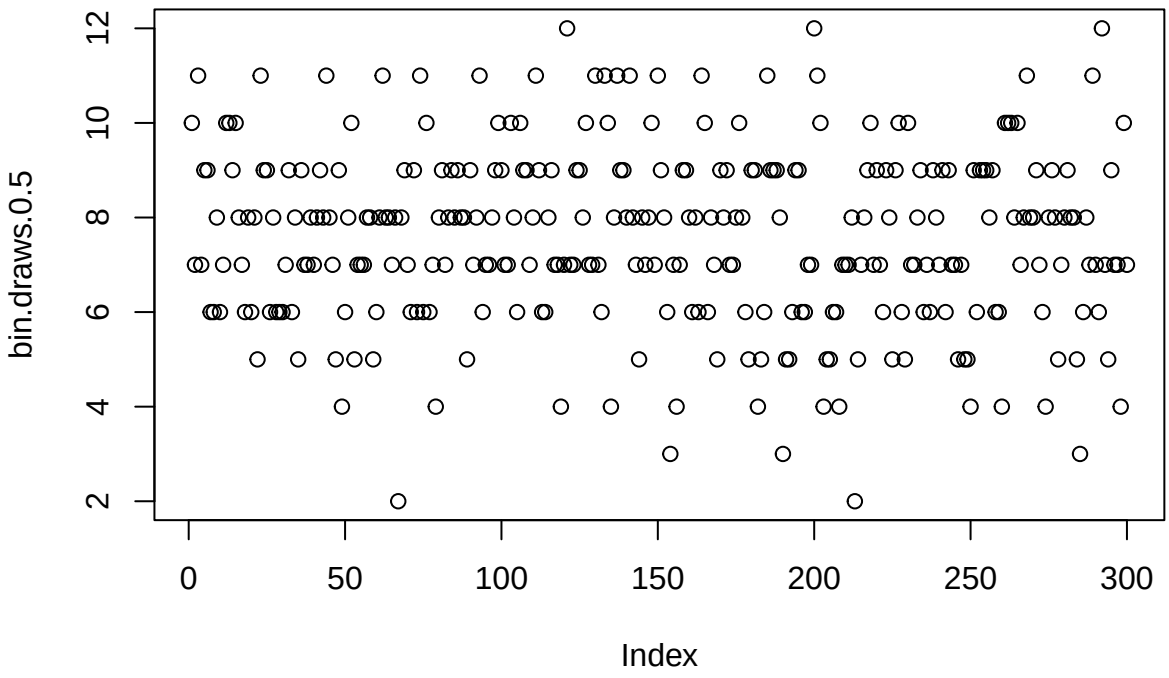
```
tabulate(bin.draws.0.5)
```

```
##  [1]  0  2  3 12 23 43 62 58 54 23 17  3
```

This shows the amounts of each observation in bin.draws.0.5. This roughly matches what we saw in the histogram.

- **2c.** Call `plot()` on `bin.draws.0.5` to display your random values from the binomial distribution. Can you interpret what the `plot()` function is doing here?

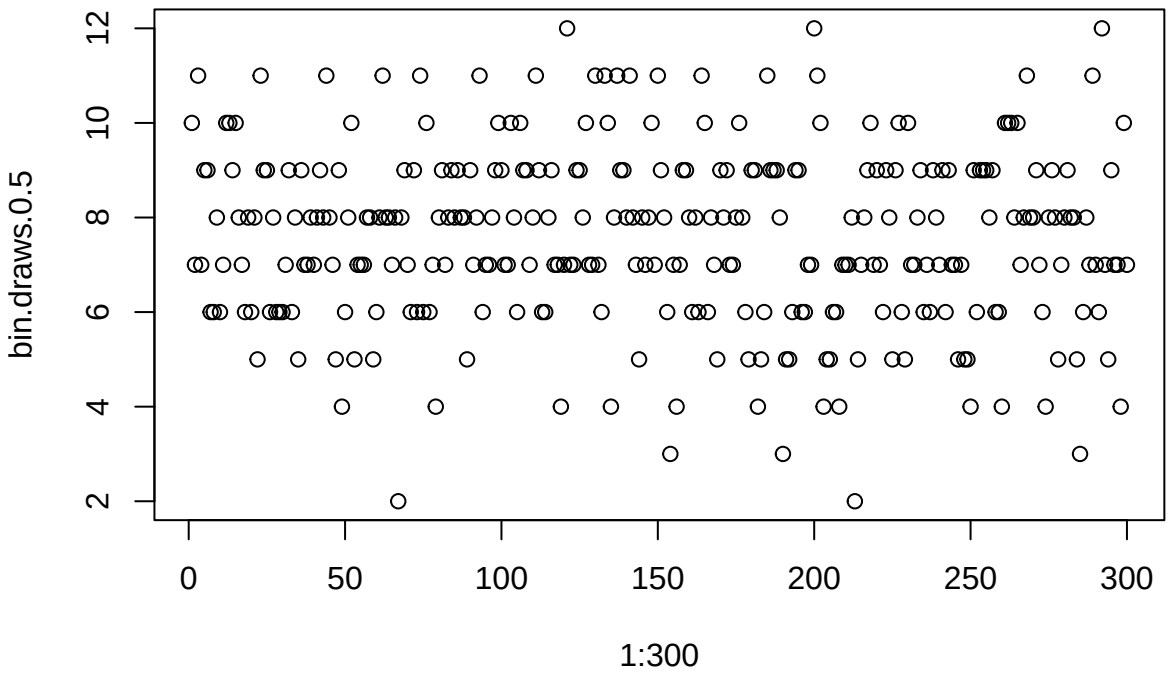
```
plot(bin.draws.0.5)
```




This function plots the result of each binomial trial in order from 1 to 300.

- **2d.** Call `plot()` with two arguments, the first being `1:300`, and the second being `bin.draws.0.5`. This creates a scatterplot of `bin.draws.0.5` (on the y-axis) versus the indices 1 through 300 (on the x-axis). Does this match your plot from the last question?

```
plot(1:300, bin.draws.0.5)
```




This matches the results from the previous plot.

## More binomials, more plots

- **3a.** Generate 300 binomials again, composed of 15 trials each, but change the probability of success to: 0.2, 0.3, 0.4, 0.6, 0.7, and 0.8, storing the results in vectors called `bin.draws.0.2`, `bin.draws.0.3`, `bin.draws.0.4.`, `bin.draws.0.6`, `bin.draws.0.7` and `bin.draws.0.8`. For each, compute the mean and standard deviation.

```
bin.draws.0.2 <- rbinom(n=300, 15, 0.2)
mean(bin.draws.0.2)
```

```
## [1] 2.963333
```

```
sd(bin.draws.0.2)
```

```
## [1] 1.517563
```

```
bin.draws.0.3 <- rbinom(n=300, 15, 0.3)
mean(bin.draws.0.3)
```

```
## [1] 4.373333
```

```
sd(bin.draws.0.3)
```

```
## [1] 1.707926
```

```
bin.draws.0.4 <- rbinom(n=300, 15, 0.4)
mean(bin.draws.0.4)
```

```
## [1] 5.933333
```

```
sd(bin.draws.0.4)
```

```
## [1] 1.995536
```

```
bin.draws.0.6 <- rbinom(n=300, 15, 0.6)
mean(bin.draws.0.6)
```

```
## [1] 9.043333
```

```
sd(bin.draws.0.6)
```

```
## [1] 1.758341
```

```
bin.draws.0.7 <- rbinom(n=300, 15, 0.7)
mean(bin.draws.0.7)
```

```
## [1] 10.56
```

```
sd(bin.draws.0.7)
```

```
## [1] 1.71743
```

```
bin.draws.0.8 <- rbinom(n=300, 15, 0.8)
mean(bin.draws.0.8)
```

```
## [1] 12.05
```

```
sd(bin.draws.0.8)
```

```
## [1] 1.499442
```

- **3b.** We'd like to compare the properties of our vectors. Create a vector of length 7, whose entries are the means of the 7 vectors we've created, in order according to the success probabilities of their underlying binomial distributions (0.2 through 0.8).

```
means <- c(mean(bin.draws.0.2), mean(bin.draws.0.3), mean(bin.draws.0.4), mean(bin.draws.0.5), mean(bin
sds <- c(sd(bin.draws.0.2), sd(bin.draws.0.3), sd(bin.draws.0.4), sd(bin.draws.0.5), sd(bin.draws.0.6),
```

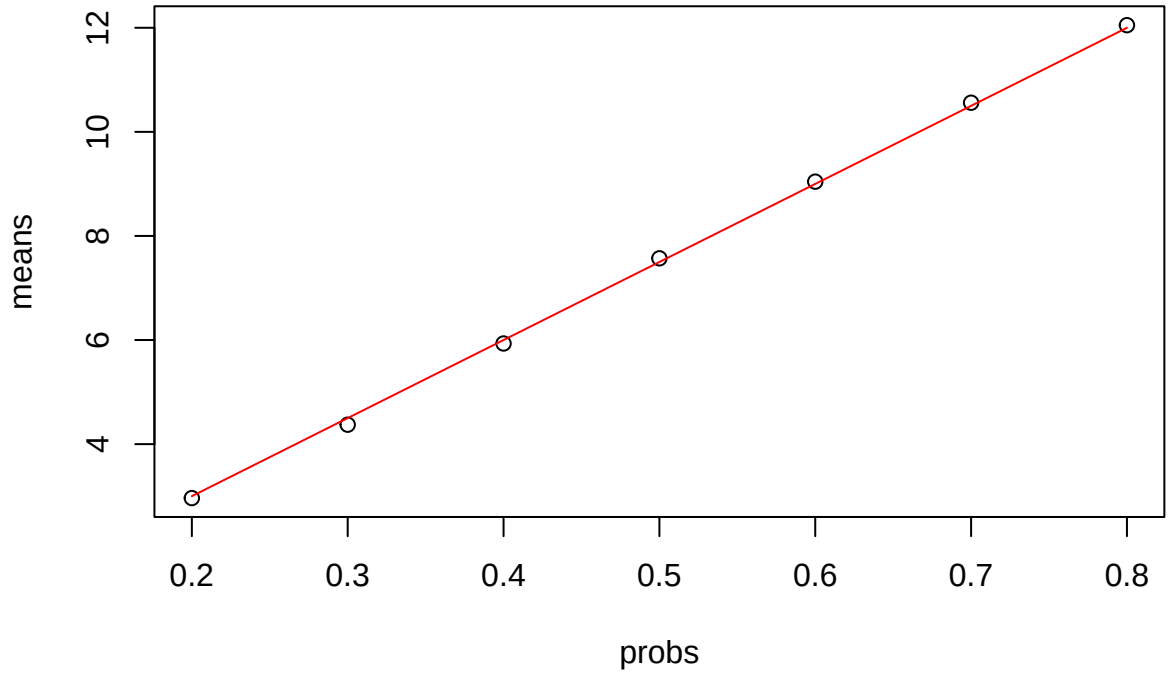
- **3c.** Using the vectors from the last part, create the following scatterplots. Explain in words, for each, what's going on.
  - The 7 means versus the 7 probabilities used to generate the draws.
  - The standard deviations versus the probabilities.
  - The standard deviations versus the means.

  **Challenge**: for each plot, add a curve that corresponds to the relationships you'd expect to see in the theoretical population (i.e., with an infinite amount of draws, rather than just 300 draws).

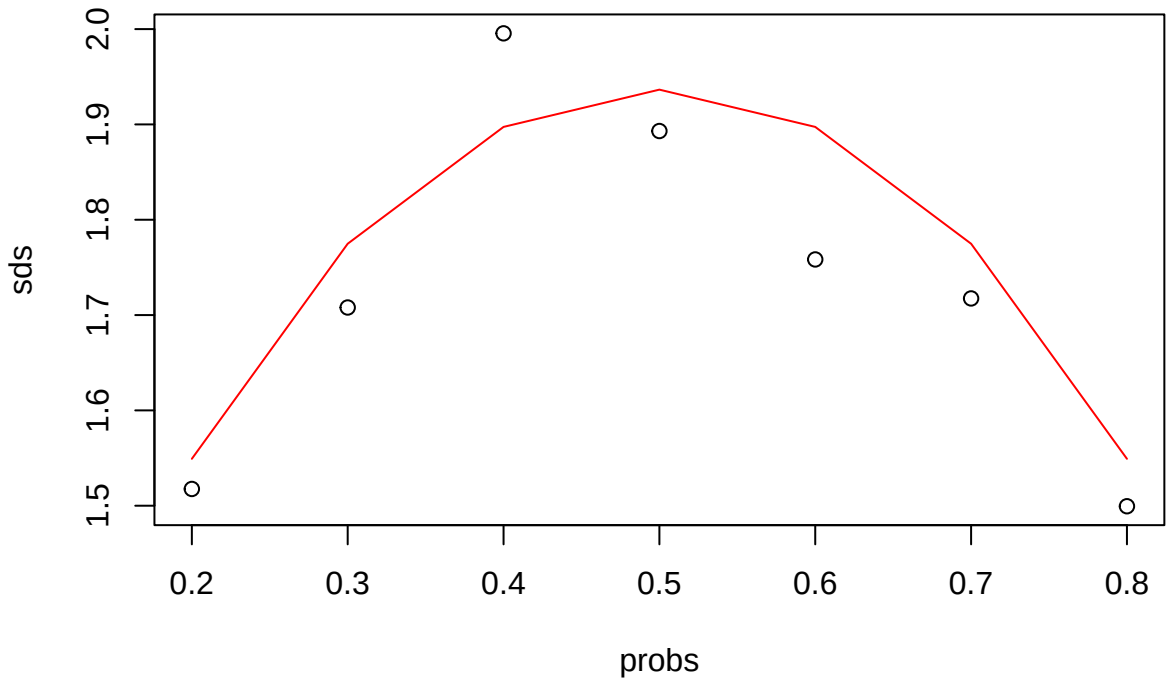
```
probs <- c()
for(i in 2:8){
  probs <- c(probs, i/10)
}

mean_line <- probs * 15
sd_line <- c()
for(i in 2:8){
  sd_line <- c(sd_line, sqrt(15 * (i/10) * (1-i/10)))
}
```

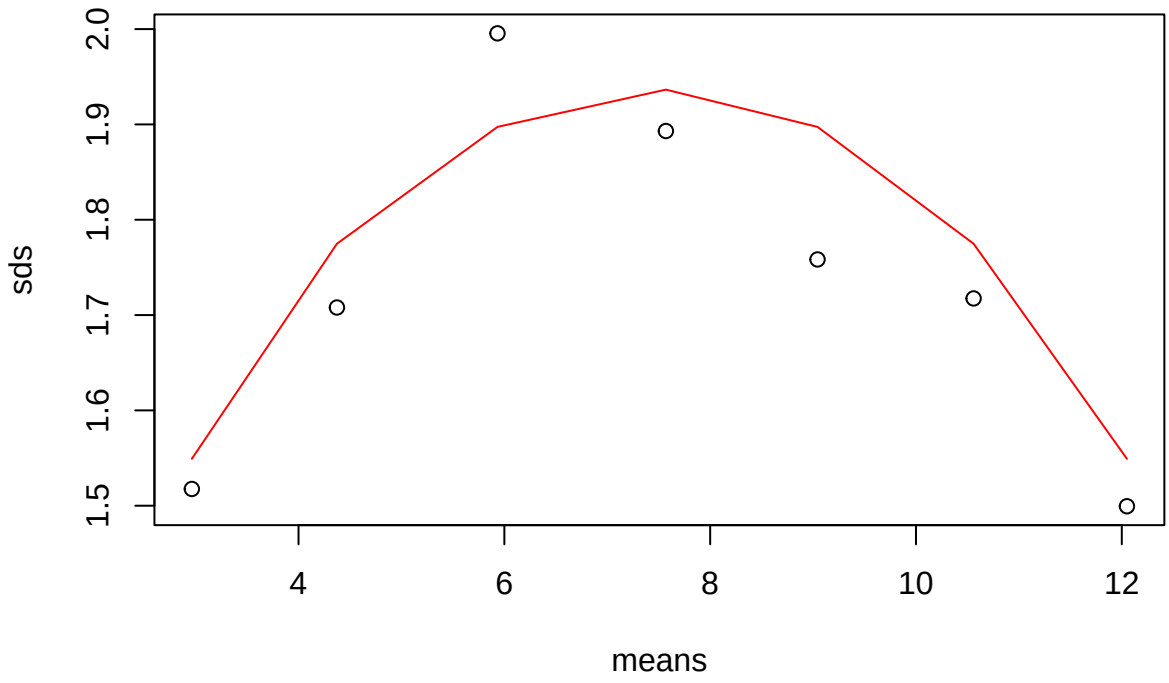
```
plot(probs, means)
lines(probs, mean_line, col="red")
```




```
plot(probs, sds)
lines(probs, sd_line, col="red")
```




```
plot(means, sds)
lines(means, sd_line, col="red")
```




As the probability increases, the mean linearly incrases. As the probability increases, the standard deviation increases and peaks at around 0.5, then decreases. The standard deviation has the same relationship with means as it has with probabilities.

## Working with matrices

- **4a.** Create a matrix of dimension 300 x 7, called `bin.matrix`, whose columns contain the 7 vectors we've created, in order of the success probabilities of their underlying binomial distributions (0.2 through 0.8). Hint: use `cbind()`.

```
bin.matrix <-  matrix(bin.draws.0.2)
bin.matrix <- cbind(bin.matrix, bin.draws.0.3)
bin.matrix <- cbind(bin.matrix, bin.draws.0.4)
bin.matrix <- cbind(bin.matrix, bin.draws.0.5)
bin.matrix <- cbind(bin.matrix, bin.draws.0.6)
bin.matrix <- cbind(bin.matrix, bin.draws.0.7)
bin.matrix <- cbind(bin.matrix, bin.draws.0.8)
colnames(bin.matrix) <- probs
```

- **4b.** Print the first five rows of `bin.matrix`. Print the element in the 66th row and 5th column. Compute the largest element in first column. Compute the largest element in all but the first column.

```
bin.matrix[1:5,]
```

```
##      0.2 0.3 0.4 0.5 0.6 0.7 0.8
## [1,]   2   6   5  10   6  10  11
```

```
## [2,]   3   4   5   7  12   9  14
## [3,]   0   7   6  11   7   9  12
## [4,]   3   5   6   7   9  11  12
## [5,]   2   2   9   9   8   9  10
```

```
bin.matrix[66,5]
```

```
## 0.6
##   9
```

```
max(bin.matrix[,-1])
```

```
## [1] 15
```

- **4c.** Calculate the column means of `bin.matrix` by using just a single function call.

```
colMeans(bin.matrix)
```

```
##       0.2       0.3       0.4       0.5       0.6       0.7       0.8
##  2.963333  4.373333  5.933333  7.570000  9.043333 10.560000 12.050000
```

- **4d.** Compare the means you computed in the last question to those you computed in Q3b, in two ways. First, using `==`, and second, using `identical()`. What do the two ways report? Are the results compatible? Explain.

```
means == colMeans(bin.matrix)
```

```
##  0.2  0.3  0.4  0.5  0.6  0.7  0.8
## TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

```
identical(means, colMeans(bin.matrix))
```

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

- **4e.** Take the transpose of `bin.matrix` and then take row means. Are these the same as what you just computed? Should they be?

```
rowMeans(t(bin.matrix))
```

```
##       0.2       0.3       0.4       0.5       0.6       0.7       0.8
##  2.963333  4.373333  5.933333  7.570000  9.043333 10.560000 12.050000
```

They are the same and should be. Transposing turns the columns into the rows, so taking the rowmean will yield the old column means.

## Warm up is over, let's go big

- **5a.** R's capacity for data storage and computation is very large compared to what was available 10 years ago. Generate 3 million numbers from Bin$(1 \times 10^6, 0.5)$ distribution and store them in a vector called `big.bin.draws`. Calculate the mean and standard deviation of this vector.

```
big.bin.draws <- rbinom(3*10^6, 10^6, 0.5)
mean(big.bin.draws)
```

```
## [1] 499999.8
```

```
sd(big.bin.draws)
```

```
## [1] 500.0116
```

- **5b.** Create a new vector, called `big.bin.draws.standardized`, which is given by taking `big.bin.draws`, subtracting off its mean, and then dividing by its standard deviation. Calculate the mean and standard deviation of `big.bin.draws.standardized`. (These should be 0 and 1, respectively, or very, very close to it; if not, you've made a mistake somewhere).

```
big.bin.draws.standardized <- (big.bin.draws - mean(big.bin.draws))/sd(big.bin.draws)
mean(big.bin.draws.standardized)
```

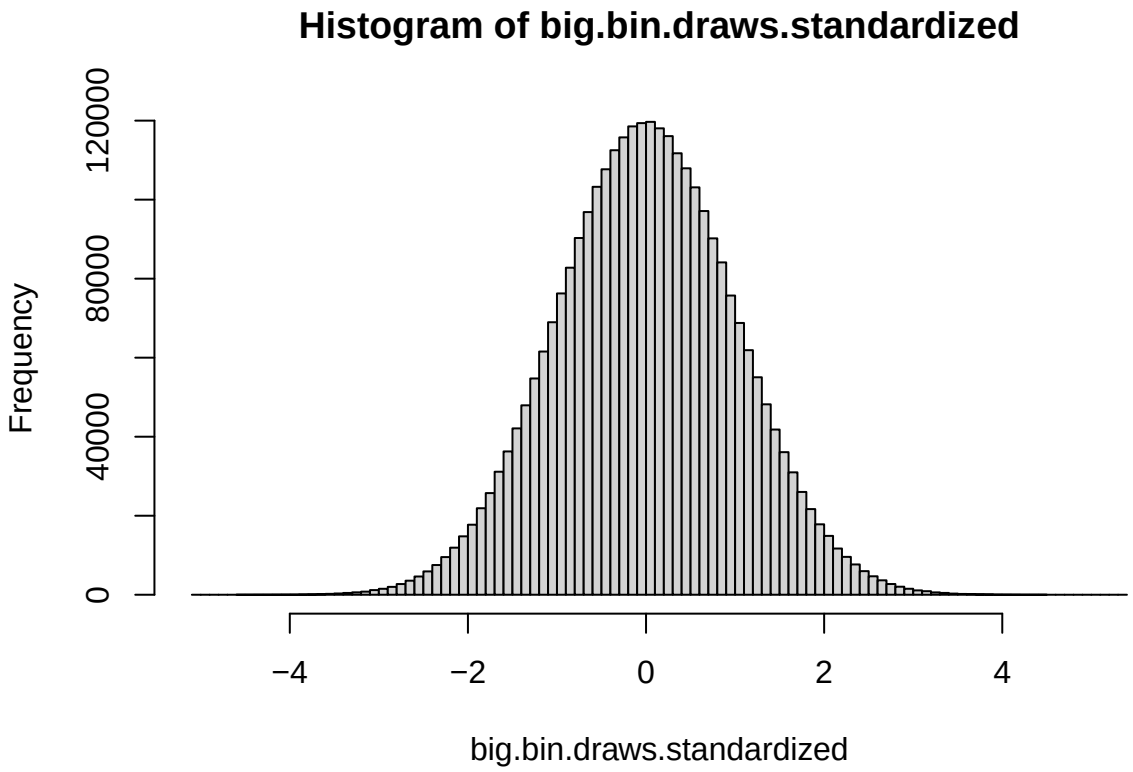
```
## [1] -4.236492e-14
```

```
sd(big.bin.draws.standardized)
```

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

- **5c.** Plot a histogram of `big.bin.draws.standardized`. To increase the number of histogram bars, set the `breaks` argument in the `hist()` function (e.g., set `breaks=100`). What does the shape of this histogram appear to be? Is this surprising? What could explain this phenomenon? Hint: rhymes with "Mental Gimmick Serum" ...

```
hist(big.bin.draws.standardized, breaks=100)
```




The shape of this histogram appears to be a bell curve. This isn't surprising because repeated samples from a distribution are normally distributed because of the Central Limit Theorem.

- **5d.** Calculate the proportion of times that an element of `big.bin.draws.standardized` exceeds 1.644854. Is this close to 0.05?

```
sum((big.bin.draws.standardized>1.644854))/(3*10^6)
```

```
## [1] 0.049991
```

This is reasonably close to 0.05.

- **5e.** Either by simulation, or via a built-in R function, compute the probability that a standard normal random variable exceeds 1.644854. Is this close to 0.05? Hint: for either approach, it would be helpful to look at the help file for the `rnorm()` function.

```
rn <- rnorm(3*10^6)
sum(rn>1.644854)/(3*10^6)
```

```
## [1] 0.05006333
```

This is pretty close to 0.05.

## Now let's go really big

- **6a.** Let's push R's computational engine a little harder. Design an expression to generate 100 million numbers from $\text{Bin}(10 \times 10^6, 50 \times 10^{-8})$, to be saved in a vector called `huge.bin.draws`, but do not evaluate this command yet. Then ask your lab partner to name three of Justin Bieber's songs and simultaneously evaluate your R command that defines `huge.bin.draws`. Which finished first, R or your partner? (Note: your partner cannot really win this game. Even if he/she finishes first, he/she still loses.)

```
t1 <- Sys.time()
huge.bin.draws <- rbinom(100*10^6, 10*10^6, 50*10^-8)
t2 <- Sys.time()
t2-t1
```

```
## Time difference of 11.84663 secs
```

R finished first.

- **6b.** Calculate the mean and standard deviation of `huge.bin.draws`. Are they close to what you'd expect? (They should be very close.) Did did longer to compute these, or to generate `huge.bin.draws` in the first place?

```
t1 <- Sys.time()
mean(huge.bin.draws)
```

```
## [1] 4.999943
```

```
sd(huge.bin.draws)
```

```
## [1] 2.235891
```

```
t2 <- Sys.time()
t2-t1
```

```
## Time difference of 1.401701 secs
```

The outputs are very close to the expected results of 5 and 2.23. It took longer to generate huge.bin.draws.

- **6c.** Calculate the median of `huge.bin.draws`. Did this median calculation take longer than the calculating the mean? Is this surprising?

```
t1 <- Sys.time()
median(huge.bin.draws)
```

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

```
t2 <- Sys.time()
t2-t1
```

```
## Time difference of 2.483857 secs
```

The median calculation took longer than calculating the mean. This isn't surprising because the median is calculated in *nlogn* time whereas the mean is calculated in linear time.

- **6d.** Calculate the exponential of the median of the logs of `huge.bin.draws`, in one line of code. Did this take longer than the median calculation applied to `huge.bin.draws` directly? Is this surprising?

```
t1 <- Sys.time()
exp(median(log(huge.bin.draws)))
```

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

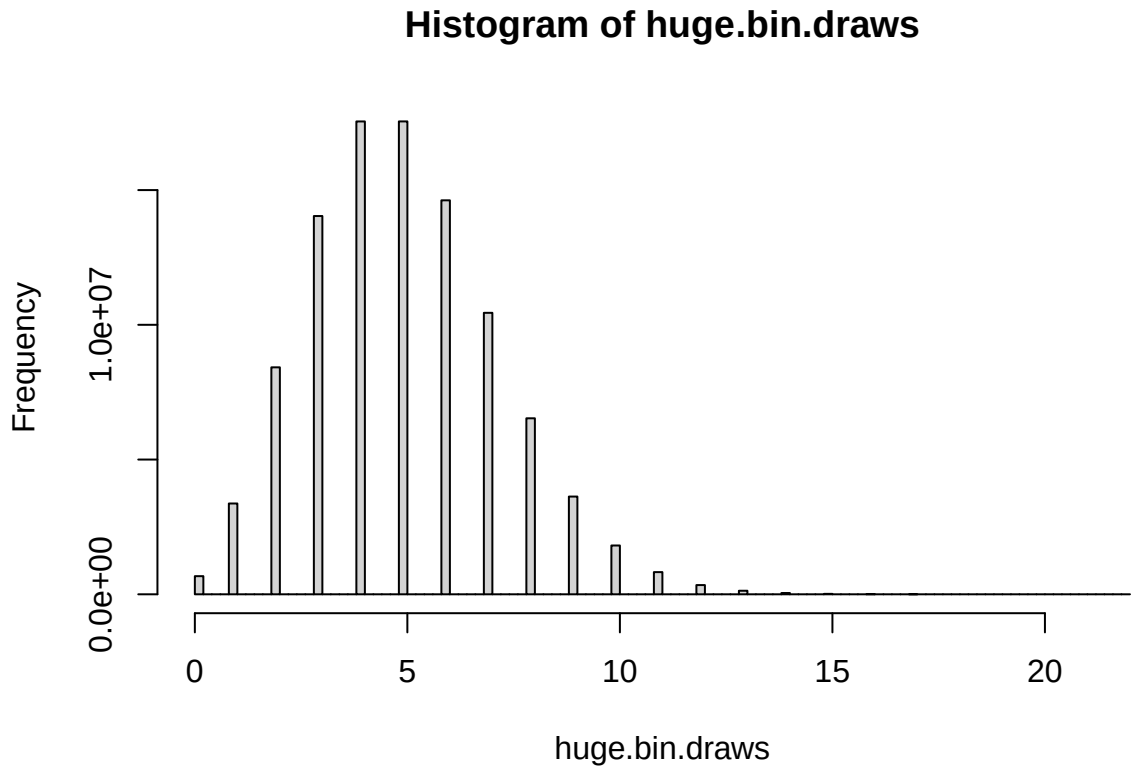
```
t2 <- Sys.time()
t2-t1
```

```
## Time difference of 7.225735 secs
```

This took longer than the direct median calculation. This isn't surprising because R has to perform two more actions on each element of the list.

- **6e.** Plot a histogram of of `huge.bin.draws`, again with a large setting of the `breaks` argument (e.g., `breaks=100`). Describe what you see; is this different from before, when we had 3 million draws? **Challenge**: Is this surprising? What distribution is this?

```
hist(huge.bin.draws, breaks=100)
```




This is different from above when we had fewer draws. This change isn't necessarily exciting because the value of the distribution stops in one direction at zero, but can expend much longer on the other direction as we add additional trials.

## Going big with lists

- **7a.** Convert `big.bin.draws` into a list using `as.list()` and save the result as `big.bin.draws.list`. Check that you indeed have a list by calling `class()` on the result. Check also that your list has the right length, and that its 1159th element is equal to that of `big.bin.draws`.

```
big.bin.draws.list <- as.list(big.bin.draws)
class(big.bin.draws.list)
```

```
## [1] "list"
```

```
length(big.bin.draws.list)
```

```
## [1] 3000000
```

```
big.bin.draws.list[1159] == big.bin.draws[1159]
```

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

- **7b.** Run the code below, to standardize the binomial draws in the list `big.bin.draws.list`. Note that `lapply()` applies the function supplied in the second argument to every element of the list supplied in the first argument, and then returns a list of the function outputs. (We'll learn much more about the `apply()` family of functions later in the course.) Did this `lapply()` command take longer to evaluate than the code you wrote in Q5b? (It should have; otherwise your previous code could have been improved, so go back and improve it.) Why do you think this is the case?

```
big.bin.draws.mean = mean(big.bin.draws)
big.bin.draws.sd = sd(big.bin.draws)
standardize = function(x) {
  return((x - big.bin.draws.mean) / big.bin.draws.sd)
}
big.bin.draws.list.standardized.slow = lapply(big.bin.draws.list, standardize)
```

CBA'd going back and setting up timer, but this one obviously applies a function to each element of the list instead of doing element-wise calculations on the vectors, so the computer uses significantly more overhead to complete this calculation.

- **7c.** Run the code below, which again standardizes the binomial draws in the list `big.bin.draws.list`, using `lapply()`. Why is it so much slower than the code in the last question? Think about what is happening each time the function is called.

```
big.bin.draws.mean = mean(big.bin.draws)
big.bin.draws.sd = sd(big.bin.draws)
standardize.slow = function(x) {
  return((x - mean(big.bin.draws)) / sd(big.bin.draws))
}
big.bin.draws.list.standardized.slow = lapply(big.bin.draws.list, standardize.slow)
```

This one is even worse because each time the function is called, it has to recalculate the mean and standard deviation of big.bin.draws instead of calculating them once and saving them as constants.

- **7d.** Lastly, let's look at memory useage. The command `object.size(x)` returns the number of bytes used to store the object `x` in your current R session. Find the number of bytes used to store `big.bin.draws` and `big.bin.draws.list`. How many megabytes (MB) is this, for each object? Which object requires more memory, and why do you think this is the case? Remind yourself: why are lists special compared to vectors, and is this property important for the current purpose (storing the binomial draws)?

```
object.size(big.bin.draws)
```

```
## 12000048 bytes
```

```
object.size(big.bin.draws.list)
```

```
## 192000048 bytes
```

The vector requires 12 MB, whereas the list requires 192 MB. The list requires more memory because it needs to allow for each element to be a different datatype. This capability isn't required for this problem because we only store integer values in our data structure. so we should obviously use vectors in this scenario.