Week 6 - Introduction to Simulations

In the previous weeks, data and control structures were discussed in great detail. This week begins an examination of simulations. Using R for statistical simulations allows the student to experience the real power of this scripting language.

Weekly Learning Objectives

1. Given a standard deck of 52 playing cards, create a simulation to calculate the probability of getting and ace.
2. Create a two simulations of function to perform coin toss results (binary). Use the sample() function for one and the rbinom() for the other.
3. Create a function to calculate the probability for a discrete distribution.
4. Create a probability and a distribution. Build a function that will determine the corresponding quantile for that probability.
5. Build a function that will calculate a z-score and form a confidence interval for a mean of a population.
6. Using the simulation you suggested in the discussions, create an R program for that simulation. Include graphs and statistical information.

```
library(ggplot2)
```

1. Probability of Ace - Card Simulation

It is based on the fact that a deck will have four aces in it. We are going to assign the numbers 13, 26, 39, and 52 the values to be aces in the simulation. Then it becomes a case of using the modulus operator. As you will remember from our earlier exercise where you encountered it, it will let you know if the number is divisible by the value (X %% 10 will equal 0 if X is a multiple of 10). Testing against one of the multiples of 13 will confirm that you have found an ace. Once you have that function established, then you simply have to loop through it as many attempts as you request and keep track of the total count.

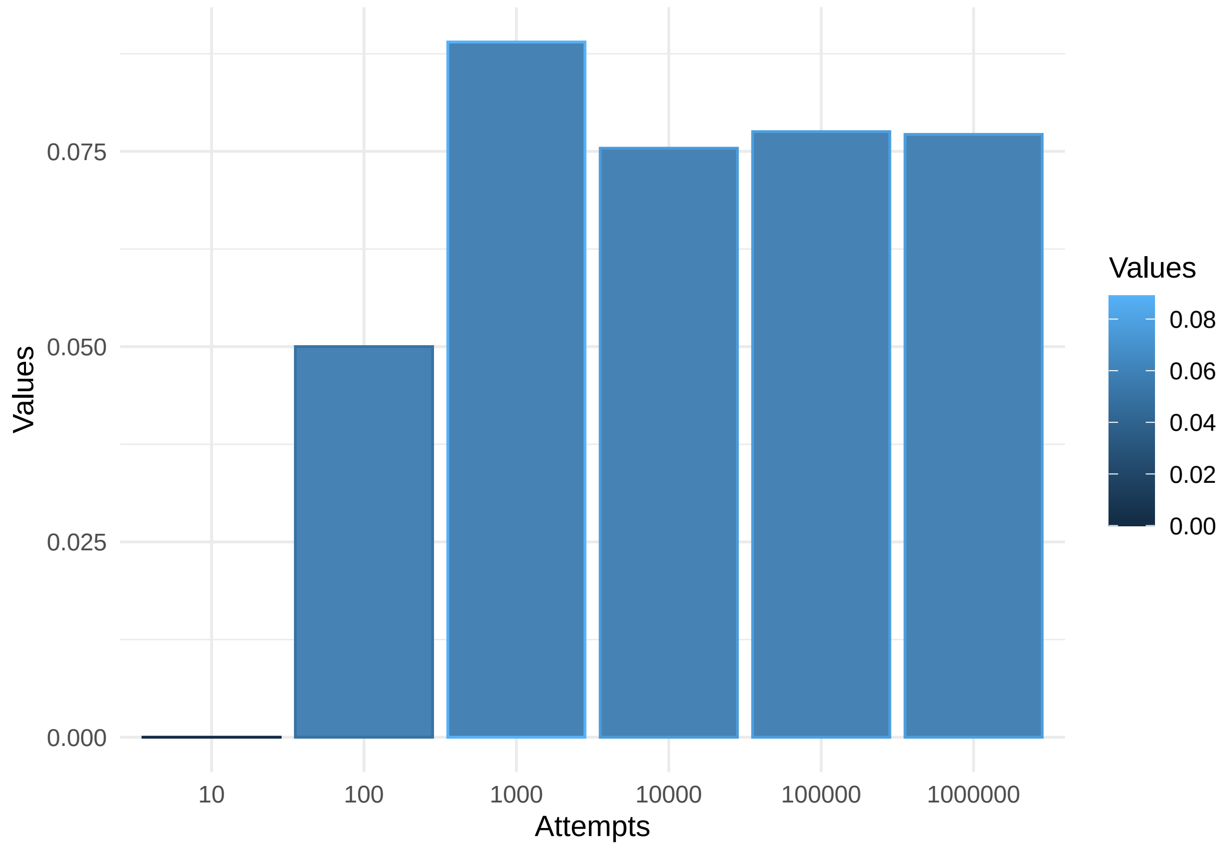
- drawAce is base function to determine if something is an ace
- drawAces is the wrapper

Full Points For full points you need to write a similar function. It is okay if you use the same multiples of 13 rule, but your function should not be a straight copy from this one. A graph is not required but nice to have.

```
drawAces <- function(attempts) {       # Wrapper function

  drawAce <- function(){               # Inner function

    foundAce = FALSE                   # Setup variable
    aCard <- sample(1:52,1)            # Create deck and pick a card
    if ((aCard %% 13) == 0) {          # Cards 13,26,39,52 are aces (all of these are modulus %% of 13)
      foundAce <- TRUE
    }
    return (foundAce)
  }

  totalAces <- 0                       # Initialize ace counter
  for (i in 1:attempts){                    # Simple for loop to attempts
    aResult <- drawAce()                    # Draw a card (notice "inner" function)
    if(aResult == TRUE) {
      totalAces <- totalAces+1
    }
  }
  return(totalAces/attempts)
}
```

Testing Ace Card Simulation Let's setup a test using powers of 10. I'm going to store it in a data frame, so afterwards I can graph it out and print the data frame.

```
# The greater number of attempts, the closer the simulation results calculate the probability
df <- data.frame(Attempts=c("10", "100", "1000","10000","100000","1000000"), Values=
        c(drawAces(10),
        drawAces(100),
        drawAces(1000),
        drawAces(10000),
        drawAces(100000),
        drawAces(100000)))

ggplot(data=df, aes(x=Attempts, y=Values, color=Values)) +
      geom_bar(stat="identity", fill="steelblue")+
      theme_minimal()
```



```r
df     # Let's print out the data frame so we can see it
```

```
##   Attempts  Values
## 1       10 0.00000
## 2      100 0.05000
## 3     1000 0.08900
## 4    10000 0.07540
## 5   100000 0.07752
## 6  1000000 0.07717
```

2. Coin Toss Simulations (sample() and rbinom())

The general syntax for sample is:

sample(x, size)

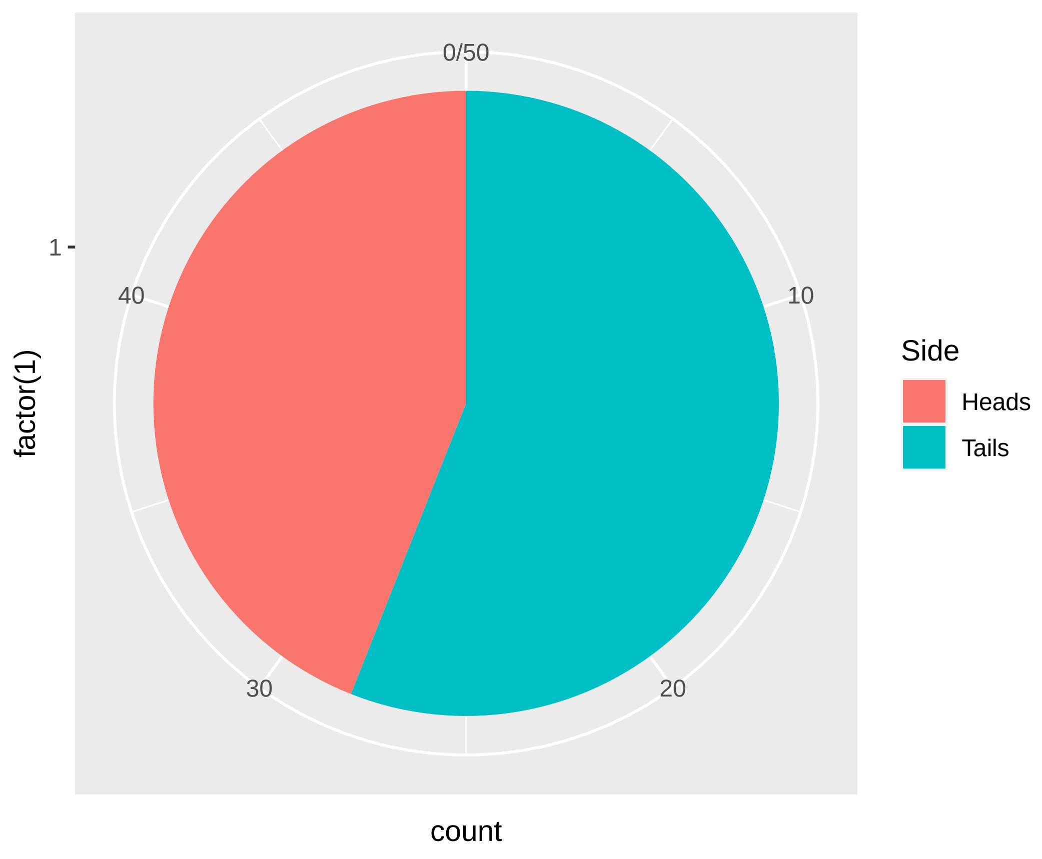
where x = vector of one ore more elements and size is a non-negative integer to give the number of items to choose.

Please remember to make these **functions**.

Full Points For full points you should write two functions. One using sample() and the other using rbinom().

Demonstration: First let's do a simple one using sample() and plot it on a pie chart. Assuming we flip the coin fifty times. HINT this can be rewritten in a function of three lines.

```r
datalist = list()
for (i in 1:50) {
  dat <- data.frame(Value = sample(0:1,1))
  if(dat == 0){
    dat$Side = "Heads"
  }
  else{
    dat$Side = "Tails"
  }
  datalist[[i]] <- dat # add it to your list
}
df = do.call(rbind, datalist)
ggplot(df, aes(x=factor(1), fill=Side))+
  geom_bar(width = 1)+
  coord_polar("y")
```



```
df    # Let's print out the data frame so we can see it
```

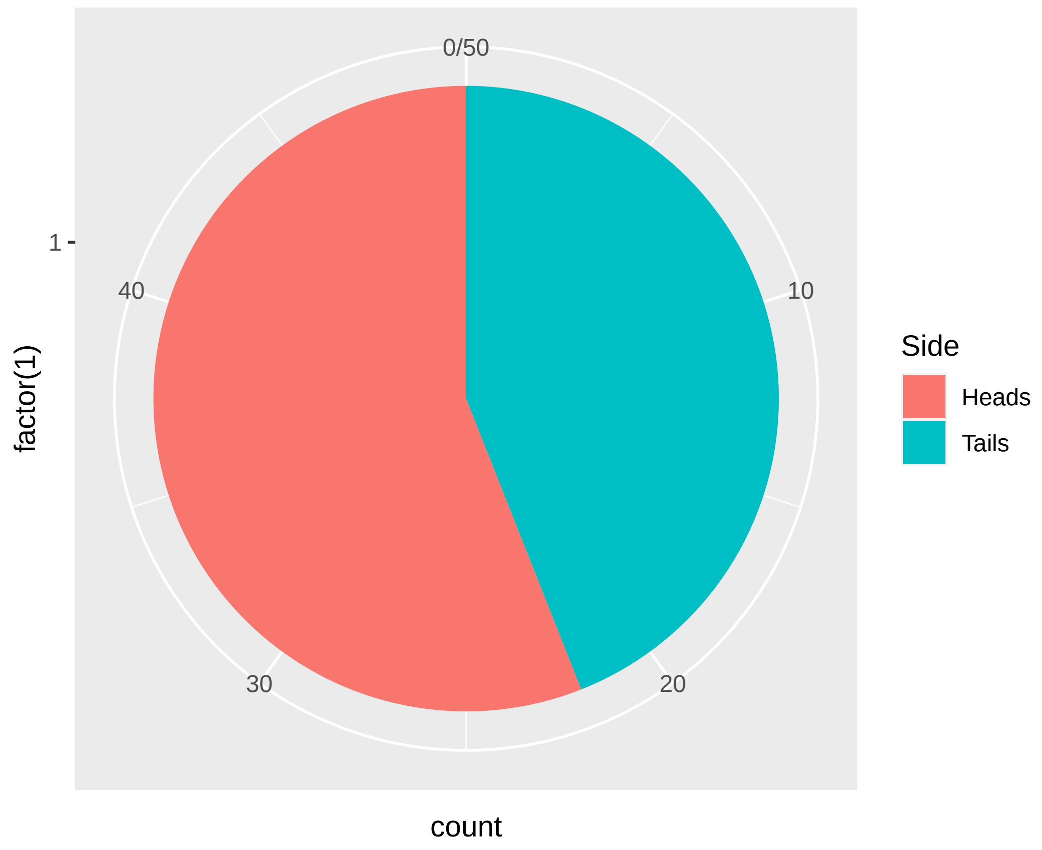
```
##    Value  Side
## 1      0 Heads
## 2      1 Tails
## 3      0 Heads
## 4      0 Heads
## 5      0 Heads
## 6      0 Heads
## 7      1 Tails
## 8      0 Heads
## 9      0 Heads
## 10     0 Heads
## 11     1 Tails
## 12     0 Heads
## 13     1 Tails
## 14     1 Tails
## 15     1 Tails
## 16     0 Heads
## 17     1 Tails
## 18     1 Tails
## 19     0 Heads
## 20     1 Tails
## 21     0 Heads
## 22     1 Tails
## 23     1 Tails
## 24     1 Tails
## 25     1 Tails
## 26     0 Heads
## 27     1 Tails
## 28     0 Heads
## 29     0 Heads
## 30     1 Tails
## 31     1 Tails
## 32     0 Heads
## 33     1 Tails
## 34     1 Tails
## 35     1 Tails
## 36     1 Tails
## 37     0 Heads
## 38     0 Heads
## 39     1 Tails
## 40     1 Tails
## 41     1 Tails
## 42     0 Heads
## 43     1 Tails
## 44     0 Heads
## 45     1 Tails
## 46     0 Heads
## 47     1 Tails
## 48     1 Tails
## 49     1 Tails
## 50     0 Heads
```

Now let's try rbinom(). The general form is:

rbinom(n, size, prob)

Where n is the number of observations, size is the number of trials (zero or more), and prob is the probability of success on each trial. Let's assume a probability of 0.5 per flip. HINT this can also be rewritten in a function of three lines.

```r
datalist = list()
for (i in 1:50) {
  dat <- data.frame(Value = rbinom(1,size=1,prob=0.5))
  if(dat == 0){
    dat$Side = "Heads"
  }
  else{
    dat$Side = "Tails"
  }
  datalist[[i]] <- dat # add it to your list
}
df = do.call(rbind, datalist)
ggplot(df, aes(x=factor(1), fill=Side))+
  geom_bar(width = 1)+
  coord_polar("y")
```



```r
df       # Let's print out the data frame so we can see it
```

```
##    Value  Side
## 1      1 Tails
## 2      0 Heads
## 3      1 Tails
## 4      0 Heads
## 5      1 Tails
## 6      1 Tails
## 7      1 Tails
## 8      0 Heads
## 9      0 Heads
## 10     1 Tails
## 11     0 Heads
## 12     0 Heads
## 13     0 Heads
## 14     0 Heads
## 15     1 Tails
## 16     1 Tails
## 17     0 Heads
## 18     0 Heads
## 19     0 Heads
## 20     1 Tails
## 21     0 Heads
## 22     0 Heads
## 23     0 Heads
## 24     1 Tails
## 25     0 Heads
## 26     0 Heads
## 27     0 Heads
## 28     1 Tails
## 29     0 Heads
## 30     0 Heads
## 31     0 Heads
## 32     1 Tails
## 33     1 Tails
## 34     1 Tails
## 35     0 Heads
## 36     1 Tails
## 37     0 Heads
## 38     0 Heads
## 39     1 Tails
## 40     1 Tails
## 41     0 Heads
## 42     1 Tails
## 43     0 Heads
## 44     1 Tails
## 45     1 Tails
## 46     0 Heads
## 47     0 Heads
## 48     1 Tails
## 49     0 Heads
## 50     1 Tails
```

3. Probability of a Discrete Distribution

This is going to be based off a binomial distribution. In order to understand this part, you have to understand the dbinom() function in R. It is used to create a binomial distribution.

The generic form for dbinom()

dbinom(x, size, prob, log=false)

- x = vector of quantiles
- size = number of trials (zero or more)
- prob = probability of success of each trial

You will also need the factoral() function

For my function (myBinomialPD) we are going to use

- x = vector
- n = number of trials
- p = probability

simply as shortened notation.

The mathematical formula for creating the distribution is the core of what is returned. You may use this formula in your submission:

(factorial(n)/(factorial(x[i])*factorial(n-x[i])))*(p^x[i])*(1-p)^(n-x[i])

Full Points For full points you should write a function similar to this one. You may use the same mathematical forumula as well as the dbinom() function.

```r
myBinomialPD <- function(x,n,p) {
  aResultSet <- vector(mode='numeric',length=length(x))    # Create a result set to return
  for (i in 1:length(x)) {
     aResultSet[i] <- (factorial(n)/(factorial(x[i])*factorial(n-x[i])))*(p^x[i])*(1-p)^(n-x[i])
  }
  return(aResultSet)
}
```

Testing the discrete distribution Let's run a test on it using the parameters:

- 1 vector
- 10 trials
- 0.6 probability

First we will run the function and then we will display the distribution as a test.

```r
myBinomialPD(1,10,0.6)
```

```
## [1] 0.001572864
```

```r
# Display Binomial distribution table as a test
for (i in 0:10) {
  aResult <- myBinomialPD(i,10,0.6)
  print(aResult)
}
```

```
## [1] 0.0001048576
## [1] 0.001572864
## [1] 0.01061683
## [1] 0.04246733
## [1] 0.1114767
## [1] 0.2006581
## [1] 0.2508227
## [1] 0.2149908
## [1] 0.1209324
## [1] 0.04031078
## [1] 0.006046618
```

Now we will compare the results to the standard R function using the same parameters for both calls

- x = 0 - 10
- trials = 10
- probability = 0.6

If the function is working, myBinomialPD() should return the same results as the standard R function dbinom()

```r
myBinomialPD(c(0:10),10,0.6)
```

```
##  [1] 0.0001048576 0.0015728640 0.0106168320 0.0424673280 0.1114767360
##  [6] 0.2006581248 0.2508226560 0.2149908480 0.1209323520 0.0403107840
## [11] 0.0060466176
```

```r
dbinom(c(0:10),10,0.6)
```

```
##  [1] 0.0001048576 0.0015728640 0.0106168320 0.0424673280 0.1114767360
##  [6] 0.2006581248 0.2508226560 0.2149908480 0.1209323520 0.0403107840
## [11] 0.0060466176
```

4. Probability and Distribution with Quantile

For this we are going to use a few additional functions.

sort(x) simply does that. It sorts x

floor(x) is a R math function, which is used to return the largest integer value which is not greater than (less than) or equal to an individual number, or an expression.

names(x) <- value sets the name of an object.

quantile(x,probs, type) produces a sample quantile

- x = vector
- probs = vector of probabilities with values
- type = a number between 1 and 7 to use for the method. In this case we are sticking to a type 3

https://www.rdocumentation.org/packages/stats/versions/3.6.1/topics/quantile

Full Points For full points you should write a function similar to this one. You may use the same mathematical forumula as well as the various functions used in this demonstration.

```
myQuantile <- function(x,y){

  x <- sort(x)     # Adding in a sort() because we want consistent splits on the quantile
  aResultSet <- vector(mode='numeric',length = length(y))

  for(i in 1:length(y)){                          # Same type of loop
    aStartPos <- (y[i] * length(x))               # Set the start position
    if(aStartPos == 0) {                          # Safety catch to make sure you do not start with a
      aStartPos = 1
    }
    aResultSet[i] <- x[floor(aStartPos)]          # Calculate
  }
  resultSetNames <- y*100                         # y is used to setup the quantile splits 0, 25, 50,
  names(aResultSet) <- resultSetNames             # names() attaches that quantile split to name the o
  return(aResultSet)
}
```

Test Quantile 1 We are going to setup a test to compare the myQuantile() function against the standard r quantile function. If everyting works, the two lines should match:

```
aPD <- dbinom(c(0:5),5,0.5)
myQuantile(aPD,c(0,0.25,0.5,0.75,1.0))
```

```
##       0      25      50      75     100
## 0.03125 0.03125 0.15625 0.15625 0.31250
```

```
quantile(aPD,c(0,0.25,0.5,0.75,1.0),type=3)
```

```
##      0%     25%     50%     75%    100%
## 0.03125 0.03125 0.15625 0.15625 0.31250
```

Test Quantile 2 Just for good measure, let's check again with a slightly different one. If everyting works, the two lines should match:

```
aPD <- dbinom(c(0:15),15,0.5)
myQuantile(aPD,c(0,0.25,0.5,0.75,1.0))
```

```
##            0           25           50           75          100 
## 3.051758e-05 4.577637e-04 1.388550e-02 9.164429e-02 1.963806e-01
```

```r
quantile(aPD,c(0,0.25,0.5,0.75,1.0),type=3)
```

```
##           0%          25%          50%          75%         100% 
## 3.051758e-05 4.577637e-04 1.388550e-02 9.164429e-02 1.963806e-01
```

5. Calculate Z-Score and Confidence Interval

What is a z-score?

Simply put, a z-score is the number of standard deviations from the mean a data point is. But more technically it's a measure of how many standard deviations below or above the population mean a raw score is. A z-score is also known as a standard score and it can be placed on a normal distribution curve.

https://www.statisticshowto.datasciencecentral.com/probability-and-statistics/z-score/

To actually get it in R, one easy way is to use the scale(x) function

Full Points Write a function to calcuate z-scores, similar to what you see here:

```r
set.seed(1234)
x= sample(1:50, 100, replace=TRUE)

m = mean(x)
s = sd(x)
zscore = (x - m)/s        # z-score = (sample - mean) / standard deviation
summary(x)                # Overall summary of the sample
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##    1.00   11.50   22.00   24.23   37.00   50.00
```

```r
summary(zscore)           # Show a summary of the z-score
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
## -1.5910 -0.8719 -0.1527  0.0000  0.8746  1.7649
```

```r
sc_zscore = scale(x)[,1]  # For good measure, let's use scale() to calculate the z-score
summary(sc_zscore)        # And compare it to the summary(zscore) above.  They are identical
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
## -1.5910 -0.8719 -0.1527  0.0000  0.8746  1.7649
```

6. Create YOUR simulation (graphs and stats)

This part of the assignment has the greatest amount of freedom in the submission. You suggested a simulation in the discussion forums. Now take that simulation and code it into a R program. Include stats as part of it.

Full Points Create your simulation and make sure it executes without issues. It should perform the action that you discussed in the forum.

Demonstration Here is an inventory fullfilment simulation. This is a bit complex, but it gives you an idea of what one could look like.

This simulation seeks to show the impact of not having enough inventory in the buffer system to fulfill orders from a warehouse. A normal distribution is used to represent a standard order fulfillment time while an exponential distribution is used for buffer replenishment on missing items. Based on this information, warehouse managers can adjust inventory strategies and fulfillment times.

Two parameters are used to generate the simulation results.

- numberofOrders - number of orders to simulate
- inventoryAvailable - % of inventory that is planned to be in the product buffer

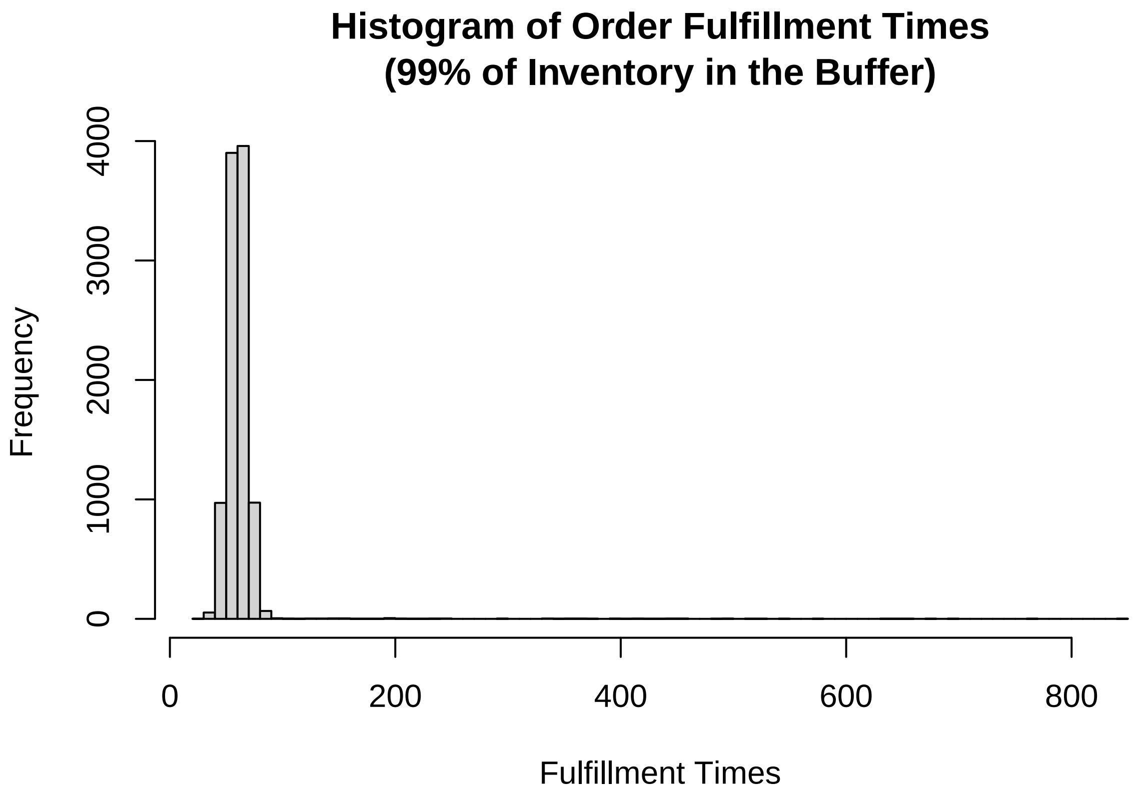
```r
inventorySimulation <- function(numberOfOrders,inventoryAvailable){

    printStats <- function (aSimResult, inventoryAvailable) {

      result<- paste("Fulfillment Time (without replenishment):\t",mean(aSimResult$fulfillTime),"\n",sep
      aReplenishment <- aSimResult$replenishTime[aSimResult$replenishTime >0]

      result<- paste(result,"Items missing from the buffer:\t\t\t",length(aReplenishment),"\n",sep="")
      result<- paste(result,"Average buffer replenishment time:\t\t",mean(aReplenishment),"\n",sep="")


      result<- paste(result,"\nOverall Fulfillment Time Summary\n",sep="")
      result<- paste(result,"Average:\t",mean(aSimResult$totalTime),"\n",sep="")
      result<- paste(result,"Median:\t\t",median(aSimResult$totalTime),"\n\n\t\tQuantile\n",sep="")

      cat(result)
      print(quantile(aSimResult$totalTime))


      hist(aSimResult$totalTime,
           main=paste("Histogram of Order Fulfillment Times\n(",inventoryAvailable*100,"% of Inventory
           xlab="Fulfillment Times",
           breaks=100)

    }

  # Randomly simulate an order availability mix
  inventoryDistribution <- runif(numberOfOrders,0,1)

  aResultSet <- data.frame(fulfillTime = numeric(numberOfOrders),replenishTime = numeric(numberOfOrders

  for (i in 1:numberOfOrders) {

    # All orders need to go through the buffer, so a base amount of time is added.  This is modeled as
    # the automated tote retrieval times are not a constant value.
    aResultSet$fulfillTime[i] <- rnorm(1,mean=60,sd=8)


    if (inventoryDistribution[i] >= inventoryAvailable) {
      # Inventory not in buffer, so time needs to be added
      aResultSet$replenishTime[i] <- rexp(1,.005)
    } else {
      #Inventory is in the buffer
      aResultSet$replenishTime[i] <- 0
    }
  }

  aResultSet$totalTime <- aResultSet$fulfillTime + aResultSet$replenishTime

  printStats(aResultSet, inventoryAvailable)
  return(aResultSet)
}
```

Testing the simulation

- Number of orders = 10,000
- Percentage in inventory (99%, 95%, 90%, and 50%)

```
sim_99 <- inventorySimulation(10000,0.99)
```

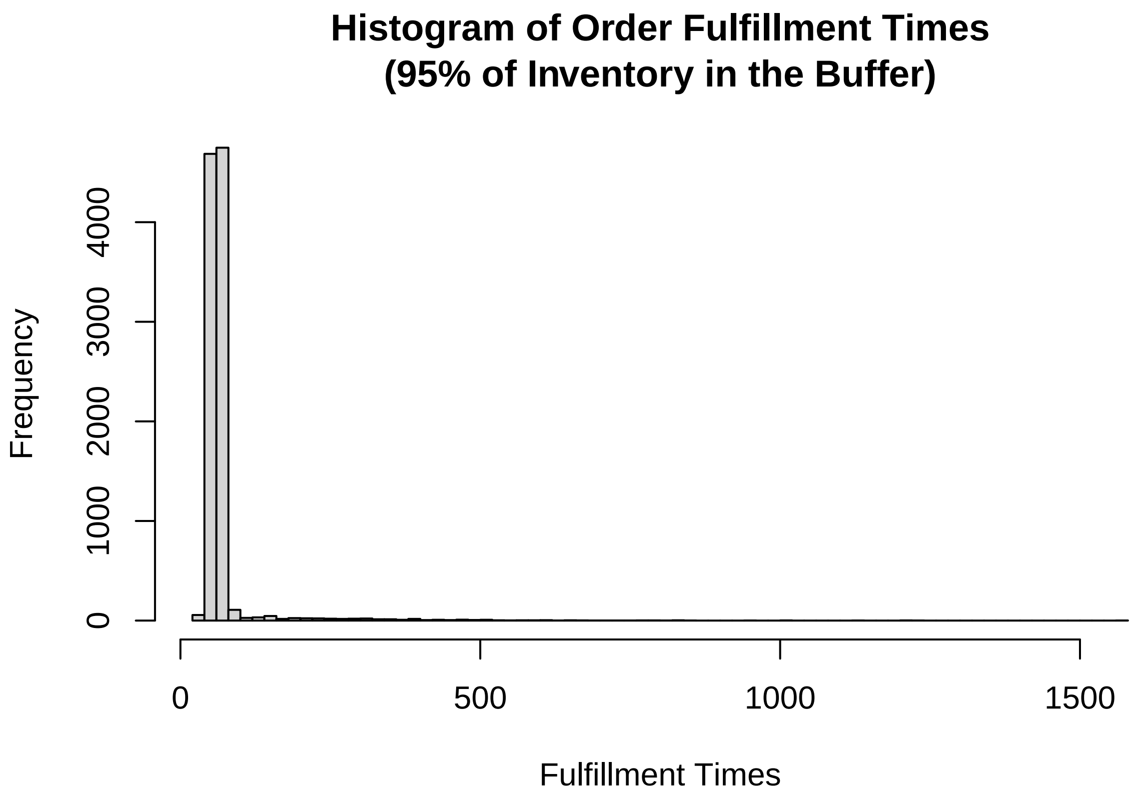
```
## Fulfillment Time (without replenishment):    60.0729383380661
## Items missing from the buffer:           90
## Average buffer replenishment time:       212.085299204299
##
## Overall Fulfillment Time Summary
## Average: 61.9817060309047
## Median:      60.1396643953484
##
##      Quantile
##        0%       25%       50%       75%      100%
##  27.98885  54.83641  60.13966  65.50779 846.90005
```



```
sim_95 <- inventorySimulation(10000,0.95)
```

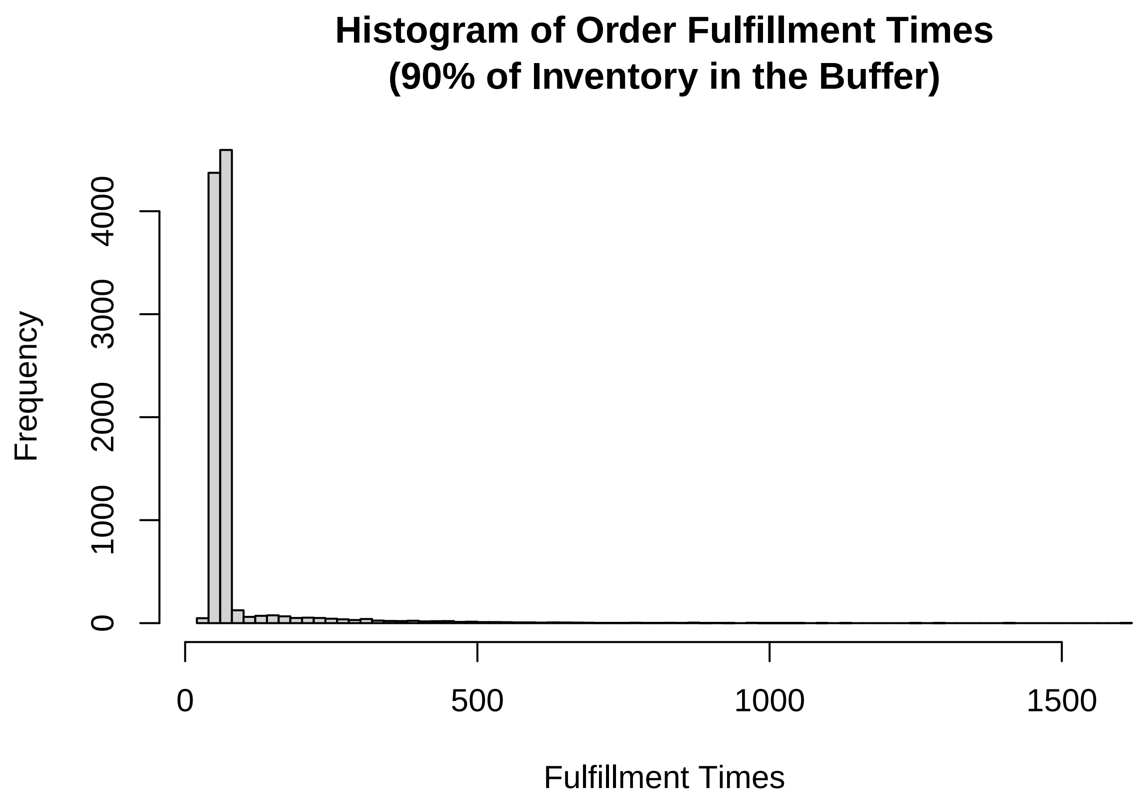
```
## Fulfillment Time (without replenishment):    60.0587521605698
## Items missing from the buffer:           495
## Average buffer replenishment time:       202.034851006736
```

```
##
## Overall Fulfillment Time Summary
## Average: 70.0594772854032
## Median:      60.5075506511991
##
##      Quantile
##         0%        25%        50%        75%       100%
##   29.81310   54.98745   60.50755   66.38783 1578.37880
```



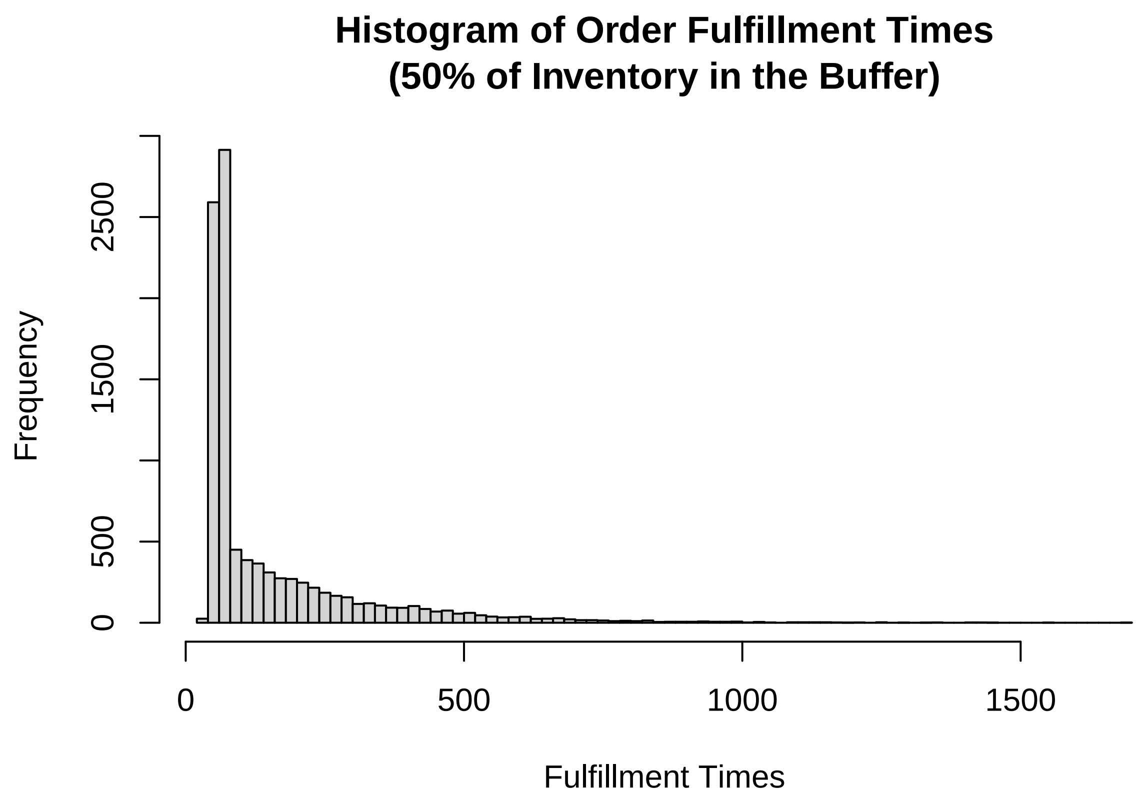
```
sim_90 <- inventorySimulation(10000,0.90)
```

```
## Fulfillment Time (without replenishment):    60.0434585472167
## Items missing from the buffer:           1054
## Average buffer replenishment time:       200.326678261363
##
## Overall Fulfillment Time Summary
## Average: 81.1578904359643
## Median:      61.2916682064258
##
##      Quantile
##         0%        25%        50%        75%       100%
##   30.87495   55.28936   61.29167   67.75163 1619.69649
```



```
sim_50 <- inventorySimulation(10000,0.50)
```

```
## Fulfillment Time (without replenishment):    59.9058616217854
## Items missing from the buffer:           4931
## Average buffer replenishment time:       196.737892020319
##
## Overall Fulfillment Time Summary
## Average: 156.917316177005
## Median:      71.576982752309
##
##      Quantile
##         0%        25%        50%        75%       100%
##   28.68152   59.61656   71.57698  193.99818 1689.07764
```



Reviewed for 2020 - MSP