ECS 132 - Project

Natalie Cowan, David Foster

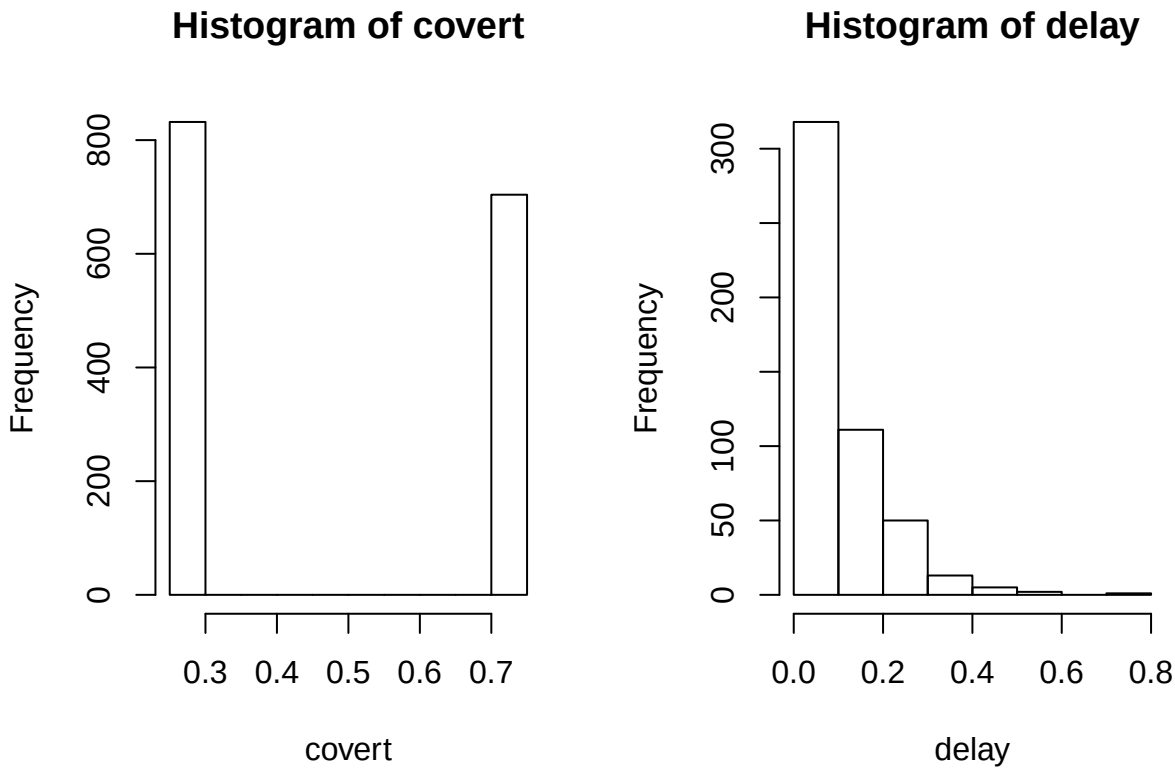
Question 1

```
library(gtools)
library(binaryLogic)

setwd("C:/Users/Natalie/Desktop/ECS132")
data <- read.csv("Traffic_data_orig.csv", header = TRUE)
delay <- numeric(length(data$Time))

for(i in 1:length(data$Time)){
   delay[i] = data$Time[i + 1] - data$Time[i]
}

letters <- asc("this is a secret message")

# create empty vector to add binary numbers
bin <- vector()

# iterate through characters in string
for(i in 1:length(letters)){
  # convert one character to binary
  x = as.binary(letters[i])

  # add padding so all characters are 8 bits
  y = fillUpToBit(x, 8, value = FALSE)

  # iterate through array of each character and add to bin
  for (j in 1:length(y)) {
  bin <- c(bin, y)
  }
}

# bits are getting translated to booleans for some reason
# so we have to iterate over them again and convert them
# back to bitst
for (i in 1:length(bin)) {
  bin[i] <- as.integer(as.logical(bin[i]))
}


# create new vector to hold converted binary message
covert <- bin

for (i in 1:length(covert)) {
  if (covert[i] == 0) {
    covert[i] <- .25
  } else if (covert[i] == 1) {
```

```
    covert[i] <- .75
  }
}
```

Question 2

```
par(mfrow=c(1,2))
hist(covert)
hist(delay)
```



Yes, Eve will be suspicious since the histogram clearly looks nothing like the original.

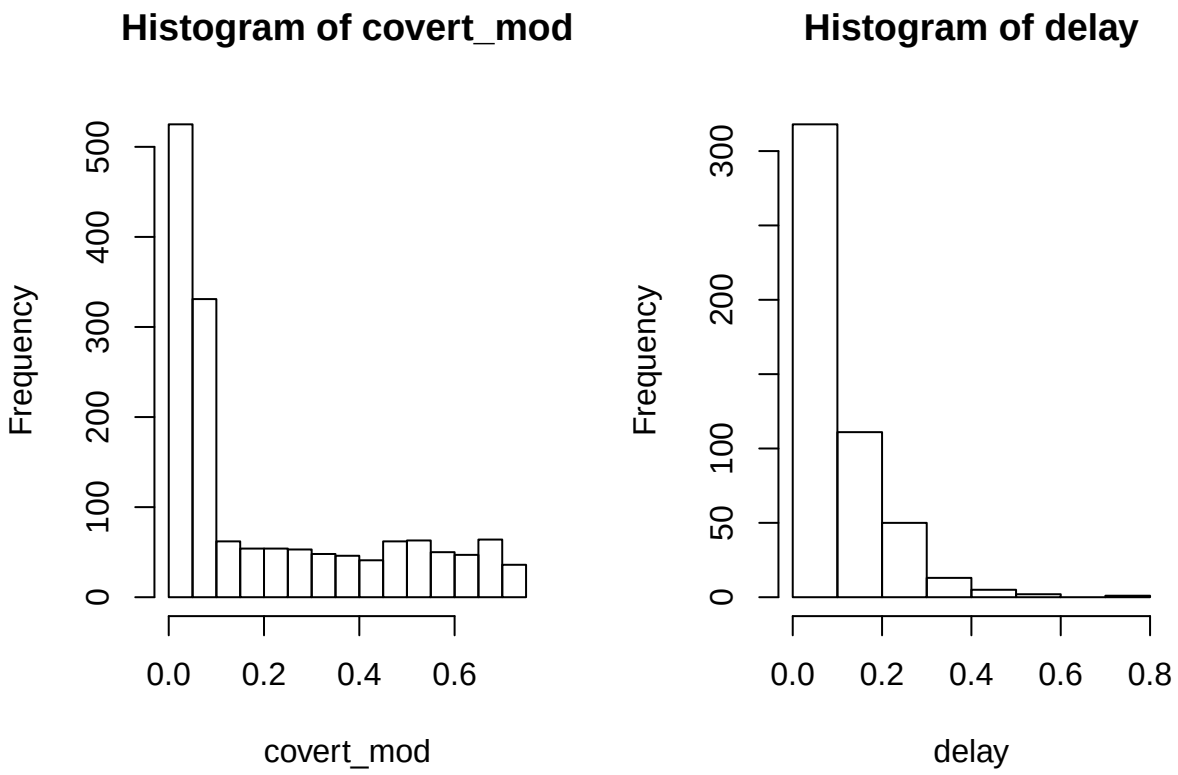
Question 3

```
# graph histograms of original and modified packet delays
min = min(delay, na.rm=TRUE)
max = max(delay, na.rm=TRUE)
m = median(delay, na.rm=TRUE)

covert_mod <- bin
```

```
for (i in 1:length(covert_mod)) {
  if (covert_mod[i] == 0) {
    covert_mod[i] = runif(1,min,m)
  } else if (covert_mod[i] == 1) {
    covert_mod[i] = runif(1,m,max)

  }
}
```

Question 4

```
par(mfrow=c(1,2))
hist(covert_mod)
hist(delay)
```



Yes, though less suspicious than previously. The graph still looks very different than the original histogram.

Question 5

Question 5.1

We can use the same method but instead of generating a random uniform delay we can generate a random delay that models an exponential distribution.

Question 5.2

When she buffers a large number of packets a queuing delay arises which causes the network quality to suffer. Additionally, time outs may occur if the server is not receiving packets within a certain timeframe.

Question 5.3

The problem with the network altering the interpacket delays is that it would make detecting the covert message difficult. One possible solution would be to use a parity "bit" to determine whether the message had been altered.

To implement a parity "bit" we would have to modify our message sending function to do the following: - Add together all the ASCII values of the string to determine the sum of all the ASCII integer values of the string - Upon sending the secret message, add the equivalent to a null-terminating character byte to the end of our string. This is for the receiver function to know when the string ends and the parity "bit" begins - After the null terminating character byte add the binary representation of the sum of the ASCII values of the string

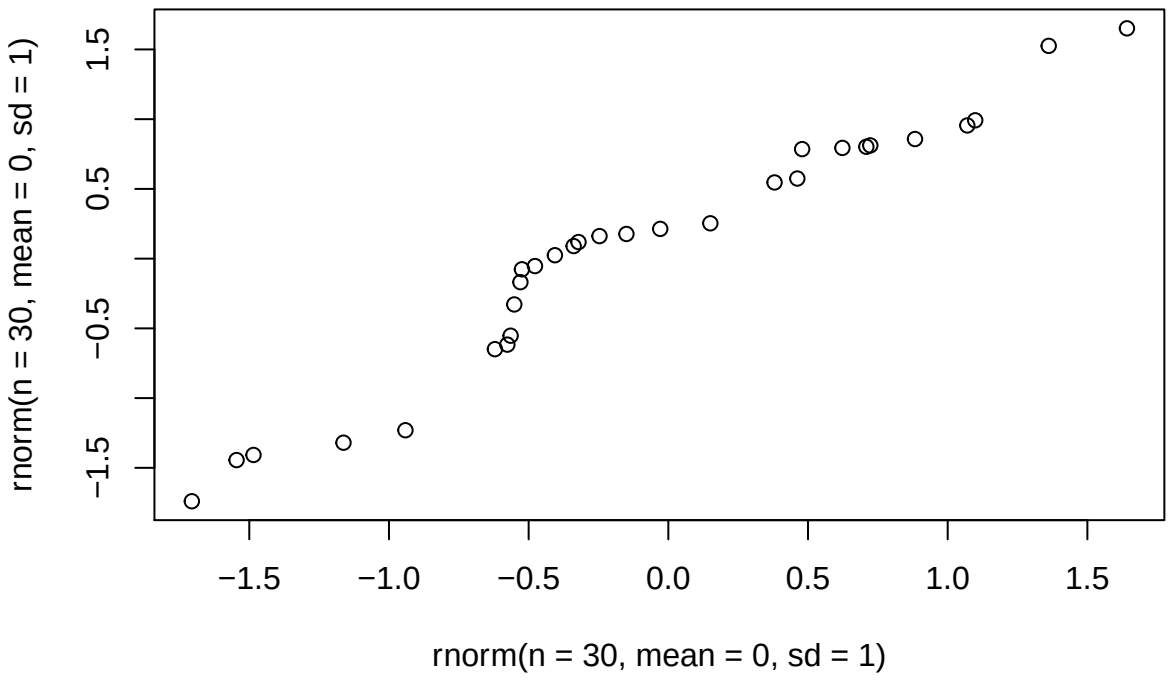
Upon reading the secret message, the reader can count the sum of the ASCII values and then compare that with the parity bit. If these values match then the reader can be assured that they received the message unaltered.

Question 4

QQ plots visualizes how far apart points are for two data sets. If two data sets are of the same distribution then their quantiles in the QQ plot should line up along the line y = x. If two data sets are not of the same distribution then they will not follow this pattern.

Question 4.1

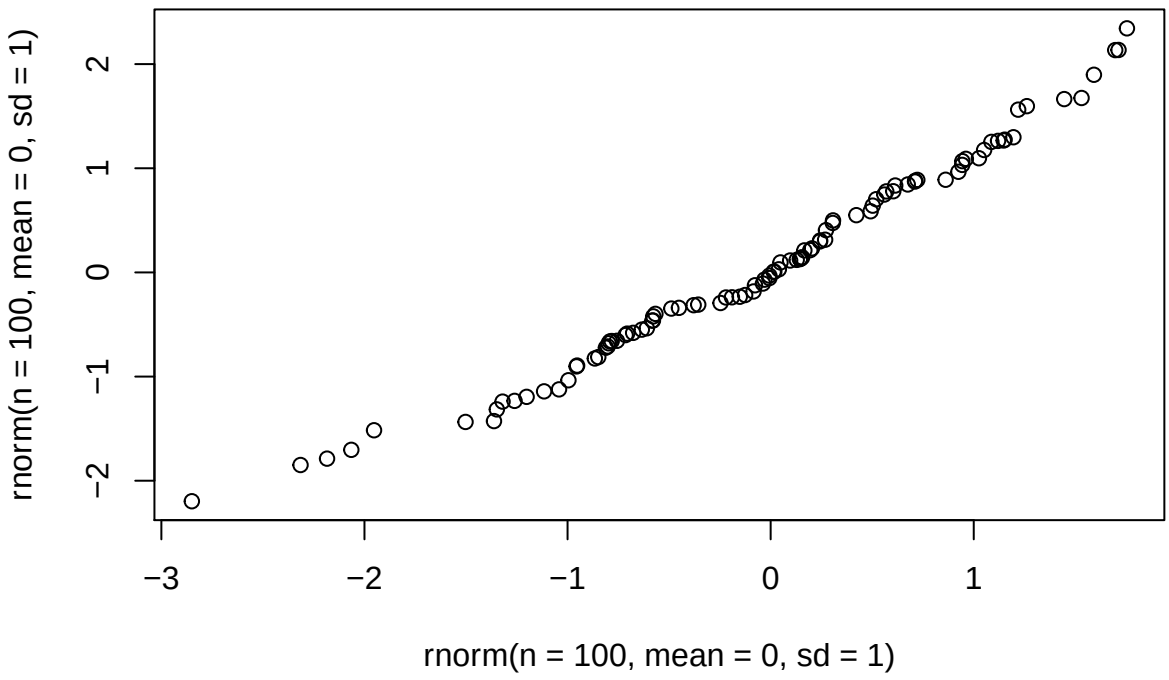
```
qqplot(rnorm(n = 30, mean = 0, sd = 1), rnorm(n = 30, mean = 0, sd = 1))
```



Question 4.2

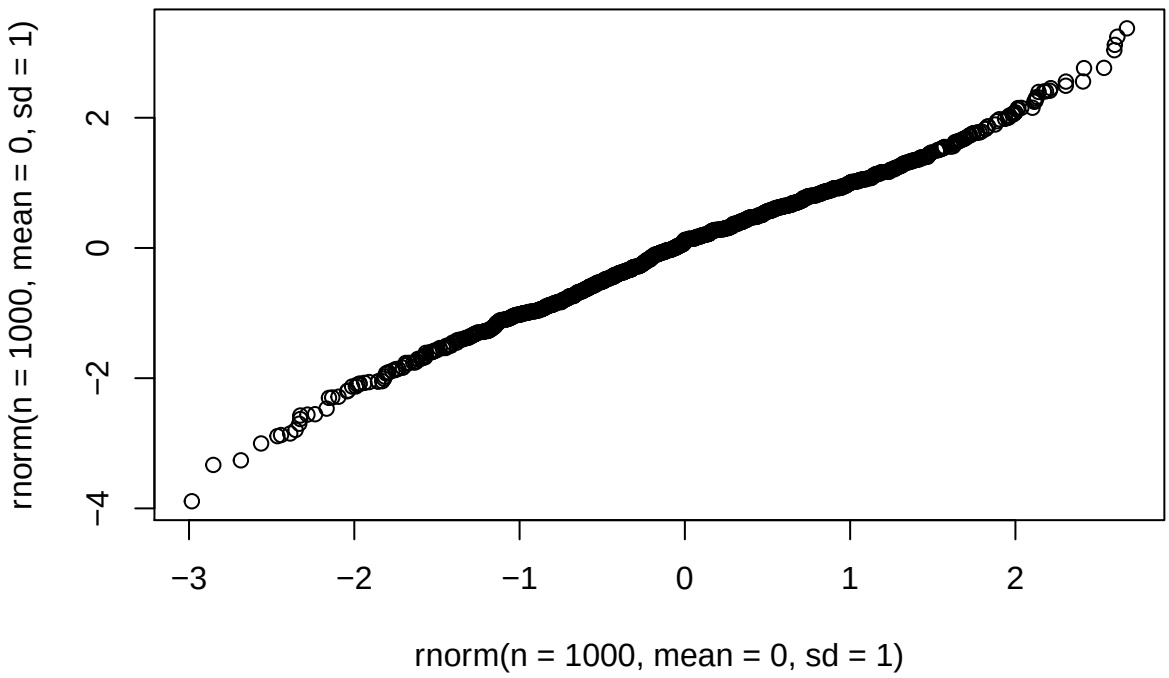
n = 100

```
qqplot(rnorm(n = 100, mean = 0, sd = 1), rnorm(n = 100, mean = 0, sd = 1))
```



n = 1000

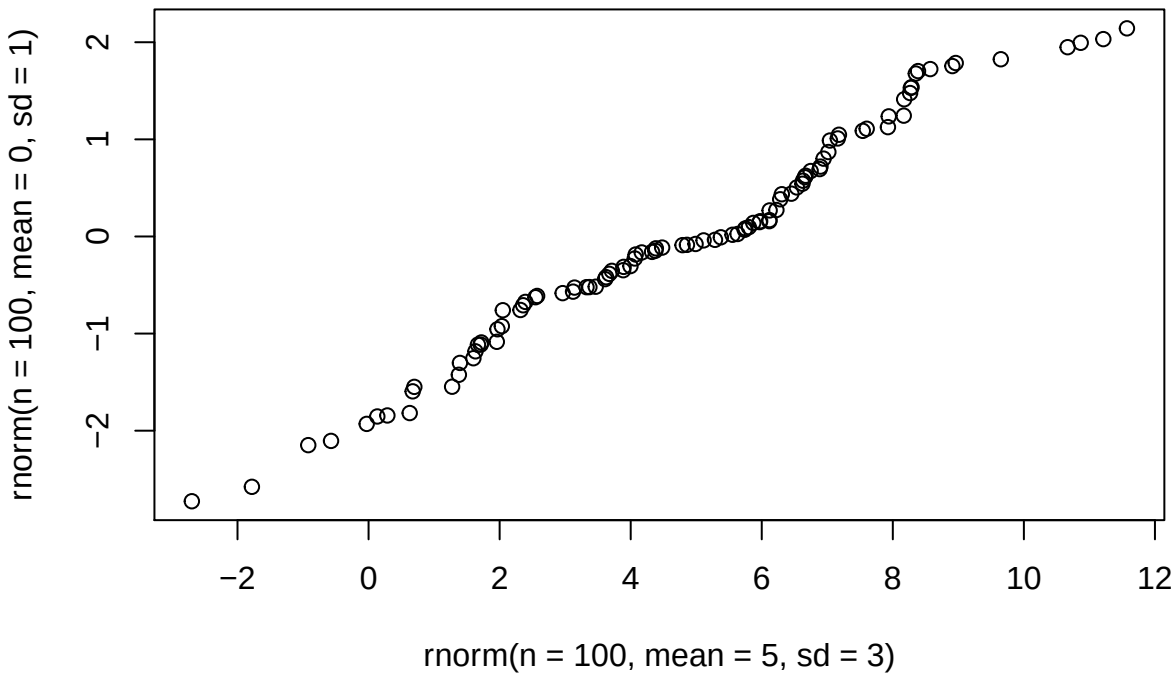
```
qqplot(rnorm(n = 1000, mean = 0, sd = 1), rnorm(n = 1000, mean = 0, sd = 1))
```



As the sample size gets larger, the graph resembles a y = x relation. Thus, the more data you have the more apparent it is that the two distributions are similar.

Question 4.3

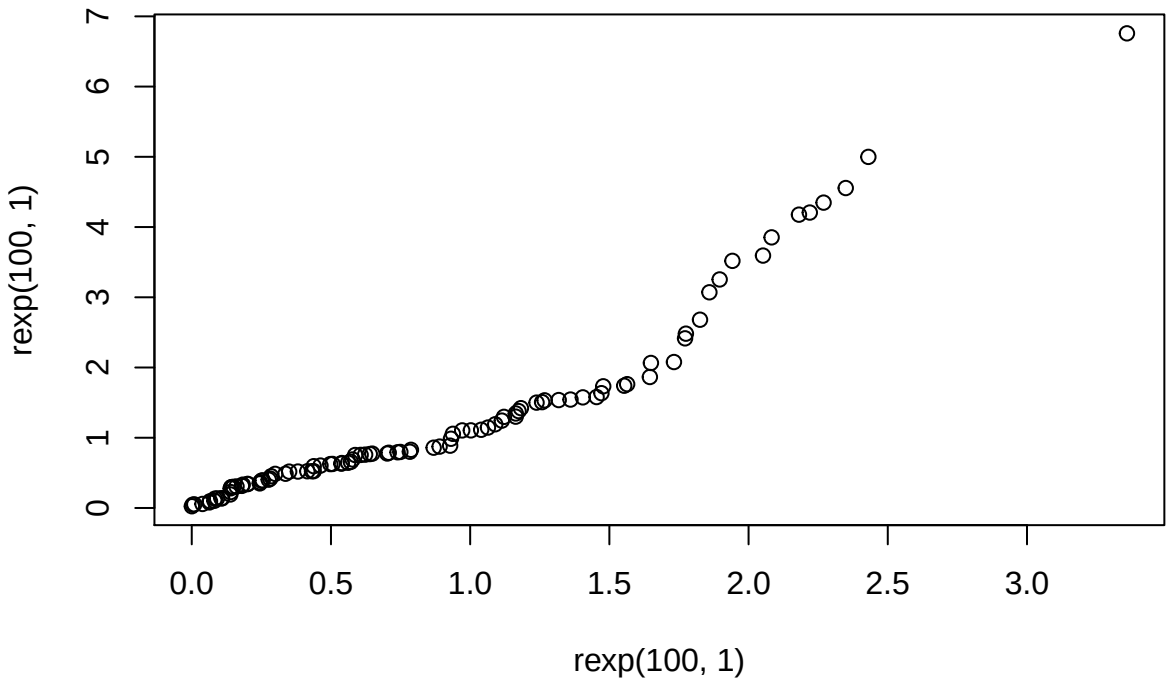
```
qqplot(rnorm(n = 100, mean = 5, sd = 3), rnorm(n = 100, mean = 0, sd = 1))
```



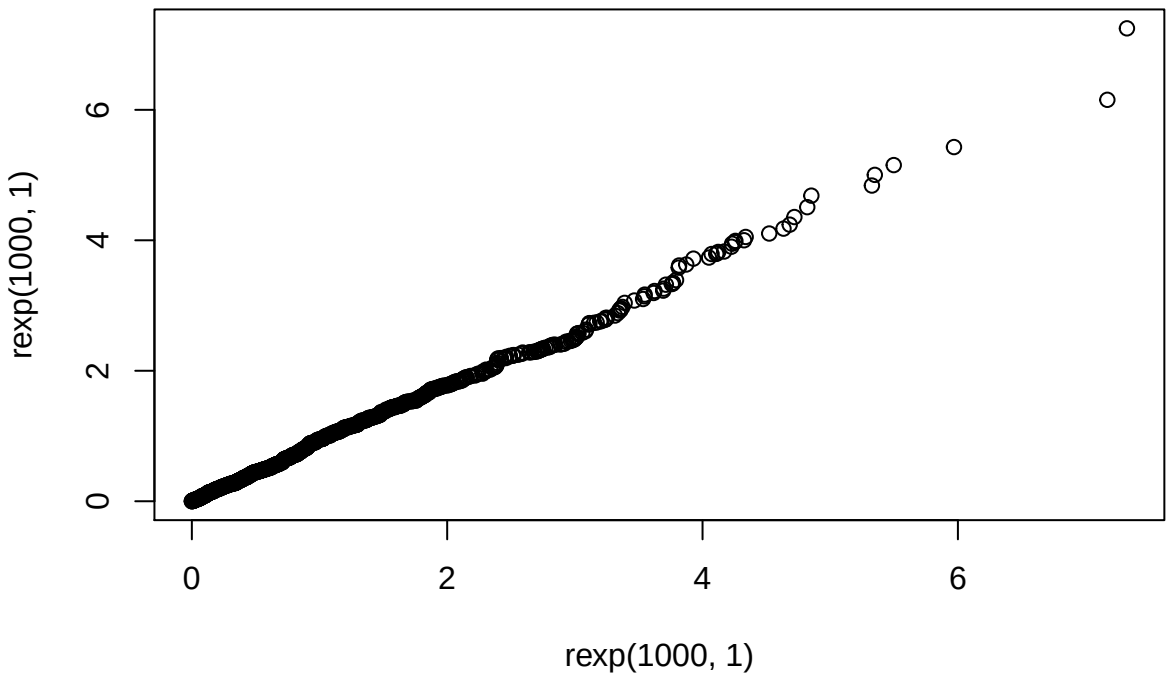
We can see here that though the two distributions have differing means and standard deviations, the distributions are both normal and thus fall along a relatively straight line.

Question 4.4

```
qqplot(rexp(100, 1), rexp(100,1))
```




```
qqplot(rexp(1000, 1), rexp(1000,1))
```

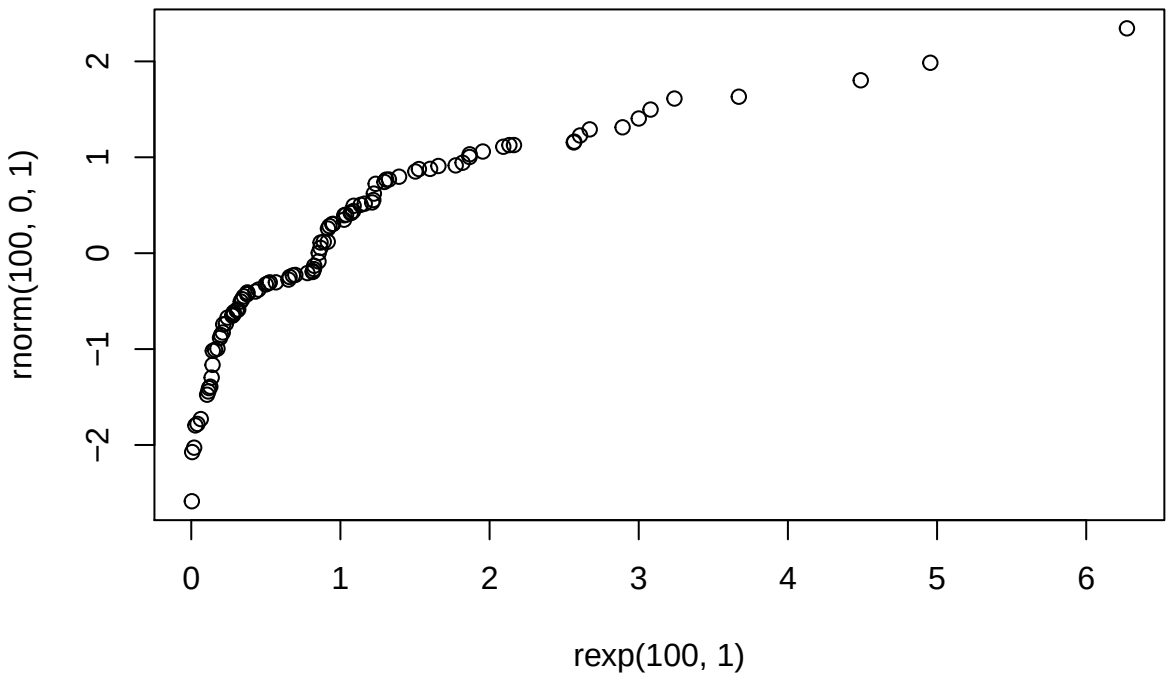


In both graphs, the data is much denser towards the start of the graph since exponential distributions have greater frequency towards the lower end of the range. However, once our sample size gets larger, the graph gets denser and models a straighter line.

Question 4.5

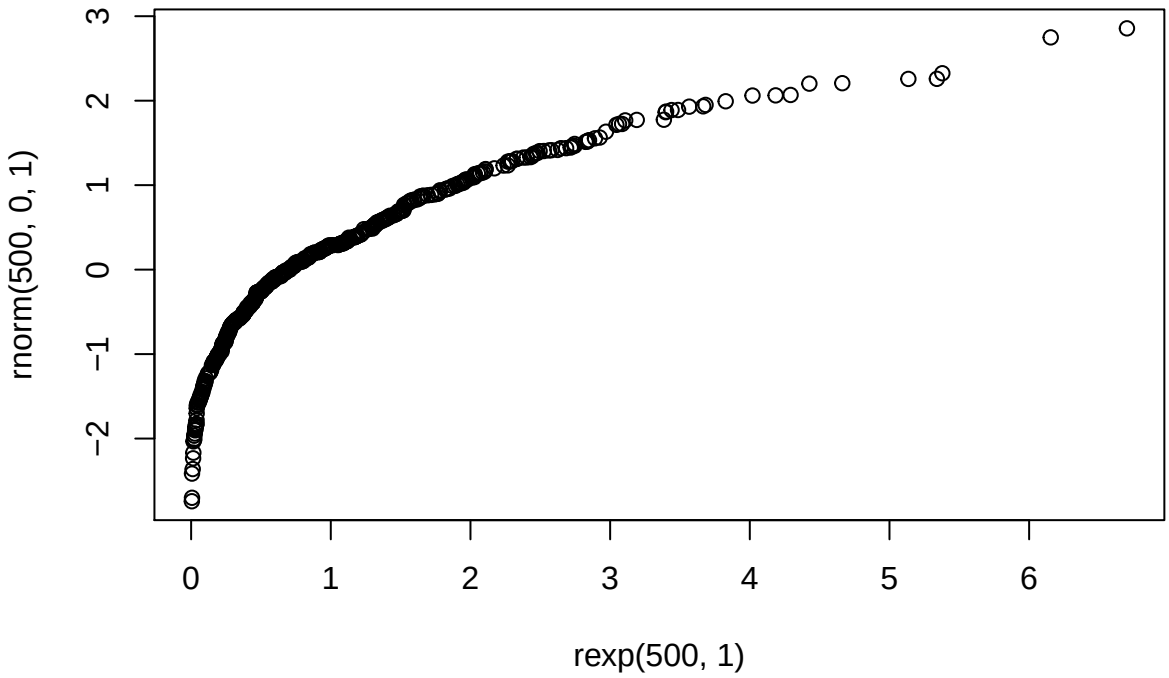
n = 100

```
qqplot(rexp(100, 1), rnorm(100,0,1))
```



n = 500

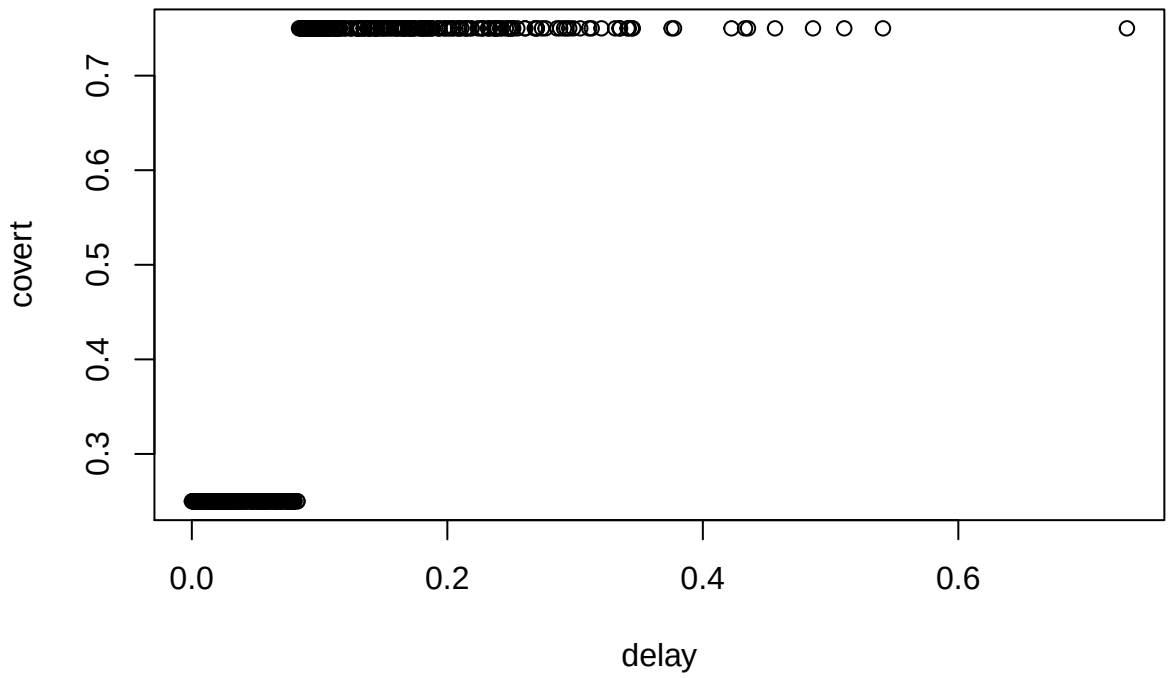
```
qqplot(rexp(500, 1), rnorm(500,0,1))
```



The graph models a curve during the head of the graph since there is a large frequency during the lower end of the graph with an exponential distribution.

Question 4.6

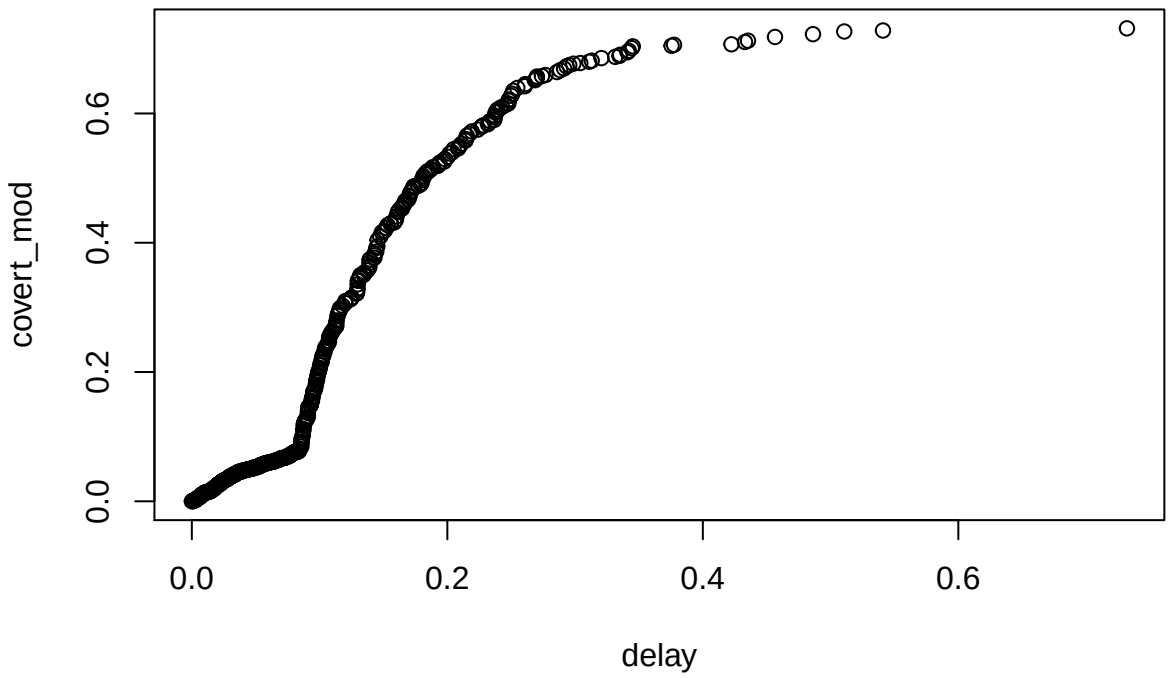
```
qqplot(delay,covert)
```



The QQ plot for the naive covert interpacket delay very clearly does not belong to the same distribution as the unmodified interpacket delay; in fact the line formed by the quantiles jumps at one point since there is a large gap in our distribution modeled in our original covert stream, and the plot don't adhere to the y = x centerline at all. If Eve the warden were to view a QQ plot of these two datasets she should determine that the packet delays are being modified.

Question 4.7

```
qqplot(delay,covert_mod)
```

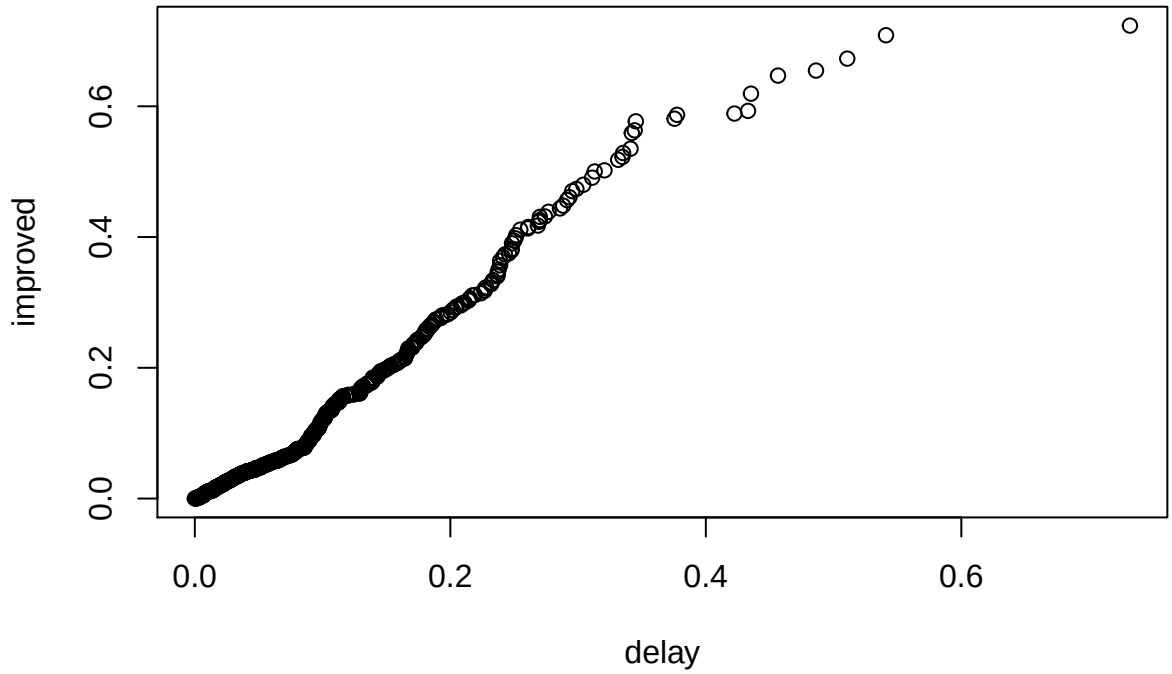


Comparing the next, better version of the covert interpacket delay with the unmodified version shows better results. The QQ plot of these two data sets somewhat adhere to the centerline, not so much initially but as the number of delays approaches infinity the quantiles head towards centerline, and we notice a curve similar to that in question 5 due to the large discrepancy in data being sent out between the first few bins of data. The covert_mod stream is sending out many more packets compared to delay within the same timeframe, which accounts for the curve. If Eve were to look at this QQ plot her suspicions should be slightly raised, thus leading her to investigate the the phenomena more closely.

Question 4.8

```
min = min(delay, na.rm=TRUE)
max = max(delay, na.rm=TRUE)
m = median(delay, na.rm=TRUE)

improved <- bin

for (i in 1:length(improved)) {
  if (improved[i] == 0) {
    # Get a random number from min to median with an exponential distribution of rate= 1
    rate <- 5
    C <- exp(-rate*m)
    D <- exp(-rate*min)
    X <- runif(1,C,D)
    improved[i] = (1/rate)*log(1/X)
```

```
  } else if (improved[i] == 1) {
    # Get a random number from median to max with an exponential distribution of rate= 1
    rate <- 5
    C <- exp(-rate*max)
    D <- exp(-rate*m)
    X = runif(1,C,D)
    improved[i] = (1/rate)*log(1/X)
  }
}

#Compare the original delays with our improved version
qqplot(delay,improved)
```



We see now from the qqplot that we have made our distribution much more convincing. When we send packets with delays that model an exponential distribution with a rate = 5, they are now very similar to the original, unmodified delays. Eve would have a difficult time indeed detecting our covert packets using this implementation.

Implementation

EXPONENTIAL

```
covertStreamExp <- function(i, m){
  packetDelay <- rexp(m + 12, 1)
```

```
packetTime <- packetDelay
packetTime[1] = 0

min = min(packetDelay, na.rm=TRUE)
max = max(packetDelay, na.rm=TRUE)
med = median(packetDelay, na.rm=TRUE)

for(index in 2:m + 12){
  packetTime[index] = packetTime[index] + packetTime[index - 1]
}


# Randomly generate message of m bits
message <- numeric(0)
bits <- c(0,1)

for(index in 1:m){
  message[index] = sample(bits,1)
}

buffer <- c()

#When we can send out the first packet
time = packetTime[i]

messageIndex <- 1

for(index in 1:i){
  buffer = c(buffer,1)
}

for(index in 1:m+12){

  #Underflow
  if(length(buffer) == 0)
    return(0)

  #pop first element from buffer
  buffer = buffer[-1]

  # generate a delay for our packet
  if(message[messageIndex] == 0){
    rate <- 1
    C <- exp(-rate*med)
    D <- exp(-rate*min)
    X <- runif(1,C,D)
    delay = (1/rate)*log(1/X)
    messageIndex = messageIndex + 1
  }else if(message[messageIndex] == 1){
    rate <- 1
    C <- exp(-rate*max)
    D <- exp(-rate*med)
    X <- runif(1,C,D)
```

```
      delay = (1/rate)*log(1/X)
      messageIndex = messageIndex + 1

    }

    #Check if more packets need to be buffered
    for(index2 in 1:m+12){
      if( (packetTime[index2] > time)  & (packetTime[index2] < time + delay)){

        #Overflow
        if(length(buffer) == 20)
          return(1)

        buffer = c(buffer,1)
        prob <<- prob + 1
      }
    }

    #increment the time
    time = time + delay

    if(messageIndex == m)
      return(777)

  }
}

prob = 0
p <- c()


probability <-function(i,m){
  trials <- rep(0,0)
  for(index in 1:100){
    trials[index] = covertStreamExp(i,m)
  }

  return(trials)
}

  m = 16
  run = probability(2,m)
  PU = (length(which(run == 0)))/length(run)
  PO = (length(which(run == 1)))/length(run)
  PS = (length(which(run == 777)))/length(run)
  p[1] = prob /(100* (m + 12))

  PU #Probability of Underflow
```

```
## [1] 0.75
```

```
  PO #Probability of Overflow
```

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

```
  PS #Probability of Success
```

```
## [1] 0.25
```

```
  prob = 0
  run = probability(6,m)
  PU = (length(which(run == 0)))/length(run)
  PO = (length(which(run == 1)))/length(run)
  PS = (length(which(run == 777)))/length(run)
  p[2] = prob /(100* (m + 12))


  PU #Probability of Underflow
```

```
## [1] 0.29
```

```
  PO #Probability of Overflow
```

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

```
  PS #Probability of Success
```

```
## [1] 0.71
```

```
  prob = 0
  run = probability(10,m)
  PU = (length(which(run == 0)))/length(run)
  PO = (length(which(run == 1)))/length(run)
  PS = (length(which(run == 777)))/length(run)
  p[3] = prob /(100* (m + 12))


  PU #Probability of Underflow
```

```
## [1] 0.04
```

```
  PO #Probability of Overflow
```

```
## [1] 0.01
```

```
  PS #Probability of Success
```

```
## [1] 0.95
```

```
  prob= 0
  run = probability(14,m)
  PU = (length(which(run == 0)))/length(run)
  PO = (length(which(run == 1)))/length(run)
  PS = (length(which(run == 777)))/length(run)
  p[4] = prob /(100* (m + 12))

  PU #Probability of Underflow
```

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

```
  PO #Probability of Overflow
```

```
## [1] 0.02
```

```
  PS #Probability of Success
```

```
## [1] 0.98
```

```
  prob = 0
  run = probability(18,m)
  PU = (length(which(run == 0)))/length(run)
  PO = (length(which(run == 1)))/length(run)
  PS = (length(which(run == 777)))/length(run)
  p[5] = prob /(100* (m + 12))


  PU #Probability of Underflow
```

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

```
  PO #Probability of Overflow
```

```
## [1] 0.32
```

```
  PS #Probability of Success
```

```
## [1] 0.68
```

```
  m = 32
  prob = 0
  run = probability(2,m)
  PU = (length(which(run == 0)))/length(run)
  PO = (length(which(run == 1)))/length(run)
  PS = (length(which(run == 777)))/length(run)
  p[6] = prob /(100* (m + 12))

  PU #Probability of Underflow
```

```
## [1] 0.89
```

```
  PO #Probability of Overflow
```

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

```
  PS #Probability of Success
```

```
## [1] 0.11
```

```
  prob = 0
  run = probability(6,m)
  PU = (length(which(run == 0)))/length(run)
  PO = (length(which(run == 1)))/length(run)
  PS = (length(which(run == 777)))/length(run)
  p[7] = prob /(100* (m + 12))

  PU #Probability of Underflow
```

```
## [1] 0.49
```

```
  PO #Probability of Overflow
```

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

```
  PS #Probability of Success
```

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

```
  prob = 0
  run = probability(10,m)
  PU = (length(which(run == 0)))/length(run)
  PO = (length(which(run == 1)))/length(run)
  PS = (length(which(run == 777)))/length(run)
  p[8] = prob /(100* (m + 12))


  PU #Probability of Underflow
```

```
## [1] 0.17
```

```
  PO #Probability of Overflow
```

```
## [1] 0.01
```

```
  PS #Probability of Success
```

```
## [1] 0.82
```

```
  prob= 0
  run = probability(14,m)
  PU = (length(which(run == 0)))/length(run)
  PO = (length(which(run == 1)))/length(run)
  PS = (length(which(run == 777)))/length(run)
  p[9] = prob /(100* (m + 12))


  PU #Probability of Underflow
```

```
## [1] 0.05
```

```
  PO #Probability of Overflow
```

```
## [1] 0.13
```

```
  PS #Probability of Success
```

```
## [1] 0.82
```

```
  prob = 0
  run = probability(18,m)
  PU = (length(which(run == 0)))/length(run)
  PO = (length(which(run == 1)))/length(run)
  PS = (length(which(run == 777)))/length(run)
  p[10] = prob /(100* (m + 12))


  PU #Probability of Underflow
```

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

```
  PO #Probability of Overflow
```

```
## [1] 0.42
```

```
  PS #Probability of Success
```

```
## [1] 0.58
```

UNIFORM DISTRIBUTION

```
covertStreamExp <- function(i, m){
  packetDelay <- runif(m + 12, 0,1)
  packetTime <- packetDelay
  packetTime[1] = 0

  min = min(packetDelay, na.rm=TRUE)
  max = max(packetDelay, na.rm=TRUE)
```

```
med = median(packetDelay, na.rm=TRUE)

for(index in 2:m + 12){
  packetTime[index] = packetTime[index] + packetTime[index - 1]
}


# Randomly generate message of m bits
message <- numeric(0)
bits <- c(0,1)

for(index in 1:m){
  message[index] = sample(bits,1)
}

buffer <- c()

#When we can send out the first packet
time = packetTime[i]

messageIndex <- 1

for(index in 1:i){
  buffer = c(buffer,1)
}

for(index in 1:m+12){

  #Underflow
  if(length(buffer) == 0)
    return(0)

  #pop first element from buffer
  buffer = buffer[-1]

  # generate a delay for our packet
  if(message[messageIndex] == 0){
    delay = runif(1,0,1)
    messageIndex = messageIndex + 1
  }else if(message[messageIndex] == 1){
    delay = runif(1,0,1)
    messageIndex = messageIndex + 1

  }

  #Check if more packets need to be buffered
  for(index2 in 1:m+12){
    if( (packetTime[index2] > time)  & (packetTime[index2] < time + delay)){

      #Overflow
      if(length(buffer) == 20)
        return(1)
```

```
        buffer = c(buffer,1)
        prob <<- prob + 1
      }
    }

    #increment the time
    time = time + delay

    if(messageIndex == m)
      return(777)

  }
}

prob = 0
p1 <- c()

probability <-function(i,m){
  trials <- rep(0,0)
  for(index in 1:100){
    trials[index] = covertStreamExp(i,m)
  }

  return(trials)
}

  m = 16
  run = probability(2,m)
  PU = (length(which(run == 0)))/length(run)
  PO = (length(which(run == 1)))/length(run)
  PS = (length(which(run == 777)))/length(run)
  p1[1] = prob /(100* (m + 12))


  PU #Probability of Underflow
```

```
## [1] 0.54
```

```
  PO #Probability of Overflow
```

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

```
  PS #Probability of Success
```

```
## [1] 0.46
```

```
  prob = 0
  run = probability(6,m)
  PU = (length(which(run == 0)))/length(run)
  PO = (length(which(run == 1)))/length(run)
  PS = (length(which(run == 777)))/length(run)
```

```
  p1[2] = prob /(100* (m + 12))


  PU #Probability of Underflow
```

```
## [1] 0.05
```

```
  PO #Probability of Overflow
```

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

```
  PS #Probability of Success
```

```
## [1] 0.95
```

```
  prob = 0
  run = probability(10,m)
  PU = (length(which(run == 0)))/length(run)
  PO = (length(which(run == 1)))/length(run)
  PS = (length(which(run == 777)))/length(run)
  p1[3] =  prob /(100* (m + 12))


  PU #Probability of Underflow
```

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

```
  PO #Probability of Overflow
```

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

```
  PS #Probability of Success
```

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

```
  prob = 0
  run = probability(14,m)
  PU = (length(which(run == 0)))/length(run)
  PO = (length(which(run == 1)))/length(run)
  PS = (length(which(run == 777)))/length(run)
  p1[4] = prob /(100* (m + 12))

  PU #Probability of Underflow
```

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

```
  PO #Probability of Overflow
```

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

```
  PS #Probability of Success
```

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

```
  prob = 0
  run = probability(18,m)
  PU = (length(which(run == 0)))/length(run)
  PO = (length(which(run == 1)))/length(run)
  PS = (length(which(run == 777)))/length(run)
  p1[5] = prob /(100* (m + 12))

  PU #Probability of Underflow
```

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

```
  PO #Probability of Overflow
```

```
## [1] 0.15
```

```
  PS #Probability of Success
```

```
## [1] 0.85
```

```
  m = 32
  prob = 0
  run = probability(2,m)
  PU = (length(which(run == 0)))/length(run)
  PO = (length(which(run == 1)))/length(run)
  PS = (length(which(run == 777)))/length(run)
  p1[6] = prob /(100* (m + 12))

  PU #Probability of Underflow
```

```
## [1] 0.69
```

```
  PO #Probability of Overflow
```

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

```
  PS #Probability of Success
```

```
## [1] 0.31
```

```
  prob = 0
  run = probability(6,m)
  PU = (length(which(run == 0)))/length(run)
  PO = (length(which(run == 1)))/length(run)
  PS = (length(which(run == 777)))/length(run)
  p1[7] = prob /(100* (m + 12))

  PU #Probability of Underflow
```

```
## [1] 0.18
```

```
  PO #Probability of Overflow
```

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

```
  PS #Probability of Success
```

```
## [1] 0.82
```

```
  prob = 0
  run = probability(10,m)
  PU = (length(which(run == 0)))/length(run)
  PO = (length(which(run == 1)))/length(run)
  PS = (length(which(run == 777)))/length(run)
  p1[8] = prob /(100* (m + 12))

  PU #Probability of Underflow
```

```
## [1] 0.01
```

```
  PO #Probability of Overflow
```

```
## [1] 0.02
```

```
  PS #Probability of Success
```

```
## [1] 0.97
```

```
  prob = 0
  run = probability(14,m)
  PU = (length(which(run == 0)))/length(run)
  PO = (length(which(run == 1)))/length(run)
  PS = (length(which(run == 777)))/length(run)
  p1[9] = prob /(100* (m + 12))

  PU #Probability of Underflow
```

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

```
  PO #Probability of Overflow
```

```
## [1] 0.04
```

```
  PS #Probability of Success
```

```
## [1] 0.96
```

```
  prob = 0
  run = probability(18,m)
  PU = (length(which(run == 0)))/length(run)
  PO = (length(which(run == 1)))/length(run)
  PS = (length(which(run == 777)))/length(run)
  p1[10] = prob /(100* (m + 12))

  PU #Probability of Underflow
```

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

```
  PO #Probability of Overflow
```

```
## [1] 0.41
```

```
  PS #Probability of Success
```

```
## [1] 0.59
```

3

```
gambRuin <- function(q,i,B){
  p = q-1

  if(q != p){
    win <- (1-(q/p)^i)/(1-(q/p)^B)
  } else {
    win <- i/B
  }
  loss <- 1 -  win

  cat("win: ", win, "\n")
  cat("loss: ", loss, "\n")

}

gambRuin(p[1],2,20)
```

```
## win:  0.9217637
## loss:  0.07823628
```

```
gambRuin(p[2],6,20)
```

```
## win:  0.9470438
## loss:  0.05295623
```

```
gambRuin(p[3],10,20)
```

```
## win:  0.9354111
## loss:  0.06458885
```

```
gambRuin(p[4],14,20)
```

```
## win:  0.9858442
## loss:  0.01415576
```

```
gambRuin(p[5],18,20)
```

```
## win:  0.9999999
## loss:  5.426941e-08
```

```
gambRuin(p[6],2,20)
```

```
## win:  0.9338776
## loss:  0.06612245
```

```
gambRuin(p[7],6,20)
```

```
## win:  0.6875576
## loss:  0.3124424
```

```
gambRuin(p[8],10,20)
```

```
## win:  0.047236
## loss:  0.952764
```

```
gambRuin(p[9],14,20)
```

```
## win:  0.1444146
## loss:  0.8555854
```

```
gambRuin(p[10],18,20)
```

```
## win:  0.9995873
## loss:  0.0004127014
```

```r
gambRuin <- function(p,i,B){
  q = p-1

  if(q != p){
    win <- (1-(q/p)^i)/(1-(q/p)^B)
  } else {
    win <- i/B
  }
  loss <- 1 -  win

  cat("win: ", win, "\n")
  cat("loss: ", loss, "\n")

}

gambRuin(p1[1],2,20)
```

```
## win:  6.477722e-07
## loss:  0.9999994
```

```r
gambRuin(p1[2],6,20)
```

```
## win:  0.1249866
## loss:  0.8750134
```

```r
gambRuin(p1[3],10,20)
```

```
## win:  0.3381205
## loss:  0.6618795
```

```r
gambRuin(p1[4],14,20)
```

```
## win:  0.3916362
## loss:  0.6083638
```

```r
gambRuin(p1[5],18,20)
```

```
## win:  0.2653798
## loss:  0.7346202
```

```r
gambRuin(p1[6],2,20)
```

```
## win:  1.568046e-06
## loss:  0.9999984
```

```r
gambRuin(p1[7],6,20)
```

```
## win:  0.9473487
## loss:  0.05265126
```

```
gambRuin(p1[8],10,20)
```

```
## win:  0.999011
## loss:  0.0009889883
```

```
gambRuin(p1[9],14,20)
```

```
## win:  0.999739
## loss:  0.0002609891
```

```
gambRuin(p1[10],18,20)
```

```
## win:  0.7187303
## loss:  0.2812697
```

4

There are several methods available to deal with buffer overflow and underflow. One method we have already conducted: for a given buffer size adjust i manually to receive the best results for a simulation. Another method can be used in conjunction with the previously mentioned one: increase the max size of the buffer. Doing so without adjusting i will automatically reduce the occurences of overflows, and incrementing i to take advantage of the new buffer size will decrease the occurences of underflow as well. Finally, instead of adjusting i manually one could theoretically write code to programatically adjust i depending on occurrences of overflow/underflow.