Assignment 01

Syed Muhammad Adeel Ibrahim

March 23, 2019

## Including Q1

Q1. Use seq(), rep() and/or other commonly-used operators/functions and/or the recycling rule, but deﬁnitely not c(), nor any explicit loop, to create the following sequences.

# [1] 0 15 30 45 60 75 90 105 120 135 150 165 180 195 210 225

# [17] 240 255 270 285 300 315 330 345

seq(0, 345, by = 15)

## [1] 0 15 30 45 60 75 90 105 120 135 150 165 180 195 210 225 240  
## [18] 255 270 285 300 315 330 345

# [1] 1 2 4 8 16 32 64 128 256

2 ^ seq(0, 8)

## [1] 1 2 4 8 16 32 64 128 256

# [1] 1 1 1 1 2 2 2 3 3 4

rep(1:4, rep(4:1))

## [1] 1 1 1 1 2 2 2 3 3 4

# [1] 10 11 12 13 20 21 22 23 30 31 32 33 40 41 42 43

rep(0:3, times = 4) + rep(1:4, each = 4) \* 10

## [1] 10 11 12 13 20 21 22 23 30 31 32 33 40 41 42 43

# [1] “x^1/1 + -x^2/2 + x^3/3 + -x^4/4 + x^5/5 + -x^6/6”

x <- seq(1, 6)  
y <- rep(x, times = 3)  
z <- paste(x, x, sep = "/")  
s <- ifelse(x %% 2 == 0, "-", "")  
equationVector <- paste(s, z, sep = "x^")  
paste(equationVector, collapse = " + ")

## [1] "x^1/1 + -x^2/2 + x^3/3 + -x^4/4 + x^5/5 + -x^6/6"

## Including Q2

Q2. This question makes use of data on 10,000 electric scooter trips in Austin, Texas.1 Download the ﬁle “AustinDockless.csv” from Canvas and run the following code to load the data into an R data frame.

dockless <- read.csv(“AustinDockless.csv”)

dockless <- read.csv("AustinDockless.csv")  
  
docklessCols <- dockless[ ,c('Trip.Duration', 'Trip.Distance', 'Hour', 'Origin.Cell.ID', 'Destination.Cell.ID')]  
  
# docklessCols

Find the length of the ten longest trips.

distanceOrderbyLongest <- sort(docklessCols$Trip.Distance, decreasing = TRUE)  
  
top10Distance <- distanceOrderbyLongest[1:10]  
  
top10Distance

## [1] 19355 19263 19038 18394 14507 14451 12779 12601 12472 12263

Find the start and end Cell IDs for trips longer than 19km

tripAbove19km <- docklessCols[which(docklessCols$Trip.Distance > 19000), c('Origin.Cell.ID', 'Destination.Cell.ID') ]  
  
tripAbove19km

## Origin.Cell.ID Destination.Cell.ID  
## 4669 14074 14074  
## 9518 15019 14227  
## 9621 15019 14228

Find the longest trip (distance) between midnight and 8am.

tripBtw0to8 <- docklessCols[which(docklessCols$Hour >= 0 & docklessCols$Hour <= 8),]  
  
tripBtw0to8LongestDistance <- sort(tripBtw0to8$Trip.Distance, decreasing = TRUE)   
  
topDistanceTripBtw0to8 <- tripBtw0to8LongestDistance[1]  
  
topDistanceTripBtw0to8

## [1] 11977

Find the average speed and the distance covered for the top ten fastest trips (trips with highest average speed), where average speed is calculated as distance divided by duration.

docklessCols['Trip.Speed'] <- docklessCols$Trip.Distance / docklessCols$Trip.Duration  
  
tripOrderByFastestSpeed <- docklessCols[order(docklessCols$Trip.Speed, decreasing = TRUE), c('Trip.Distance', 'Trip.Speed')]  
  
top10FastestSpeed <- tripOrderByFastestSpeed[1:10,]  
  
top10FastestSpeed

## Trip.Distance Trip.Speed  
## 8909 6145 53.903509  
## 1799 356 19.777778  
## 9885 1286 14.613636  
## 3674 5333 11.346809  
## 978 357 10.818182  
## 193 110 10.000000  
## 4327 645 9.626866  
## 5917 3440 8.775510  
## 5982 3055 8.415978  
## 4776 1245 8.190789

BONUS [no marks]: Where in the city did that fastest ride occur? Is there a reasonable explanation for why it is so fast (and so much faster than the next fastest) ?

## Including Q3

Q3. This question also works with the electronic scooter data. We will just focus on the distance variable Trip.Distance.

Calculate the mean and standard deviation of the distance data

Trip.Distance <- docklessCols$Trip.Distance

Mean Trip Distance

distMean <- mean(Trip.Distance)  
distMean

## [1] 1615.266

Standard Deviation Trip Distance

distSD <- sd(Trip.Distance)  
distSD

## [1] 1674.185

Calculate the median and upper and lower quartiles for the distance data (HINT: use the quantile() function).

Median of Distance

distMedian <- median(Trip.Distance)  
distMedian

## [1] 1142

upper and lower quartiles of distance

distQuartiles <- quantile(Trip.Distance, seq(.25, .75, .25))  
distQuartiles

## 25% 50% 75%   
## 491.75 1142.00 2182.50

Generate 10,000 random values from a Normal distribution (HINT: use the rnorm() function) with the same mean and standard deviation as the distance data and calculate the median and upper and lower quartiles of these random values.

set.seed(1234)  
  
randomDistance10000 <- rnorm(10000, distMean, distSD)  
  
randomDistanceMedian <- median(Trip.Distance)  
  
randQuartiles <- quantile(randomDistance10000, seq(.25, .75, .25))  
randQuartiles

## 25% 50% 75%   
## 508.9781 1623.0258 2736.4287

Generate 1,000,000 random values from a Normal distribution with the same mean and standard deviation as the distance data and, dealing with each consecutive subset of 10,000 values, calculate 100 upper quartiles (one for each of the 100 subsets of 10,000 values)

set.seed(1234)  
  
randomDistance1000000 <- rnorm(1000000, distMean, distSD)  
  
  
randomizedDistanceMatrix <- matrix(randomDistance1000000, 10000)  
  
randQuartiles <- numeric(ncol(randomizedDistanceMatrix))  
  
for(i in 1:ncol(randomizedDistanceMatrix)) {  
 randQuartiles[i] <- quantile(randomizedDistanceMatrix[, i], c(.75))  
}  
  
randQuartiles

## [1] 2736.429 2748.037 2734.750 2778.251 2727.928 2747.678 2737.341  
## [8] 2733.882 2736.504 2772.283 2747.166 2740.335 2737.877 2749.035  
## [15] 2742.073 2747.603 2731.202 2745.210 2762.028 2722.479 2755.105  
## [22] 2784.287 2770.057 2744.672 2788.803 2732.448 2719.164 2716.093  
## [29] 2716.177 2748.662 2796.661 2770.102 2739.757 2696.621 2745.113  
## [36] 2758.199 2774.130 2745.969 2738.225 2766.942 2755.778 2757.481  
## [43] 2755.060 2768.428 2761.173 2745.528 2742.979 2721.388 2759.575  
## [50] 2750.646 2753.603 2747.491 2757.320 2770.252 2740.586 2739.986  
## [57] 2727.110 2787.447 2729.941 2744.196 2699.110 2777.860 2740.209  
## [64] 2734.215 2767.500 2755.582 2746.498 2755.516 2749.469 2772.825  
## [71] 2741.689 2695.096 2733.864 2763.335 2707.142 2758.902 2766.222  
## [78] 2740.095 2752.072 2726.040 2753.004 2747.965 2684.081 2757.887  
## [85] 2710.427 2748.324 2729.187 2781.094 2770.067 2735.959 2733.398  
## [92] 2750.425 2710.730 2767.942 2697.985 2764.782 2704.821 2728.067  
## [99] 2786.141 2723.497

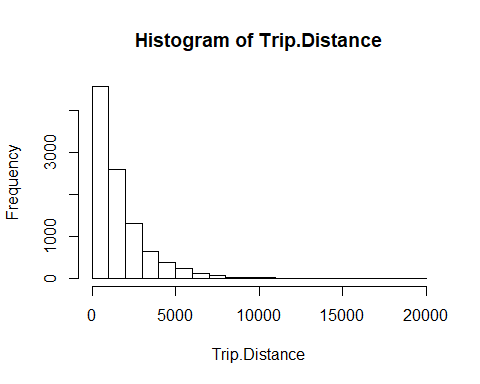
What proportion of the upper quartiles from the random data are less than the distance upper quartile

pval = sum(distQuartiles[which(randQuartiles < distQuartiles[2])])  
  
pval

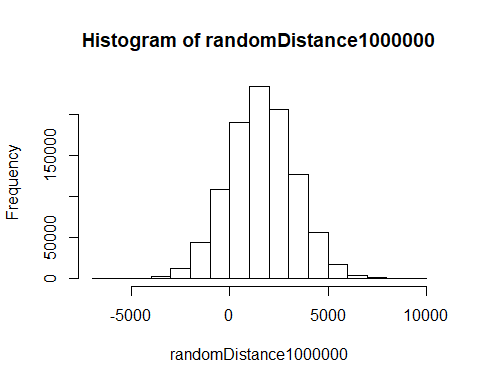
## [1] 0

What does that tell us ? (HINT: look at the plots below)

hist(Trip.Distance)



hist(randomDistance1000000)



Graph shows that we could only get Positive value for distance, which also means that negative value probablity would add up to be most after zero and gradually reduce when we move forward to right.

## Including Q4

Q4. Write a function called segments that takes three arguments: from, to, and rep. The function should generate a matrix with two columns, the ﬁrst column containing from values and the second column containing to values. The argument rep should default to FALSE.

Here is the simplest use of the function.

segments <- function(from, to, rep = FALSE) {  
 options(warn = 1)  
   
 if (from > 1) {  
 from = 1  
 warning(" 'from' value(s) larger than 1 reduced to 1 \n")  
 }  
   
 if (from < -1) {  
 from = 1  
 warning(" 'from' value(s) less than -1 raised to -1 \n")  
 }  
   
 if (to > 1) {  
 to = 1  
 warning(" 'to' value(s) larger than 1 reduced to 1 \n")  
 }  
   
 if (to < -1) {  
 to = 1  
 warning(" 'to' value(s) less than -1 raised to -1 \n")  
 }  
   
 c(from = from, to = to)  
}  
  
segments(.1, 1.2)

## Warning in segments(0.1, 1.2): 'to' value(s) larger than 1 reduced to 1

## from to   
## 0.1 1.0

segments(-1.2, .1)

## Warning in segments(-1.2, 0.1): 'from' value(s) less than -1 raised to -1

## from to   
## 1.0 0.1

The values in from and to can be vectors and they should both “recycle” to the longest length.

segments <- function(from, to, rep = FALSE) {  
 options(warn = 1)  
   
 # validation function for checking limits  
 validateValue <- function(name, value) {  
 if (any(value > 1)) {  
 value[value > 1] = 1  
 warning(paste0(" '", name, "' value(s) larger than 1 reduced to 1 \n"))  
 }  
   
 if (any(value < -1)) {  
 value[value < -1] = -1  
 warning(paste0(" '", name, "' value(s) less than -1 raised to -1 \n"))  
 }  
   
 value  
 }  
   
 # creating equal number of elements for from/to  
 if (length(from) > 1 & length(to) == 1) {  
 to <- rep(to, length(from))  
 }  
   
 if (length(to) > 1 & length(from) == 1) {  
 from <- rep(from, length(to))  
 }  
   
 # general Code for merging row and columns  
 if (length(from) == length(to)) {  
 rowBind <- rbind(from, to)  
 vector <- c(rowBind)  
 matrix <- matrix(vector, ncol = length(from))  
 dimnames(matrix) <- list(c('from', 'to'))  
 matrix <- t(matrix)  
 matrix[, "from"] <- validateValue("from", matrix[, "from"])  
 matrix[, "to"] <- validateValue("to", matrix[, "to"])  
 matrix  
 }  
   
}  
  
segments(c(.1, .2), c(.3, .4))

## from to  
## [1,] 0.1 0.3  
## [2,] 0.2 0.4

segments(1:4/10, .9)

## from to  
## [1,] 0.1 0.9  
## [2,] 0.2 0.9  
## [3,] 0.3 0.9  
## [4,] 0.4 0.9

The rows of the matrix should be in order from smallest from value to largest from value.

segments <- function(from, to, rep = FALSE) {  
 options(warn = 1)  
   
 # validation function for checking limits  
 validateValue <- function(name, value) {  
 if (any(value > 1)) {  
 value[value > 1] = 1  
 warning(paste0(" '", name, "' value(s) larger than 1 reduced to 1 \n"))  
 }  
   
 if (any(value < -1)) {  
 value[value < -1] = -1  
 warning(paste0(" '", name, "' value(s) less than -1 raised to -1 \n"))  
 }  
   
 value  
 }  
   
 # creating equal number of elements for from/to  
 # Descending elements  
 if (length(from) > 1 & length(to) == 1) {  
 from <- sort(from, decreasing = FALSE)  
 to <- rep(to, length(from))  
 }  
   
 if (length(to) > 1 & length(from) == 1) {  
 to <- sort(to, decreasing = FALSE)  
 from <- rep(from, length(to))  
 }  
   
 # general Code for merging row and columns   
 if (length(from) == length(to)) {  
 rowBind <- rbind(from, to)  
 vector <- c(rowBind)  
 matrix <- matrix(vector, ncol = length(from))  
 dimnames(matrix) <- list(c('from', 'to'))  
 matrix <- t(matrix)  
 matrix[, "from"] <- validateValue("from", matrix[, "from"])  
 matrix[, "to"] <- validateValue("to", matrix[, "to"])  
 matrix  
 }  
   
}  
  
segments(4:1/10, .9)

## from to  
## [1,] 0.1 0.9  
## [2,] 0.2 0.9  
## [3,] 0.3 0.9  
## [4,] 0.4 0.9

Any negative values in from and to should be treated as values measuring backward from 1.

segments <- function(from, to, rep = FALSE) {  
 options(warn = 1)  
   
 # validation function for checking limits  
 validateValue <- function(name, value) {  
 if (any(value > 1)) {  
 value[value > 1] = 1  
 warning(paste0(" '", name, "' value(s) larger than 1 reduced to 1 \n"))  
 }  
   
 if (any(value < -1)) {  
 value[value < -1] = -1  
 warning(paste0(" '", name, "' value(s) less than -1 raised to -1 \n"))  
 }  
   
 negitiveFloat = value > -1 & value < 0  
 value[negitiveFloat] = 1 + value[negitiveFloat]  
   
 value  
 }  
   
 # creating equal number of elements for from/to  
 # Descending elements  
 if (length(from) > 1 & length(to) == 1) {  
 from <- sort(from, decreasing = FALSE)  
 to <- rep(to, length(from))  
 }  
   
 if (length(to) > 1 & length(from) == 1) {  
 to <- sort(to, decreasing = FALSE)  
 from <- rep(from, length(to))  
 }  
   
 # general Code for merging row and columns   
 if (length(from) == length(to)) {  
 rowBind <- rbind(from, to)  
 vector <- c(rowBind)  
 matrix <- matrix(vector, ncol = length(from))  
 dimnames(matrix) <- list(c('from', 'to'))  
 matrix <- t(matrix)  
 matrix[, "from"] <- validateValue("from", matrix[, "from"])  
 matrix[, "to"] <- validateValue("to", matrix[, "to"])  
 matrix  
 }  
   
}  
  
segments(.1, -.1)

## from to  
## [1,] 0.1 0.9

segments(c(.1, .2), c(.7, -.2))

## from to  
## [1,] 0.1 0.7  
## [2,] 0.2 0.8

If any from values are larger than the corresponding to values, they should be swapped.

segments <- function(from, to, rep = FALSE) {  
 options(warn = 1)  
   
 # validation function for checking limits  
 validateValue <- function(name, value) {  
 if (any(value > 1)) {  
 value[value > 1] = 1  
 warning(paste0(" '", name, "' value(s) larger than 1 reduced to 1 \n"))  
 }  
   
 if (any(value < -1)) {  
 value[value < -1] = -1  
 warning(paste0(" '", name, "' value(s) less than -1 raised to -1 \n"))  
 }  
   
 negitiveFloat = value > -1 & value < 0  
 value[negitiveFloat] = 1 + value[negitiveFloat]  
   
 value  
 }  
   
 # creating equal number of elements for from/to  
 # Descending elements  
 if (length(from) > 1 & length(to) == 1) {  
 from <- sort(from, decreasing = FALSE)  
 to <- rep(to, length(from))  
 }  
   
 if (length(to) > 1 & length(from) == 1) {  
 to <- sort(to, decreasing = FALSE)  
 from <- rep(from, length(to))  
 }  
   
 # general Code for merging row and columns   
 if (length(from) == length(to)) {  
 rowBind <- rbind(from, to)  
 vector <- c(rowBind)  
 matrix <- matrix(vector, ncol = length(from))  
 dimnames(matrix) <- list(c('from', 'to'))  
 matrix <- t(matrix)  
 matrix[, "from"] <- validateValue("from", matrix[, "from"])  
 matrix[, "to"] <- validateValue("to", matrix[, "to"])  
   
 element <- matrix[, "from"] > matrix[, "to"]   
 fromValue <- matrix[, "from"][element]   
 toValue <- matrix[, "to"][element]  
 matrix[, "from"][element] <- toValue  
 matrix[, "to"][element] <- fromValue  
   
 matrix  
 }  
   
}  
  
 segments(.9, .1)

## from to  
## [1,] 0.1 0.9

If rep is TRUE, the function should“repeat”the values in the argument until they exceed 1.

segments <- function(from, to, rep = FALSE) {  
 options(warn = 1)  
   
 # validation function for checking limits  
 validateValue <- function(name, value) {  
 if (any(value > 1)) {  
 value[value > 1] = 1  
 warning(paste0(" '", name, "' value(s) larger than 1 reduced to 1 \n"))  
 }  
   
 if (any(value < -1)) {  
 value[value < -1] = -1  
 warning(paste0(" '", name, "' value(s) less than -1 raised to -1 \n"))  
 }  
   
 negitiveFloat = value > -1 & value < 0  
 value[negitiveFloat] = 1 + value[negitiveFloat]  
   
 value  
 }  
   
 # rep parameter check for repeat functionality  
 if (!rep) {  
 # creating equal number of elements for from/to  
 # Descending elements  
 if (length(from) > 1 & length(to) == 1) {  
 from <- sort(from, decreasing = FALSE)  
 to <- rep(to, length(from))  
 }  
   
 if (length(to) > 1 & length(from) == 1) {  
 to <- sort(to, decreasing = FALSE)  
 from <- rep(from, length(to))  
 }  
 } else {  
 # code for repeat values  
 if (length(from) == 1 & length(to) == 1) {  
 if (from > to) {  
 temp <- from  
 from <- to  
 to <- temp  
 }  
 from <- seq(from = from, to = 1, by = to)  
 to <- seq(from = to, to = 1, by = to)  
   
 if (length(to) > length(from)) {  
 from <- c(from, 1)  
 }  
 if (length(to) < length(from)) {  
 to <- c(to, 1)  
 }  
   
 }  
 }  
   
 # general Code for merging row and columns   
 if (length(from) == length(to)) {  
 rowBind <- rbind(from, to)  
 vector <- c(rowBind)  
 matrix <- matrix(vector, ncol = length(from))  
 dimnames(matrix) <- list(c('from', 'to'))  
 matrix <- t(matrix)  
 matrix[, "from"] <- validateValue("from", matrix[, "from"])  
 matrix[, "to"] <- validateValue("to", matrix[, "to"])  
   
 element <- matrix[, "from"] > matrix[, "to"]   
 fromValue <- matrix[, "from"][element]   
 toValue <- matrix[, "to"][element]  
 matrix[, "from"][element] <- toValue  
 matrix[, "to"][element] <- fromValue  
   
 matrix  
 }  
   
}  
  
 segments(.1, .2, rep=TRUE)

## from to  
## [1,] 0.1 0.2  
## [2,] 0.3 0.4  
## [3,] 0.5 0.6  
## [4,] 0.7 0.8  
## [5,] 0.9 1.0

segments <- function(from, to, rep = FALSE) {  
 options(warn = 1)  
   
 # validation function for checking limits  
 validateValue <- function(name, value) {  
 if (any(value > 1)) {  
 value[value > 1] = 1  
 warning(paste0(" '", name, "' value(s) larger than 1 reduced to 1 \n"))  
 }  
   
 if (any(value < -1)) {  
 value[value < -1] = -1  
 warning(paste0(" '", name, "' value(s) less than -1 raised to -1 \n"))  
 }  
   
 negitiveFloat = value > -1 & value < 0  
 value[negitiveFloat] = 1 + value[negitiveFloat]  
   
 value  
 }  
   
 if (!rep) {  
 # creating equal number of elements for from/to  
 # Descending elements  
 if (length(from) > 1 & length(to) == 1) {  
 from <- sort(from, decreasing = FALSE)  
 to <- rep(to, length(from))  
 }  
   
 if (length(to) > 1 & length(from) == 1) {  
 to <- sort(to, decreasing = FALSE)  
 from <- rep(from, length(to))  
 }  
 } else {  
 # code for repeat values  
 if (length(from) == 1 & length(to) == 1) {  
 if (from > to) {  
 temp <- from  
 from <- to  
 to <- temp  
 }  
 from <- seq(from = from, to = 1, by = to)  
 to <- seq(from = to, to = 1, by = to)  
   
 if (length(to) > length(from)) {  
 from <- c(from, 1)  
 }  
 if (length(to) < length(from)) {  
 to <- c(to, 1)  
 }  
 } else {  
 if (length(from) > 1 & length(to) == 1) {  
 to <- rep(to, length(from))  
 }  
 if (length(to) > 1 & length(from) == 1) {  
 from <- rep(from, length(to))  
 }  
 vec <- sort(c(from, to), decreasing = FALSE)  
 tv <- ceiling(1/vec[length(vec)])  
 vec1 <- rep(vec, times = tv)  
 vec2 <- rep(0:(tv-1) \* rep(vec[length(vec)]), each = length(vec))  
 vec <- vec1 + vec2  
 mat <- matrix(vec, nrow = 2)  
 rownames(mat) <- c('from', 'to')  
 from <- mat["from",]  
 to <- mat["to",]  
 }  
 }  
   
 # general Code for merging row and columns  
 if (length(from) == length(to)) {  
 rowBind <- rbind(from, to)  
 vector <- c(rowBind)  
 matrix <- matrix(vector, ncol = length(from))  
 dimnames(matrix) <- list(c('from', 'to'))  
 matrix <- t(matrix)  
 matrix[, "from"] <- validateValue("from", matrix[, "from"])  
 matrix[, "to"] <- validateValue("to", matrix[, "to"])  
 element <- matrix[, "from"] > matrix[, "to"]   
 fromValue <- matrix[, "from"][element]   
 toValue <- matrix[, "to"][element]  
 matrix[, "from"][element] <- toValue  
 matrix[, "to"][element] <- fromValue  
   
 # Duplicate row check, takes place for repeat = TRUE  
 if (rep) {  
 duplicateUperLimit <- matrix[, "from"] == matrix[, "to"] & matrix[, "from"] == 1  
 matrix <- matrix[!duplicateUperLimit,]  
   
 f <- matrix[, "from"]  
 t <- matrix[, "to"]  
 if (f[length(f)] != 1 & t[length(t)] != 1) {  
 matrix <- rbind(matrix, 1)  
 }  
 }  
   
 matrix  
 }  
   
}  
  
 segments(c(.3, .1), .2, rep=TRUE)

## Warning in validateValue("from", matrix[, "from"]): 'from' value(s) larger than 1 reduced to 1

## Warning in validateValue("to", matrix[, "to"]): 'to' value(s) larger than 1 reduced to 1

## from to  
## [1,] 0.1 0.2  
## [2,] 0.2 0.3  
## [3,] 0.4 0.5  
## [4,] 0.5 0.6  
## [5,] 0.7 0.8  
## [6,] 0.8 0.9  
## [7,] 1.0 1.0