Week 5 Programming Structures

Contents

Weekly Learning Objectives 1
QuickSort 3
Binary Search Tree 4
Bus Terminal Simulation 7

With the previous weeks completion of various structures for storing data, this week turns to the control structures and syntax of the R language. While the storage of data elements is very important, the real power of the language starts to become apparent when control structures and programming constructs are introduced. Upon completion of this weeks learning, students will have a good grasp of the core R language, and the learning will begin to introduce the use of simulations.

```
knitr::opts_knit$set(root.dir = "D:/Projects/Introduction-to-R/")
```

Weekly Learning Objectives

1. Create a use-case for an R application that uses one of the more advanced features of the language.
2. Create a function to prove the Central Limit Theorem.
3. Create a function to perform a quickstort.
4. Create a binary search tree function.
5. Create a discrete event simulation for a queue system involving a bus terminal.

Use Case

This is essentially a discussion forum exercise. Consider some of the more advanced features of the R language that you have been learning, and present a real-world example of how R could be used to build that out. You do not need to post code, just the use case. For example, in this week's assignments you will build out a bus queue simulation. Explain how that use case would work and specific things you would consider before you start coding for it. Please **DO NOT** use the bus simulation as your use case!

Central Limit Theorem

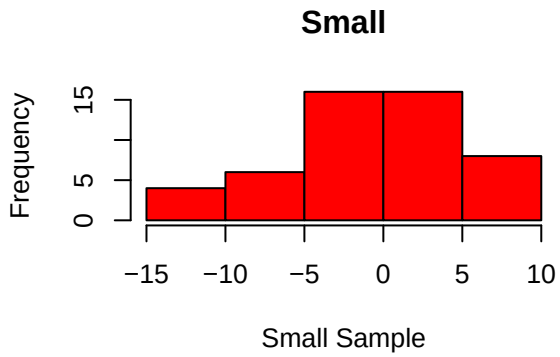
Central Limit Theorem: If you take repeated samples from a population with a finite variance and calculate their averages, then the averages will be normally distributed.

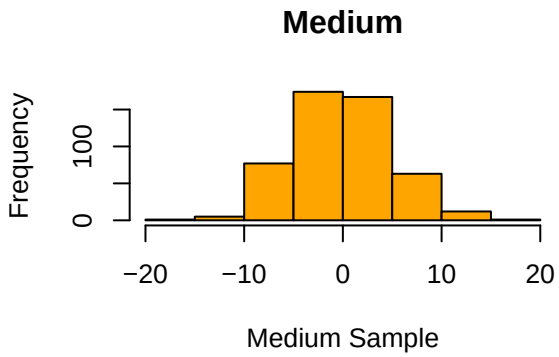
So essentially this is a function to take four samples of various sizes and compare their graphs and means. As the sample size increases, it should approach normal distribution. If this is the case, the graphs will show that easily.

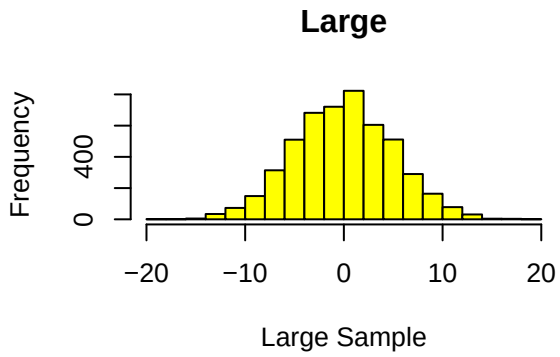
Full Points For full points, you need to create a function similar to this. It should display multiple graphs and have a data table that similiarly shows the averages converging on a normal distribution.

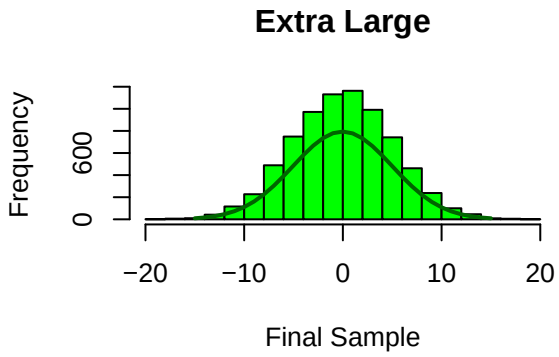
```
CLTProof <- function() {
  n1 <- 50           # Create four sample size variables
  n2 <- 500
  n3 <- 5000
  n4 <- 7500
```

```r
  xBar1 <- rnorm(10000, sd = 5)     # generates multivariate normal random variates (10,000 of them)
  xBarPop <- mean(xBar1)            # calculate the mean
  xBarPopsd <- sd(xBar1)            # calculate the standard deviation

  xBar2 <- sample(xBar1, n1)        # takes n samples from the random number pool xBar1
  xBar3 <- sample(xBar1, n2)
  xBar4 <- sample(xBar1, n3)
  xBar5 <- sample(xBar1, n4)
  Labels <-                         # optional step - setup some pretty labels
    c("Population",
      "Small Sample",
      "Medium Sample",
      "Large Sample",
      "Larger Sample")
  meansCompare <- c(xBarPop, mean(xBar2), mean(xBar3), mean(xBar4), mean(xBar5))
  meansTable <- data.frame(Labels, meansCompare)
  par(mfrow = c(2, 2))                 # For the plot, setup a 2x2 grid
  hist(xBar2,
       main = "Small",
       col = "red",
       xlab = "Small Sample")
  hist(xBar3,
       main = "Medium",
       col = "orange",
       xlab = "Medium Sample")
  hist(xBar4,
       main = "Large",
       col = "yellow",
       xlab = "Large Sample")
  hist(xBar5,
       main = "Extra Large",
       col = "green",
       xlab = "Final Sample")

  xBar5Mn <- mean(xBar5)                          # draw a bell curve over the last graph
  xBar5sd <- sd(xBar5)                            # calculate standard deviation
  x <- seq(-15, 15, 1)                            # Generate a distribution
  y <- dnorm(x, xBar5Mn, xBar5sd) * 10000
  lines(x, y, lwd = 2, col = "dark green")        # plot the line - it should overlay the bell

  meansTable   # print the means table comparison
}

CLTProof()     # execute the function you created
```










```
##          Labels meansCompare
## 1    Population  -0.04025652
## 2  Small Sample  -1.01548804
## 3 Medium Sample  -0.10573808
## 4  Large Sample  -0.02833922
## 5 Larger Sample  -0.02936407
```

QuickSort

The following function implements a more "traditional" quicksort algorithm using array indexes. Partitioning is done "manually" without using R easy filtering features. The use of arrays would be less efficient than vectorization, but this function is focused on flow control.

- aVector = vector to sort
- lowIndex = starting position
- highIndex = ending position

pseudo code

- store position
- using the index of the vector, determine if a swap is needed or not
- swap where appropriate
- repeat until the end position

This function also uses recursion, meaning that it calls itself as part of the process.

```
quickSort <- function(aVector, lowIndex, highIndex) {
  if (lowIndex < highIndex)
```

```
  {
    pivot <- aVector[highIndex];

    i <- (lowIndex - 1)

    for (j in lowIndex:(highIndex-1)) {
      if (aVector[j] <= pivot) {
        i<-i+1

        #Exchange values
        aTemp<-aVector[i]
        aVector[i]<-aVector[j]
        aVector[j]<-aTemp
      }
    }
    # Exchange values
    aTemp <-aVector[(i+1)]
    aVector[i+1] <- aVector[highIndex]
    aVector[highIndex] <- aTemp

    anIndex <- i + 1

    aVector <- quickSort(aVector, lowIndex, anIndex - 1);
    aVector <- quickSort(aVector, anIndex + 1, highIndex);
  }
  return(aVector)
}

#Test QuickSort
myNum <- c(56,12,23,87,10,45,27,72,23,20,100)
mySortedNum <- quickSort(myNum,1,length(myNum))
mySortedNum
```

```
##  [1]  10  12  20  23  23  27  45  56  72  87 100
```

```
myNum <- c(10,9,8,7,6,5,4,3)
mySortedNum <- quickSort(myNum,1,length(myNum))
mySortedNum
```

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

```
myNum <- c(1,2,1,2)
mySortedNum <- quickSort(myNum,1,length(myNum))
mySortedNum
```

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

How can it be rewritten to be more efficient?

1. Second parameter is always one, so you could use as a constant.
2. Third parameter is the length of the first, so it could be calculated inside function
3. While recursive functions are faster, they can be tricky to understand. This could be rewritten

Binary Search Tree

Here is an example of a binary search tree of size 9, depth 3, with 8 as the root.

https://github.com/mrpgmr67/Introduction-to-R/blob/master/Data/Binary%20Search%20Tree.JPG

Details: https://yourbasic.org/algorithms/binary-search-tree/

Functional Decomposition of Code

1. createTree - Generates the tree
2. addNode - Extends the tree
3. printTree - Displays the tree

Each node on the left has a value smaller than the root. Each value on the right has a value bigger than the root. The tree in this function will be horizontal instead of vertical.

Full Points For full points, you need to have a similar function to what you see here. Using the same function is permitted, provided that you have comments explaining each step in the process.

```
createTree <- function (aValue) {

  aTree <- matrix(data=c(NA,NA,NA),nrow=3,ncol=3)
  aTree[1,2] <- aValue
  colnames(aTree)<-c("Left","Value","Right")
  aSize<-3
  anIndex<-2

  aResult <- list(nextIndex=anIndex, matrixSize=aSize, treeMatrix=aTree)
  return(aResult)
}

addNode <-function(aTree,aValue) {

  nextIndex <- aTree["nextIndex"][[1]]
  matrixSize <- aTree["matrixSize"][[1]]
  treeMatrix <- aTree["treeMatrix"][[1]]

  # Add more space to the matrix if needed
  if (nextIndex > matrixSize) {
    treeMatrix <- rbind(treeMatrix, matrix(c(NA,NA,NA),nrow=3,ncol=3))
    matrixSize <- matrixSize + 3
  }

  # Set starting location
  anIndex <- 1
  while (anIndex > 0){
    if (aValue <= treeMatrix[anIndex,"Value"]) {
      # Add Value
      if (is.na(treeMatrix[anIndex,"Left"])) {
        treeMatrix[anIndex,"Left"]<-nextIndex
        treeMatrix[nextIndex,"Value"]<-aValue
        anIndex<-0
      } else {
        # Traverse down a level to the Left
        anIndex <- treeMatrix[anIndex,"Left"][[1]]
      }
    } else {
      if (is.na(treeMatrix[anIndex,"Right"])) {
        treeMatrix[anIndex,"Right"]<-nextIndex
```

```
        treeMatrix[nextIndex,"Value"]<-aValue
        anIndex<-0
      } else {
        # Traverse down a level to the Right
        anIndex <- treeMatrix[anIndex,"Right"][[1]]
      }
    }
  }
  nextIndex<-nextIndex+1
  aResult <- list(nextIndex=nextIndex, matrixSize=matrixSize, treeMatrix=treeMatrix)
  return(aResult)
}

printTree <- function(aTree, aSpace=1, anIndex=1){

  treeMatrix <- aTree["treeMatrix"][[1]]
  aCount <- 5

  if(is.na(treeMatrix[anIndex,"Value"][[1]]) == TRUE) {
    return()
  }

  aSpace <- aSpace+aCount

  printTree(aTree, aSpace, treeMatrix[anIndex,"Right"][[1]])
  if (exists("printString") != TRUE) {
    printString<-""
  }
  cat(rep(".",(aSpace-aCount-1)), sep="")
  cat(treeMatrix[anIndex,"Value"][[1]],"\n")
  printTree(aTree, aSpace, treeMatrix[anIndex, "Left"][[1]])

  return()
}

# Test Binary Tree
aTree <- createTree(8)        # Create Basic Tree
p<-printTree(aTree)           # Print it out
```

```
## 8
```

```
aTree <- addNode(aTree,5)     # Add a node with value 5
p<-printTree(aTree)
```

```
## 8
## .....5
```

```
aTree <- addNode(aTree,20)    # Add a node of value 20
p<-printTree(aTree)
```

```
## .....20
## 8
## .....5
```

```
aTree <- addNode(aTree,6)     # Add a node of value 6
```

```
p<-printTree(aTree)
```

```
## .....20
## 8
## ..........6
## .....5
```

```
aTree <- addNode(aTree,2)    # Add a node of value 2
p<-printTree(aTree)
```

```
## .....20
## 8
## ..........6
## .....5
## ..........2
```

```
aTree <- addNode(aTree,21)   # Add a node of value 21
p<-printTree(aTree)
```

```
## ..........21
## .....20
## 8
## ..........6
## .....5
## ..........2
```

```
aTree <- addNode(aTree,18)   # Add a node of value 18
p<-printTree(aTree)
```

```
## ..........21
## .....20
## ..........18
## 8
## ..........6
## .....5
## ..........2
```

Bus Terminal Simulation

Create a discrete event simulation for a queue system involving a bus terminal. It should include an event list, along with components to add and subtract from that list as buses arrive and depart from the terminal. The following functions should be present: - Core simulation function - Variable initialization - Add event - Event response - Get next event - Function to print the results of the simulation

https://github.com/mrpgmr67/Introduction-to-R/blob/master/Data/Bus%20Simulation.JPG

Full Points For full points you need separate functions for each of these items, plus a core "wrapper" function to encompass the entire thing. This can be solved using very few steps, or it can be more complex. Here is a complex example.

Each event will be represented by a data frame row consisting of the following components:
- evnttime, the time the event is to occur; - evnttype, a character string for the programmer-defined event type; - optional application-specific components, e.g.the job's arrival time in a queuing app

a global list named "sim" holds the events data frame, evnts, and # current simulated time, currtime; there is also a component dbg, which indicates debugging mode

```
evntrow <- function(evnttm,evntty,appin=NULL) {
  rw <- c(list(evnttime=evnttm,evnttype=evntty),appin)
  return(as.data.frame(rw))
}

# insert event with time evnttm and type evntty into event list;
# appin is an optional set of application-specific traits of this event,
# specified in the form a list with named components
schedevnt <- function(evnttm,evntty,appin=NULL) {
  newevnt <- evntrow(evnttm,evntty,appin)
  # if the event list is empty, set it to consist of evnt and return
  if (is.null(sim$evnts)) {
    sim$evnts <<- newevnt
    return()
  }
  # otherwise, find insertion point
  inspt <- binsearch((sim$evnts)$evnttime,evnttm)
  # now "insert," by reconstructing the data frame; we find what
  # portion of the current matrix should come before the new event and
  # what portion should come after it, then string everything together
  before <-
    if (inspt == 1) NULL else sim$evnts[1:(inspt-1),]
  nr <- nrow(sim$evnts)
  after <- if (inspt <= nr) sim$evnts[inspt:nr,] else NULL
  sim$evnts <<- rbind(before,newevnt,after)
}

# binary search of insertion point of y in the sorted vector x; returns
# the position in x before which y should be inserted, with the value
# length(x)+1 if y is larger than x[length(x)]; could be changed to C
# code for efficiency
binsearch <- function(x,y) {
  n <- length(x)
  lo <- 1
  hi <- n
  while(lo+1 < hi) {
    mid <- floor((lo+hi)/2)
    if (y == x[mid]) return(mid)
    if (y < x[mid]) hi <- mid else lo <- mid
  }
  if (y <= x[lo]) return(lo)
  if (y < x[hi]) return(hi)
  return(hi+1)
}

# start to process next event (second half done by application
# programmer via call to reactevnt())
getnextevnt <- function() {
  head <- sim$evnts[1,]
  # delete head
  if (nrow(sim$evnts) == 1) {
    sim$evnts <<- NULL
  } else sim$evnts <<- sim$evnts[-1,]
```

```
  return(head)
}

# simulation body
# arguments:
#    initglbls:  application-specific initialization function; inits
#      globals to statistical totals for the app, etc.; records apppars
#      in globals; schedules the first event
#    reactevnt: application-specific event handling function, coding the
#       proper action for each type of event
#    prntrslts:  prints application-specific results, e.g. mean queue
#       wait
#    apppars:  list of application-specific parameters, e.g.
#      number of servers in a queuing app
#    maxsimtime:  simulation will be run until this simulated time
#    dbg:  debug flag; if TRUE, sim will be printed after each event
dosim <- function(initglbls,reactevnt,prntrslts,maxsimtime,apppars=NULL,
                  dbg=FALSE) {
  sim <<- list()
  sim$currtime <<- 0.0  # current simulated time
  sim$evnts <<- NULL  # events data frame
  sim$dbg <<- dbg
  initglbls(apppars)
  while(sim$currtime < maxsimtime) {
    head <- getnextevnt()
    sim$currtime <<- head$evnttime  # update current simulated time
    reactevnt(head)  # process this event
    if (dbg) print(sim)
  }
  prntrslts()
}

# Bus terminal application specific functions
# This simulation assumes a normal distribution for the servicing and arrival of a bus at a terminal.
# The key to the simulation will largely be the identification of the right distribution.  A normal
# distribution would not be a proper choice if there are going to be values near zero, which would resu
# in an error trap.  By adjusting the values in the error trap, the normal distribution would be lost.
terminalInitGlbls <- function(apppars) {
  terminal.Glbls <<- list()
  # simulation parameters
  terminal.Glbls$arrvrate <<- apppars$arrvrate
  terminal.Glbls$arrvsd   <<- apppars$arrvsd
  terminal.Glbls$srvrate <<- apppars$srvrate
  terminal.Glbls$srvsd <<- apppars$srvsd
  # server queue, consisting of arrival times of queued jobs
  terminal.Glbls$srvq <<- vector(length=0)
  # statistics
  terminal.Glbls$busesDeparted <<- 0  # jobs done so far
  terminal.Glbls$totwait <<- 0.0  # total wait time so far

  aTime <- rnorm(1,terminal.Glbls$arrvrate,terminal.Glbls$arrvsd)
  # Prevent buses from arriving "in the past" or at the same time
  ifelse (aTime <= 0,1,aTime)
```

```
  arrvtime <- sim$currtime + aTime
  schedevnt(arrvtime,"arrv",list(arrvtime=arrvtime))
}

# application-specific event processing function called by dosim()
# in the general DES library
terminalEvent <- function(head) {
  if (head$evnttype == "arrv") {  # arrival

    if (length(terminal.Glbls$srvq) == 0) {
      terminal.Glbls$srvq <<- head$arrvtime

      aTime <- rnorm(1,terminal.Glbls$srvrate,terminal.Glbls$srvsd)

      # Error Trap - Bus must stop for at least 1 minute to handle empty buses
      ifelse (aTime <= 0,1,aTime)

      srvdonetime <- sim$currtime + aTime

      schedevnt(srvdonetime,"srvdone",list(arrvtime=head$arrvtime))
    } else {
      terminal.Glbls$srvq <<- c(terminal.Glbls$srvq,head$arrvtime)
    }

    # generate next arrival using a normal distribution
    aTime <- rnorm(1,terminal.Glbls$arrvrate,terminal.Glbls$arrvsd)
    # Error Trap - Prevent buses from arriving "in the past" or at the same time
    ifelse (aTime <= 0,1,aTime)
    arrvtime <- sim$currtime + aTime
    schedevnt(arrvtime,"arrv",list(arrvtime=arrvtime))
  } else {

    terminal.Glbls$busesDeparted <<- terminal.Glbls$busesDeparted + 1
    terminal.Glbls$totwait <<- terminal.Glbls$totwait + sim$currtime - head$arrvtime
    # remove from queue
    terminal.Glbls$srvq <<- terminal.Glbls$srvq[-1]
    # more still in the queue?
    if (length(terminal.Glbls$srvq) > 0) {
      # schedule new service

      aTime <- rnorm(1,terminal.Glbls$srvrate,terminal.Glbls$srvsd)
      # Bus must stop for at least 1 minute to handle empty situations
      ifelse (aTime <= 0,1,aTime)

      srvdonetime <- sim$currtime + aTime
      schedevnt(srvdonetime,"srvdone",list(arrvtime=terminal.Glbls$srvq[1]))
    }
  }
}

terminalSimResultsPrint <- function() {
  print("Total Wait Time:")
  print(terminal.Glbls$totwait)
```

```
  print("Total Buses Serviced:")
  print(terminal.Glbls$busesDeparted)
  print("Bus mean times (minutes):")
  print(terminal.Glbls$totwait/terminal.Glbls$busesDeparted)
}

# Bus termininal is open for 12 hours a day (i.e. 720 minutes)
# A simulation with typical parameters
dosim(terminalInitGlbls,terminalEvent,terminalSimResultsPrint,720,list(arrvrate=45,arrvsd=10,srvrate=45
```

```
## [1] "Total Wait Time:"
## [1] 1278.625
## [1] "Total Buses Serviced:"
## [1] 14
## [1] "Bus mean times (minutes):"
## [1] 91.33038
```

```
# A simulation with quick service rates
dosim(terminalInitGlbls,terminalEvent,terminalSimResultsPrint,720,list(arrvrate=45,arrvsd=10,srvrate=5,s
```

```
## [1] "Total Wait Time:"
## [1] 75.62933
## [1] "Total Buses Serviced:"
## [1] 15
## [1] "Bus mean times (minutes):"
## [1] 5.041955
```

```
# A simulation with quick arrivals
dosim(terminalInitGlbls,terminalEvent,terminalSimResultsPrint,720,list(arrvrate=5,arrvsd=1,srvrate=45,s
```

```
## [1] "Total Wait Time:"
## [1] 5297.112
## [1] "Total Buses Serviced:"
## [1] 16
## [1] "Bus mean times (minutes):"
## [1] 331.0695
```

Reviewed for 2020 - MSP