BIO 754 - Lecture Week 03

09-03-2017

Solutions to Homework 02

Question 1

a. Create a numeric vector called `v` containing numbers `3.13, 5.06, 19.04, 12,22, 13.1, 2.02, 5.49`. What is the average and *variance* of these numbers?

```
v = c(3.13, 5.06, 19.04, 12,22, 13.1, 2.02, 5.49)
mean(v)
```

```
## [1] 10.23
```

```
sd(v)^2  # or
```

```
## [1] 56.36134
```

```
var(v)
```

```
## [1] 56.36134
```

b. Calculate the mean without the last element, but do this *without* redefining the vector `v`, but using subsetting and the function `length`:

```
mean(v[-length(v)])
```

```
## [1] 10.90714
```

Question 2

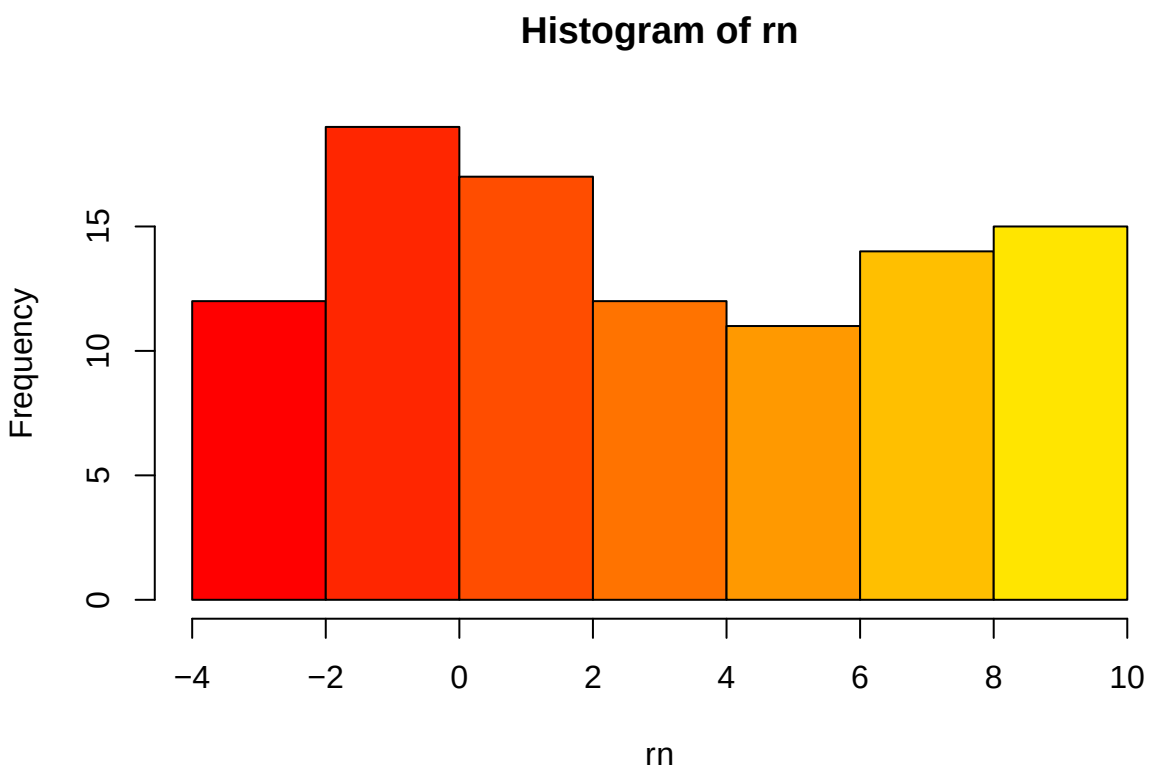
a. Obtain 100 random numbers from a uniform distribution ranged between `-4` and `10`. Decide how to use `runif` for this purpose by checking its help page. To ensure everyone gets the same results, use `set.seed(2)` before generating the random numbers. Report the 50th element of this vector:

```
set.seed(2)
rn = runif(100, min=-4, max=10)
rn[50]
```

```
## [1] 7.345562
```

b. Plot the histogram of `rn`, with `5` breaks and colors according to your taste:

```
hist( rn, col = rainbow(40), breaks = 5 )
```



This is again a uniform distribution.

c. How many elements of `rn` are either greater than `7` OR less than `-0.5`?

```
sum(rn>7 | rn< -0.5)
```

```
## [1] 52
```

d. Find the elements of `rn` with values between `-0.5` and `0.5` and save them as another object `rn2`. Check the length and content of `rn2`.

```
rn2 = rn[ rn < 0.5 & rn >- 0.5]
length(rn2)
```

```
## [1] 7
```

e. Sort the vector `rn2` using the function `sort` (we did not learn this in class, so check yourself), in decreasing order, and reassign the result to `rn2`. Then name the elements of sorted vector with the letters of the alphabet, using the built-in data object `letters`. The result should appear as follows:

```
# this produces a matrix, which is not what we wish
rbind(letters[1:length(rn2)], rn2)
```

```
##     [,1]                  [,2]                [,3]
##     "a"                   "b"                 "c"
## rn2 "-0.0118119604885578" "0.158150292001665" "-0.144383220467716"
##     [,4]                  [,5]                [,6]
##     "d"                   "e"                 "f"
## rn2 "-0.355799505021423" "-0.140185626689345" "0.345335364807397"
##     [,7]
##     "g"
## rn2 "0.0328362504951656"
```

```
# this is the correct solution
rn2 = sort(rn2, decreasing = T)
names(rn2) = letters[1:length(rn2)]
rn2
```

```
##           a           b           c           d           e           f
##  0.34533536  0.15815029  0.03283625 -0.01181196 -0.14018563 -0.14438322
##           g
## -0.35579951
```

```
# btw
LETTERS
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

e. Replace the negative elements of the `rn` object with their squares. Check the first 6 and last 6 elements of the sorted `rn` to ensure there are no negative numbers (do this without reassigning the sorted vector to `rn`):

```
rn[rn<0] = rn[rn<0]^2
head(sort(rn))
```

```
## [1] 0.0001395224 0.0196520099 0.0208465144 0.0328362505 0.1265932878
## [6] 0.1581502920
```

```
tail(sort(rn))
```

```
## [1]  9.84463 11.62498 13.31735 14.39660 14.85483 15.21369
```

Question 3

a. Create a matrix called `mymat` with 200 rows and 5 columns containing random numbers generated from a normal distribution with mean `60` and standard deviation `25`. Use `set.seed(-5)` to make sure everyone gets the same results. Show only the first 8 rows of the matrix. Check its dimensions and its class.

```
set.seed(-5)
mymat = matrix(rnorm(200*5, mean=60, sd=25), 200, 5)
head(mymat, n=8)
```

```
##           [,1]     [,2]      [,3]     [,4]      [,5]
## [1,]  75.66923 65.38460  62.50368 63.36658 109.33900
## [2,]  38.14873 85.83083 103.12907 58.95615  48.92736
## [3,] 104.38819 76.64287  48.64392 25.97761  81.37482
## [4,]  50.64862 97.10207  43.38806 82.84109  96.07837
## [5,]  33.42766 69.76444  43.56181 65.41335  73.47764
## [6,]  74.27570 37.14353  35.71751 59.61248  62.28456
## [7,]  91.33614 83.10238  31.85347 42.69549  45.11716
## [8,]  58.88148 41.95815  55.98638 58.92821  59.88862
```

```
dim(mymat)
```

```
## [1] 200   5
```

```
class(mymat)
```

```
## [1] "matrix"
```

b. Create a five element character vector `cnames` with values `C1,C2,C3,C4,C5` and use it to assign column names to `mymat`. Also, give row names to `mymat` using numbers from 1 to 200. Check the last 3 rows of the matrix:

```
cnames = c("C1","C2","C3","C4","C5")
colnames(mymat) = cnames
rownames(mymat) = seq(1,200)
tail(mymat, 3)
```

```
##           C1       C2       C3       C4       C5
## 198 74.12492 65.98861 45.92591 77.69991 68.18033
## 199 66.54808 44.15721 36.11378 37.67415 54.66687
## 200 76.09303 90.28674 74.49736 92.79072 36.70233
```

c. Convert `mymat` into a data frame. Then create a character vector called `myclass` that repeats the words `"classA"` and `"classB"` 100 times *each* (check the `rep` help page to learn how to do this). Join `myclass` as a new column of `mymat`. Study the new data frame using the function `summary`:

```
mymat = data.frame(mymat)
myclass = rep(c("classA", "classB"), each=100)
mymat = cbind(mymat, myclass)
summary(mymat)
```

```
##        C1                C2               C3               C4
##  Min.   : -6.158   Min.   :-12.38   Min.   :-17.65   Min.   : -2.573
##  1st Qu.: 42.488   1st Qu.: 47.22   1st Qu.: 40.12   1st Qu.: 46.900
##  Median : 60.370   Median : 63.05   Median : 54.42   Median : 58.859
##  Mean   : 60.142   Mean   : 63.01   Mean   : 57.37   Mean   : 60.520
##  3rd Qu.: 75.243   3rd Qu.: 77.68   3rd Qu.: 74.75   3rd Qu.: 75.566
##  Max.   :127.207   Max.   :133.68   Max.   :123.75   Max.   :135.121
##        C5           myclass
##  Min.   :-10.70   classA:100
```

```
##  1st Qu.: 43.35   classB:100
##  Median : 58.05
##  Mean   : 58.32
##  3rd Qu.: 70.71
##  Max.   :132.39
```

```
# btw, this also works:
rep(c("classA", "classB"), c(100, 100))
```

```
##   [1] "classA" "classA" "classA" "classA" "classA" "classA" "classA"
##   [8] "classA" "classA" "classA" "classA" "classA" "classA" "classA"
##  [15] "classA" "classA" "classA" "classA" "classA" "classA" "classA"
##  [22] "classA" "classA" "classA" "classA" "classA" "classA" "classA"
##  [29] "classA" "classA" "classA" "classA" "classA" "classA" "classA"
##  [36] "classA" "classA" "classA" "classA" "classA" "classA" "classA"
##  [43] "classA" "classA" "classA" "classA" "classA" "classA" "classA"
##  [50] "classA" "classA" "classA" "classA" "classA" "classA" "classA"
##  [57] "classA" "classA" "classA" "classA" "classA" "classA" "classA"
##  [64] "classA" "classA" "classA" "classA" "classA" "classA" "classA"
##  [71] "classA" "classA" "classA" "classA" "classA" "classA" "classA"
##  [78] "classA" "classA" "classA" "classA" "classA" "classA" "classA"
##  [85] "classA" "classA" "classA" "classA" "classA" "classA" "classA"
##  [92] "classA" "classA" "classA" "classA" "classA" "classA" "classA"
##  [99] "classA" "classA" "classB" "classB" "classB" "classB" "classB"
## [106] "classB" "classB" "classB" "classB" "classB" "classB" "classB"
## [113] "classB" "classB" "classB" "classB" "classB" "classB" "classB"
## [120] "classB" "classB" "classB" "classB" "classB" "classB" "classB"
## [127] "classB" "classB" "classB" "classB" "classB" "classB" "classB"
## [134] "classB" "classB" "classB" "classB" "classB" "classB" "classB"
## [141] "classB" "classB" "classB" "classB" "classB" "classB" "classB"
## [148] "classB" "classB" "classB" "classB" "classB" "classB" "classB"
## [155] "classB" "classB" "classB" "classB" "classB" "classB" "classB"
## [162] "classB" "classB" "classB" "classB" "classB" "classB" "classB"
## [169] "classB" "classB" "classB" "classB" "classB" "classB" "classB"
## [176] "classB" "classB" "classB" "classB" "classB" "classB" "classB"
## [183] "classB" "classB" "classB" "classB" "classB" "classB" "classB"
## [190] "classB" "classB" "classB" "classB" "classB" "classB" "classB"
## [197] "classB" "classB" "classB" "classB"
```

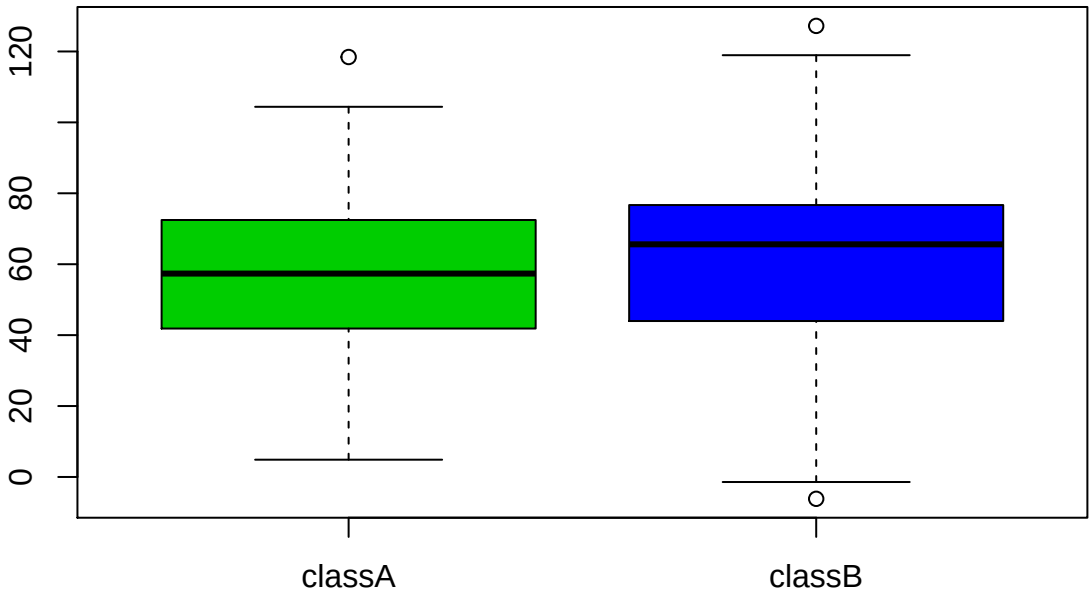
```
# but not this
rep(c("classA", "classB"), 100)
```

```
##   [1] "classA" "classB" "classA" "classB" "classA" "classB" "classA"
##   [8] "classB" "classA" "classB" "classA" "classB" "classA" "classB"
##  [15] "classA" "classB" "classA" "classB" "classA" "classB" "classA"
##  [22] "classB" "classA" "classB" "classA" "classB" "classA" "classB"
##  [29] "classA" "classB" "classA" "classB" "classA" "classB" "classA"
##  [36] "classB" "classA" "classB" "classA" "classB" "classA" "classB"
##  [43] "classA" "classB" "classA" "classB" "classA" "classB" "classA"
##  [50] "classB" "classA" "classB" "classA" "classB" "classA" "classB"
##  [57] "classA" "classB" "classA" "classB" "classA" "classB" "classA"
##  [64] "classB" "classA" "classB" "classA" "classB" "classA" "classB"
##  [71] "classA" "classB" "classA" "classB" "classA" "classB" "classA"
```

```
##  [78] "classB" "classA" "classB" "classA" "classB" "classA" "classB"
##  [85] "classA" "classB" "classA" "classB" "classA" "classB" "classA"
##  [92] "classB" "classA" "classB" "classA" "classB" "classA" "classB"
##  [99] "classA" "classB" "classA" "classB" "classA" "classB" "classA"
## [106] "classB" "classA" "classB" "classA" "classB" "classA" "classB"
## [113] "classA" "classB" "classA" "classB" "classA" "classB" "classA"
## [120] "classB" "classA" "classB" "classA" "classB" "classA" "classB"
## [127] "classA" "classB" "classA" "classB" "classA" "classB" "classA"
## [134] "classB" "classA" "classB" "classA" "classB" "classA" "classB"
## [141] "classA" "classB" "classA" "classB" "classA" "classB" "classA"
## [148] "classB" "classA" "classB" "classA" "classB" "classA" "classB"
## [155] "classA" "classB" "classA" "classB" "classA" "classB" "classA"
## [162] "classB" "classA" "classB" "classA" "classB" "classA" "classB"
## [169] "classA" "classB" "classA" "classB" "classA" "classB" "classA"
## [176] "classB" "classA" "classB" "classA" "classB" "classA" "classB"
## [183] "classA" "classB" "classA" "classB" "classA" "classB" "classA"
## [190] "classB" "classA" "classB" "classA" "classB" "classA" "classB"
## [197] "classA" "classB" "classA" "classB"
```

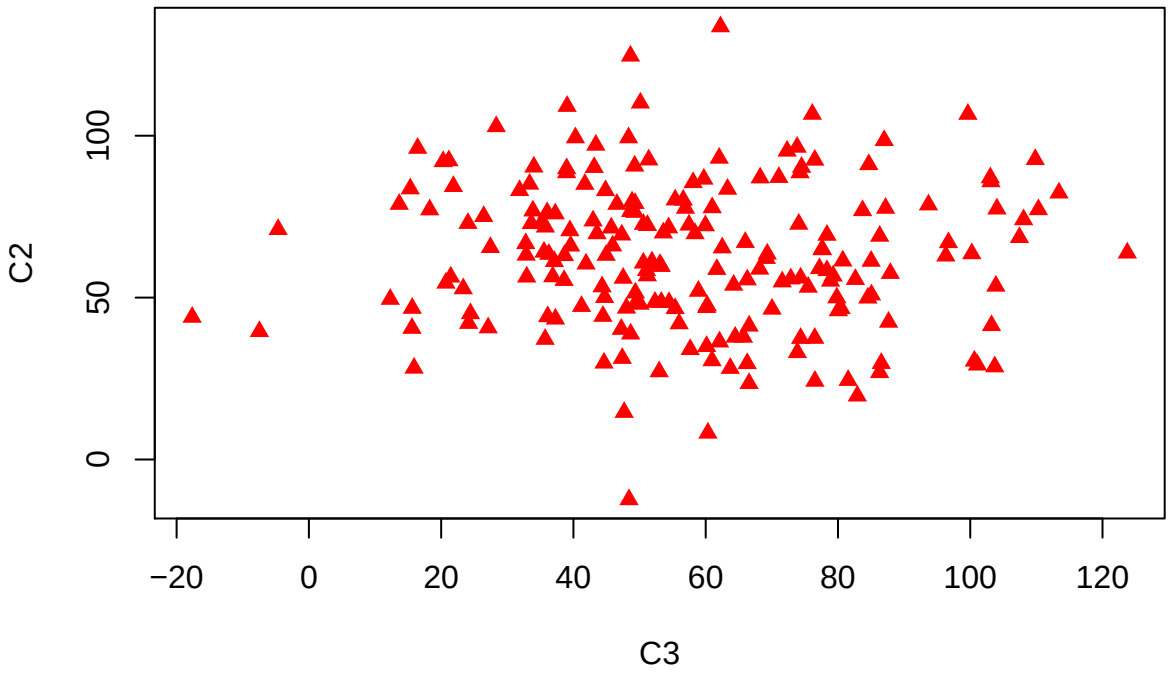
d. Draw a boxplot of the data in the first column of `mymat` after sorting it into the two classes defined in the `myclass` column, using the formula notation:

```
boxplot(C1 ~ myclass, mymat, col=3:4)
# yet another way
boxplot(mymat$C1 ~ mymat$myclass, col=3:4)
```



e. Plot `C2` against `C3` (`C3` should be on the x axis). Represent points as filled triangles (find the correct option yourselves):

```
plot(C2 ~ C3, mymat, pch=17, col="red")
```



Question 4

a. This question follows our discussion about the absolute deviation from expectation as a function of sample size. First, you will retrieve `100` random values from the uniform distribution (between `0` and `1`), each time calculate the proportion of values less than 0.5, and then the **absolute** deviation from the expectation. You will run this 5 times and store the results in a vector called `dev100` (you will need to create the vector first and then add the new results). Write your code by using an numeric variable `N` to specify the sample size in your code (as we did in class). Setting the seed to `1`, your results should appear as follows:

```
set.seed(1)
N = 100
dev100 = abs( 0.5 - sum( runif(N) < 0.5 )/N )
# you could also have used mean in this case:
dev100 = c(dev100, abs( 0.5 - mean( runif(N) < 0.5 ) ) )
dev100 = c(dev100, abs( 0.5 - mean( runif(N) < 0.5 ) ) )
dev100 = c(dev100, abs( 0.5 - mean( runif(N) < 0.5 ) ) )
dev100 = c(dev100, abs( 0.5 - mean( runif(N) < 0.5 ) ) )
dev100
```

```
## [1] 0.02 0.04 0.12 0.07 0.03
```

The following piece of code is an example of how to create a vector by adding new elements at each step using `c()`. Soon you will learn how to run such processes in loops:

```
x = runif(1)
x = c(x, runif(1))
x = c(x, runif(1))
x
```

```
## [1] 0.5541771 0.6882752 0.6580576
```

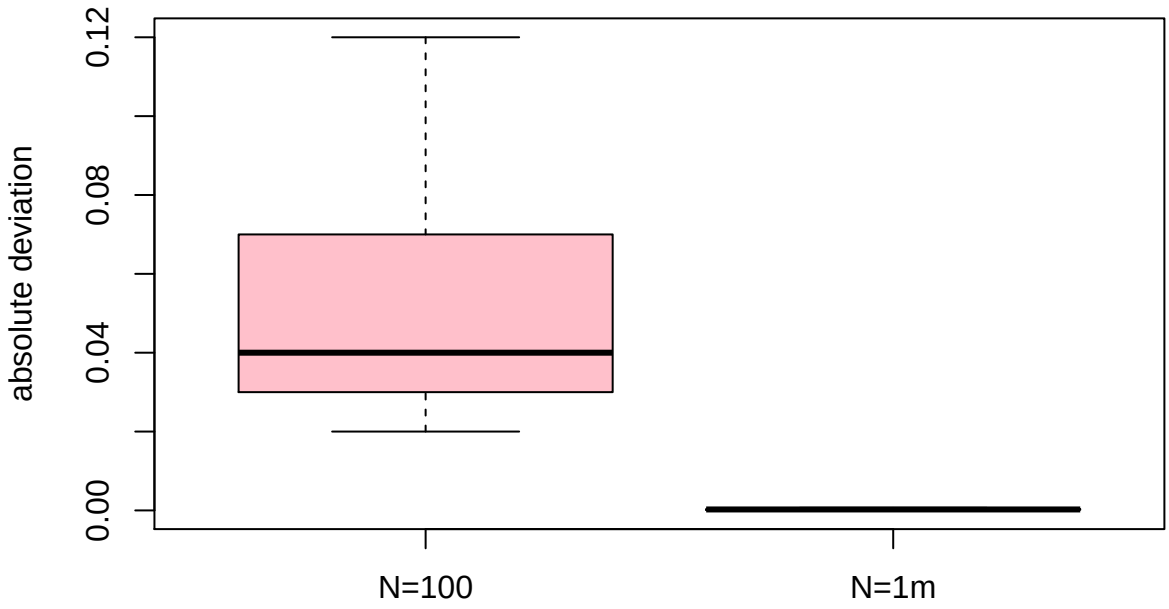
b. Now repeat the same, but sampling `1000000` values, and storing the results in a vector called `dev1m` (set the seed again to `1`):

```
set.seed(1)
N = 1000000
dev1m = abs( 0.5 - sum( runif(N) < 0.5 )/N )
dev1m = c(dev1m, abs( 0.5 - sum( runif(N) < 0.5 )/N ) )
dev1m = c(dev1m, abs( 0.5 - sum( runif(N) < 0.5 )/N ) )
dev1m = c(dev1m, abs( 0.5 - sum( runif(N) < 0.5 )/N ) )
dev1m = c(dev1m, abs( 0.5 - sum( runif(N) < 0.5 )/N ) )
dev1m
```

```
## [1] 0.000370 0.000648 0.000242 0.000139 0.000057
```

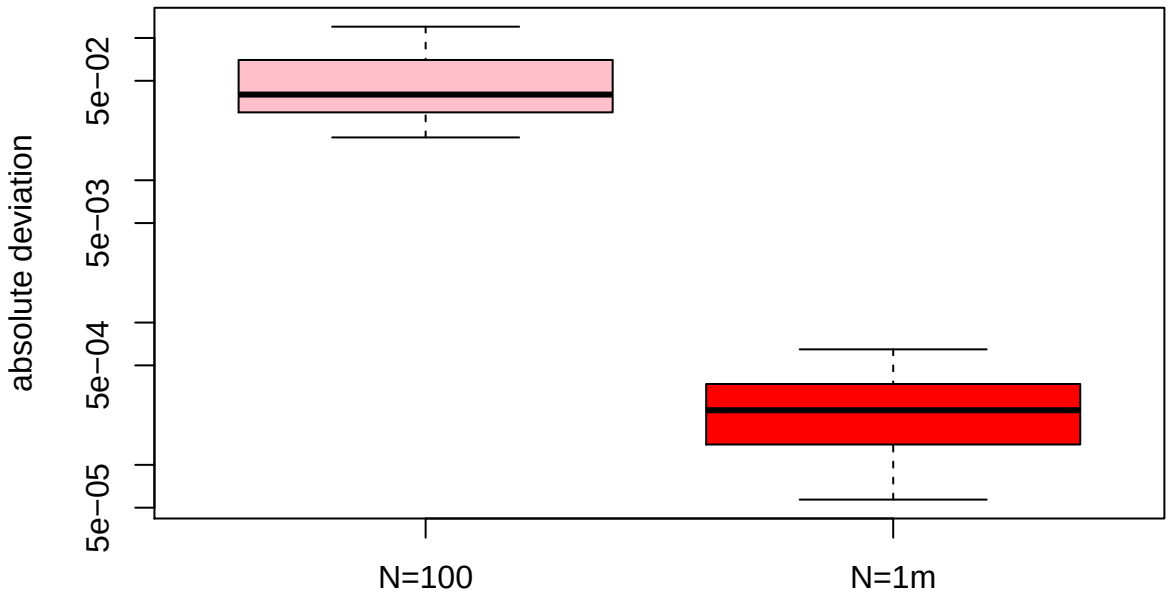
c. Now compare the two distributions using the boxplot:

```
boxplot(dev100, dev1m, names = c("N=100", "N=1m"), ylab="absolute deviation", col=c("pink", "red"))
```



d. Because you cannot observe variance in the `1m` case, try log transforming the y axis (find the correct argument and its usage by checking the `boxplot` function help page):

```
boxplot(dev100, dev1m, names = c("N=100", "N=1m"), ylab="absolute deviation", log = "y", col=c("pink",
```

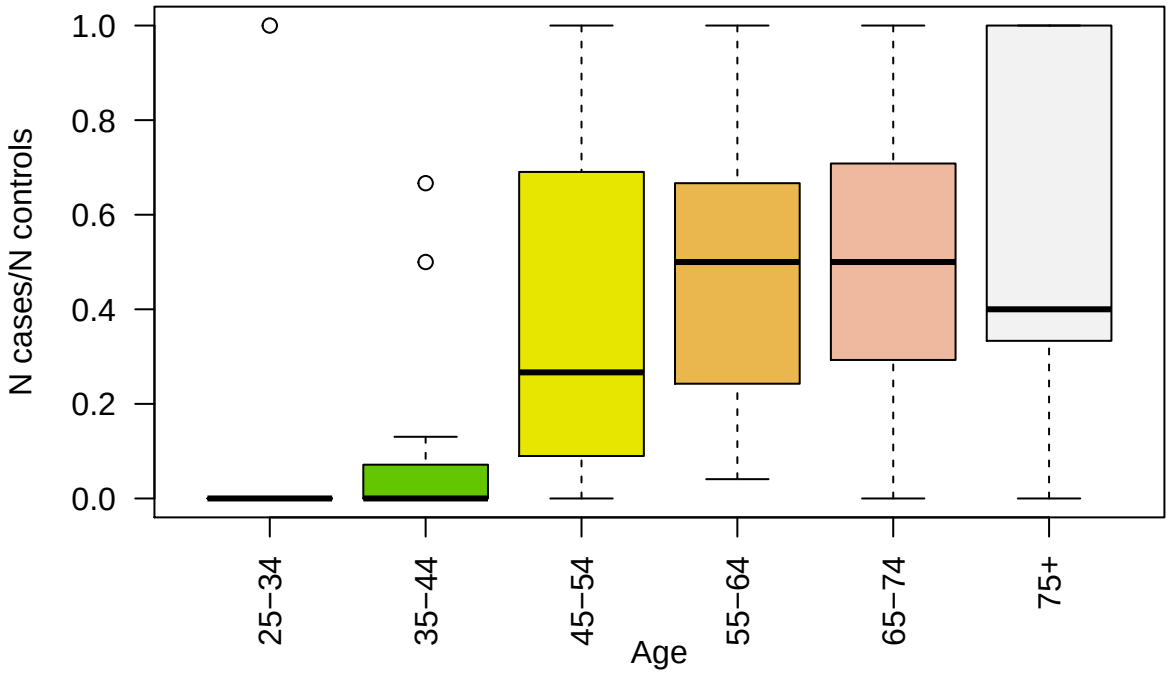


Question 5:

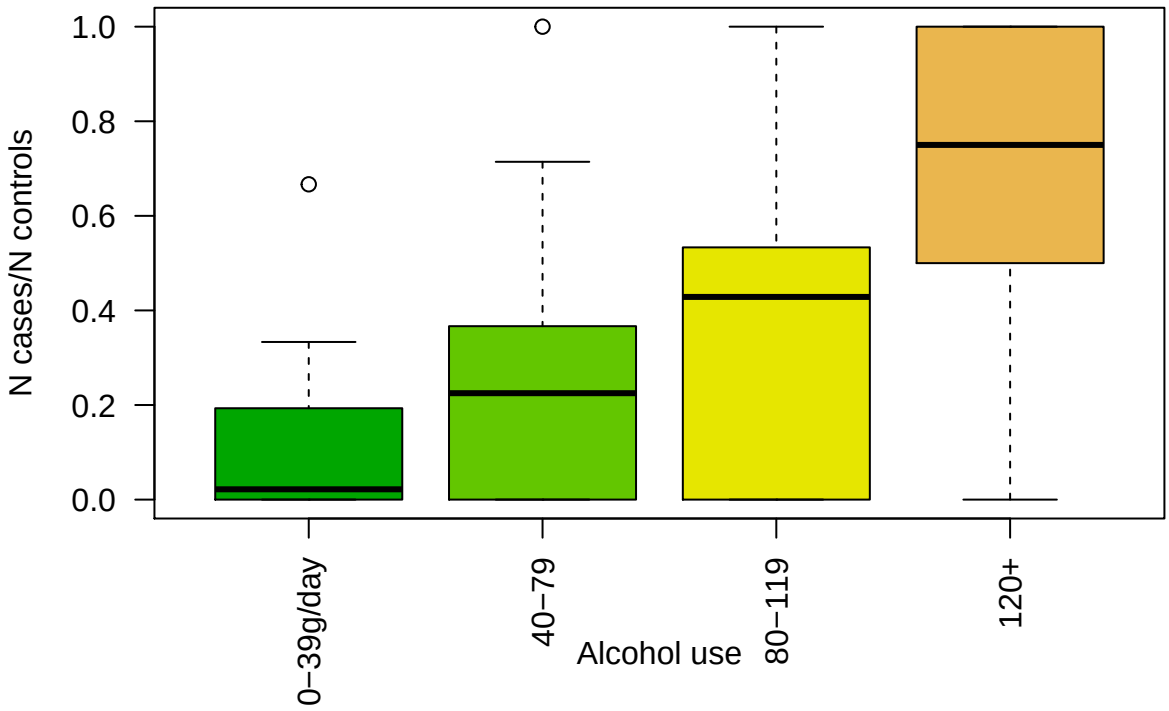
In the `esoph` dataset, the variable of interest regarding cancer risk should be the ratio of the number of cases to the number of controls.

Plot the ratio of cases to the controls, against age, alcohol use, and tobacco use.

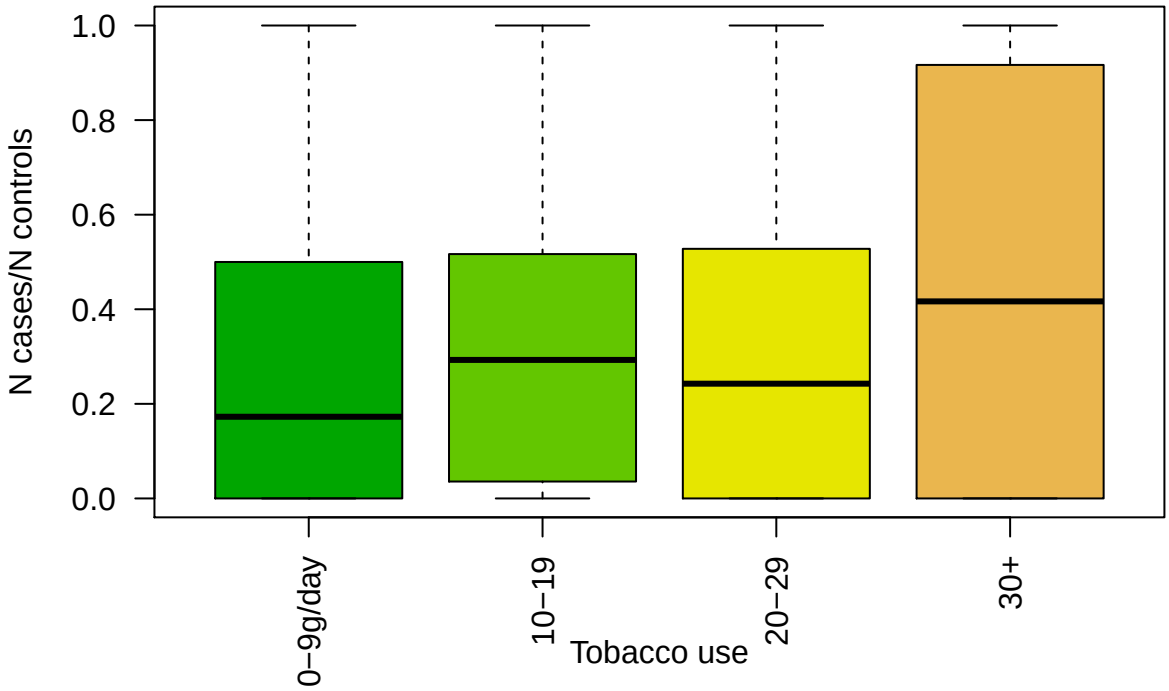
```
boxplot((ncases/ncontrols) ~ agegp, data = esoph, las = 2, col=terrain.colors(6), ylab="N cases/N contr
```




```
boxplot(ncases/ncontrols ~ alcgp, data = esoph, las = 2, col=terrain.colors(6), ylab="N cases/N controls
```




```
boxplot(ncases/ncontrols ~ tobgp, data = esoph, las = 2, col=terrain.colors(6), ylab="N cases/N controls
```



Which variable seems to show the strongest effect? It seems that alcohol use may be the most influential.

Loops with `for`, `sapply`, and `apply`

As you experienced when working on the homework, when studying some quantitative characteristic where there is uncertainty, we are frequently faced with running the same command multiple times to obtain reliable

information. Like many computer languages we can use built-in structures that repeat a command multiple times.

There are different control-flow constructs in R and in many other programming languages, and the simplest one is `for`.

```
for (i in 1:6) {  # the syntax - the parantheses and curly brackets - are important
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
```

```
i
```

```
## [1] 6
```

Here `i` is a variable we create while running the `for` command. Each round of the loop, `i` receives a new value, dictated by the vector within the parantheses. The loop then runs the `print` command, and starts over again with a new value of `i`.

This is a two line loop:

```
for (i in 1:6) {
  j = i^2
  print(j)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
## [1] 36
```

```
i
```

```
## [1] 6
```

```
j
```

```
## [1] 36
```

When you write `for (i in 1:6) {` and press enter, you see a `+` at the beginning of the new line. This suggests your code is not yet finished and R waits for the next part. In this case, it waits for the second parenthesis `}`.

The above command could also be written as `for (i in 1:6) { j=i^2; print(j) }` on the same line. Note that different commands on the same line have to be separated by a `;`.

The only difference is that when you write long chunk of code inside of a `for` loop, using new lines and indentation is easier for readers to understand your code.

Here is how we can create a vector called `j` using a `for` loop. We first need to create an empty vector, and then we fill it in, using the `c` function:

```
j = c()  # empty j
length(j)
```

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

```
j
```

```
## NULL
```

```
for (i in 1:6) {
  j = c(j, i^2)
  print(j)  # this is to follow what happens at each step of the loop
}
```

```
## [1] 1
## [1] 1 4
## [1] 1 4 9
## [1]  1  4  9 16
## [1]  1  4  9 16 25
## [1]  1  4  9 16 25 36
```

```
j
```

```
## [1]  1  4  9 16 25 36
```

An alternative way would be using an index:

```
j = c()  # redefine j as an empty vector
for (i in 1:6) {
  j[i] = i^2
  print(j)
}
```

```
## [1] 1
## [1] 1 4
## [1] 1 4 9
## [1]  1  4  9 16
## [1]  1  4  9 16 25
## [1]  1  4  9 16 25 36
```

```
j
```

```
## [1]  1  4  9 16 25 36
```

Let's now repeat the first part of Homework Q4 using a loop, this time collecting 100 points:

```
dev100 = c()
length(dev100)
```

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

```
N = 100
for (i in 1:100) {  # i is just a counter here. it is not used inside the function
  dev100[i] = abs( 0.5 - mean( runif(N) < 0.5 ) )  # calculate the difference and add to the vector
}
length(dev100)
```

```
## [1] 100
```

In R there is an alternative way to run loops on vectors, called `sapply`. It is a special kind of a function running for loops. It can be faster than `for`, is more convenient to use (you don't have to define your output object in advance). Also, the variables created when it is run are *not* available in the global environment.

Let's repeat the `for` loop with `sapply`.

```
myvec = sapply(1:6, function(h) {
  k = h^2
  return(k)
})
myvec
```

```
## [1]  1  4  9 16 25 36
```

```
# these objects are not anymore available in the global environment
h
```

```
## Error in eval(expr, envir, enclos): object 'h' not found
```

```
k
```

```
## Error in eval(expr, envir, enclos): object 'k' not found
```

The `return` function is useful when you have multiple commands in the loop, but here it is not necessary. Curly brackets are also not necessary with a single command. Hence, this also works:

```
myvec = sapply(1:6, function(h) h^2 )
myvec
```

```
## [1]  1  4  9 16 25 36
```

Let's repeat the second part of Homework Q4 using a `sapply` loop, again collecting 100 points:

```
N = 1e6
dev1m = sapply(1:100, function(i) {  # i is the counter this time
  abs( 0.5 - sum( runif(N) < 0.5 )/N )
})  # remember closing both parantheses
length(dev1m)
```

```
## [1] 100
```

```
dev1m
```

```
##   [1] 0.000428 0.000700 0.000299 0.000052 0.000156 0.000488 0.000227
##   [8] 0.000299 0.001164 0.000394 0.000361 0.000192 0.000244 0.000104
##  [15] 0.000308 0.000072 0.000350 0.000106 0.000408 0.000231 0.000204
##  [22] 0.000034 0.000798 0.000209 0.001373 0.000464 0.000282 0.000020
##  [29] 0.001181 0.000093 0.000336 0.001138 0.000336 0.000061 0.000205
##  [36] 0.000045 0.000163 0.001055 0.000165 0.000529 0.000415 0.000170
##  [43] 0.000078 0.000359 0.000806 0.000266 0.000758 0.000302 0.000406
##  [50] 0.001267 0.000805 0.000591 0.000956 0.000146 0.000525 0.000222
##  [57] 0.000129 0.000521 0.000640 0.000404 0.000050 0.000929 0.000038
##  [64] 0.001021 0.000118 0.000942 0.000260 0.000053 0.000239 0.000078
##  [71] 0.000456 0.000750 0.000029 0.000076 0.000261 0.000353 0.000604
##  [78] 0.000300 0.000386 0.000425 0.000284 0.000640 0.000495 0.000936
##  [85] 0.000129 0.000392 0.000709 0.001219 0.000627 0.000541 0.000516
##  [92] 0.000022 0.000182 0.000144 0.000376 0.000061 0.000397 0.000700
##  [99] 0.000001 0.000916
```

`apply` is a very useful recursive function tailored for matrices in R. It applies a specific subfunction on rows or columns of a matrix, which is defined by its second argument. This

```
me = matrix(1:300, 50, 6)
apply(me, 1, sum) # apply the function on rows
```

```
##  [1]  756  762  768  774  780  786  792  798  804  810  816  822  828  834
## [15]  840  846  852  858  864  870  876  882  888  894  900  906  912  918
## [29]  924  930  936  942  948  954  960  966  972  978  984  990  996 1002
## [43] 1008 1014 1020 1026 1032 1038 1044 1050
```

In this case, `apply` is running a shortcut version of the below type of command:

```
mem = c()
for (i in 1:50) {
  mem[i] = sum(me[i,])  # choose each row and calculate the sum
}
mem
```

```
##  [1]  756  762  768  774  780  786  792  798  804  810  816  822  828  834
## [15]  840  846  852  858  864  870  876  882  888  894  900  906  912  918
## [29]  924  930  936  942  948  954  960  966  972  978  984  990  996 1002
## [43] 1008 1014 1020 1026 1032 1038 1044 1050
```

```
apply(me, 2, sum) # apply the function on columns
```

```
## [1]  1275  3775  6275  8775 11275 13775
```

```
# this function runs the same
colSums(me)
```

```
## [1]  1275  3775  6275  8775 11275 13775
```

```
apply(me, 2, min) # find the min value of each column
```

```
## [1]   1  51 101 151 201 251
```

We may also have a function returning not a single value but a vector in `apply`. For example, there is a function for cumulative sum called `cumsum`. The following returns a matrix:

```
mecumsum = apply(me, 2, cumsum)
dim(mecumsum)
```

```
## [1] 50  6
```

```
tail(mecumsum)
```

```
##       [,1] [,2] [,3] [,4]  [,5]  [,6]
## [45,] 1035 3285 5535 7785 10035 12285
## [46,] 1081 3381 5681 7981 10281 12581
## [47,] 1128 3478 5828 8178 10528 12878
## [48,] 1176 3576 5976 8376 10776 13176
## [49,] 1225 3675 6125 8575 11025 13475
## [50,] 1275 3775 6275 8775 11275 13775
```

If the function to apply involves multiple commands, then you will need to use the following syntax, with the curly brackets:

```
mex = apply(me, 2, function(x) {  # at each step, x is the vector
  meanx = mean(x)
  sdx = sd(x)
  rnorm(1, mean=meanx, sd=sdx )  #
})
head(mex)
```

```
## [1]  49.57330  70.53011 131.83426 185.10962 209.07198 281.64779
```

There are yet other ways to run loops in R, on pairs of vectors and on lists. We will learn about them later.

The `sample` function for random sampling

A very useful function in R for simulation experiments is called `sample`. Its usage is `sample(x, size, replace)`, where the function randomly chooses a specified number of elements (`size`) from any vector `x`, with or without **replacement**. Sampling without replacement is the default and is just a permutation of the vector `x`. With replacement means some elements can be chosen multiple times, and others not. Each have probability being chosen `1/n`, each time.

```
myvec = 30:42
set.seed(1)
sample(myvec)
```

```
##  [1] 33 34 36 39 31 37 40 42 35 30 41 32 38
```

```r
sample(myvec)
```

```
##  [1] 34 39 35 37 38 33 40 36 31 32 30 41 42
```

```r
sample(myvec)
```

```
##  [1] 30 34 39 33 41 38 40 31 37 32 36 42 35
```

```r
sample(myvec, size = 2)  # only choose 2 elements
```

```
## [1] 35 39
```

`sample` also uses the same RNG, so it depends on the seed:

```r
set.seed(1)
sample(myvec)
```

```
##  [1] 33 34 36 39 31 37 40 42 35 30 41 32 38
```

`sample` with replacement:

```r
set.seed(1)
sample(myvec, replace = T)
```

```
##  [1] 33 34 37 41 32 41 42 38 38 30 32 32 38
```

```r
sample(myvec, replace = T)
```

```
##  [1] 34 40 36 39 42 34 40 42 32 38 31 33 35
```

This won't work:

```r
sample(myvec, size = 20)  # choose 20 elements
```

```
## Error in sample.int(length(x), size, replace, prob): cannot take a sample larger than the population
```

But this will:

```r
sample(myvec, size = 20, replace = T)  # choose 20 elements with replacement
```

```
##  [1] 30 34 41 34 36 37 36 32 40 38 40 31 39 35 40 38 40 37 36 40
```

Functions to compare and manipulate vectors

`unique`, `table`, `identical`, `duplicated`, `reverse`, `order`, etc

```r
myvec2 = sample(myvec, replace = T)
myvec2
```

```
##  [1] 30 36 39 39 36 41 35 33 30 31 34 36 38
```

```r
# check whether two vectors are identical
identical(myvec, sort( myvec2) )
```

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

```r
# check unique elements of the new vector
unique(myvec2)
```

```
## [1] 30 36 39 41 35 33 31 34 38
```

```r
# same thing, more convenient to study
sort(unique(myvec2))
```

```
## [1] 30 31 33 34 35 36 38 39 41
```

```r
# check how many elements are lost
length(myvec2)
```

```
## [1] 13
```

```r
# check which elements are lost from vector1 compared to vector2
setdiff(myvec, myvec2)
```

```
## [1] 32 37 40 42
```

```r
# check the frequency of elements, similar to a histogram
table(myvec2)
```

```
## myvec2
## 30 31 33 34 35 36 38 39 41
##  2  1  1  1  1  3  1  2  1
```

```r
# check which elements are chosen multiple times
myvec2t = table(myvec2)
myvec2t[myvec2t > 1]
```

```
## myvec2
## 30 36 39
##  2  3  2
```

```
names(myvec2t)[myvec2t > 1]
```

```
## [1] "30" "36" "39"
```

```
as.numeric(names(myvec2t)[myvec2t > 1])
```

```
## [1] 30 36 39
```

```
# a shorter way to do the same:
duplicated(myvec2)  # returns TRUE only for occurrances >1 of any element
```

```
##  [1] FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE
## [12]  TRUE FALSE
```

```
# to see better how this works:
rbind(myvec2, duplicated(myvec2))
```

```
##        [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
## myvec2   30   36   39   39   36   41   35   33   30    31    34    36
##           0    0    0    1    1    0    0    0    1     0     0     1
##        [,13]
## myvec2    38
##            0
```

```
unique(myvec2[duplicated(myvec2)])
```

```
## [1] 39 36 30
```

```
# sign of elements
sign(-3:3)
```

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

Another useful approach here for comparing and studying vectors is the `%in%` matching operator, returning `TRUE` if *any* element of vector 1 matches those of vector 2. This is useful because the equality operator `==` only checks for one-to-one matching.

```
myvec %in% myvec2
```

```
##  [1]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE
## [12]  TRUE FALSE
```

```
length(myvec %in% myvec2)
```

```
## [1] 13
```

```
myvec == myvec2
```

```
##  [1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE
```

```
length(myvec == myvec2)
```

```
## [1] 13
```

```
myvec2 %in% myvec  # this is all true, because all
```

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

```
length(myvec2 %in% myvec)
```

```
## [1] 13
```

```
summary(myvec)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##      30      33      36      36      39      42
```

```
summary(myvec2)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   30.00   33.00   36.00   35.23   38.00   41.00
```

```
min(myvec)
```

```
## [1] 30
```

```
min(myvec2)
```

```
## [1] 30
```

```
max(myvec)
```

```
## [1] 42
```

```
max(myvec2)
```

```
## [1] 41
```

```r
which.min(myvec)  # the index
```

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

```r
which.min(myvec2)
```

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

```r
# the ranks (average assigned for ties)
rank(myvec2)
```

```
##  [1]  1.5  8.0 11.5 11.5  8.0 13.0  6.0  4.0  1.5  3.0  5.0  8.0 10.0
```

```r
data.frame( myvec2 = myvec2, rank = rank(myvec2) )
```

```
##    myvec2 rank
## 1      30  1.5
## 2      36  8.0
## 3      39 11.5
## 4      39 11.5
## 5      36  8.0
## 6      41 13.0
## 7      35  6.0
## 8      33  4.0
## 9      30  1.5
## 10     31  3.0
## 11     34  5.0
## 12     36  8.0
## 13     38 10.0
```

```r
# the indices of elements from smallest to largest
order(myvec2)
```

```
##  [1]  1  9 10  8 11  7  2  5 12 13  3  4  6
```

```r
# which you can use to sort
myvec2[order(myvec2)]
```

```
##  [1] 30 30 31 33 34 35 36 36 36 38 39 39 41
```

```r
# the index of the max element
which.max(myvec)
```

```
## [1] 13
```

```r
which.max(myvec2)
```

```
## [1] 6
```

```r
myvec2[which.max(myvec2)]
```

```
## [1] 41
```

```r
# a 2D comparison of elements
table(myvec, myvec2)
```

```
##      myvec2
## myvec 30 31 33 34 35 36 38 39 41
##    30  1  0  0  0  0  0  0  0  0
##    31  0  0  0  0  0  1  0  0  0
##    32  0  0  0  0  0  0  0  1  0
##    33  0  0  0  0  0  0  0  1  0
##    34  0  0  0  0  0  1  0  0  0
##    35  0  0  0  0  0  0  0  0  1
##    36  0  0  0  0  1  0  0  0  0
##    37  0  0  1  0  0  0  0  0  0
##    38  1  0  0  0  0  0  0  0  0
##    39  0  1  0  0  0  0  0  0  0
##    40  0  0  0  1  0  0  0  0  0
##    41  0  0  0  0  0  1  0  0  0
##    42  0  0  0  0  0  0  1  0  0
```

```r
# reverse the order
rev(myvec2)
```

```
##  [1] 38 36 34 31 30 33 35 41 36 39 39 36 30
```