Lecture 7: solutions

Statistical Computing with R

Exercise 1

1.

In the top left of `RStudio` you find a button *file*, hover over and select *New Project...*. Select *New Directory* and *New Project*, give the directory (the folder) a name and a suiting location.

2.

Go to *file*, select *New file* and *R Markdown*.

3.

Try to *Knit* your document. Make sure that *output: pdf_document* in the YAML header.

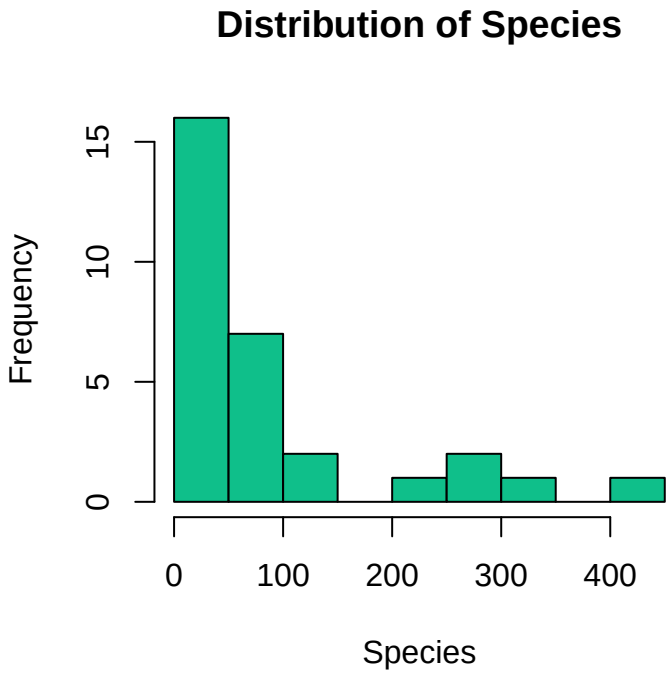
Exercise 2

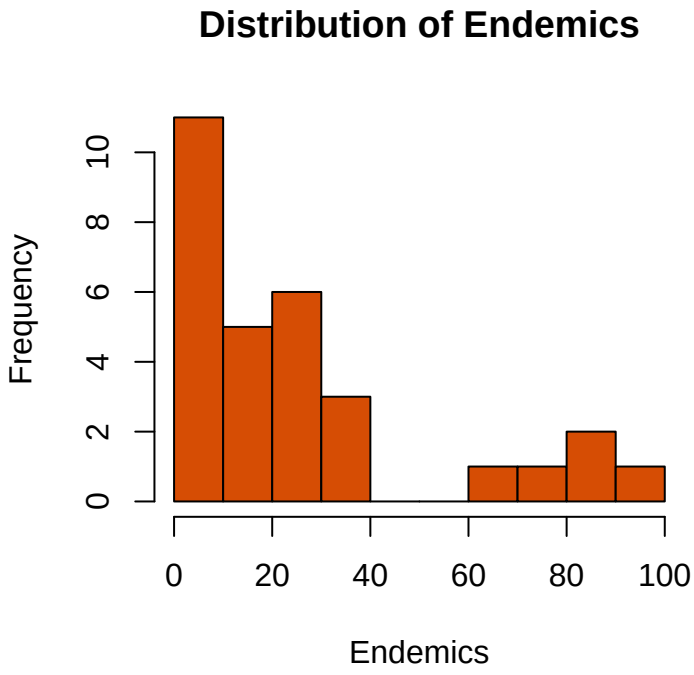
We will use the function `randomColor()` from the library `randomcoloR` inside the plot functions, to generate a different "random" color for each of the plots.

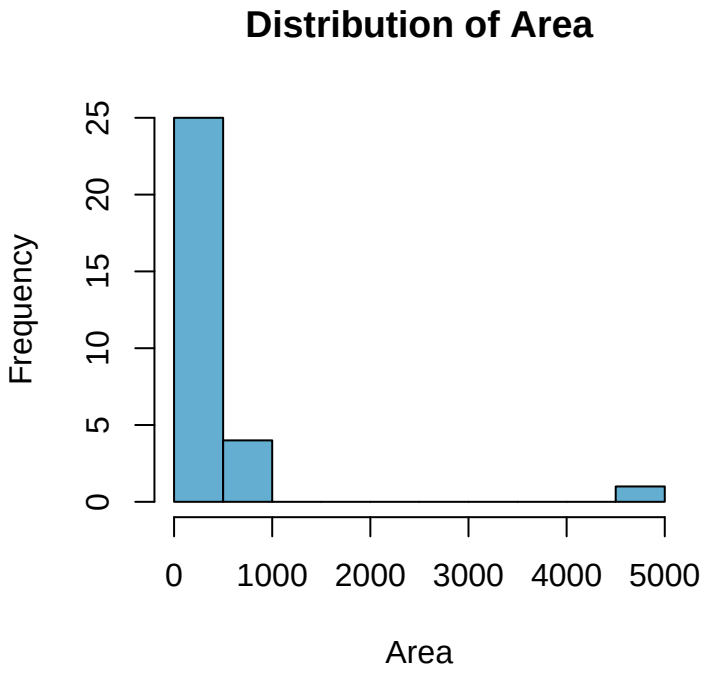
```
library(randomcoloR)

f <- function(df){

  # check the type of data in each column
  vartype <- sapply(df, class)

  # check if the dataframe contains any
  if(sum(c("character", "numeric", "factor") %in% vartype) == 0){
    print("This dataframe contains no numeric, character or factor values")
  }

  else {
    #This code avoids that the plots created by the for loop overlap the page
    #margins. This is not part of the course material, and you do not have to
    #be able to use/understand this code.
    my_plot_hook <- function(x, options)
    paste("\n", knitr::hook_plot_tex(x, options), "\n")
    knitr::knit_hooks$set(plot = my_plot_hook)

    # print histograms and barplots
    for(i in 1:ncol(df)){
      if(vartype[i] == "numeric"){
        hist(df[,i], xlab = names(df)[i], col = randomColor(1), breaks = 10,
             main = paste('Distribution of', names(df)[i]))
      } else if(vartype[i] == "character" | vartype[i] == "factor"){
        barplot(table(df[,i]), col = randomColor(1),
             main = paste('Frequency Distribution of', names(df)[i]) )
      }
    }
  }
}
```

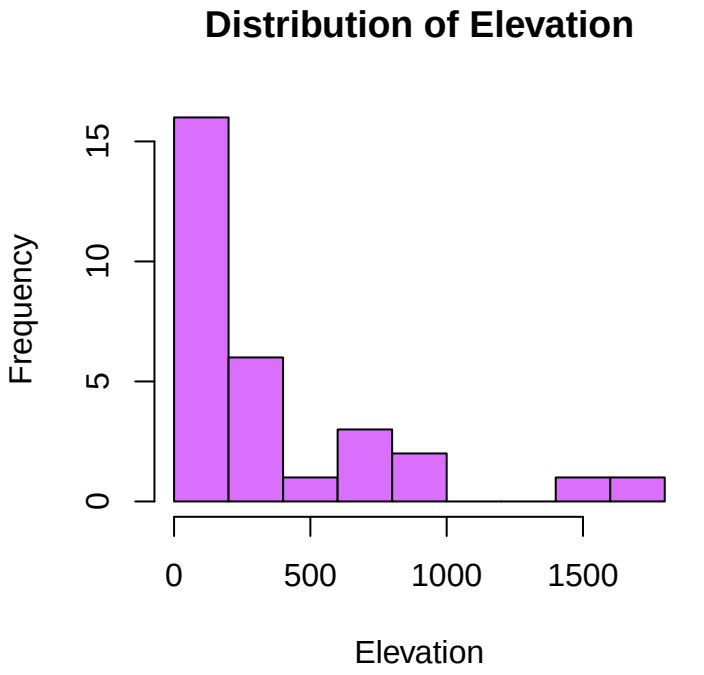
Now, let's check our function with the `gala` and `amlxray` datasets.

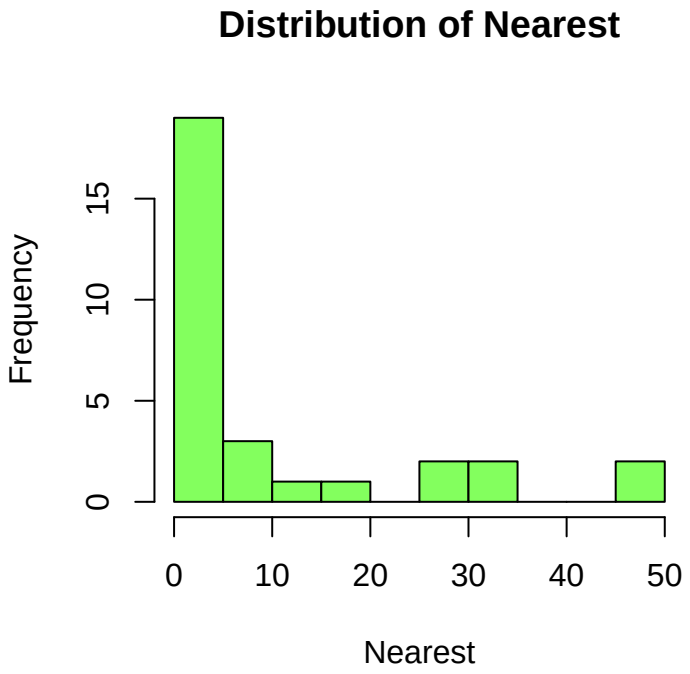
```
library(faraway)

f(gala)
```

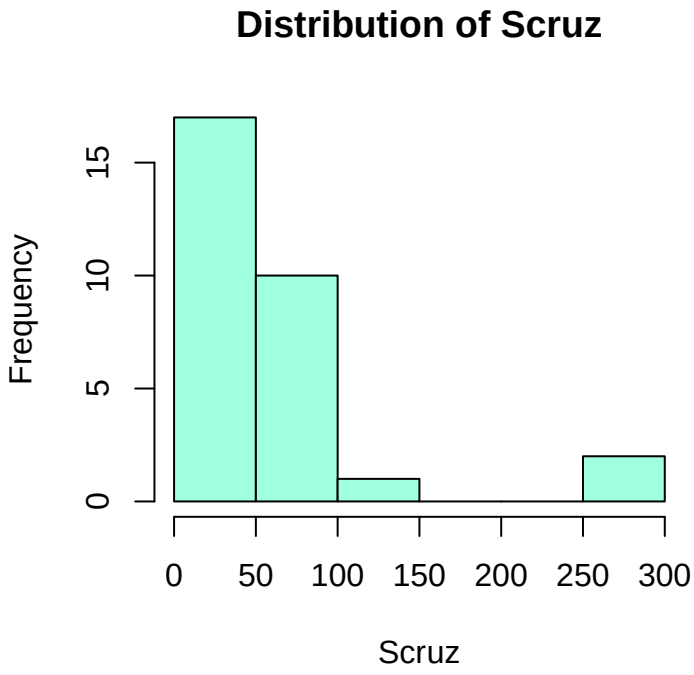

Distribution of Species
Frequency
0
5
10
15
0
100
200
300
400
Species

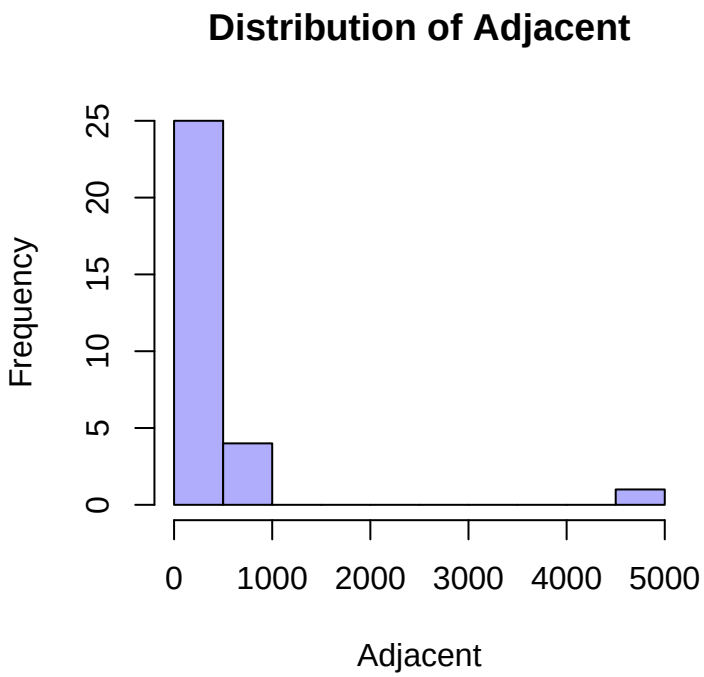

Distribution of Endemics
Frequency
0
2
4
6
8
10
0
20
40
60
80
100
Endemics


Distribution of Area
Frequency
0
5
10
15
20
25
0
1000
2000
3000
4000
5000
Area

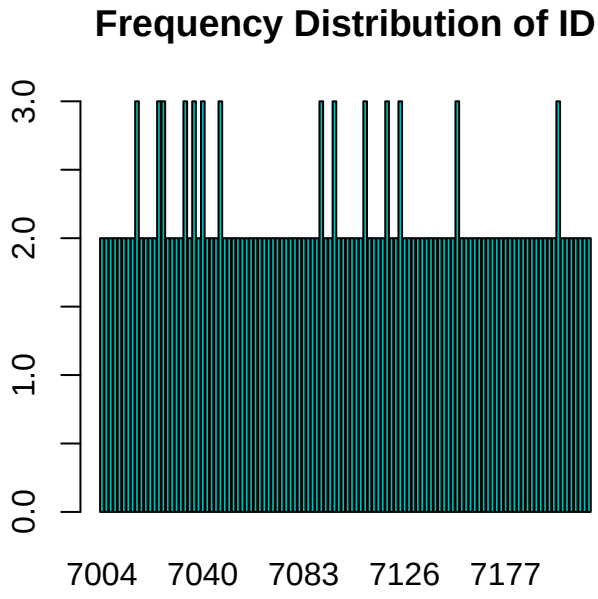

Distribution of Elevation
Frequency
0
5
10
15
0
500
1000
1500
Elevation

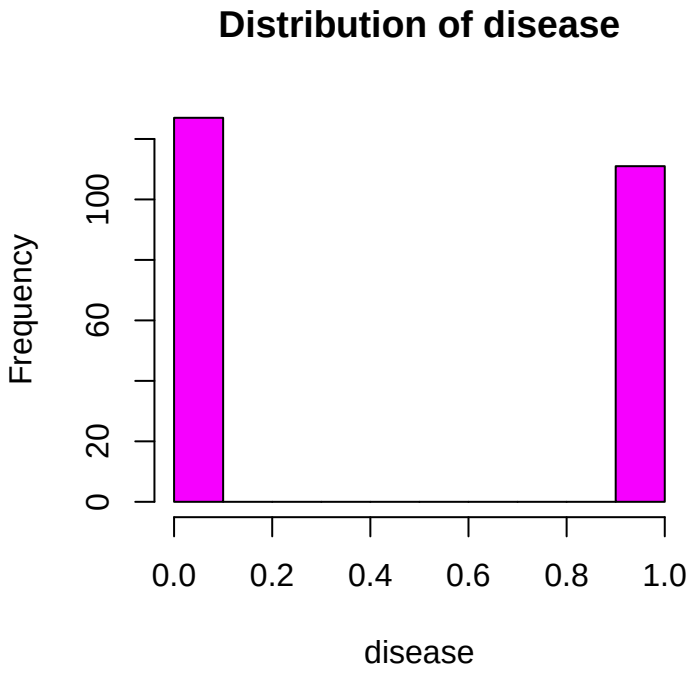

Distribution of Nearest
Frequency
0
5
10
15
0
10
20
30
40
50
Nearest

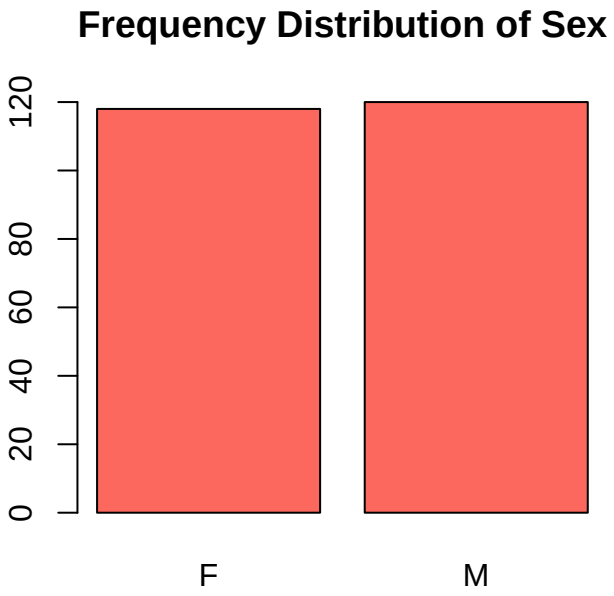

Distribution of Scruz
Frequency
0
5
10
15
0
50
100
150
200
250
300
Scruz

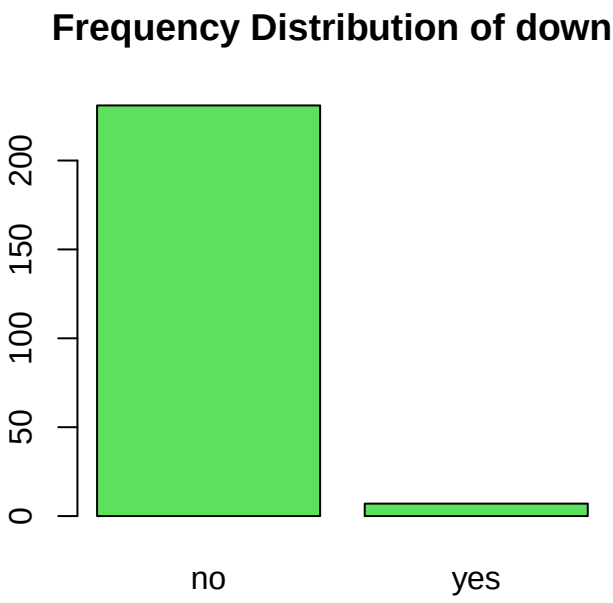

Distribution of Adjacent
Frequency
0
5
10
15
20
25
0
1000
2000
3000
4000
5000
Adjacent

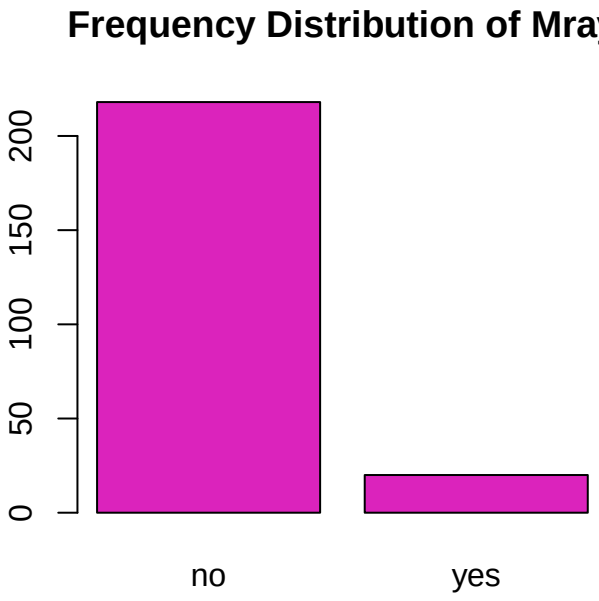
```
f(amlxray)
```

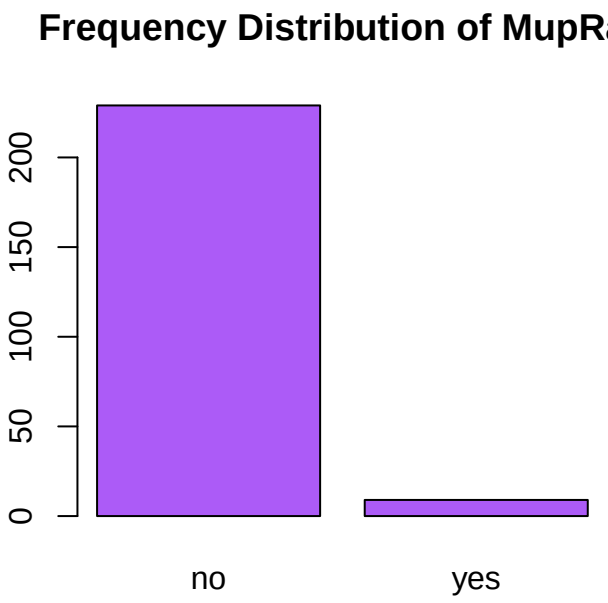

Frequency Distribution of ID
0.0
1.0
2.0
3.0
7004
7040
7083
7126
7177

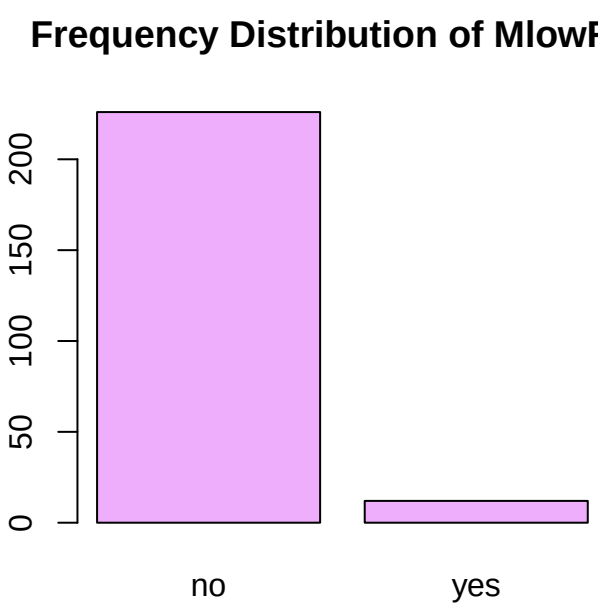

Distribution of disease
Frequency
0
20
60
100
0.0
0.2
0.4
0.6
0.8
1.0
disease

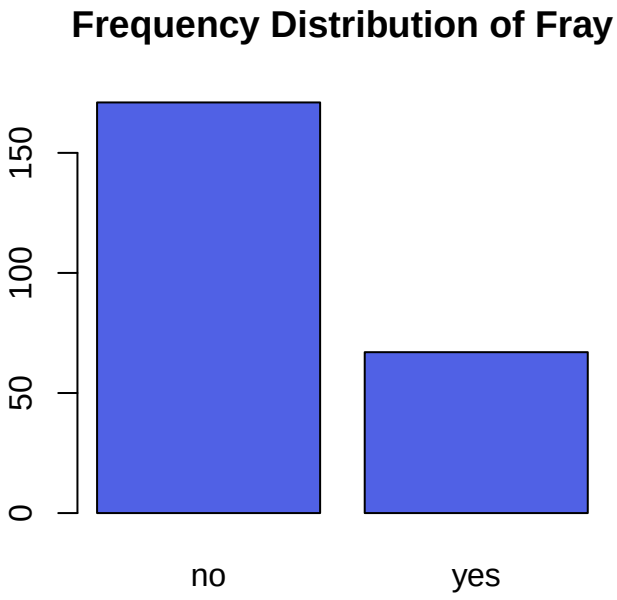

Frequency Distribution of Sex
0
20
40
60
80
120
F
M

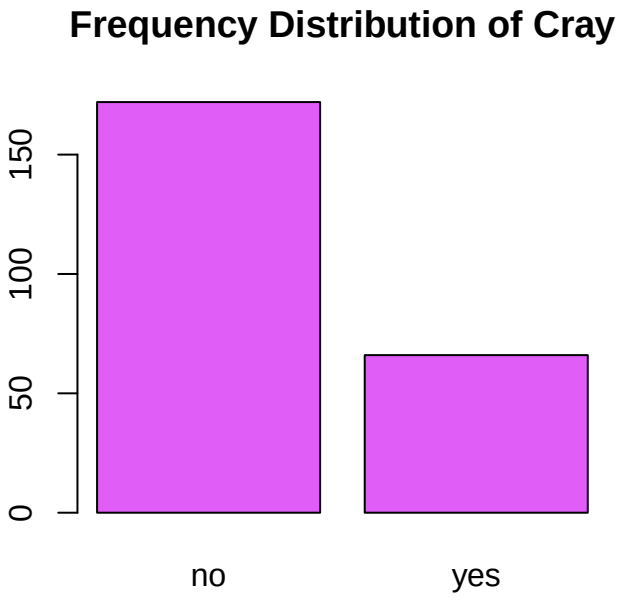

Frequency Distribution of downs
0
50
100
150
200
no
yes


Frequency Distribution of Mray
0
50
100
150
200
no
yes


Frequency Distribution of MupRay
0
50
100
150
200
no
yes


Frequency Distribution of MlowRay
0
50
100
150
200
no
yes


Frequency Distribution of Fray
0
50
100
150
no
yes


Frequency Distribution of Cray
0
50
100
150
no
yes

Exercise 3

```
library(MASS)
```

1.

There are many ways to calculate statistics of variables. An easy way to do this is to use the function `apply`, which requires the three following inputs:

- An object (typically, a matrix or data frame).
- The `MARGIN` to which you want to apply the function (1 for computations by row, 2 for computations by column, c(1, 2) for computations by row *and* column).
- The function that you want to apply.

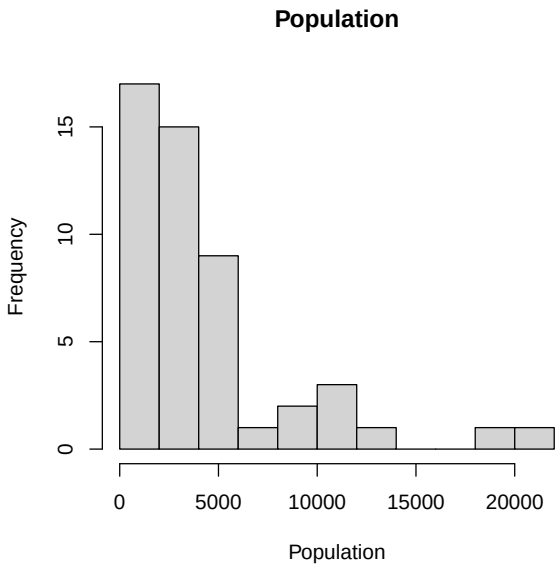
```
# obtaining variable names
df <- data.frame("variable"=colnames(state.x77))

# calculating the mean
df$mean <- colMeans(state.x77)

# calculating the median using apply
df$median <- apply(state.x77, 2, median)

# calculating the standard deviation using apply
df$std_dev <- apply(state.x77, 2, sd)

# calculating the mean using apply
library(moments)
df$skewness <- apply(state.x77, 2, skewness)

# show df
df
```

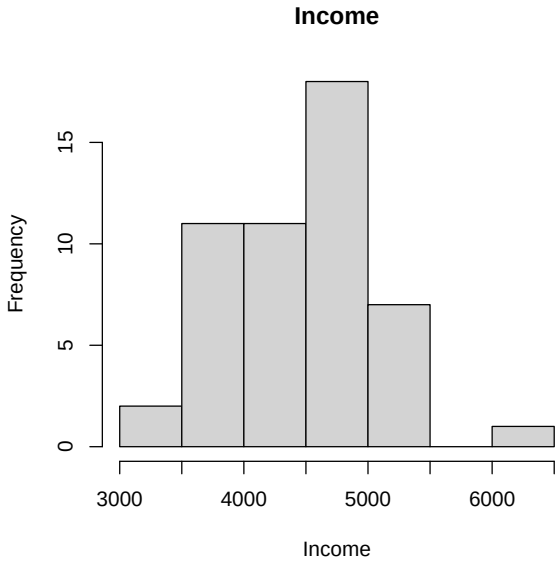
```
##     variable       mean    median      std_dev   skewness
## 1 Population  4246.4200  2838.500 4.464491e+03  1.9813947
## 2     Income  4435.8000  4519.000 6.144699e+02  0.2109882
## 3 Illiteracy     1.1700     0.950 6.095331e-01  0.8437669
## 4   Life Exp    70.8786    70.675 1.342394e+00 -0.1582224
## 5     Murder     7.3780     6.850 3.691540e+00  0.1333186
## 6    HS Grad    53.1080    53.250 8.076998e+00 -0.3290666
## 7      Frost   104.4600   114.500 5.198085e+01 -0.3776493
## 8       Area 70735.8800 54277.000 8.532730e+04  4.2244553
```

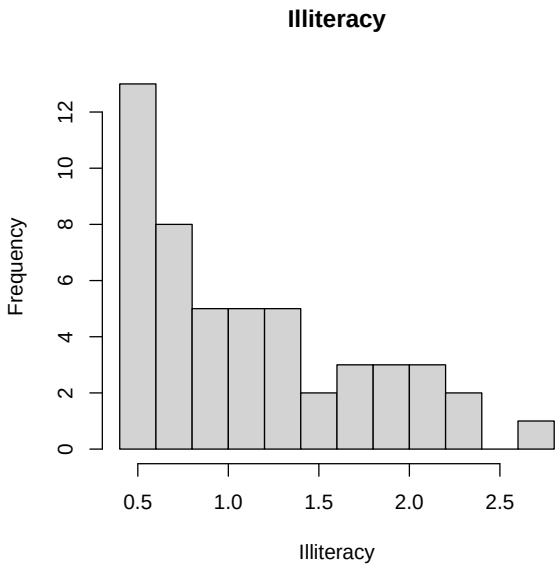
2.

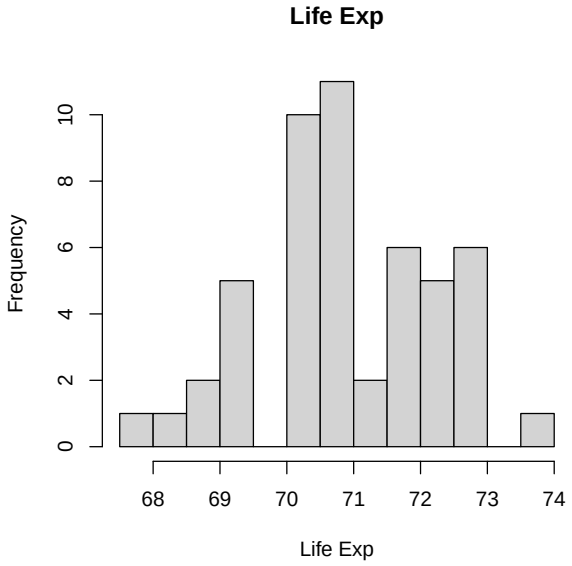
There are many ways to solve this exercise. The solution below uses a `while` loop (covered in today's lecture). A `for`-loop would work just as well. For the `main` argument (the title of the plot), we can retrieve the column name and use that as input.

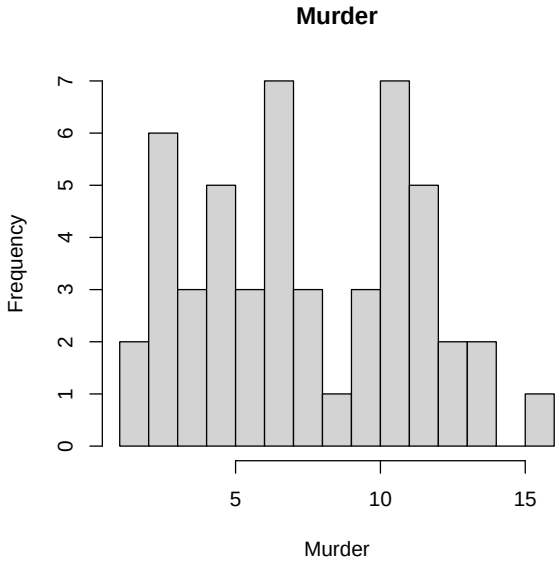
```
i = 1
while(i <= dim(state.x77)[2]){
  hist(state.x77[,i], main = colnames(state.x77)[i],
       breaks = 10, xlab = colnames(state.x77)[i])
  i = i + 1
}
```

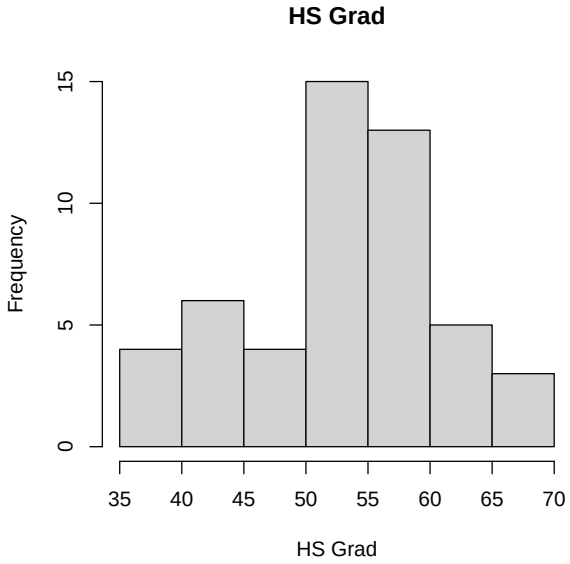

Population
Frequency
0
5
10
15
0
5000
10000
15000
20000
Population

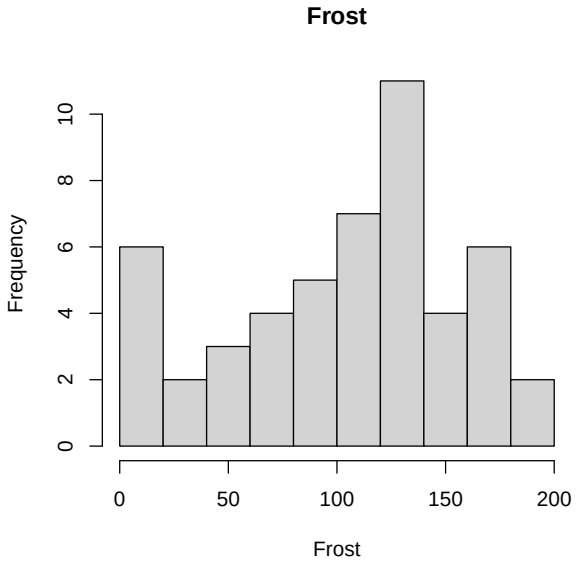

Income
Frequency
0
5
10
15
3000
4000
5000
6000
Income

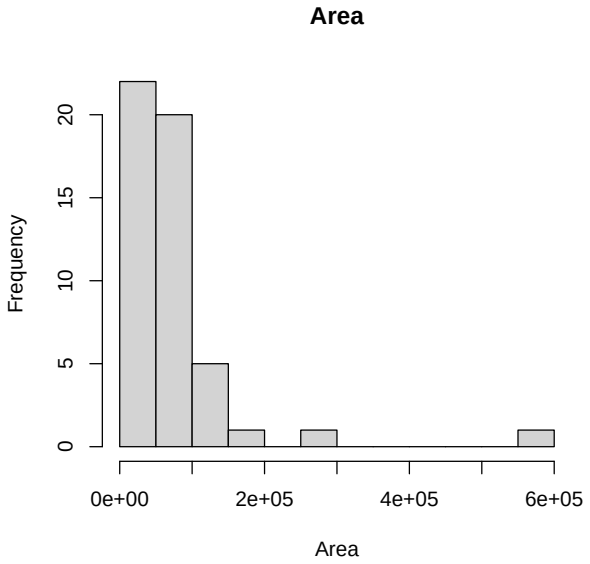

Illiteracy
Frequency
0
2
4
6
8
10
12
0.5
1.0
1.5
2.0
2.5
Illiteracy


Life Exp
Frequency
0
2
4
6
8
10
68
69
70
71
72
73
74
Life Exp


Murder
Frequency
0
1
2
3
4
5
6
7
5
10
15
Murder


HS Grad
Frequency
0
5
10
15
35
40
45
50
55
60
65
70
HS Grad





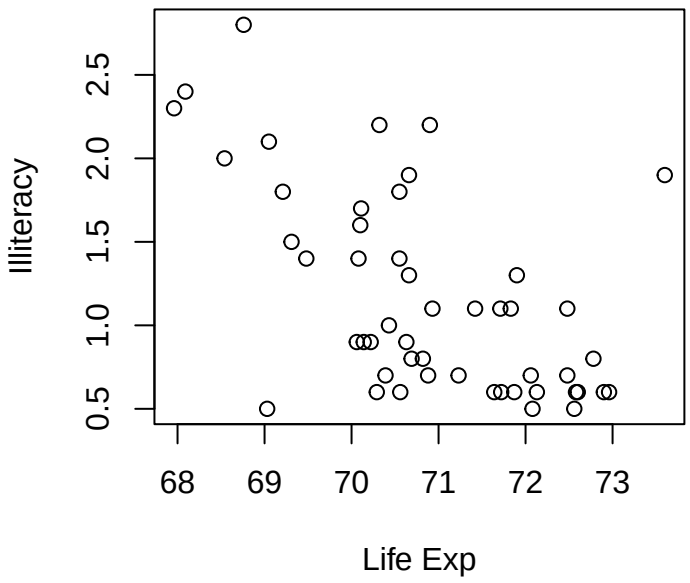
3.

Murder, income, and HS grade seems to be relatively symmetrically distributed over all states. Population, illiteracy, and area are positively skewed, and frost and Life exp seem to be negatively skewed.

4.

The data depicts a downward trend, indicating that lower levels of illiteracy, thus higher levels of literacy, are linked to higher life expectancy. There is a lot of research being done on the relations between life expectancy, income, education, and literacy which can give more inside in this phenomenon.

```
# we use `` to refer to variables with spaces in the name.
plot(Illiteracy ~ `Life Exp`, data = state.x77)
```



5.

The correlation test shows a moderately strong negative correlation (-0.588) that is statistically significant $p < 0.01$. A possible explanation for this result might be that life expectancy depends on a number of factors, including socioeconomic factors such as literacy, education level, and income.

```
cor.test(state.x77[,"Illiteracy"], state.x77[,"Life Exp"])
```

```
##
##  Pearson's product-moment correlation
##
## data:  state.x77[, "Illiteracy"] and state.x77[, "Life Exp"]
## t = -5.0427, df = 48, p-value = 6.969e-06
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  -0.7448226 -0.3708811
## sample estimates:
##        cor
## -0.5884779
```

Exercise 4

1.

```
# 1.
c1 = c(0, 1)
c2 = expand.grid(c1, c1)
c2s = rowSums(c2)
c2
```

```
##   Var1 Var2
## 1    0    0
## 2    1    0
## 3    0    1
## 4    1    1
```

`c2` appears to be all possible combinations of two independent binary experiments. `c2s` appears to be the number of successes of each of the combinations. The dimensions of `c2` are $2^2 = 4$ by 2. `expand.grid()` seems to create a data frame with all possible combinations of the input vectors.

```
c2s
```

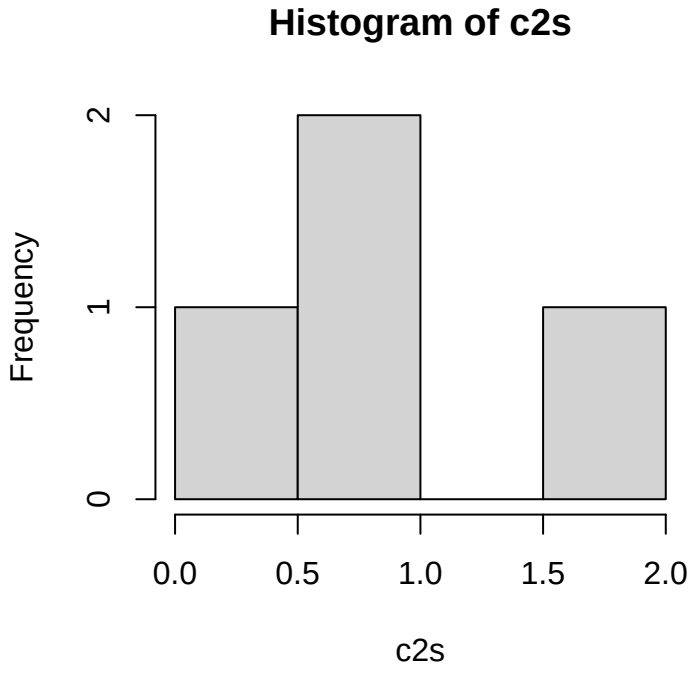
```
## [1] 0 1 1 2
```

`rowSums` gives the sum of all values of each row. Instead of `rowSums`, you could also use `apply(c2, 1, sum)`.

```
dim(c2)
```

```
## [1] 4 2
```

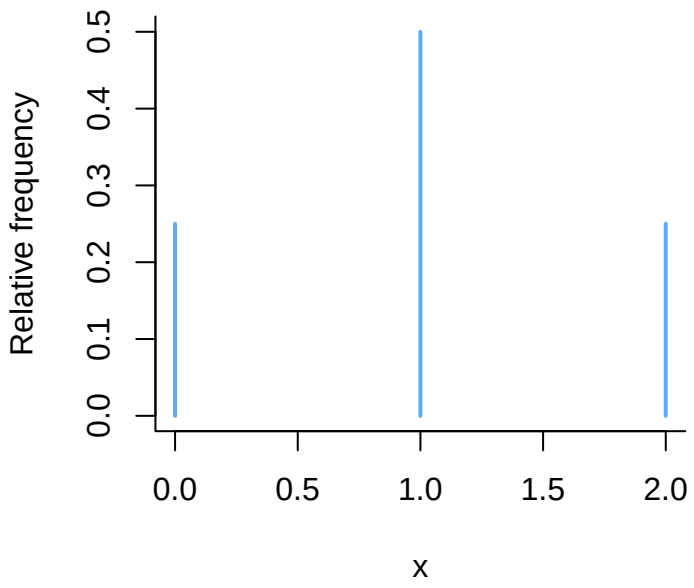
```
hist(c2s)
```



The histogram depicts all outcomes of 2 independent Bernoulli trials and their frequencies. In this case, a

histogram is not the best way to visualize this discrete distribution. Instead, the `pmf` function of the `ptmixed` package could be used (slide 41 L5).

```
library(ptmixed)
pmf(c2s, absolute = F, lwd = 2, col = 'steelblue2')
```



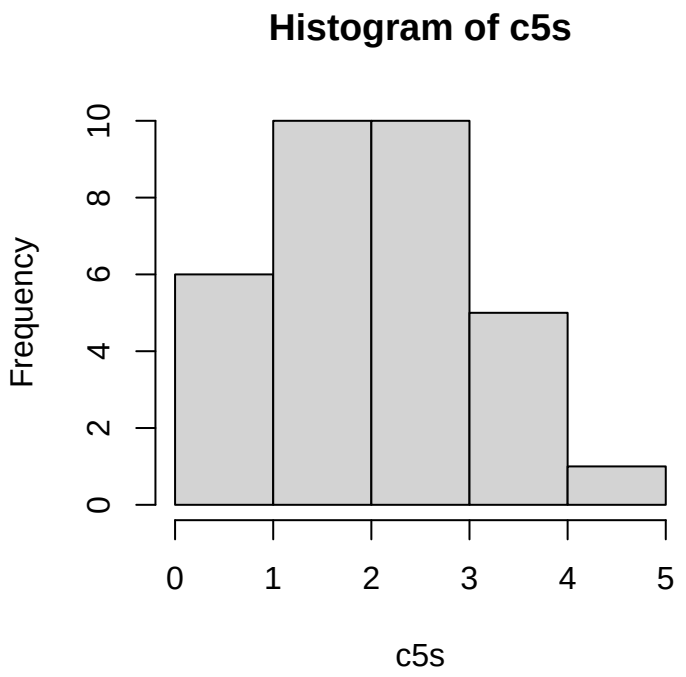
2.

The dimensions of `c5` are now $2^5 = 32$ and 5. The histogram seems to be getting a peak in the center.

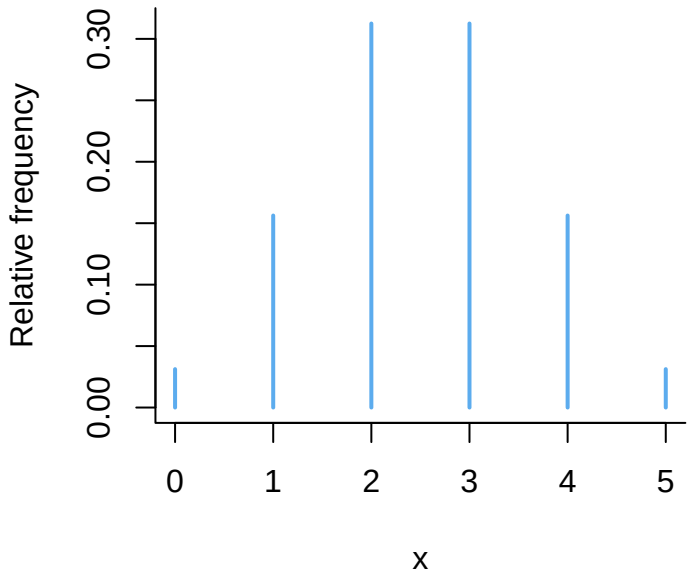
```
# 3.
c5 <- expand.grid(c1, c1, c1, c1, c1)
c5s <- rowSums(c5)
dim(c5)
```

```
## [1] 32  5
```

```
hist(c5s)
```



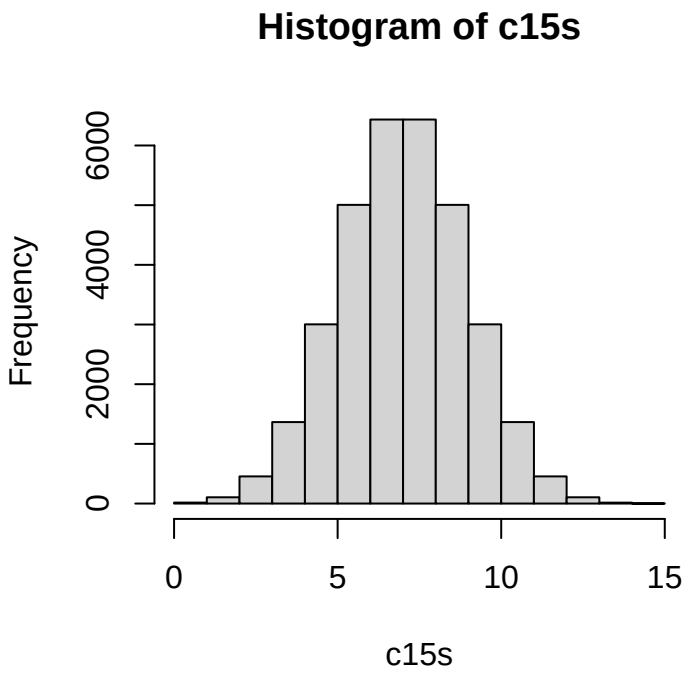

```
pmf(c5s, absolute = F, lwd = 2, col = 'steelblue2')
```



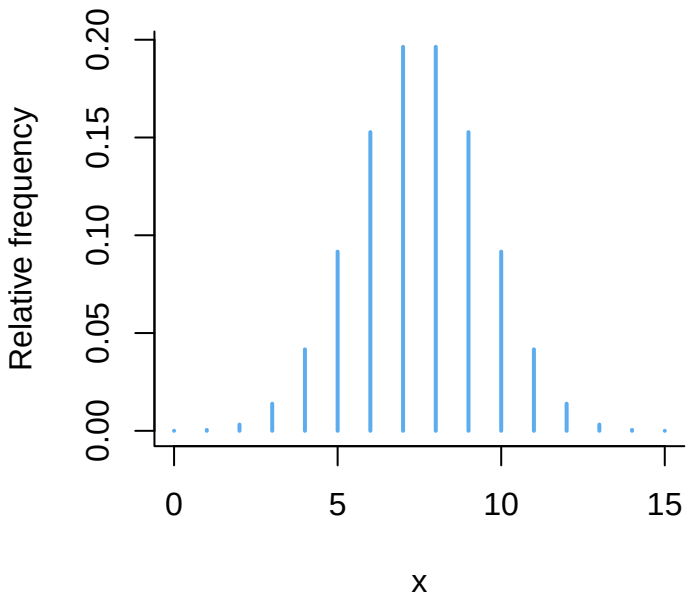
3.

The dimensions of `c15` will be $2^{15} = 32768$ by 15, because there are 2^{15} possible combinations of 15 binary variables. The histogram and pmf plot starts to look like a normal distribution.

```
# 4.
c15 <- expand.grid(replicate(15, c1, simplify = FALSE))
c15s <- rowSums(c15)
hist(c15s)
```




```
pmf(c15s, absolute = F, lwd = 2, col = 'steelblue2')
```



4.

The number of successes range between 0, no successes at all, and 15, all trials were successful. The mean is $n \cdot p = 15 \cdot 0.5 = 7.5$, and the standard deviation is $\sqrt{np(1-p)} \approx 1.94$. These numbers are not surprising, because they are the analytical solutions for the expected value and standard deviation of the binomial distribution with $n = 15$ and $p = 0.5$.

```
range(c15s)
```

```
## [1]  0 15
```

```
mean(c15s)
```

```
## [1] 7.5
```

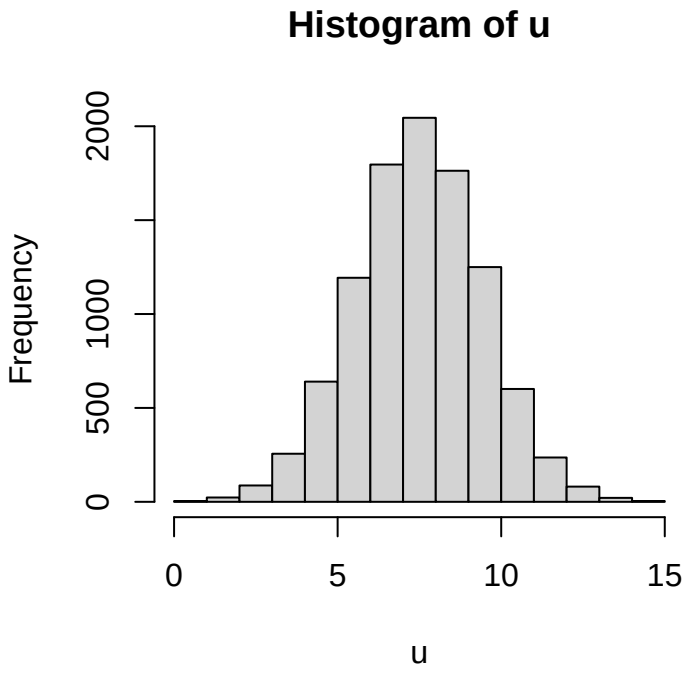
```
sd(c15s)
```

```
## [1] 1.936521
```

5.

We draw 1000 numbers from a normal distribution with $\mu = 7.5$ and $\sigma = 1.936521$. We can observe that the histogram looks similar to the one for `c15s`. The range of values `c15s` can take is between 0, no successes, and 15, all success. Thus we have 16 possible outcomes. To make the histogram of the normal draws more similar, the `breaks = 16` argument is used to bin the draws in a similar fashion.

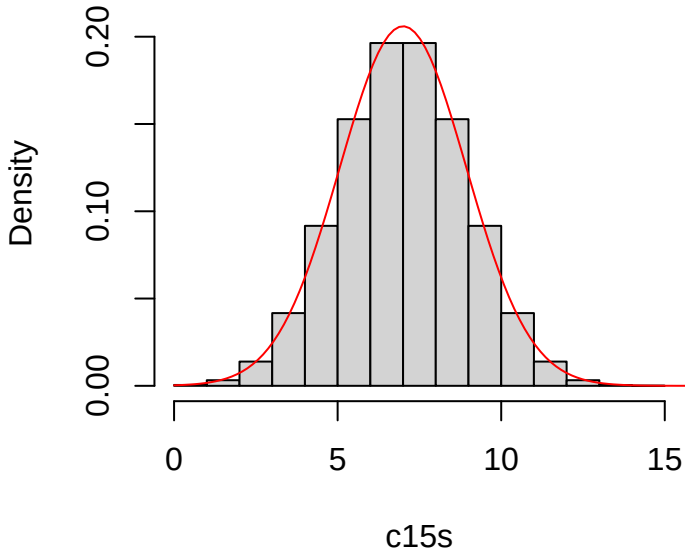
```
# 5.
u <- rnorm(1e4, mean(c15s), sd(c15s))
hist(u, breaks = 16, xlim = c(0, 15))
```



6.

First plot the histogram of `c15s`, with, instead of the frequencies, the density on the y-axis, by `prob = T`. Then, add the normal distribution by `add = T`, using `dnorm(x, mean(c15s)-0.5, sd(c15s))`. The -0.5 is to center the distribution nicely. We can now clearly see that the binomial distribution starts to approach the normal distribution for a large number of trials.

```
# 6.
hist(c15s, prob = T, main = "", ylim = c(0, 0.22))
curve(dnorm(x, mean(c15s)-0.5, sd(c15s)),
      from = 0,
      to = 16,
      ylab = "density",
      add = T,
      col = "red")
```


Density
0.00
0.10
0.20
0
5
10
15
c15s

Exercise 5

Let's first create a function which returns a vector of the first n elements of the Fibonacci sequence.

Notice that the n number is $n - 1 + n - 2$.

```
fibonacci_vector <- function(n){
  x <- numeric(n)
  # Allocate the first
  # two elements of the sequence
  x[c(1, 2)] <- 1

  if(n == 1){
    return(x[1])
  }

  if(n == 2){
    return(x[c(1, 2)])
  }

  for (i in 2:(n-1)){
    x[i+1] <- x[i] + x[i-1]
  }
  return(x)
}
```

To find out how many elements in the sequence we need to to sum to reach 22000, we can use the following `while()` loop:

```
i <- 0
x <- 0
while (x <= 22000){
  i <- i + 1
  x <- sum(fibonacci_vector(i))
}
print(i)
```

```
## [1] 21
```

An alternative, less efficient solution would be that of using a `for()` loop. In this case we can use a big number which we know we won't get and an `if()` statement that stops the loop. To stop a loop we can use `break`. See `?break`.

```
x <- 0
for (j in 1:10^5){
  x <- sum(fibonacci_vector(j))
  if (x >= 22000){
    print(j)
    break
  }
}
```

```
## [1] 21
```

Another way of doing this is using *recursion*. This function calls itself subtracting from the n original input until it reaches $n <= 1$. Then it returns the value successively until it reaches the initial call to the function. This returns the value of the n number of the series.

```r
fibonacci_number <- function(n){
  if(n <= 1){
    return(n)
  }
  return(fibonacci_number(n-1) + fibonacci_number(n-2))
}
```

To calculate how many iterations it takes:

```r
i <- 0
x <- 0
while (x <= 22000){
  i <- i + 1
  x <- x + fibonacci_number(i)
}
print(i)
```

```
## [1] 21
```

Exercise 6

Part 1

1.

Load the packages required to read JPEG (JPEG = jpg). It converts the image to a 3D array. This is 3 2D arrays for each color channel.

```
library(jpeg)
my_image <- readJPEG("academiegebouw.jpg")
is(my_image)
```

```
## [1] "array"     "structure" "vector"
```

We can confirm that seeing the dimensions of the object.

```
dim(my_image)
```

```
## [1]  666 1000    3
```

2.

Here we can compute the mean for each row/column pair. So basically we are getting the mean of color for each pixel. Please see in `?apply()` the description of the `MARGIN` argument.

```
my_BW_image <- apply(my_image, c(1, 2), mean)
```

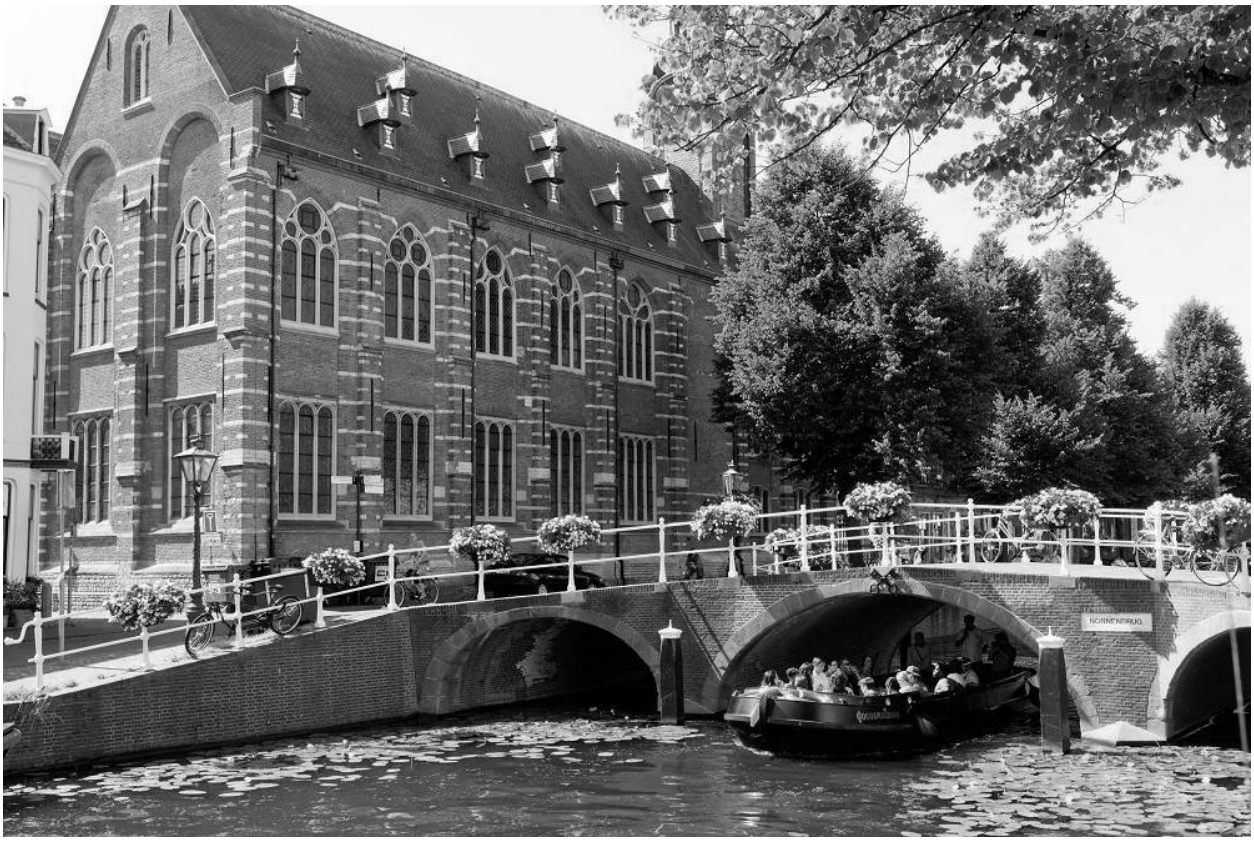
We have summarised the 3 2D arrays in one 2D array:

```
dim(my_BW_image)
```

```
## [1]  666 1000
```

Now, we can convert the image to jpg. Because it is a 2D array it converts it to a black and white image. See `?writeJPG()`.

```
writeJPEG(my_BW_image, "Exercise_5_image_BW.jpg")
```



3.

`expand.grid()` is going to create a `data.frame` with all the combinations row/column in the following way: One column with rows and one column with columns. For example, the new column with rows will have row 1 repeated for each column, so 1000 times. Then another 1000 for row 2 and so on. In this way we have all the pixels uniquely defined in two columns.

Then, one option is to get the color of each pixel and put each of the colors channels in a different vector. This can be easily done using the columns previously defined with `expand.grid()`.

Eventually we have the 5 columns we need, the only thing left to do is create the new `data.frame`.

```
my_conversor_fl <- function(image){
  # Compute all combinations
  row_column <- expand.grid(1:dim(image)[1], 1:dim(image)[2])

  #Notice row_column is a data.frame
  row <- row_column[, 1]
  column <- row_column[, 2]

  #Initialize
  r <- numeric(nrow(row_column))
  g <- numeric(nrow(row_column))
  b <- numeric(nrow(row_column))

  for(i in 1:nrow(row_column)){
    r[i] <- image[row[i], column[i], 1]
    g[i] <- image[row[i], column[i], 2]
```

```
    b[i] <- image[row[i], column[i], 3]
  }


  return(
    data.frame("row" = row, "column" = column,
               "red" = r, "green" = g, "blue" = b)
    )
}
```

A less readable but more efficient way of doing this is using `sapply()`. Please see `?sapply`. Notice that it simplifies the output to a vector.

The following works because when `sapply()` simplifies the array to a vector (1D), it does it in order. The order in which it simplifies the array is the same order in which the columns of `expand.grid` output are constructed. The first m (in this case 1000) values will correspond to the first row, then the next m values to the second row and so on. This is exactly what we want. This is called "flatten" an array, it means to collapse an array into one dimension.

To be more precise, here `sapply()` outputs a vector for each color channel, there are 3 color channels so the final output will be a matrix of 3 columns (one vector for each color channel).

```
my_conversor <- function(image){
  # Compute all combinations
  row_column <- expand.grid(1:dim(image)[1], 1:dim(image)[2])

  #Notice row_column is a data.frame
  row <- row_column[, 1]
  column <- row_column[, 2]

  # We use sapply because it simplifies the 2d
  # matrix to a vector. It therefore returns a
  # matrix nx3.
  RGB <- sapply(c(1, 2, 3), function(x){
    image[, , x]
  })

  r <- RGB[, 1]
  g <- RGB[, 2]
  b <- RGB[, 3]

  return(
    data.frame("row" = row, "column" = column,
               "red" = r, "green" = g, "blue" = b)
    )
}
my_df <- my_conversor(my_image)
head(my_df)
```

```
##   row column       red green      blue
## 1   1      1 0.9921569     1 0.9960784
## 2   2      1 0.9921569     1 0.9960784
## 3   3      1 0.9921569     1 0.9960784
## 4   4      1 0.9921569     1 0.9960784
## 5   5      1 0.9921569     1 0.9960784
```

```
## 6   6      1 0.9921569     1 0.9960784
```

And we can check that both solutions are identical:

```
all.equal(my_conversor(my_image), my_conversor_fl(my_image))
```

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

```
my_df <- my_conversor(my_image)
```

4

See `?unique`.

The following code basically checks how many rows are in a matrix where all duplicate rows are removed.

```
nrow(unique(my_df[, c("red","green","blue")]))
```

```
## [1] 240350
```

Part 2

5.

In the code below we apply the k-means algorithm to our 3 dimensional data (the three colors). We must not take into account the rows or the columns since we want to make clusters of the different colors disregarding where are they in the image.

```
set.seed(1) #this initialization does not rise any warning.
#If not converged just try again.
my_km <- kmeans(my_df[, c("red","green","blue")], 40)
```

```
## Warning: Quick-TRANSfer stage steps exceeded maximum (= 33300000)
```

```
my_km$centers
```

```
##           red     green        blue
## 1  0.33179537 0.4117095 0.144678595
## 2  0.02816655 0.0302308 0.009549286
## 3  0.14576858 0.2466115 0.031839052
## 4  0.24471069 0.1703304 0.071894330
## 5  0.13875172 0.1419582 0.075313835
## 6  0.56704299 0.5716047 0.227369589
## 7  0.95397933 0.9594900 0.711596857
## 8  0.39767414 0.3038741 0.208222396
## 9  0.36028363 0.3882194 0.025958216
## 10 0.79376681 0.8662162 0.204287798
## 11 0.25051292 0.3229651 0.030429561
## 12 0.47417239 0.4880998 0.201135266
## 13 0.78293129 0.8250853 0.440857914
## 14 0.92284643 0.8233348 0.734960197
## 15 0.65232078 0.1851214 0.291692381
## 16 0.63761821 0.7172964 0.153986320
## 17 0.61294424 0.6593315 0.668791877
## 18 0.50951158 0.3721624 0.277318066
## 19 0.92758360 0.9581307 0.988365795
## 20 0.74657726 0.6215513 0.530325707
## 21 0.89676704 0.9409275 0.414500245
## 22 0.85777152 0.7165977 0.622679549
```

```
## 23 0.80541610 0.8020376 0.759536762
## 24 0.65404214 0.5472617 0.428864885
## 25 0.05456676 0.1194924 0.027045056
## 26 0.53299768 0.4561428 0.384369624
## 27 0.23865859 0.3278493 0.385138255
## 28 0.74734103 0.7075426 0.640287794
## 29 0.67936448 0.7239091 0.350870051
## 30 0.48773073 0.5677734 0.054735873
## 31 0.54920641 0.5581346 0.525372062
## 32 0.40844882 0.3722559 0.320476394
## 33 0.30288209 0.2989902 0.257521545
## 34 0.14160631 0.2377329 0.287888711
## 35 0.21885922 0.2327204 0.187684257
## 36 0.45861106 0.4186034 0.103348489
## 37 0.30511927 0.2394080 0.144931192
## 38 0.86800223 0.8938916 0.569004318
## 39 0.94071246 0.9153378 0.855528928
## 40 0.33645486 0.4276789 0.493060425
```

Notice that we reduce the 235722 to just 40!

6.

The groups are given by row of our data frame in `my_km$cluster`,

```
length(my_km$cluster)
```

```
## [1] 666000
```

```
head(my_km$cluster)
```

```
## [1] 19 19 19 19 19 19
```

an the value correspond to a specific row in `my_km$centers`. We can just take the value of the center which correspond to the group which corresponds to the row in our data frame.

```
head(my_km$centers)
```

```
##          red     green        blue
## 1 0.33179537 0.4117095 0.144678595
## 2 0.02816655 0.0302308 0.009549286
## 3 0.14576858 0.2466115 0.031839052
## 4 0.24471069 0.1703304 0.071894330
## 5 0.13875172 0.1419582 0.075313835
## 6 0.56704299 0.5716047 0.227369589
```

So basically, this below creates the new data frame just by repeating the value of the centers for each row.

```
head(my_km$centers[my_km$cluster, ])
```

```
##          red     green      blue
## 19 0.9275836 0.9581307 0.9883658
## 19 0.9275836 0.9581307 0.9883658
## 19 0.9275836 0.9581307 0.9883658
## 19 0.9275836 0.9581307 0.9883658
## 19 0.9275836 0.9581307 0.9883658
## 19 0.9275836 0.9581307 0.9883658
```

Here we can see that the first 5 pixels correspond to group 11.

Therefore the new data frame:

```
my_df[, c("red","green","blue")] <- my_km$centers[my_km$cluster, ]
```

7.

A possible solution for this exploits the fact that the order of the rows in `expand.grid` is the same as the order in which `R` rearranges a vector to transform it to a matrix by default (`byrow = F`). This is the basically the reverse of what we have seen with the simplification of an array with `sapply`. In other words, we are reshaping a 1D vector into a 2D array and this is, by default, performed in the order that we want.

So we fill the matrix by columns, i.e, filling numbers from the vector since the number of columns specify is satisfied. Using this we can fill in the values in the array really fast.

```
my_c_img = array(dim = c(666, 1000, 3))
RGB = c("red","green","blue")

for(i in 1:3){
  #The "magic" here is happening inside the matrix() function
  my_c_img[, , i] <- matrix(my_df[, RGB[i]], c(666, 1000))
}

writeJPEG(my_c_img, "Exercise_5_image_40k.jpg")
```

We have sumarised a 16MB array into a 2.7MB `kmeans` object. Having those lines of code allows us to decompress the image and make it 16MB again.