STAT 243: Problem Set 5

Cristobal Pais

18 de octubre de 2017

Problem 2

a) Integers stored using double precision floating point representation

In order to represent integers using the double precision floating point format, we need to simply use the formula provided $(-1)^S \times 1.d \times 2^{e-1023}$ (for the representation) and detect the pattern behind the different numbers computations.

As a general introduction, we can remember that a binary representation of any number follows this format: Sign S (1 bit), exponent e (11 bit), and decimal ("mantissa") d (52 bits).

i) **Number 1**: As a simple starting point, we can illustrate the format using small numbers. Clearly, the simplest number that we can actually represent is the number one, that can be easily written as follows:

$$1 \quad = \quad (-1)^0 \times 1 \times 2^{1023-1023} \tag{1}$$

Here, we notice that the sign value $S = 0$ and the exponent of the last term $e = 1023$ in order to cancel the exponentiation. In addition, all the decimal numbers (known as "mantissa") are set to $d = 0$. Therefore, the number is represented as follows (in binary notation):

$$\underbrace{0}_{S} \,|\, \underbrace{01111111111}_{e} \,|\, \underbrace{00000...00000}_{d} \tag{2}$$

Where we can easily check that the exponent corresponds to 1023 and the representation of one by the following code:

```r
# Hide warnings
options(warn=-1)

# Loading libraries
library("pryr")

# Exponent (1023)
bits(1023)
```

```
## [1] "01000000 10001111 11111000 00000000 00000000 00000000 00000000 00000000"
```

```r
bits(1023L)
```

```
## [1] "00000000 00000000 00000011 11111111"
```

```r
# Number 1
bits(1)
```

```
## [1] "00111111 11110000 00000000 00000000 00000000 00000000 00000000 00000000"
```

ii) **Number 2**: Following the same logic, we can easily represent number two as follows:

$$2 = (-1)^0 \times 1 \times 2^{1024-1023} \tag{3}$$

Since 1024 is represented by 10000000000, it is clear that two will be stored as:

$$\underbrace{0}_{S} \,|\, \underbrace{10000000000}_{e} \,|\, \underbrace{00000...00000}_{d} \tag{4}$$

From this, we can easily predict that all exact powers of two will be stored with an ending of 52 zeros (d). This comes from the fact that we are using the number 2 as the base for our calculations and thus, we will always multiply the "hidden one" by a power of 2 in order to obtain those numbers, without modifying the values of d.

```r
# Exponent (1024)
bits(1024)
```

```
## [1] "01000000 10010000 00000000 00000000 00000000 00000000 00000000 00000000"
```

```r
bits(1024L)
```

```
## [1] "00000000 00000000 00000100 00000000"
```

```r
# Number 1
bits(2)
```

```
## [1] "01000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000"
```

iii) **Number 3**: The case of number three is the first one where we actually need to modify the structure of d since it is an odd number different from one. In this case, we can easily pick $d = 1.5$, $S = 0$ (as always with positive integers), and $e = 1024$ such that we have:

$$3 = (-1)^0 \times 1.5 \times 2^{1024-1023} \tag{5}$$

In order to visualize the elements that represent the integer numbers, we create the following functions $Binary2Decimal()$, $Mantissa2Decimal()$, and $FloatingFormat()$ that will help us to understand the pattern and format of more complex numbers:

```r
# Declaring the function: x string or binary number
Binary2Decimal <- function(x) {
  # Process the string and transform to decimal format
  sum(2^(which(rev(unlist(strsplit(as.character(x), "")) == 1)) -1))
}
```

```r
# Declaring the function: x string or binary number
Mantissa2Decimal <- function(x) {
  # Process the string and transform to decimal format
  sum(2^(- which(unlist(strsplit(as.character(x), "")) == 1)))
}

# Declaring the function: input x = number
FloatingFormat <- function(x){
  # Set working directory
  setwd("C:/Users/Lenovo/Desktop/UCBerkeley/3er Semestre/STATS 243/HW/HW5/")

  # Loading libraries and functions
  library("pryr")
  source("Binary2Decimal.R")
  source("Mantissa2Decimal.R")

  # Calling the bits() function
  y <- bits(x)
  y <- gsub(" ", "", y)

  # Extract Elements
  S <- gsub(" ", "", substr(y, 1, 1))
  e <- gsub(" ", "", substr(y, 2, 12))
  d <- gsub(" ", "", substr(y, 13, nchar(y)))

  # Prints out the different elements of the number representation
  print(paste("Sign (S):", S, "- Length:", nchar(S), "- Number:",
              Binary2Decimal(S), sep = " "))
  print(paste("Exponent (e):", e, "- Length:", nchar(e), "- Number:",
              Binary2Decimal(e), sep = " "))
  print(paste("Mantissa (d):", d, "- Length:", nchar(d), sep = " "))
  print(paste("              Number: ", Mantissa2Decimal(d), sep = " "))
  print(paste("Number representation: ", x, " = (-1)^",
              Binary2Decimal(S), " * (",
              1 + Mantissa2Decimal(d), ") * 2^(",
              Binary2Decimal(e), " - 1023)",
              sep = ""))
}
```

Using the previous function, we can easily check our analysis for the number three:

```r
# Number three elements
FloatingFormat(3)
```

```
## [1] "Sign (S): 0 - Length: 1 - Number: 0"
## [1] "Exponent (e): 10000000000 - Length: 11 - Number: 1024"
## [1] "Mantissa (d): 1000000000000000000000000000000000000000000000000000 - Length: 52"
## [1] "              Number:  0.5"
## [1] "Number representation: 3 = (-1)^0 * (1.5) * 2^(1024 - 1023)"
```

iv) **General Integer Number**: Based on the previous simple examples analysis, we can easily see that every integer number n from 1, 2, ..., $2^{53} - 2$, $2^{53} - 1$ can be represented by the floating point format.

As we already discussed, power of two can be represented by just changing the value of the exponent e from 1024 to 1076 (since $e = 1076 - 1023 = 53$ and 1076 can be represented with eleven bits), and keeping $d = 0$, $S = 0$.

For odd numbers such as 3, 5, 7, etc., we can just compute the power of 2 value that is the closest one **from below** to the number that we want to construct and simply modify the mantissa d value such that its multiplication with the power of two found is equal to the number we are looking for. For helping us to visualize the computations, we include the following auxiliary functions $ClosesPower2Below()$ and $ComputeD()$:

```
# Declaring the function
ClosestPower2below <- function(x){
  # Variable to return initialization
  exponent <- 0

  # Loop for finding the value
  while (TRUE){
    if (2 ^ (exponent + 1) > x){
      break
    }
    else{
      exponent <- exponent + 1
    }
  }

  # Return exponent value
  return(exponent)
}

# Declaring the function
ComputeD <- function(x, power2){
  # Return the value of d (with 1 at the beginning)
  return(x / (2 ^ power2))
}
```

Here we present a series of examples:

1. **Number** 15: the closest power of two from below is $2^3 = 8$ and thus, the value of d can be easily obtained by solving:

$$1.d = \frac{15}{2^3} = 1.875 \Rightarrow d = 857 \tag{6}$$

We can easily check this using our $FloatingFormat()$ function:

```
# Calling the function
FloatingFormat(15)
```

```
## [1] "Sign (S): 0 - Length: 1 - Number: 0"
## [1] "Exponent (e): 10000000010 - Length: 11 - Number: 1026"
## [1] "Mantissa (d): 1110000000000000000000000000000000000000000000000000 - Length: 52"
## [1] "              Number:  0.875"
## [1] "Number representation: 15 = (-1)^0 * (1.875) * 2^(1026 - 1023)"
```

2. **Number** 337: we compute the closest power of two using our previously declared function for finding the exponent $e_2 = 8$. Thus, $e = 1023 + e_2 = 1031$. Now, we can easily compute the value of $d = 1.3164062$ by calling the $ComputeD()$ function.

   We check this using our $FloatingFormat()$ function:

```
# Calling the function
FloatingFormat(337)
```

```
## [1] "Sign (S): 0 - Length: 1 - Number: 0"
## [1] "Exponent (e): 10000000111 - Length: 11 - Number: 1031"
## [1] "Mantissa (d): 0101000100000000000000000000000000000000000000000000 - Length: 52"
## [1] "                Number:  0.31640625"
## [1] "Number representation: 337 = (-1)^0 * (1.31640625) * 2^(1031 - 1023)"
```

3. Number 103335: Using the same procedure as before, we can easily obtain $e = 1039$ and $d = 1.576767$.

   We check this using our $FloatingFormat()$ function:

```
# Calling the function
FloatingFormat(103335)
```

```
## [1] "Sign (S): 0 - Length: 1 - Number: 0"
## [1] "Exponent (e): 10000001111 - Length: 11 - Number: 1039"
## [1] "Mantissa (d): 1001001110100111000000000000000000000000000000000000 - Length: 52"
## [1] "                Number:  0.576766967773438"
## [1] "Number representation: 103335 = (-1)^0 * (1.57676696777344) * 2^(1039 - 1023)"
```

Therefore, the general case for odd numbers is covered.

In the case of even numbers (and no power of two numbers) such as - for example - 18, we proceed exactly as with the odd numbers since we again need to find the closest power of two value from below (obtaining e) and then we can easily compute the value of d. Again, $S = 0$ for all cases.

Here we present a series of examples:

1. **Number** 18: the closest power of two from below is $2^4 = 16$ and thus, the value of d can be easily obtained by solving:

$$1.d = \frac{18}{2^4} = 1.125 \Rightarrow d = 125 \tag{7}$$

   We can easily check this (check e) using our $FloatingFormat()$ function:

```
# Calling the function
FloatingFormat(18)
```

```
## [1] "Sign (S): 0 - Length: 1 - Number: 0"
## [1] "Exponent (e): 10000000011 - Length: 11 - Number: 1027"
## [1] "Mantissa (d): 0010000000000000000000000000000000000000000000000000 - Length: 52"
## [1] "                Number:  0.125"
## [1] "Number representation: 18 = (-1)^0 * (1.125) * 2^(1027 - 1023)"
```

2. **Number** 338: we compute the closest power of two using our previously declared function for finding the exponent $e_2 = 8$. Thus, $e = 1023 + e_2 = 1031$. Now, we can easily compute the value of $d = 1.3203125$ by calling the $ComputeD()$ function.

   We check this using our $FloatingFormat()$ function:

```
# Calling the function
FloatingFormat(338)
```

```
## [1] "Sign (S): 0 - Length: 1 - Number: 0"
## [1] "Exponent (e): 10000000111 - Length: 11 - Number: 1031"
## [1] "Mantissa (d): 0101001000000000000000000000000000000000000000000000 - Length: 52"
## [1] "              Number:  0.3203125"
## [1] "Number representation: 338 = (-1)^0 * (1.3203125) * 2^(1031 - 1023)"
```

3. **Number** 450002: Using the same procedure as before, we can easily obtain $e = 1041$ and $d = 1.7166214$.

   We check this using our $FloatingFormat()$ function:

```
# Calling the function
FloatingFormat(450002)
```

```
## [1] "Sign (S): 0 - Length: 1 - Number: 0"
## [1] "Exponent (e): 10000010001 - Length: 11 - Number: 1041"
## [1] "Mantissa (d): 1011011101110100100000000000000000000000000000000000 - Length: 52"
## [1] "              Number:  0.716621398925781"
## [1] "Number representation: 450002 = (-1)^0 * (1.71662139892578) * 2^(1041 - 1023)"
```

In order to show that show that 2^{53} and $2^{53}+2$ can be represented exactly but $2^{53}+1$ cannot, so the spacing of numbers of this magnitude is 2, we can proceed as follows:

a) Based on the structure of the floating point representation, we have that between $2^{52} = 4,503,599,627,370,496$ and $2^{53} = 9,007,199,254,740,992$ we can represent all the integer numbers.

b) Then, from 2^{53} to 2^{54}, we can easily see that **everything** is multiplied by a factor of 2. Therefore, we can only represent **even numbers** and thus, $2^{53}+1$ cannot be represented in this interval.

c) We can obtain a general formula for the spacing by noting that it can be calculated as a fraction of the numbers in the range of interest from 2^k to 2^{k+1} will be 2^{52+1}. Therefore, for 2^{54} we have that the spacing is $2^{54-52} = 2^2 = 4$, matching the expected value as stated in the problem description.

   The explanation behind this is based on the fact that all the numbers following the floating point format are in the interval $[2^e, 2^{e+1})$. where e is the exponent. Therefore, we can easily notice that the length of the interval is $2^{e+1} - 2^e = 2^e$. Now, every interval will have exactly the same number of "spacing" which can be fully determined by knowing the precision of our numbers, say a. Then, it is clear that the size of each space inside each interval will be the length of the interval, divided by the number of spaces: $\frac{2^e}{2^{a-1}} = 2^{e-(a-1)} = 2^{e+1-a}$.

   Clearly, each "space" will be a power of two (since we are using 2 as our base). Thus, every time we are increasing the value of the exponent, we are doubling the size of the "spacing". On the other hand, if we increment the value of a, the new space will be half of the original one. In our case, since we are not modifying the precision a in our computations, we are doubling the value of the "spacing" with every new magnitude.

   We can check our analysis by executing the following code:

```r
# Calling the function
FloatingFormat(2^53 - 1)
```

```
## [1] "Sign (S): 0 - Length: 1 - Number: 0"
## [1] "Exponent (e): 10000110011 - Length: 11 - Number: 1075"
## [1] "Mantissa (d): 1111111111111111111111111111111111111111111111111111 - Length: 52"
## [1] "               Number:  1"
## [1] "Number representation: 9007199254740991 = (-1)^0 * (2) * 2^(1075 - 1023)"
```

```r
ComputeD(2^53 - 1, ClosestPower2below(2^53 - 1))
```

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

```r
FloatingFormat(2^53)
```

```
## [1] "Sign (S): 0 - Length: 1 - Number: 0"
## [1] "Exponent (e): 10000110100 - Length: 11 - Number: 1076"
## [1] "Mantissa (d): 0000000000000000000000000000000000000000000000000000 - Length: 52"
## [1] "               Number:  0"
## [1] "Number representation: 9007199254740992 = (-1)^0 * (1) * 2^(1076 - 1023)"
```

```r
ComputeD(2^53, ClosestPower2below(2^53))
```

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

```r
FloatingFormat(2^53 + 1)
```

```
## [1] "Sign (S): 0 - Length: 1 - Number: 0"
## [1] "Exponent (e): 10000110100 - Length: 11 - Number: 1076"
## [1] "Mantissa (d): 0000000000000000000000000000000000000000000000000000 - Length: 52"
## [1] "               Number:  0"
## [1] "Number representation: 9007199254740992 = (-1)^0 * (1) * 2^(1076 - 1023)"
```

```r
ComputeD(2^53 + 1, ClosestPower2below(2^53 + 1))
```

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

```r
FloatingFormat(2^53 + 2)
```

```
## [1] "Sign (S): 0 - Length: 1 - Number: 0"
## [1] "Exponent (e): 10000110100 - Length: 11 - Number: 1076"
## [1] "Mantissa (d): 0000000000000000000000000000000000000000000000000001 - Length: 52"
## [1] "               Number:  2.22044604925031e-16"
## [1] "Number representation: 9007199254740994 = (-1)^0 * (1) * 2^(1076 - 1023)"
```

```r
ComputeD(2^53 + 2, ClosestPower2below(2^53 + 2))
```

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

Based on the outputs, we can see that for $2^{53}-1$ the reported value for $d \approx 1$ and thus we obtain 2 as the full value inside the second parenthesis. If we perform the division while displaying more digits, we obtain 1.9999999999999998. Then, 2^{53} is clearly a power of two so we can simply represent it with a null mantissa and $e = 1076$. However, for $2^{53}+1$ we can notice how our function is not able to handle it, obtaining exactly the same result as before. This is due to the fact that the mantissa value cannot be represented due to its magnitude (in fact, we obtain a zero value for d) and thus the number cannot be represented using this floating format. On the other hand, we can easily see that $2^{53}+2$ can be represented with $d = 2.22044604925031e-16$ (not displayed in the number representation formula for visualization purposes), checking our previous analysis.

For completeness, we can test $2^{53}+3$ and $2^{53}+4$:

```
# Calling the function
FloatingFormat(2^53 + 3)
```

```
## [1] "Sign (S): 0 - Length: 1 - Number: 0"
## [1] "Exponent (e): 10000110100 - Length: 11 - Number: 1076"
## [1] "Mantissa (d): 0000000000000000000000000000000000000000000000000010 - Length: 52"
## [1] "                Number:  4.44089209850063e-16"
## [1] "Number representation: 9007199254740996 = (-1)^0 * (1) * 2^(1076 - 1023)"
```

```
ComputeD(2^53 + 3, ClosestPower2below(2^53 + 3))
```

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

```
FloatingFormat(2^53 + 4)
```

```
## [1] "Sign (S): 0 - Length: 1 - Number: 0"
## [1] "Exponent (e): 10000110100 - Length: 11 - Number: 1076"
## [1] "Mantissa (d): 0000000000000000000000000000000000000000000000000010 - Length: 52"
## [1] "                Number:  4.44089209850063e-16"
## [1] "Number representation: 9007199254740996 = (-1)^0 * (1) * 2^(1076 - 1023)"
```

```
ComputeD(2^53 + 4, ClosestPower2below(2^53 + 4))
```

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

Again, we can see how the odd number cannot be represented by our function, obtaining exactly the same value as the even one indicating us that the spacing, in this case, is of value 2.

Finally, the same analysis can be made for 2^{54} magnitude:

```
# Calling the function
FloatingFormat(2^54)
```

```
## [1] "Sign (S): 0 - Length: 1 - Number: 0"
## [1] "Exponent (e): 10000110101 - Length: 11 - Number: 1077"
## [1] "Mantissa (d): 0000000000000000000000000000000000000000000000000000 - Length: 52"
## [1] "                Number:  0"
## [1] "Number representation: 18014398509481984 = (-1)^0 * (1) * 2^(1077 - 1023)"
```

```r
ComputeD(2^54, ClosestPower2below(2^54))
```

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

```r
FloatingFormat(2^54 + 1)
```

```
## [1] "Sign (S): 0 - Length: 1 - Number: 0"
## [1] "Exponent (e): 10000110101 - Length: 11 - Number: 1077"
## [1] "Mantissa (d): 0000000000000000000000000000000000000000000000000000 - Length: 52"
## [1] "              Number:  0"
## [1] "Number representation: 18014398509481984 = (-1)^0 * (1) * 2^(1077 - 1023)"
```

```r
ComputeD(2^54 + 1, ClosestPower2below(2^54 + 1))
```

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

```r
FloatingFormat(2^54 + 2)
```

```
## [1] "Sign (S): 0 - Length: 1 - Number: 0"
## [1] "Exponent (e): 10000110101 - Length: 11 - Number: 1077"
## [1] "Mantissa (d): 0000000000000000000000000000000000000000000000000000 - Length: 52"
## [1] "              Number:  0"
## [1] "Number representation: 18014398509481984 = (-1)^0 * (1) * 2^(1077 - 1023)"
```

```r
ComputeD(2^54 + 2, ClosestPower2below(2^54 + 2))
```

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

```r
FloatingFormat(2^54 + 3)
```

```
## [1] "Sign (S): 0 - Length: 1 - Number: 0"
## [1] "Exponent (e): 10000110101 - Length: 11 - Number: 1077"
## [1] "Mantissa (d): 0000000000000000000000000000000000000000000000000001 - Length: 52"
## [1] "              Number:  2.22044604925031e-16"
## [1] "Number representation: 18014398509481988 = (-1)^0 * (1) * 2^(1077 - 1023)"
```

```r
ComputeD(2^54 + 3, ClosestPower2below(2^54 + 3))
```

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

```r
FloatingFormat(2^54 + 4)
```

```
## [1] "Sign (S): 0 - Length: 1 - Number: 0"
## [1] "Exponent (e): 10000110101 - Length: 11 - Number: 1077"
## [1] "Mantissa (d): 0000000000000000000000000000000000000000000000000001 - Length: 52"
## [1] "              Number:  2.22044604925031e-16"
## [1] "Number representation: 18014398509481988 = (-1)^0 * (1) * 2^(1077 - 1023)"
```

```
ComputeD(2^54 + 4, ClosestPower2below(2^54 + 4))
```

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

Notice that - as expected - for the case $2^{54} + 4$, we have $2.22044604925031e - 16 * 2^{54} = 3.99999999999999$.

Problem 3

In order to perform the comparisons, we will use the microbenchmark library to obtain the execution times and the pryr library for helping us to trace the size/memory usage of each element. In addition, we define an auxiliary function for generating explicit copies (in memory) of current values.

a) Copy of large vector: integers vs numeric

```
# Hide warnings and max number of digits
options(warn=-1)
options(digits=3)

# Loading libraries
library("microbenchmark")
library("ggplot2")
library("pryr")

# Auxiliary function: CopyExplicit()
CopyExplicit <- function(x){
  # Reference
  y <- x

  # Modification: for new copy
  y[1] <- x[1]

  # Return the copy
  return(y)
}

# Test data (numeric)
x <- as.numeric(1:1e8)
y <- as.numeric(1:1e7)
z <- as.numeric(1:1e6)
w <- as.numeric(1:1e5)
t <- as.numeric(1:1e4)

typeof(x)
```

```
## [1] "double"
```

```
# Test data (integer)
xL <- as.integer(x)
yL <- as.integer(y)
zL <- as.integer(z)
wL <- as.integer(w)
tL <- as.integer(t)

typeof(xL)
```

```
## [1] "integer"
```

```r
# Memory usage comparison
object_size(x)
```

```
## 800 MB
```

```r
object_size(xL)
```

```
## 400 MB
```

```r
object_size(y)
```

```
## 80 MB
```

```r
object_size(yL)
```

```
## 40 MB
```

```r
object_size(z)
```

```
## 8 MB
```

```r
object_size(zL)
```

```
## 4 MB
```

```r
object_size(w)
```

```
## 800 kB
```

```r
object_size(wL)
```

```
## 400 kB
```

```r
object_size(t)
```

```
## 80 kB
```

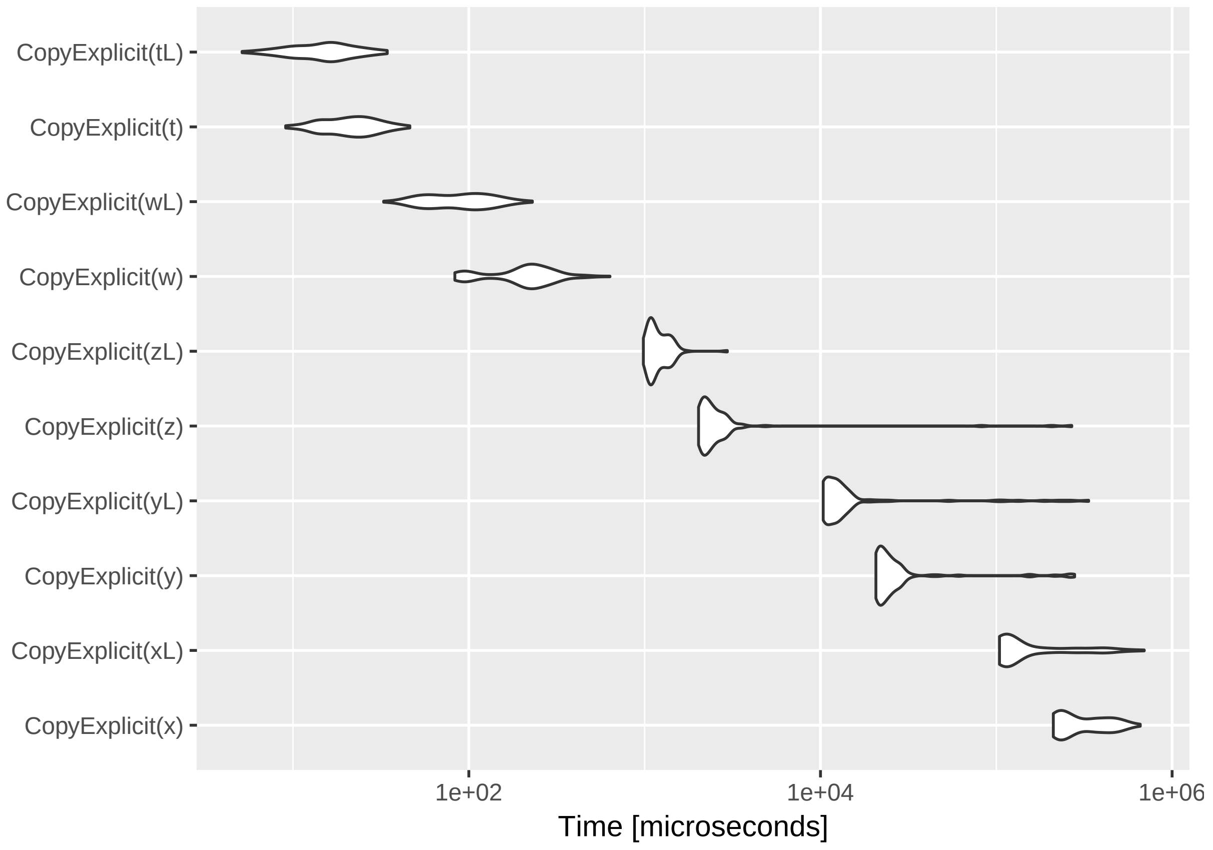
```r
object_size(tL)
```

```
## 40 kB
```

```r
# Performance comparison
CompA <- microbenchmark(CopyExplicit(x), CopyExplicit(xL), CopyExplicit(y),
                        CopyExplicit(yL), CopyExplicit(z), CopyExplicit(zL),
                        CopyExplicit(w), CopyExplicit(wL), CopyExplicit(t),
                        CopyExplicit(tL), times = 100)
print(CompA)
```

```
## Unit: microseconds
##              expr      min       lq     mean   median       uq      max
##   CopyExplicit(x) 2.11e+05 231781.5 324491.4 273343.7 414097.0 659050.3
##  CopyExplicit(xL) 1.04e+05 109627.7 179930.7 122988.7 189173.9 694430.1
##   CopyExplicit(y) 2.07e+04  21717.8  36553.3  23467.3  27044.4 279028.0
##  CopyExplicit(yL) 1.04e+04  10752.0  25594.2  12238.5  13944.0 335307.4
##   CopyExplicit(z) 2.03e+03   2145.7   7957.9   2360.3   2715.3 268713.4
##  CopyExplicit(zL) 9.83e+02   1072.4   1216.4   1145.6   1324.2   2953.6
##   CopyExplicit(w) 8.33e+01    158.9    219.3    220.3    263.5    635.2
##  CopyExplicit(wL) 3.28e+01     60.8     95.1     91.8    123.2    229.8
##   CopyExplicit(t) 9.08e+00     15.4     22.2     21.3     26.6     46.2
##  CopyExplicit(tL) 5.13e+00     10.7     15.9     15.6     18.9     34.3
##  neval
##    100
##    100
##    100
##    100
##    100
##    100
##    100
##    100
##    100
##    100
```

```r
autoplot(CompA)
```



Looking at the results, we can clearly see that the time needed for copying integer vectors tends to be less than a half of the time that is needed by R for copying numerical vectors. This follows the intuition behind the fact stated in the problem: we are moving around half as much data during the process of computation, obtaining improved running times by sacrificing precision of the recorded number.

b) Subset of $k \approx \frac{n}{2}$: integers vs numeric

We include a series of experiments in the following code for determining if there exists a significant difference between the performance of accessing k values from a vector of integers or numeric elements.

```r
# Hide warnings and max number of digits
options(warn=-1)
options(digits=3)

# Loading libraries
library("microbenchmark")
library("ggplot2")

# Test data (numeric)
x <- as.numeric(1:1e8)
y <- as.numeric(1:1e7)
z <- as.numeric(1:1e6)
w <- as.numeric(1:1e5)
t <- as.numeric(1:1e4)

typeof(x)
```

```
## [1] "double"
```

```r
# Test data (integer)
xL <- as.integer(x)
yL <- as.integer(y)
zL <- as.integer(z)
wL <- as.integer(w)
tL <- as.integer(t)

typeof(xL)
```

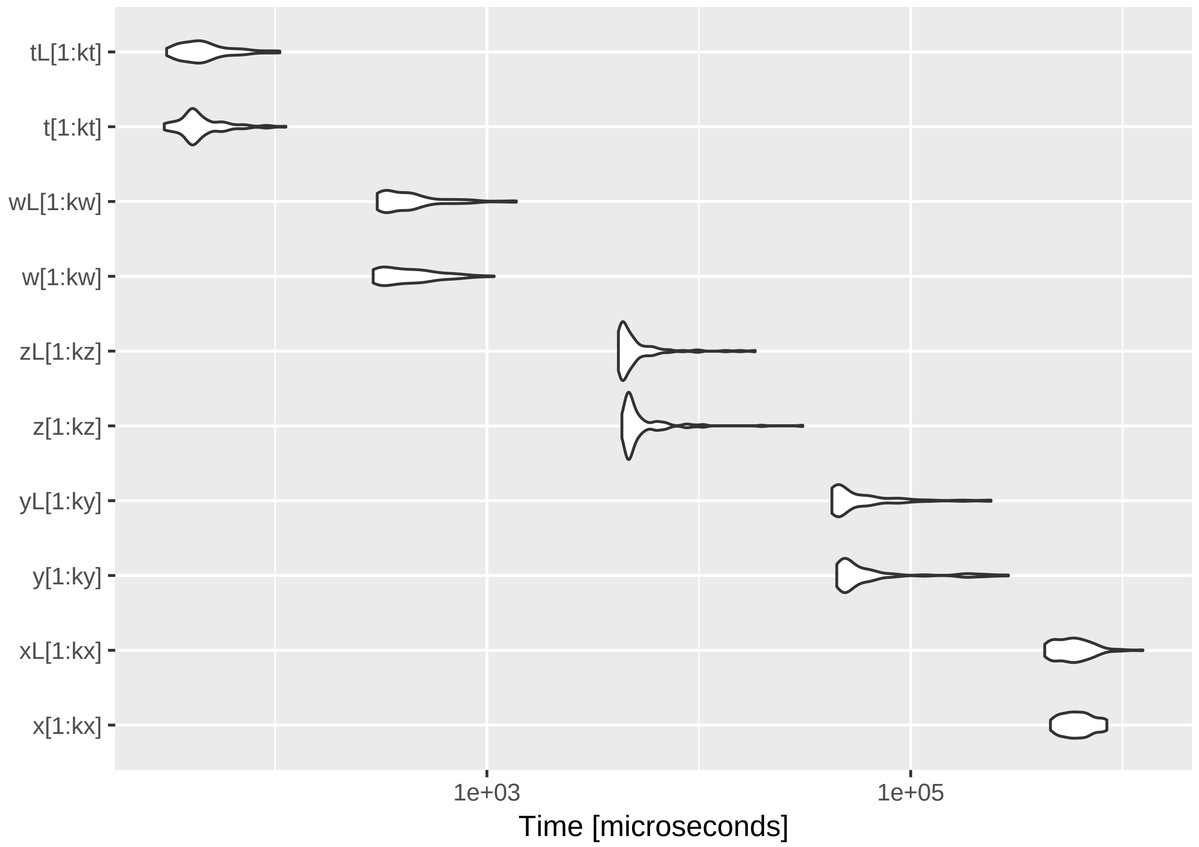
```
## [1] "integer"
```

```r
# k-values: n/2
kx <- length(x) / 2
ky <- length(y) / 2
kz <- length(z) / 2
kw <- length(w) / 2
kt <- length(t) / 2

# Performance comparison
CompB <- microbenchmark(x[1:kx], xL[1:kx], y[1:ky], yL[1:ky], z[1:kz], zL[1:kz],
                        w[1:kw], wL[1:kw], t[1:kt], tL[1:kt], times = 100)
print(CompB)
```

```
## Unit: microseconds
##      expr      min       lq     mean   median       uq     max neval
##   x[1:kx] 456507.4 537084.7 621897.6 601898.7 691499.2  842955   100
##  xL[1:kx] 428981.0 488677.2 595986.2 573403.0 661534.8 1247597   100
##    y[1:ky]  44779.7  47843.4  72670.1  52213.0  64875.6  289399   100
##   yL[1:ky]  42496.8  45055.2  62261.9  47501.6  63186.1  239468   100
##    z[1:kz]   4336.0   4637.8   5714.8   4781.7   5438.0   30959   100
##   zL[1:kz]   4173.0   4352.4   5271.7   4571.7   5129.5   18457   100
##    w[1:kw]    290.5    324.5    446.7    401.3    509.0    1082   100
##   wL[1:kw]    303.6    338.9    460.9    404.8    477.3    1377   100
##    t[1:kt]     30.0     39.1     46.3     41.5     48.8     113   100
##   tL[1:kt]     30.8     37.1     48.4     44.6     52.3     105   100
```

```r
autoplot(CompB)
```



Looking at the previous results, we can notice that running times for integer vectors with sizes greater or equal than $1e6$ are better (shorter) than the ones obtained for numerical vectors, but not as much as in the previous section. On the other hand, vectors with sizes $1e5$ and $1e4$ present a very similar performance, with no significant differences. This is expected since decreasing the size of the vectors will tend to balance the trade-off between the amount of information (integer vs numeric) and operations performance.

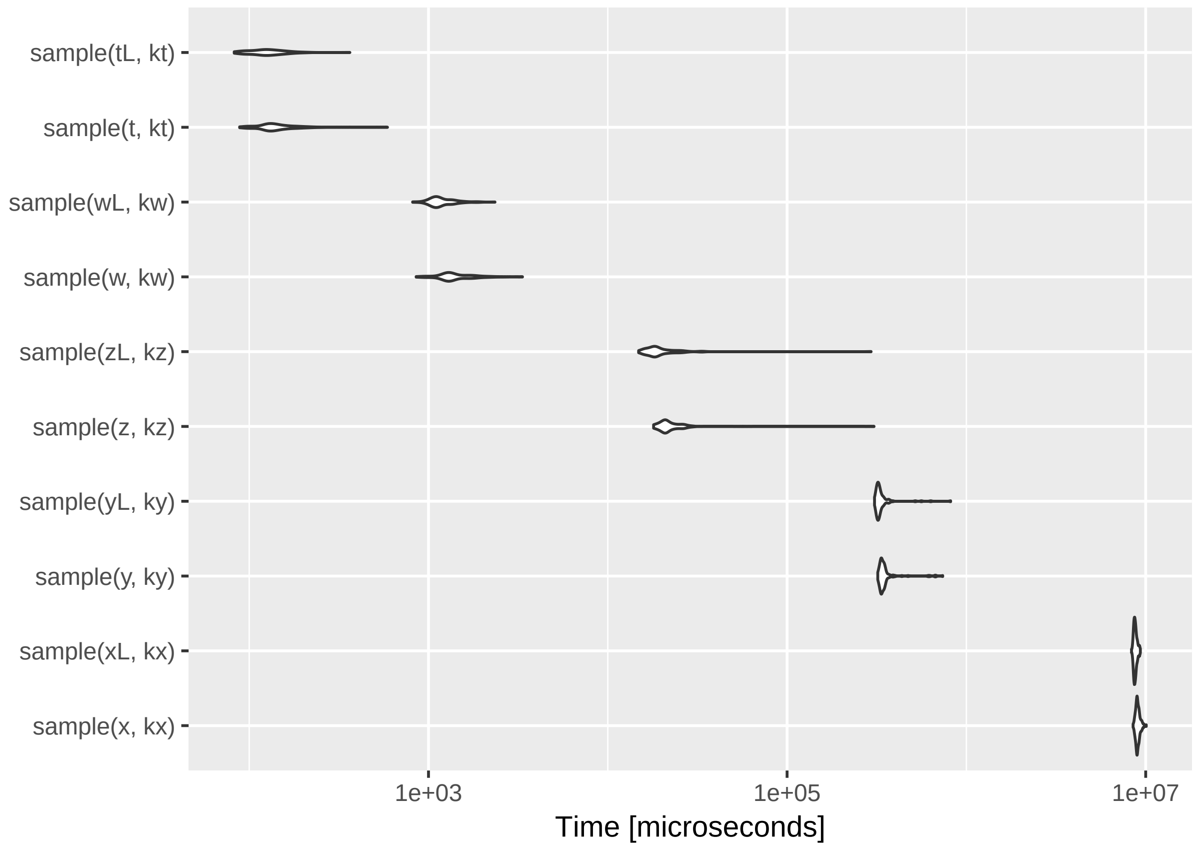
For completeness, we perform similar tests using the sample function from R, in order to take $k \approx \frac{n}{2}$ values without replacement from the vectors:

```r
# Performance comparison
CompB2 <- microbenchmark(sample(x, kx), sample(xL, kx), sample(y, ky), sample(yL, ky),
                         sample(z, kz), sample(zL, kz), sample(w, kw), sample(wL, kw),
                         sample(t, kt), sample(tL, kt), times = 100)

print(CompB2)
```

```
## Unit: microseconds
##            expr                    min                    lq
##   sample(x, kx) 8.5048872960000001e+06 8880432.11050000041723
##  sample(xL, kx) 8.3348589400000004e+06 8606929.13899999856949
##   sample(y, ky) 3.2046173999999999e+05  333146.67949999996927
##  sample(yL, ky) 3.0739842700000003e+05  318559.99650000000838
##   sample(z, kz) 1.8008046999999999e+04   20261.71749999999884
##  sample(zL, kz) 1.4856303000000000e+04   17390.44849999999860
##   sample(w, kw) 8.5307100000000003e+02    1248.02599999999984
##  sample(wL, kw) 8.1556899999999996e+02    1068.41249999999991
##   sample(t, kt) 8.8426000000000002e+01     124.15250000000000
##  sample(tL, kt) 8.2504999999999995e+01     105.20400000000001
##                    mean                 median                   uq
##  9034963.34167999960482 8987494.67650000005960 9155232.0570000000298
##  8753352.71536000072956 8712953.69000000134110 8839223.1814999990165
##   360586.06955999997444  340329.68599999998696  350784.8379999999888
##   340761.09314999997150  325173.36900000000605  336564.2924999999814
##    25197.68941999999879   21122.09100000000035   23685.0540000000001
##    22339.50319999999920   18362.53949999999895   20997.7425000000003
##     1476.95430000000010    1337.43900000000008    1646.7315000000001
##     1205.24666999999999    1137.49450000000002    1319.2805000000001
##      148.86780999999999     135.99549999999999     159.8775000000000
##      129.38648000000001     124.15200000000000     143.8905000000000
##                     max neval
##  1.0076343330000000e+07   100
##  9.3403168570000008e+06   100
##  7.3810289899999998e+05   100
##  8.1701057400000002e+05   100
##  3.0549016999999998e+05   100
##  2.9422023800000001e+05   100
##  3.3447809999999999e+03   100
##  2.3495980000000000e+03   100
##  5.9055799999999999e+02   100
##  3.6436200000000002e+02   100
```

```
autoplot(CompB2)
```



In this case, we can see that the performance difference between numerical and integer vectors is not significant, with only a slight positive difference (shorter processing time) when dealing with integer vectors. This can happen due to the fact that the *sample*() function is very efficient (C code) and the object type is not as relevant as for another kind of operations.

Problem 4: Parallelization

a) Matrix multiplication analysis

Based on the Unit 7 discussion, it might be better to break up the Y matrix into p blocks (p the number of workers available) of $m = \frac{n}{p}$ columns instead of n individual column-wise computations because:

1. If we have the case where n is less than p, then we will end up with some idle cores when performing the computations, "wasting" our resources (poor utilization). Thus, if we are dividing a number of jobs into p blocks of $m = \frac{n}{p}$, we will be able to use all the available cores/processors, obtaining a better performance in terms of running time, exploiting all the resources available. Therefore, we can exploit the tasks' granularity using more than one core/processor per task (hybrid parallelization).
2. When n is larger than p, we will create a series of n jobs that must be processed by $p << n$ and thus, the code will generate a queue of jobs in memory and maybe the allocation of these n jobs will not be balanced among the available number of cores/processors, generating a series of potential bottlenecks during the execution of the code. Also, the balancing/allocation/scheduling logic must be taken into account in order to perform a smart distribution of the tasks, adding an extra step (and difficulty) to our code, potentially leading to poor performance. Clearly, we will want to avoid situations where we have a larger number of jobs than cores/processors.
3. In addition to the previous point, we can reach a poor performance when dividing the number of tasks as n individual column-wise operations when $n >> p$ in terms of lag/latency issues: a larger amount of tasks/jobs will lead to a larger connection overhead, impacting the overall performance. Furthermore, we could end up experiencing some memory issues due to the amount of RAM that is needed during the solving process due to the potential multiple copies of objects that must be passed to each job.

b) Different parallelization approaches: Comparison

In order to perform the comparison between both approaches, we analyze each implementation in terms of memory usage and number of numerical elements passed between workers/master processes.

a) **Approach A: Full X, subset Y**

In this approach, we notice that the X matrix is not modified at any point and it is passed to all the workers AS-IS ($n \times n$ dimension). On the other hand, the Y matrix is divided into p blocks such that each subset contains $m = \frac{n}{p}$ columns. Therefore, the algorithm will clearly perform p operations: exactly one per worker.

i) **Amount of memory**

Every worker will have the full X matrix in memory ($n \times n$ elements) and a subset of the original Y matrix with dimensions $n \times m$, where $m = \frac{n}{p}$. Therefore, the total memory usage per task will be related to total amount of numbers that have to be handled, including the resulting $n \times m$ multiplication between the previous elements. Therefore, the total amount of memory used will be proportional to $n^2 + 2nm$. Clearly, the amount of memory used **per worker** in this case will be higher than in the approach B because, in A, a full copy of X is passed every time a task is performed while in B it is only a subset, as we will see in the next section.

ii) **Communication costs**:

Every worker will receive the original X matrix containing $n \times n = n^2$ numbers and a subset of the original Y matrix containing n rows and m columns, in other words, nm numbers. Thus, each worker will process a total of $n^2 + nm$ numbers.

Using these elements, each process will generate a result of dimension $n \times m$ since we are multiplying a $n \times n$ matrix by a $n \times m$ matrix. These nm numbers are then passed back to the master process when returning the result.

These operations are performed in each one of the p jobs/tasks. Thus, there is a total of $p(n^2 + 2nm)$ numbers that are passed/communicated between the master and slave processes.

b) **Approach B: subsets of X and Y**

In this approach, we notice that a subset of the original X matrix is passed to all the tasks performed, obtaining a submatrix of dimension $m \times n$. Similarly to the previous approach, the Y matrix is divided into p blocks such that each subset contains $m = \frac{n}{p}$ columns. Therefore, the algorithm will clearly perform p^2 operations: each of the p subsets of the X matrix will be multiplied by each of the p subsets of the Y matrix, leading to a total of p^2 tasks, p per worker if we want to divide it evenly.

i) **Amount of memory**

Each task will include a subset of the X matrix and a subset of the Y matrix. Both of them will contain nm elements in total and the multiplication of these two matrices will result into a $m \times m$ matrix. Therefore, the total amount of memory used will be proportional to $m^2 + 2nm$. Clearly, the amount of memory used **per task** in this case will be lower than in the approach A because, in B, a subset of X is passed every time a task is performed while in A, it is the full $n \times n$ matrix.

ii) **Communication costs**

Every worker will receive a subset of the original X matrix containing $m \times n = nm$ numbers and a subset of the original Y matrix containing n rows and m columns, in other words, nm numbers. Thus, each worker will process a total of $2nm$ numbers.

Using these elements, each process will generate a result of dimension $m \times m$ since we are multiplying a $m \times n$ matrix by a $n \times m$ matrix. These m^2 numbers are then passed back to the master process when returning the result.

These operations are performed in each one of the p^2 jobs/tasks. Thus, there is a total of $p(m^2 + 2nm)$ numbers that are passed/communicated between the master and slave processes.

Based on the previous analysis, we can construct the following summary tables:

Table 1: Summary table: Numbers to and from workers

Approach	**# of tasks**	**# per worker**	**# to worker**	**# from workers**
A	p	$n^2 + 2nm$	$n^2 + nm$	nm
B	p^2	$m^2 + 2nm$	$2nm$	m^2
Comparison (better)	**A**	**B**	**B**	**B**

Table 2: Summary table: Total numbers and memory usage

Approach	**Total numbers**	**Total memory**
A	$p(n^2 + 2nm)$	$p(n^2 + 2nm)$
B	$p^2(m^2 + 2nm)$	$p^2(m^2 + 2nm)$
Comparison (better)	**A**	**A**

Based on the summary tables, we can easily see that the second approach tends to outperform the first one when an individual analysis is performed (per task analysis). Clearly, the total amount of memory usage per task will be lower than in the first approach because we are just sending a subset of the X matrix to each task instead of the full $n \times n$ matrix while keeping the size of Y equal for both approaches. On the other hand, the number of communication operations needed for the first approach are less than in the case of the B approach because we only need to perform p tasks instead of p^2 due to the fact that we are just iterating along the p columns of the Y submatrices while in the second approach we need to iterate over the p subsets of X and p subsets of Y. In other words: we have one for loop in the approach A $(1 : p)$ while we have two nested loops in the approach B $(1 : p$ and $1 : p)$.

However, when we want to compare the total amount of memory in use **at the same time** and in use **by the entire algorithm** by all the workers inside the computer, we need to take into account the number of iterations.

1. Simultaneous amount of memory: In order to analyze the simultaneous memory usage by each approach, we need to multiply the total number of cores/workers by the individual memory needed per task. Based on the previous analysis, it is clear that the second approach will require less amount of memory usage per task since we are dealing with a subset of X instead of the full $n \times n$ matrix. Therefore, the memory usage **at the same time** is less than in the first approach.
2. Total algorithm memory: In this case, we need to multiply the total number of tasks by the individual memory usage. Looking at the table and our previous analysis, we know that for the first approach we have p tasks where each one requires to store $n^2 + 2nm$ elements while for the second approach we have p^2 tasks, each one using $m^2 + 2nm$ elements. Since $m = \frac{n}{p} \Rightarrow p\frac{m}{n}$ we can easily compare both expressions:

$$\begin{aligned}
p(n^2 + 2nm) &< p^2(m^2 + 2nm) && (8)\\
(n^2 + 2nm) &< p(m^2 + 2nm) && (9)\\
n^2 + 2nm &< \frac{n}{m}(m^2 + 2nm) && (10)\\
n^2 + 2nm &< nm + 2n^2 && (11)\\
m &< n && (12)
\end{aligned}$$

   Since $m = \frac{n}{p}$ and $p \geq 1$, we have that the previous expression is always true and thus, we can conclude that the first approach is more efficient in terms of the **total amount of memory** used during the entire execution of the algorithm.

Problem 5: Floating point mysteries

In order to give a reasonable explanation to the problem stated, we need to understand how R (and many other programming languages) are dealing with floating point numbers storage and which kind of approximations/truncation are performed when keeping numbers (such as decimals) in memory.

We know that floating point numbers are stored in a computer as binary fractions, where a base of 2 is used. As a simple example, we have that the decimal fraction 0.125 can be represented as a binary fraction by 0.001 because its value can be translated by $\frac{0}{2} + \frac{0}{4} + \frac{1}{8} = 0.125$, so they are equivalent. However, we will see that some decimal numbers are not as easy as the previous case to be represented by binary fractions, producing rounding errors that will generate inconsistencies when two floating numbers are being compared.

For example, let's take the value 0.1. This decimal number cannot be represented exactly as a base 2 fraction because we obtain a repeating (infinite) fraction. For completeness, we will use a $floatToBin()$ function for displaying the binary representation of the numbers:

```r
# Declaring the function
floatToBin <- function(x){
  # Select the integer part of the number
  intpart <- floor(x)

  # Decimal part of x
  decpart <- x - intpart

  # Binary representation of the integer part
  intbin <- R.utils::intToBin(intpart)

  # Binary representation of the fractional part
  decbin <- stringr::str_pad(R.utils::intToBin(decpart * 2^31), 31, pad = "0")

  # Generate the number
  sub("[.]?0+$", "", paste0(intbin, ".", decbin))
}

# Testing it with 0.1
floatToBin(0.1)
```

```
## [1] "0.0001100110011001100110011001100110011"
```

```r
# Compare with out FloatingFormat function
FloatingFormat(0.1)
```

```
## [1] "Sign (S): 0 - Length: 1 - Number: 0"
## [1] "Exponent (e): 01111111011 - Length: 11 - Number: 1019"
## [1] "Mantissa (d): 1001100110011001100110011001100110011001100110011010 - Length: 52"
## [1] "               Number:  0.6"
## [1] "Number representation: 0.1 = (-1)^0 * (1.6) * 2^(1019 - 1023)"
```

As we can see, there is a clear (repetitive) pattern when we represent the 0.1 decimal number as a binary fraction. In fact, it is an infinite representation that has been truncated by R in order to display a finite number. Therefore, an approximation is being made behind the scenes by R.

We know from the first problem that floating point numbers are approximated using binary fractions where the numerator includes 53 bits, including a "hidden" 1 at the beginning and then we use denominators of a power of two. In this case, 0.1 is represented by the binary fraction ($3602879701896397/(2^{55})$), that is close to 0.1, but not exact.

Another interesting example is the following:

```
# Direct operation (FALSE)
0.1 + 0.1 + 0.1 == 0.3
```

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

```
# Individual roundind (FALSE)
round(0.1,1) + round(0.1,1) + round(0.1,1) == 0.3
```

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

```
# Global rounding (TRUE)
round(0.1 + 0.1 + 0.1, 1) == round(0.3, 1)
```

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

The first case is expected since we already know that 0.1 is not being stored exactly in the computer and thus, approximations are being made. The second case is also false since we can also guess that 0.3 does not have an exact binary fraction representation and thus, it is not equal to its real value. We can check this fact with the following code:

```
# Check representation
floatToBin(0.3)
```

```
## [1] "0.010011001100110011001100110011"
```

```
FloatingFormat(0.3)
```

```
## [1] "Sign (S): 0 - Length: 1 - Number: 0"
## [1] "Exponent (e): 01111111101 - Length: 11 - Number: 1021"
## [1] "Mantissa (d): 0011001100110011001100110011001100110011001100110011 - Length: 52"
## [1] "               Number:  0.2"
## [1] "Number representation: 0.3 = (-1)^0 * (1.2) * 2^(1021 - 1023)"
```

As with 0.1, we can see that a clear infinite pattern lies on the 0.3 binary representation and thus, an approximation is being made.

Finally, we have that the third case is true since we are rounding the sum of the three 0.1 fractions and we are also rounding the 0.3 number, obtaining an "exact" value on both sides of the equality.

At this point, we know that there exists a "representation error" for some decimal fractions that cannot be represented exactly as binary fractions, leading to problems when two or more floating point numbers are being compared to R. R implements the IEEE-754 double precision standard where 754 doubles contains 53 bits of precision, as we saw in the first problem. Thus, R (the computer) will convert the decimal number (say 0.1 for illustration purposes) to the closest fraction of the form $\frac{d}{2^e}$, where d is the integer value containing the 53 bits.

Thus, we have:

$$\frac{1}{10} \approx \frac{d}{2^e} \Leftrightarrow d \approx \frac{2^e}{10} \tag{13}$$

Now, since d contains 53 bits so it is $\geq 2^{52}$ and $< 2^{53}$, we can see that the best value for the exponent e will be 56 since $\frac{2^{56}}{10}$ satisfies the previous conditions. Then, we can calculate d as the rounded quotient, obtaining a remainder of 6. Since this value is larger than half of 10, we will need to round up in order to get the best approximation: 7205759403792794.

Of course, since we rounded up the number, the final result is larger than the original value 0.1. On the other hand, if we had rounded down, it would be smaller than 0.1. Therefore, it is not possible to keep 0.1 as an exact value in the computer when using the binary fraction representation.

All the previous analysis can be extended to all the numbers but there is a special exception: power of two numbers. Since R is storing the numbers if a binary fraction representation (base 2), all the numbers that have an exact power of two representation are stored exactly. Therefore, values such as 2, 0.5, 0.125, etc. are easily stored in R, without introducing approximation errors. For example:

```r
# Testing power of two numbers
floatToBin(0.5)
```

```
## [1] "0.1"
```

```r
floatToBin(0.125)
```

```
## [1] "0.001"
```

```r
floatToBin(2)
```

```
## [1] "10"
```

```r
floatToBin(8)
```

```
## [1] "1000"
```

```r
# Non-power of two numbers
floatToBin(0.15)
```

```
## [1] "0.0010011001100110011001100110011"
```

```r
floatToBin(0.2)
```

```
## [1] "0.0011001100110011001100110011001"
```

```r
floatToBin(0.33)
```

```
## [1] "0.0101010001111010111000010100011"
```

```r
floatToBin(0.66)
```

```
## [1] "0.10101000111101011100001010001111"
```

Clearly, we can see how the power of two numbers are easily stored in the binary fraction format without introducing any approximation.

Finally, we can introduce a new function that will help us to analyze and compare floating point numbers in order to understand why some operations are consistent with some numbers and not for others. The idea of the following function is to display the decimal, hexadecimal, and binary representation of each decimal number given as an input (vector), allowing us to compare all their elements and realize why they are not equal. As an example, we will test the expression $0.3 + 0.6 == 0.9$ and we will check why the result is not true:

```r
# Testing expression (FALSE)
0.3 + 0.6 == 0.9
```

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

```r
# Auxiliary function declaration: input vector of relevant numbers to compare
fullComparison <- function(nums){

  # Create a data frame with all relevant information
  ComparisonDF <- data.frame("decimal-2" = nums,
                             "decimal-17" = format(nums, digits=17),
                             hexadecimal = sprintf("%+13.13a", nums),
                             "binary" = floatToBin(nums))

  # Change the name of the rows
  attributes(ComparisonDF)$row.names <- as.character(nums)
  attributes(ComparisonDF)$row.names[3] <- paste(nums[1], "+", nums[2])


  # Print the DF
  ComparisonDF
}

# Calling the function: numbers, sum and expected result
nums <- c(0.3, 0.6, 0.3 + 0.6, 0.9)
fullComparison(nums)
```

```
##             decimal.2          decimal.17           hexadecimal
## 0.3               0.3 0.29999999999999999 +0x1.3333333333333p-2
## 0.6               0.6 0.59999999999999998 +0x1.3333333333333p-1
## 0.3 + 0.6         0.9 0.89999999999999991 +0x1.ccccccccccccccp-1
## 0.9               0.9 0.90000000000000002 +0x1.cccccccccccccdp-1
##                                     binary
## 0.3        0.01001100110011001100110011001
## 0.6         0.1001100110011001100110011001
## 0.3 + 0.6 0.11100110011001100110011001100
## 0.9       0.11100110011001100110011001100
```

Based on the information presented in [1] *We need to align binary points for addition. The shift is calculated by converting hex to binary, shifting one bit to the right to get the same p-1 exponent, regrouping four bits into hex characters, and allowing the last bit to fall of*:

$$1.001100110011...0011 \times 2^{-2} \Rightarrow .100110011001...1001 \times 2^{-1} \tag{14}$$

Hence, the explanation behind the FALSE result consists of the fact that the floating point representation of 0.3 and 0.6 have to be aligned on the binary point before the addition. Once the numbers are completely aligned, by shifting the smaller number right one position, we end up with a situation where the last bit of the smaller number does not have any place to be allocated and thus, it is lost. Thus, the final sum will be one bit smaller in comparison to the floating point binary representation of the expected 0.9 result.

Now, based on all the previous discussion, we can analyze the "mysteries" stated in the problem:

1. $0.2 + 0.3 == 0.5$

   As with the example, we use the $fullComparison()$ function and the $floatToBin()$ function for understanding why this expression is TRUE:

```
# Perform comparison
fullComparison(c(0.2, 0.3, 0.2 + 0.3, 0.5))
```

```
##             decimal.2          decimal.17           hexadecimal
## 0.2               0.2 0.20000000000000001 +0x1.999999999999ap-3
## 0.3               0.3 0.29999999999999999 +0x1.3333333333333p-2
## 0.2 + 0.3         0.5 0.50000000000000000 +0x1.0000000000000p-1
## 0.5               0.5 0.50000000000000000 +0x1.0000000000000p-1
##                                      binary
## 0.2         0.00110011001100110011001100110011001
## 0.3          0.0100110011001100110011001100110011
## 0.2 + 0.3                                   0.1
## 0.5                                         0.1
```

```
floatToBin(0.2 + 0.3)
```

```
## [1] "0.1"
```

```
floatToBin(0.5)
```

```
## [1] "0.1"
```

In this case, we can easily see from the results above that there is no difference between the sum and the stored 0.5 value. This means that 0.2 and 0.3 are stored as binary fractions in such a way that they are equivalent to this power of two (-1) number and no last bit is lost during the summation. In fact (for completeness), we can check in the web (such as in http://www.binary convert.com/) how some decimals numbers are being represented in the IEEE-745 format standard and we can easily check that for 0.2 the most accurate representation is $2.00000000000000011102230246252e - 1$ and for 0.3 it is $2.99999999999999988897769753748e - 1$. Using these numbers, we can easily check that their sum is exactly 0.5.

2. $0.01 + 0.49 == 0.5$

   Similar to the previous case, we repeat the analysis using our auxiliary functions:

```
fullComparison(c(0.01, 0.49, 0.01 + 0.49, 0.5))
```

```
##              decimal.2          decimal.17           hexadecimal
## 0.01              0.01 0.01000000000000000 +0x1.47ae147ae147bp-7
## 0.49              0.49 0.48999999999999999 +0x1.f5c28f5c28f5cp-2
## 0.01 + 0.49       0.50 0.50000000000000000 +0x1.0000000000000p-1
## 0.5               0.50 0.50000000000000000 +0x1.0000000000000p-1
##                                       binary
## 0.01           0.0000001010001111010111000101
## 0.49         0.0111110101110000101000111101011
## 0.01 + 0.49                                0.1
## 0.5                                        0.1
```

```
floatToBin(0.01 + 0.49)
```

```
## [1] "0.1"
```

```
floatToBin(0.5)
```

```
## [1] "0.1"
```

Again, we can see in this case that no last bit is lost and thus, the way R (and other programming languages) stores 0.01 and 0.49 allows us to obtain exactly the same value stored for 0.5 when the summation is performed.

Before checking the last expression, it is interesting to check if this particular pattern is due to the fact that 0.5 is a power of two number. In order to test this idea, we can perform a series of test for different decimal numbers summation such that their sum is equal to 0.5 and check if we obtain the same TRUE result as before. In addition, we can test the different power of two values such as 0.125.

```
# Do not output warnings
options(warn = -1)

# Power of two tests
# Testing loop 0.5
a <- seq(0.01, 0.49, 0.01)
b <- seq(0.49, 0.01, -0.01)

BoolVector <- integer(length(a))
aux <- 1

for (i in seq_along(a)){
  BoolVector[aux] = (a + b == 0.5)
  aux <- aux + 1
}

# Check for FALSE values
if(length(BoolVector[BoolVector == FALSE]) > 0){
  print("Some test are not TRUE")
}
```

```r
# Testing loop 0.25
a <- seq(0.01, 0.24, 0.01)
b <- seq(0.24, 0.01, -0.01)

BoolVector <- integer(length(a))
aux <- 1

for (i in seq_along(a)){
  BoolVector[aux] <- (a + b == 0.25)
  aux <- aux + 1
}

# Check for FALSE values
if(length(BoolVector[BoolVector == FALSE]) > 0){
  print("Some test are not TRUE")
}

# Testing loop 0.125
a <- seq(0.01, 0.124, 0.01)
b <- seq(0.124, 0.01, -0.01)

BoolVector <- integer(length(a))
aux <- 1

for (i in seq_along(a)){
  BoolVector[aux] <- (a + b == 0.125)
  aux <- aux + 1
}

# Check for FALSE values
if(length(BoolVector[BoolVector == FALSE]) > 0){
  print("Some test are not TRUE")
}
```

```
## [1] "Some test are not TRUE"
```

Based on the previous output, we can clearly see that our idea is not right: we also have some inconsistencies when the resulting number is a power of two, indicating us that the relevant factor for determining if the summation of two floating point numbers is equal to their expected result lies on the approximation (representation) used for storing them inside the computer. Depending on this, we can have a situation where the expressions will be TRUE or FALSE, but it is not completely related to the expected result (like a power of two property).

3. $0.2 + 0.1 == 0.3$

   Since we know that this expression is FALSE, we expect that the last bit of summation to be different in comparison to the expected value, as we already discussed above:

```r
# Comparison
fullComparison(c(0.2, 0.1, 0.2 + 0.1, 0.3))
```

```
##           decimal.2          decimal.17           hexadecimal
## 0.2             0.2 0.20000000000000001 +0x1.999999999999ap-3
## 0.1             0.1 0.10000000000000001 +0x1.999999999999ap-4
## 0.2 + 0.1       0.3 0.30000000000000004 +0x1.3333333333334p-2
## 0.3             0.3 0.29999999999999999 +0x1.3333333333333p-2
##                                      binary
## 0.2       0.001100110011001100110011001
## 0.1         0.00011001100110011001100110011
## 0.2 + 0.1  0.010011001100110011001100110011
## 0.3        0.010011001100110011001100110011
```

```r
floatToBin(0.2 + 0.1)
```

```
## [1] "0.010011001100110011001100110011"
```

```r
floatToBin(0.3)
```

```
## [1] "0.010011001100110011001100110011"
```

Based on the $fullComparison()$ function output, we can clearly see from the hexadecimal representation that the last bit is lost when the summation is performed, obtaining a different value in comparison to the expected 0.3 stored in R and thus, the expression is FALSE. Note that the binary representation looks identical but we are not taking into account all the digits and the truncation that R is performing for each individual term (0.2 and 0.1 are stored as approximations when they are transformed to a binary fraction format, as we discussed at the beginning of this problem).

Therefore, based on our previous discussion, we now understand why some summations obtain the expected result (and thus, the evaluation is TRUE) while other expressions fail: depending on the approximation performed when the number is stored as a binary fraction, bits can be lost when summations are performed, obtaining a different result in comparison to the binary representation of the expected value in R, leading to FALSE evaluations. In addition, we tested that this is not a condition linked to the power of two results, they can also fail the evaluation of the expression if no power of two numbers are used as the summation factors.