Case Study 1

AKSTA Statistical Computing 107.258

Maximilian Hagn

21. April 2022

Ratio of Fibonacci numbers

For-Loop

In the following code snippet the ratio of the nth Fibonacci number to the following number is calculated. For this purpose, the start values are determined first. Then a for-loop is executed n - 1 times. In each step the next number is calculated and stored. After the last and the next number has been calculated, the loop is terminated and the ratio is returned.

```
fibonacciRatioForLoop <- function(n) {

  F_x <- 0
  F_1 <- 0
  F_2 <- 1

  for(x in 0:(n-1)) {
    F_x = F_1 + F_2
    F_1 = F_2
    F_2 = F_x
  }

  return(F_2/F_1)

}
```

While-Loop

In the following code snippet, the same principle is applied as in the calculation with the for-loop. The only difference is that in addition it must be counted in which iteration the program is, so that the termination condition can take effect.

```
fibonacciRatioWhileLoop <- function(n) {

  F_x <- 0
  F_1 <- 0
  F_2 <- 1
  x = 0

  while(x <= (n-1)) {
    f_i = F_1 + F_2
    F_1 = F_2
```

```
    F_2 = f_i
    x = x + 1
  }

  return(F_2/F_1)

}
```

Benchmark

To accomplish this task, the package "microbenchmark" must be installed first, with the following code:

```
package.install(microbenchmark)
```

Afterwards the library can be included:

```
library(microbenchmark)
```

To perform the benchmark test, the two expressions whose runtime is to be checked must be passed to the function. In the first step, the runtime is calculated for n=100:

```
n <- 100
benchmark_100 = microbenchmark(fibonacciRatioForLoop(n), fibonacciRatioWhileLoop(n))
benchmark_100
```

```
## Unit: microseconds
##                          expr   min     lq      mean median    uq      max neval
##    fibonacciRatioForLoop(n)  5.82  6.135  31.63561  7.480  7.68  2452.20   100
##  fibonacciRatioWhileLoop(n) 10.17 10.555 138.56083 11.795 11.99 12702.27   100
```

It can be observed that the median of the execution times of the for-loop takes less time than the execution of the While loop. Furthermore, the minimum measured runtime of the for-loo is lower, while the maximum measured time is also higher than that of the while-loop. It can thus be stated that the for-loop has a greater spread of execution times and the while-loop thus acts more consistently in terms of time consumption.

In the next step, the runtime is calculated for n=1000:

```
n <- 1000
benchmark_1000 = microbenchmark(fibonacciRatioForLoop(n), fibonacciRatioWhileLoop(n))
benchmark_1000
```

```
## Unit: microseconds
##                          expr   min      lq      mean  median     uq    max neval
##    fibonacciRatioForLoop(n) 53.98  58.850  62.43531  60.345  66.67  74.82   100
##  fibonacciRatioWhileLoop(n) 97.87 102.255 106.61973 108.350 110.13 129.59   100
```

It can be stated again that the for-loop takes less time than the while-loop. This time, however, it should be noted that the times of both variants slowly converge.

Plot

The library "microbenchmark" offers a possibility to visualize the results with the help of the function autoplot(). Therefore the library "ggplot2" must be installed first:

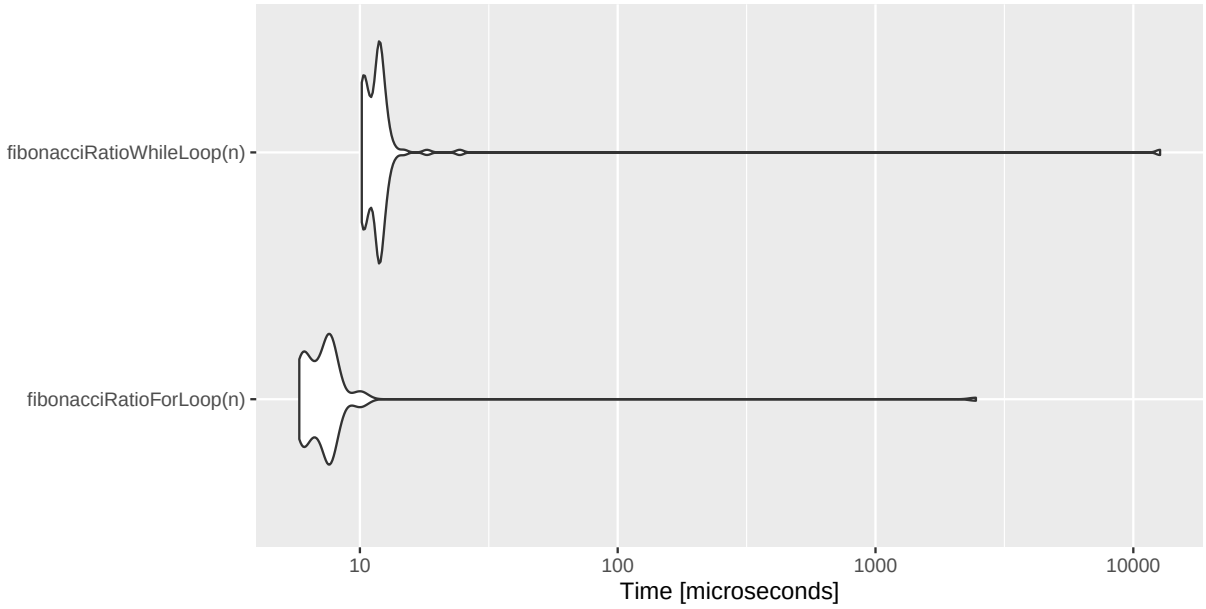
```
package.install(ggplot2)
```

Afterwards the library can be included:

```
library(ggplot2)
```

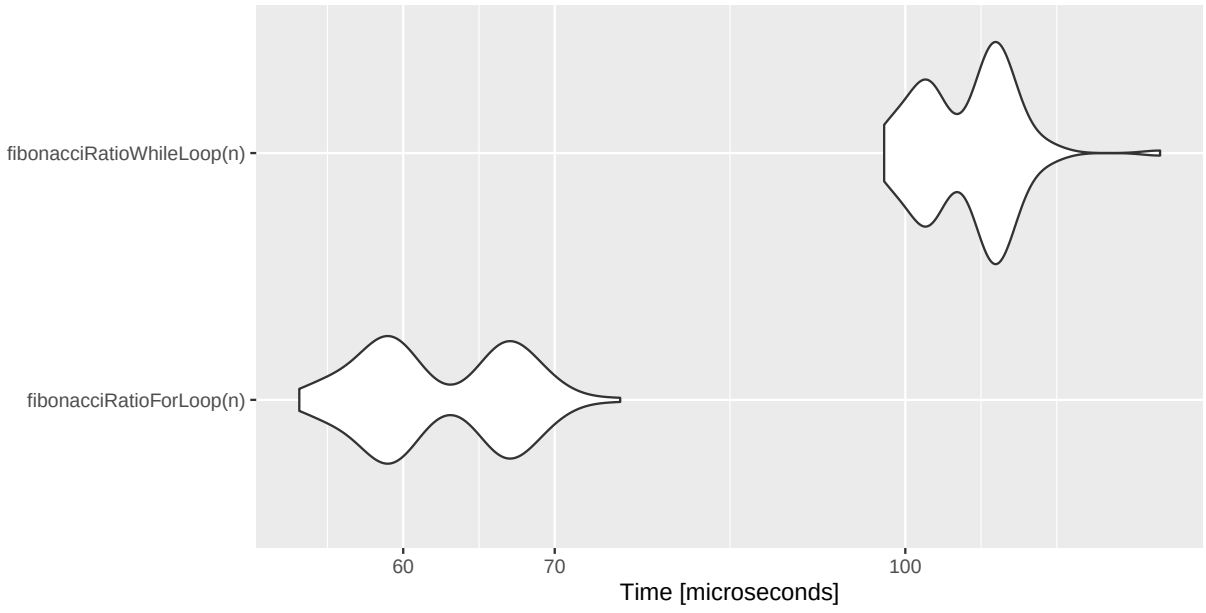
Subsequently, the result of microbenchmark can be visualized by inserting the result into the autoplot function. This is done first again for n=100:

```
autoplot(benchmark_100)
```



Afterwards, the result of n=1000 can also be displayed:

```
autoplot(benchmark_1000)
```



Hier kommt eine Erklärung hin. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt

ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

The golden ratio

```
goldenRatio <- (sqrt(5) + 1)/2
testValues = c(1, 5, 10, 25, 40, 50, 100, 500, 1000)

CheckEquality_1 <- function(values) {
  for(i in values){
     cat(sprintf("fibonacci(%d): %s\n", i, fibonacciRatioForLoop(i) == goldenRatio))
  }
}

CheckEquality_1(testValues)
```

```
## fibonacci(1): FALSE
## fibonacci(5): FALSE
## fibonacci(10): FALSE
## fibonacci(25): FALSE
## fibonacci(40): TRUE
## fibonacci(50): TRUE
## fibonacci(100): TRUE
## fibonacci(500): FALSE
## fibonacci(1000): FALSE
```

```
goldenRatio <- (sqrt(5) + 1)/2
testValues = c(1, 5, 10, 25, 40, 50, 100, 500, 1000)

CheckEquality_2 <- function(values) {
  for(i in values){
    cat(sprintf("fibonacci(%d): %s\n", i, all.equal(fibonacciRatioForLoop(i), goldenRatio)))
  }
}
CheckEquality_2(testValues)
```

```
## fibonacci(1): Mean relative difference: 0.618034
## fibonacci(5): Mean relative difference: 0.01127124
## fibonacci(10): Mean relative difference: 9.135527e-05
## fibonacci(25): TRUE
## fibonacci(40): TRUE
## fibonacci(50): TRUE
## fibonacci(100): TRUE
## fibonacci(500): TRUE
## fibonacci(1000): TRUE
```

Game of craps

The game starts by throwing two dice. These are stored in the variables diceResult_1 and diceResult_2 in my program and generated by runif(). The first parameter describes that one number should be generated, whereas the second and third parameters describe the limits. The calculated number is then rounded down. This process could also be done by generating only a random number between 2 and 12, but here the approach to generate two numbers was chosen to represent the abstract concept of the game. After the sum of the two dice has been calculated, a check is made to see if the numbers rolled together add up to 7 or 11. If this condition is met, a string containing the diced value and "Won!" is returned. If not satisfied, the number of dice pips are stored and written to the output that the player must roll again. The while-loop is now executed until either the condition to win or the condition to lose is met. The player wins if the sum of the current rolls equals the sum of the first rolls. If this condition is met, a string is returned that contains the current value of the dice and "Won!". If the player rolls the value seven or eleven after the first round, the current value and "Lost!" are returned. To better track the progress of the game, after the first round, as well as after all rolls, the number that was rolled is also returned. The first round and the remaining rounds are intentionally treated separately in this implementation, since checking each round if it is the first round would have a negative effect on performance.

The code is shown in the following snippet:

```
game_of_craps <- function() {

  diceResult_1 <- floor(runif(1, min=1, max=6))
  diceResult_2 <- floor(runif(1, min=1, max=6))
  diceSum = diceResult_1 + diceResult_2

  if(diceSum == 7 || diceSum == 11){
     return(paste("x = ", diceSum, "- Won!"))
  }

  firstRoll <- diceSum
  print(paste("x = ", diceSum, "- Roll Again!"))

  while(TRUE) {

    diceResult_1 <- floor(runif(1, min=1, max=7))
    diceResult_2 <- floor(runif(1, min=1, max=7))
    diceSum = diceResult_1 + diceResult_2

      if(diceSum == firstRoll){
        return(paste("x = ", diceSum, "- Won!"))
      }

      if(diceSum == 7 || diceSum == 11){
        return(paste("x = ", diceSum, "- Lost!"))
      }

      print(paste("x = ", diceSum, "- Roll Again!"))
  }
}
```

Three runs of the game are given below:

```
game_of_craps()
```

```
## [1] "x =  7 - Won!"
```

```
game_of_craps()
```

```
## [1] "x =  9 - Roll Again!"
```

```
## [1] "x =  9 - Won!"
```

```
game_of_craps()
```

```
## [1] "x =  7 - Won!"
```

Readable and efficient code

Code Wrapped in Function “foobar0”

In this implementation, the code was inserted into the “foobar0” function. For this purpose, it was only additionally changed that the value of x is returned.

```
foobar0 <- function(x, z) {
  set.seed(1)

  if (sum(x >= .001) < 1) {
    stop("step 1 requires 1 observation(s) with value >= .001")
  }

  fit <- lm(x ~ z)
  r <- fit$residuals
  x <- sin(r) + .01

  if (sum(x >= .002) < 2) {
    stop("step 2 requires 2 observation(s) with value >= .002")
  }

  fit <- lm(x ~ z)
  r <- fit$residuals
  x <- 2 * sin(r) + .02
  if (sum(x >= .003) < 3) {
   stop("step 3 requires 3 observation(s) with value >= .003")
  }

  fit <- lm(x ~ z)
  r <- fit$residuals
  x <- 3 * sin(r) + .03

  if (sum(x >= .004) < 4) {
    stop("step 4 requires 4 observation(s) with value >= .004")
  }

  fit <- lm(x ~ z)
  r <- fit$residuals
  x <- 4 * sin(r) + .04

  return(x)
}
```

Rewrite

For the implementation, the repetitive pieces of code were moved to their own functions. The input was outsourced to the function “checkInput” and the calculation of the values to the function “computeValues”. These functions are executed one after the other in the main function “foobar”.

```
checkInput <- function(x, step) {

  if (sum(x >= (step * .001)) < step) {
    stop(paste("step ", step, " requires ", step, " observation(s) with value >= ", (step * .001)))
  }

}

computeValues <- function(x, z, step) {

  fit <- lm(x ~ z)
  r <- fit$residuals
  x <- step * sin(r) + (.01 * step)
  return(x)

}

foobar <- function(x, z) {

  set.seed(1)
  checkInput(x, 1)
  x <- computeValues(x, z, 1)
  checkInput(x, 2)
  x <- computeValues(x, z, 2)
  checkInput(x, 3)
  x <- computeValues(x, z, 3)
  checkInput(x, 4)
  x <- computeValues(x, z, 4)

  return(x)

}
```

Equality

If the two implementations are compared with the function “all.equal”, the result returns true. Thus, it can be stated that the semantics of the program remains the same.

```
normValue <- rnorm(100)
all.equal(foobar0(normValue, normValue), foobar(normValue, normValue))
```

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