Case Study 1

AKSTA Statistical Computing

Fani Sentinella-Jerbić

30.03.2023

Ratio of Fibonacci numbers

1.

The task was to write R functions which return the sequence $r_n = F_{n+1}/F_n$ using for and while loops.

I used the version of Fibonacci sequence starting with 1 because otherwise the first member of the r_n sequence would be 1/0 which is undefined.

My functions use two vectors, one storing the Fibonacci numbers, and one storing the special sequence. Both sequences are simultaneously calculated inside the loops.

FOR version

```
fib_for <- function(n){
  if (n == 1) return(c(1))

  fib <- c(1, 1, 2)
  r <- c(1, 2)

  if (n > 2){
    for (i in 2:n){
      fib[i+1] = fib[i-1] + fib[i]
      r[i] = fib[i+1]/fib[i]
    }
  }
  return(r)
}
fib_for(8)
```

```
## [1] 1.000000 2.000000 1.500000 1.666667 1.600000 1.625000 1.615385 1.619048
```

WHILE version

```
fib_while <- function(n){
  if (n == 1) return(c(1))

  fib <- c(1,1, 2)
  r <- c(1, 2)

  if (n > 2){
    i = 2
    while (i <= n){
      fib[i+1] = fib[i-1] + fib[i]
      r[i] = fib[i+1]/fib[i]
      i = i + 1
    }
    return(r)
  }
}
fib_while(8)
```

```
## [1] 1.000000 2.000000 1.500000 1.666667 1.600000 1.625000 1.615385 1.619048
```

2.

Benchmarking the two functions for $n = 100$ and $n = 1000$:

```
library(microbenchmark)
```

```
## Warning: package 'microbenchmark' was built under R version 4.2.3
```

```
microbenchmark(fib_for(100), fib_while(100))
```

```
## Unit: microseconds
##            expr   min    lq    mean median     uq   max neval
##    fib_for(100) 106.0 125.6 170.794  153.1 173.25 783.7   100
##  fib_while(100) 111.1 138.3 193.512  166.3 205.95 974.5   100
```

```
microbenchmark(fib_for(1000), fib_while(1000))
```

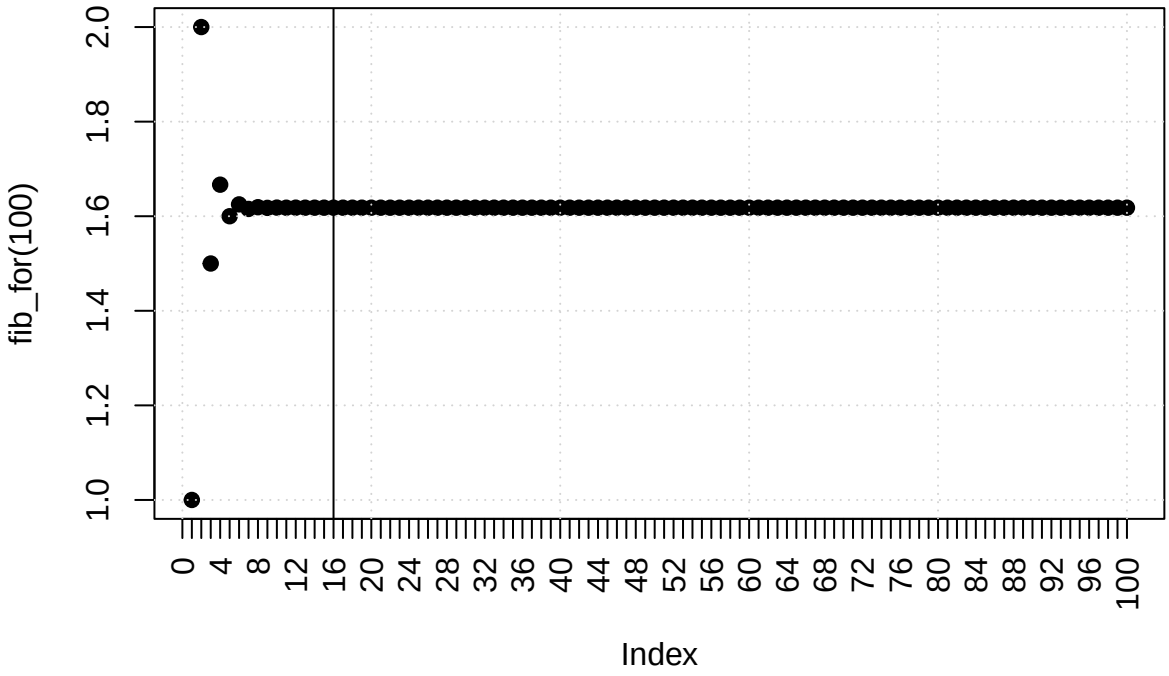
```
## Unit: microseconds
##             expr   min      lq     mean  median      uq    max neval
##    fib_for(1000) 835.0  995.40 1436.761 1146.50 1505.55 8805.4   100
##  fib_while(1000) 949.8 1069.75 1506.250 1202.65 1521.45 9073.5   100
```

FOR loop seems to be faster than the WHILE loop. Probably because it doesn't need to execute a condition check for continuation.

3.

Plotting the sequence for $n = 100$:

```
plot(fib_for(100), pch = 19, xaxt='n')
axis(1, at = seq(0, 100, by = 1), las=2)
abline(v=16)
grid()
```



The sequence converges very fast towards the "perfect number" or "golden ratio". From the plot it is hard to see when it actually reaches it, so I'm considering the actual values to the 6th decimal:

```
fib_for(20)
```

```
##  [1] 1.000000 2.000000 1.500000 1.666667 1.600000 1.625000 1.615385 1.619048
##  [9] 1.617647 1.618182 1.617978 1.618056 1.618026 1.618037 1.618033 1.618034
## [17] 1.618034 1.618034 1.618034 1.618034
```

It is reached for $n = 16$.

The golden ratio

Showing the golden ratio satisfies the Fibonacci-like relationship:

```
phi = (sqrt(5)+1)/2
eq = 0
al = 0
for (n in 1:100) {
  lhs = phi^(n+1)
  rhs = phi^n + phi^(n-1)
  if(lhs==rhs) eq = eq + 1
  if(all.equal(lhs, rhs)) al = al + 1
  #print(paste("n =", n))
  #print(paste("==       ", lhs==rhs))
  #print(paste("all.equal", all.equal(lhs, rhs)))
}
```

```
print(eq)
```

```
## [1] 67
```

```
print(al)
```

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

all.equal returned TRUE for all n values, whereas == returned TRUE only for 67 n values.

== can be sensitive to small differences in floating-point values due to rounding errors, causing the equality check to result in FALSE.

all.equal considers the float values with a specific tolerance level, which makes it less likely to be affected by rounding errors.

Game of craps

My implementation uses a temporary variable of **xtarget**. If the target is not yet set, we are in the first iteration. The dice are rolled by using **sample** function. If the sum of the dice is equal to 7 or 11, game is won. Otherwise, I save this sum to **xtarget** and the loop starts the next iteration. The dice are rolled. If the new sum is equal to 7 or 11, the game is lost. If the new sum equals the first sum, the game is won. Otherwise, the loop starts the next iteration and new dice are rolled. The game continues so on until an outcome is reached. The roll variable is only used for printing the game flow.

```
roll = 1
xtarget <- NULL

repeat {
  dice <- sample(1:6, 2)
  x <- sum(dice)
  print(paste("ROLL", roll, ":", dice[1], dice[2]))

  if(is.null(xtarget)){
    if(x == 7 | x == 11){
      print("WIN")
      break
    }
    xtarget <- x
  } else {
    if(x == 7 | x == 11){
      print("LOSE")
      break
    }
    if(x == xtarget){
      print("WIN")
      break
    }
  }
  roll <- roll + 1
}
```

```
## [1] "ROLL 1 : 3 6"
## [1] "ROLL 2 : 6 4"
## [1] "ROLL 3 : 3 6"
## [1] "WIN"
```

Readable and efficient code

Original version

```
set.seed(1)
x <- rnorm(100)
z <- rnorm(100)
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
test <- x
```

My version

I extracted conditional stopping and model fitting to two seperate functions **validate** and **special_fit**. The code could be even further simplified if I made assumptions on the step number always having to be equal to the numbers inside the conditions and fits, but I didn't want to "overassume".

```
validate <- function(x, a, b){
  if (sum(x >= a) < b) {
    stop(paste("step", b, "requires", b, "observation(s) with value >=", a))
  }
}

special_fit <- function(x, z, a, b){
  fit <- lm(x ~ z)
  r <- fit$residuals
  a * sin(r) + b

}
```

```
foobar <- function(x, z){
  validate(x, .001, 1)
  x <- special_fit(x, z, 1, .01)

  validate(x, .002, 2)
  x <- special_fit(x, z, 2, .02)

  validate(x, .003, 3)
  x <- special_fit(x, z, 3, .03)

  validate(x, .004, 4)
  x <- special_fit(x, z, 4, .04)
  x
}

set.seed(1)
x <- rnorm(100)
z <- rnorm(100)
all.equal(test, foobar(x,z))
```

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