Case Study 1 AKSTA Statistical Computing

Hanna Kienast Iulia Mihaela Enache Kateryna Ponomarenko

2024-03-25

Task 1

a) Using for loop

We will add the check if the given number is positive integer and equals at least 1.

Then we create numeric vector filled with NA of length n + 1 to store numbers of Fibonacci sequence there and initialize first two Fibonacci numbers.

In the case n is 2, we return ratio of 2 first numbers, else we calculate in a loop all numbers of sequence up to n + 1 inclusively.

We initialize numeric vector of length n to store ratios and calculate in a loop final results.

```
for_impl_fib_ratio <- function(n) {
  if (!is.integer(n) || n < 2) {
    stop("n must be a positive integer greater than or equal to 2")
  }

  fib <- numeric(n + 1)
  fib[1:2] <- 1

  for (i in 3:(n + 1)) {
    fib[i] <- fib[i - 1] + fib[i - 2]
  }

  r <- numeric(n)
  for (i in 1:n) {
    r[i] <- fib[i + 1] / fib[i]
  }

  return(r)
}

print(for_impl_fib_ratio(4L))
```

```
## [1] 1.000000 2.000000 1.500000 1.666667
```

a) Using while loop

Rewriting the function to use while loop by setting conditions for 2 while loops and updating i variable in each loop.

```r
while_impl_fib_ratio <- function(n) {
  if (!is.integer(n) || n < 2) {
    stop("n must be a positive integer greater than or equal to 2")
  }

  fib <- numeric(n + 1)
  fib[1:2] <- 1

  if (n == 2) {
    return(fib[2] / fib[1])
  }

  i <- 3
  while(i <= (n + 1)) {
    fib[i] <- fib[i - 1] + fib[i - 2]
    i <- i + 1
  }

  r <- numeric(n)

  i <- 1
  while(i <= n) {
    r[i] <- fib[i + 1] / fib[i]
    i <- i + 1
  }

  return(r)
}

print(for_impl_fib_ratio(4L))
```

```
## [1] 1.000000 2.000000 1.500000 1.666667
```

b) Using microbenchmark function

It was decided to run evaluation for each function 1000 times

```r
library(microbenchmark)
```

```
## Warning: package 'microbenchmark' was built under R version 4.3.3
```

```r
benchmark_200 <- microbenchmark(
  for_loop = for_impl_fib_ratio(200L),
  while_loop = while_impl_fib_ratio(200L),
  times = 1000L
)

benchmark_2000 <- microbenchmark(
  for_loop = for_impl_fib_ratio(2000L),
  while_loop = while_impl_fib_ratio(2000L),
  times = 1000L
```

```r
)

print(benchmark_200)
```

```
## Unit: microseconds
##        expr    min     lq     mean median      uq       max neval
##    for_loop 26.500 27.201 37.91419 27.701 46.7005   337.501  1000
##  while_loop 37.801 38.502 66.88031 39.001 70.5515 15488.801  1000
```

```r
print(benchmark_2000)
```

```
## Unit: microseconds
##        expr     min       lq     mean   median       uq     max neval
##    for_loop 246.202 258.9015 370.0225 281.5005 480.0510 11419.9  1000
##  while_loop 362.601 374.5010 508.6598 399.1505 590.0005  1903.2  1000
```

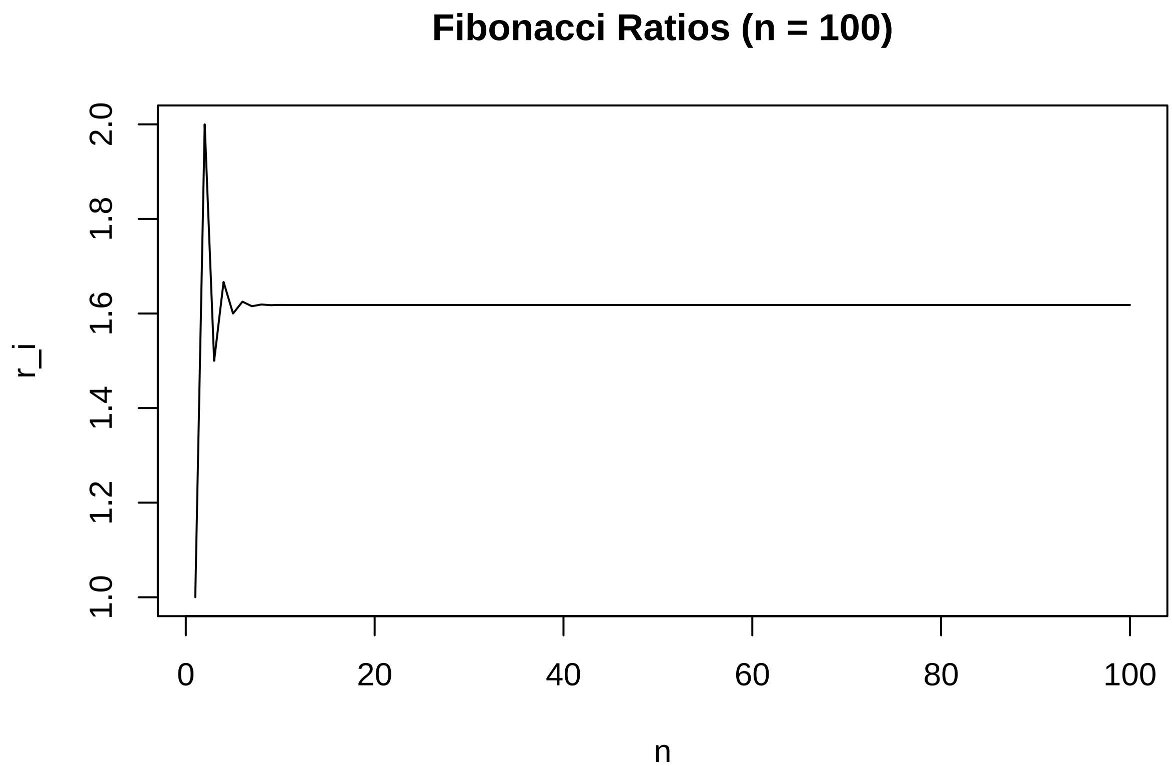
Conclusion: we can summarize that for loop is more efficient, while the mean and median execution time are lower than these indicators for while loop. It may be because in for loop, we don't need to initialize counter variable and update it manually. However, we noticed, that in case n = 2000 the max execution time of for loop is greater than while loop's time. We think, that potential reason might be some external factors, for example system load or CPU usage. But mean and median are more indicative for execution time distribution analysis, so we conclude the better efficiency of for loop.

c)

We will plot sequence for n = 100 to see, where is stabilizes

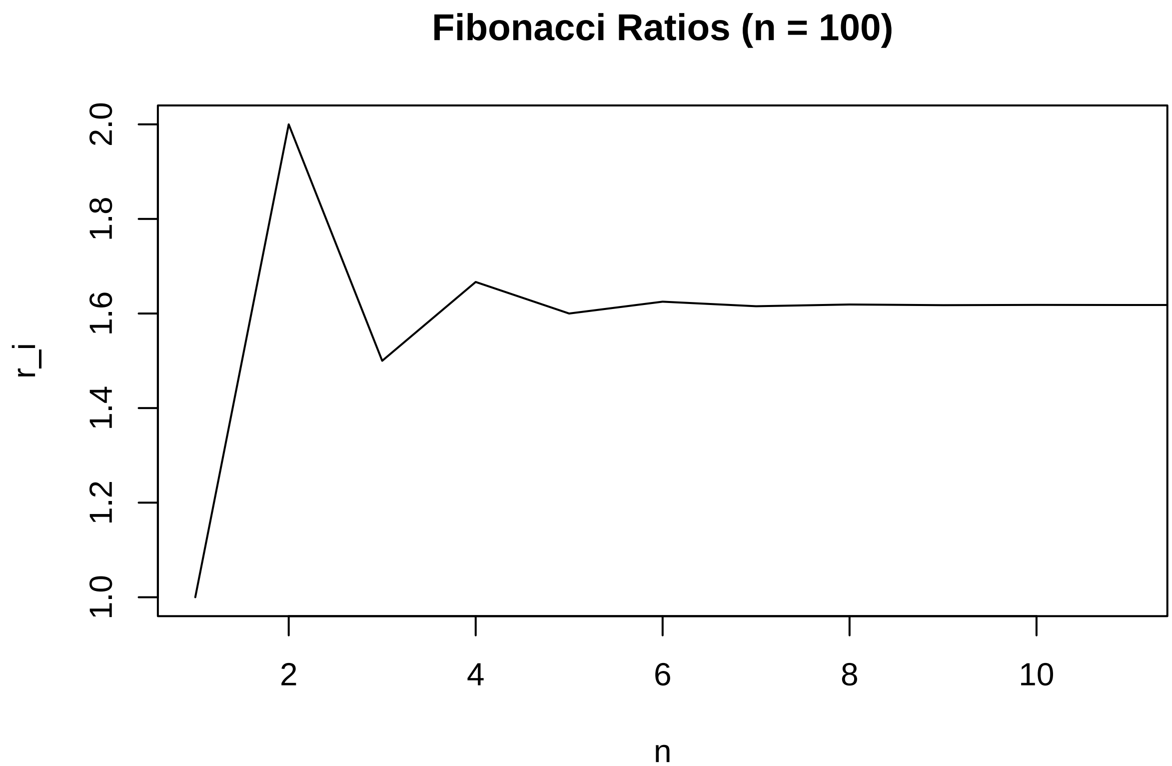
```r
# Call the function for n = 100
seq <- for_impl_fib_ratio(100L)

# Plot the sequence
plot(seq, type = "l", xlab = "n", ylab = "r_i", main = "Fibonacci Ratios (n = 100)")
```



We see, that sequence stabilizes roughly at n = 10, we can zoom in this sector of the plot to detect the preciser number.

```
plot(seq, type = "l", xlab = "n", ylab = "r_i", main = "Fibonacci Ratios (n = 100)", xlim = c(1, 11))
```



As we can see, ratios converge to the value ≈ 1.6, starting from the n ≈ 5 and stabilizing more from n ≈ 8.