

# 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 2.

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.

```

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

```
library(microbenchmark)
```

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

```

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
)

```

)

```
print(benchmark_200)
```

```
## Unit: microseconds
##      expr   min    lq   mean median    uq   max neval
## for_loop 28.7 29.7 31.8864   30.0 30.80  153.0  1000
## while_loop 40.8 41.9 56.5106   42.2 42.75 11354.6  1000
```

```
print(benchmark_2000)
```

```
## Unit: microseconds
##      expr   min    lq   mean median    uq   max neval
## for_loop 250.7 263.8 302.4009 282.85 297.40 1997.4  1000
## while_loop 362.9 380.6 448.5071 407.75 433.95 9548.8  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.

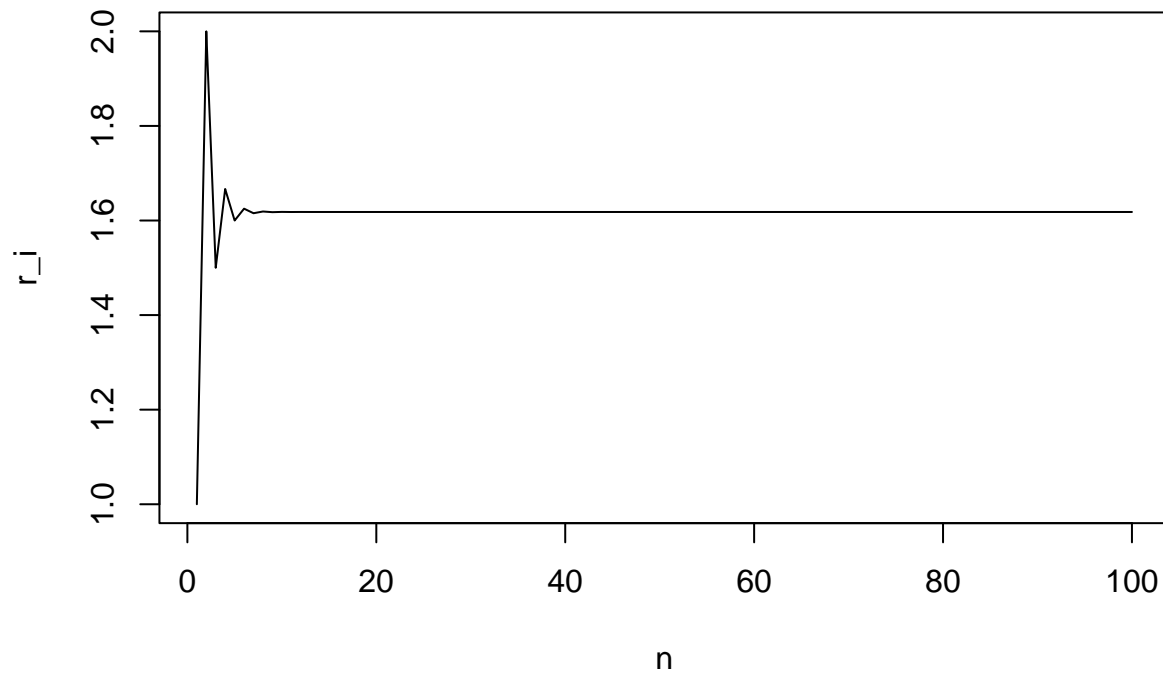
c)

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

```
# 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)")
```

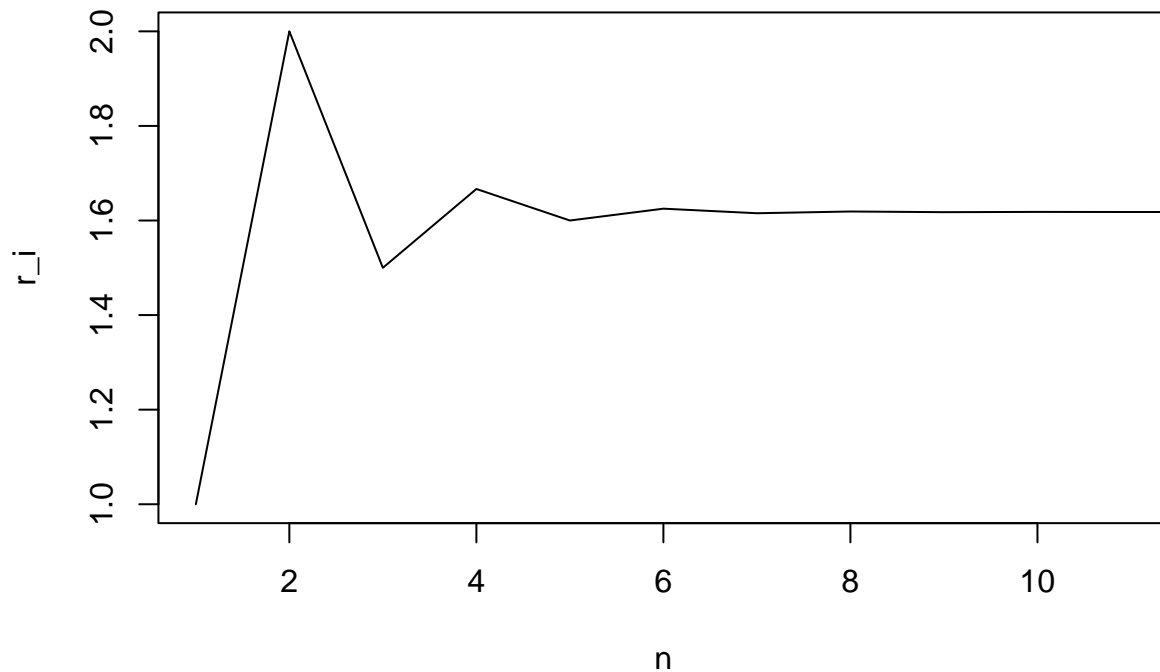
## Fibonacci Ratios (n = 100)



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))
```

## Fibonacci Ratios (n = 100)



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

### Task 3

#### a) Using recursion

First attempt to implement the function with the usage of recursion wasn't successful, because for  $n$  values greater than 100, the execution time turned out to be hours. Due to that, we decided to use memoization technique to optimize our code.

We define the helper function inside of main function, which is recursive and calculates the Fibonacci numbers with memoization using the environment. We define base cases for  $n = 0$  (calculate  $\phi^1$ ) and  $n = 1$  (calculate  $\phi^2$ ), which is  $\frac{\sqrt{5}+1}{2}$  and  $(\frac{\sqrt{5}+1}{2})^2 = \frac{\sqrt{3}+1}{2}$  correspondingly. Then we calculate recursively all following element of the sequence. Also, we check if  $n$  is integer that equals at least to 1.

Our both functions will return  $\phi^{(n+1)}$  (if  $n = 1$ , we calculate  $\phi^2$ ).

```
recursion_golden_ratio <- function(n = 1000L) {  
  if (!is.integer(n) || n < 0) {  
    stop("n must be a non-negative integer")  
  }  
  
  memo_env <- new.env(parent = emptyenv())  
  
  compute_golden_ratio <- function(n, memo) {  
    if (n == 0) {  
      return((sqrt(5) + 1) / 2)  
    }  
  }
```

```

    } else if (n == 1) {
      return((sqrt(5) + 3) / 2)
    }

    if (!is.null(memo[[as.character(n)]])) {
      return(memo[[as.character(n)]])
    }

    res <- compute_golden_ratio(n - 1L, memo) + compute_golden_ratio(n - 2L, memo)
    memo[[as.character(n)]] <- res

    return(res)
  }

  compute_golden_ratio(n, memo_env)
}

recursion_golden_ratio(100L)

```

```
## [1] 1.281598e+21
```

## b) Using power operator

Add the same check condition and simply return power  $n + 1$  of  $\phi$

```

power_golden_ratio <- function(n) {
  if (!is.integer(n) || n < 0) {
    stop("n must be a non-negative integer")
  }

  phi <- (sqrt(5) + 1) / 2
  return(phi ^ (n + 1L))
}

power_golden_ratio(100L)

```

```
## [1] 1.281598e+21
```

## c)

We will use `sapply` function to apply defined functions to 4 specified values and store them in vectors.

While comparing values with `==`, we create logical vector, where TRUE indicates that the values are exactly equal, FALSE otherwise.

To compare values with `all.equal`, we use the `sapply` again to iterate over each pair of `recursion_vals` and `power_vals` and apply `all.equal()` for comparison.

```

recursion_vals <- sapply(as.integer(c(12, 60, 120, 300)), recursion_golden_ratio)
power_vals <- sapply(as.integer(c(12, 60, 120, 300)), power_golden_ratio)

comparison_eq <- recursion_vals == power_vals

```

```
comparison_all_equal <- sapply(seq_along(reursion_vals), function(i) {
  all.equal(reursion_vals[i], power_vals[i])
})
```

```
cat("Comparison with == operator:\n", comparison_eq, "\n")
```

```
## Comparison with == operator:
## FALSE FALSE FALSE FALSE
```

```
cat("Comparison with all.equal function:\n", comparison_all_equal, "\n")
```

```
## Comparison with all.equal function:
## TRUE TRUE TRUE TRUE
```

In the result, we observe that comparison with == outputs inequality, while comparison with all.equal - visa versa. The reason is likely due to floating-point precision. Since == operator performs an exact comparison, small differences in representation of numbers may be influence the output. On the other hand, the all.equal compares numeric vectors with some tolerance to differences in floating-point values.

## Task 4

### The game of craps

We will define the separate function to simulate rolling the six-sided die by using sample() function, which will chose a random integer between 1 and 6 inclusively.

Two first cases of rolling 1 (losing) or 6 (winning) are done by control flow if and else if statements. In the third case of rolling 2, 3, 4 or 5 (else statement) the loop does 3 iterations and breaks returning message about the victory if point number was rolled before or on the 3rd iteration.

In the case of failing to roll the point number through all 3 attempts, the message about the loss is returned. Some additional outputs about current attempt and obtained number were added.

```
roll_die <- function() {
  sample(1:6, 1)
}

play_game_of_craps <- function() {
  roll <- roll_die()

  if (roll == 6) {
    return("You got 6 ---> You win!")
  } else if (roll == 1) {
    return("You got 1 ---> You lose!")
  } else {
    point_num <- roll
    cat("You got point number ", point_num,
        ". You have three tries to roll it one more time \n")
    for (i in 1:3) {
      cat("This is try ", i, " ")
      roll <- roll_die()
    }
  }
}
```

```

cat("You got ", roll, "\n")

if (roll == point_num) {
  return(paste("You got the point number of this round",
               point_num, "on the try", i, "---> You win!", sep = " "))
}
}

return(paste("You didn't get the point number of this round",
             point_num, "---> You lose!", sep = " "))
}
}

```

```

game_res <- play_game_of_craps()
print(game_res)

```

```

## You got point number 3 . You have three tries to roll it one more time
## This is try 1 You got 5
## This is try 2 You got 4
## This is try 3 You got 2
## [1] "You didn't get the point number of this round 3 ---> You lose!"

```