Case Study 1 AKSTA Statistical Computing

Hanna Kienast Iulia Mihaela Enache Kateryna Ponomarenko

2024-03-25

1. Ratio of Fibonacci numbers

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
```

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
```

```
)

print(benchmark_200)
```

```
## Unit: microseconds
##        expr    min      lq     mean median      uq     max neval
##    for_loop 26.402 28.7510 33.53448 29.601 31.2010   175.6  1000
##  while_loop 38.101 40.1505 62.69650 42.101 43.6015 17089.4  1000
```

```
print(benchmark_2000)
```

```
## Unit: microseconds
##        expr     min      lq     mean   median       uq      max neval
##    for_loop 244.001 252.351 301.6595 264.1005 308.9515 8206.100  1000
##  while_loop 354.701 369.151 430.1069 399.2510 451.9010 1067.501  1000
```

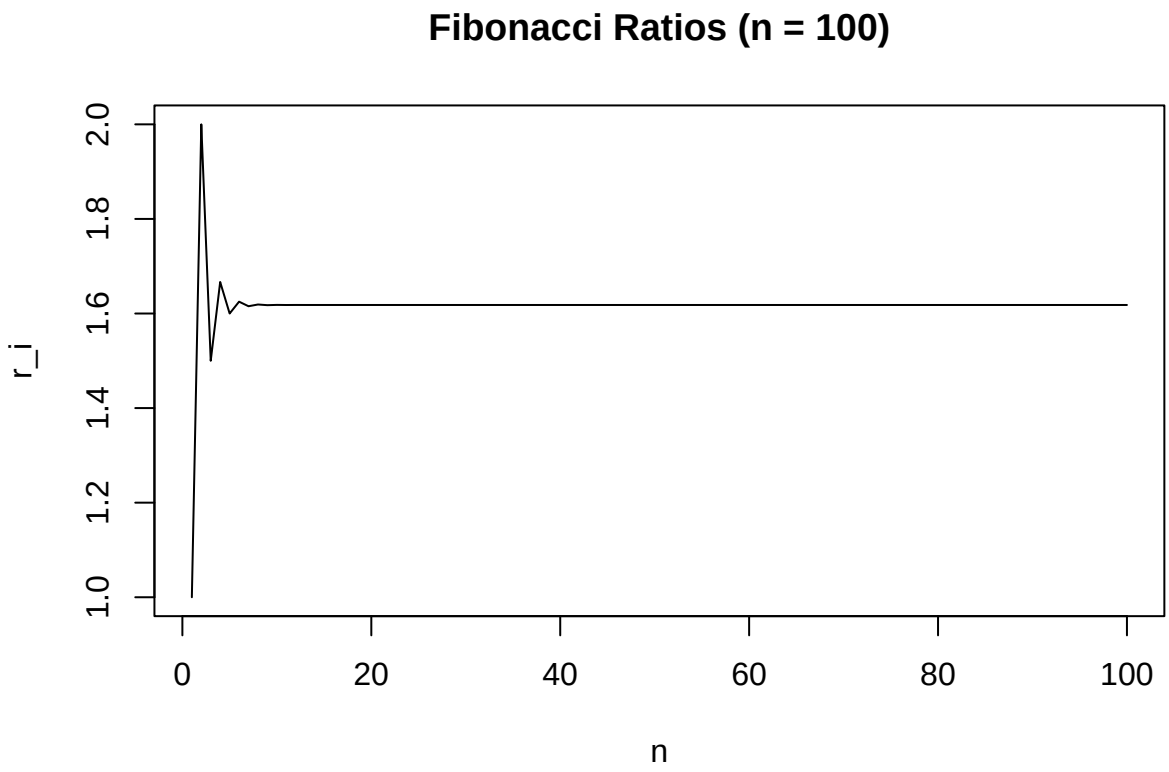
Conclusion: we can summarize that for loop is more efficient, while the mean and median execution time are lower than these indicators for while loop. It may be because in for loop, we don't need to initialize counter variable and update it manually. However, we noticed, that in case n = 2000 the max 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

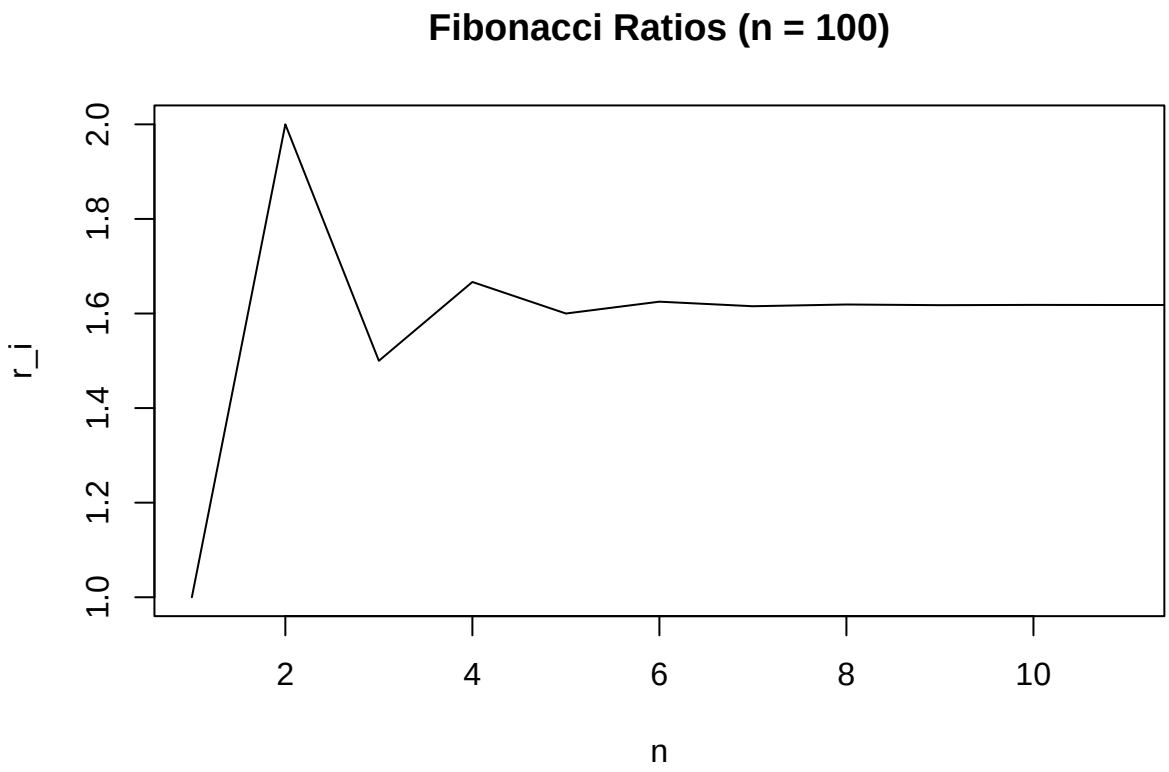
```
# Call the function for n = 100
seq <- for_impl_fib_ratio(100L)

# 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.

2. Gamma Function

a.

Write a function to compute the following for n a positive integer using the gamma function in base R. rho_n = gamma((n-1)/2)/(gamma(1/2)*gamma((n-2)/2))

```
gamma_fct <- function(n) {
  rho_n <- gamma((n-1)/2) / (gamma(1/2)*gamma((n-2)/2))
  return (rho_n)
}
```

b.

Try n = 2000. What do you observe? Why do you think the observed behavior happens?

```
print(gamma_fct(2000))
```

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

We can try for several lower n and see that from around 340 on we only get NaNs. This might happen due to numerical precision issues computing the gamma function for very large values (given gamma(n) =

(n-1)! reaches very high values very quickly). Using the lgamma function might help overcoming this issue. It returns the natural logarithm of the absolute value of the gamma function. We would only need to use the exponential function for the final result.

c.

Write an implementation which can also deal with large values of n > 1000.

```
# Function to compute the gamma function using logarithmic form
lgamma_fct <- function(n) {
  rho_n <- lgamma((n-1)/2) / (lgamma(1/2)*lgamma((n-2)/2))
  return (exp(rho_n))
}

print(lgamma_fct(1000))
```

```
## [1] 5.750145
```

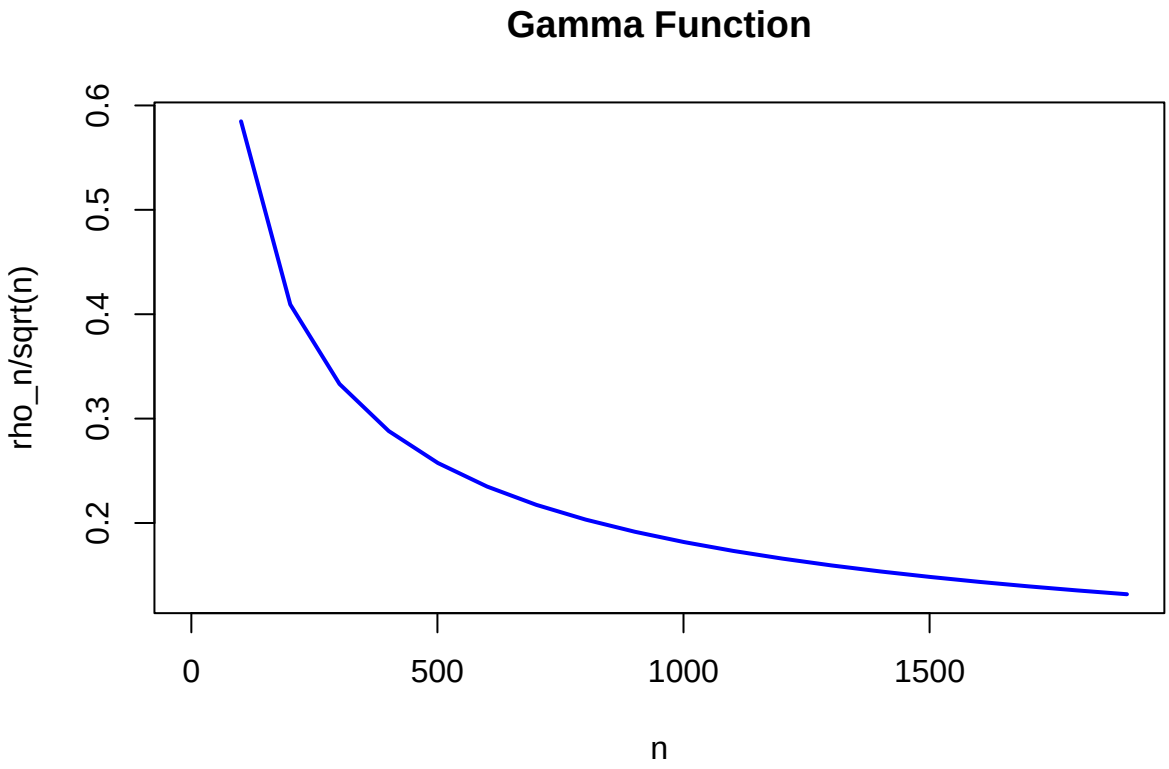
```
print(lgamma_fct(2000))
```

```
## [1] 5.744024
```

d.

Plot rho_n/sqrt(n) for different values of n. Can you guess the limit of rho_n/sqrt(n) for n -> inf?

```
x <- seq(1, 2000, by = 100)
y <- lgamma_fct(x)/sqrt(x)
plot(x, y, type = "l", col = "blue", lwd = 2, main = "Gamma Function", xlab = "n", ylab = "rho_n/sqrt(n)
```



We can see that for large values of x (or n) the function converges to 0.

3. The golden ratio

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 ϕ^1) and n = 1 (calculate ϕ^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 ϕ^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 ϕ

```
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
```

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

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

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

```r
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.

4. 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.

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

5. Readable and efficient code

a.

Explain (in text) what the code does. First, a seed is set (for reproducibility). It generates values for vectors x and y, whereas x are random values from a normal distribution and y is a linear combination of x and an intercept and other random values. Then a dataframe is created out of those. The line "cat(...)" would print the word "Step" and then the number of the step. The next steps are done for subsets of the data: A linear regression model "lm" is fit on one subset of the data. Predictions are computed for y values (using another subset of the data). The RMSE value between the true y values and the predictions is computed and stored in the vector r. After all is done for the 4 subsets the vector r with all RMSE values is returned.

```
set.seed(1)
x <- rnorm(1000)
y <- 2 + x + rnorm(1000)
df <- data.frame(x, y)

cat("Step", 1, "\n")
```

```
## Step 1
```

```
fit1 <- lm(y ~ x, data = df[-(1:250),])
p1 <- predict(fit1, newdata = df[(1:250),])
r <- sqrt(mean((p1 - df[(1:250),"y"])^2))

cat("Step", 2, "\n")
```

```
## Step 2
```

```
fit2 <- lm(y ~ x, data = df[-(251:500),])
p2 <- predict(fit2, newdata = df[(251:500),])
r <- c(r, sqrt(mean((p2 - df[(251:500),"y"])^2)))

cat("Step", 3, "\n")
```

```
## Step 3
```

```
fit3 <- lm(y ~ x, data = df[-(501:750),])
p3 <- predict(fit3, newdata = df[(501:750),])
r <- c(r, sqrt(mean((p3 - df[(501:750),"y"])^2)))

cat("Step", 4, "\n")
```

```
## Step 4
```

```
fit4 <- lm(y ~ x, data = df[-(751:1000),])
p4 <- predict(fit4, newdata = df[(751:1000),])
r <- c(r, sqrt(mean((p4 - df[(751:1000), "y"])^2)))

r
```

```
## [1] 1.0249956 0.9952113 1.0685500 1.0707264
```

b.

Explain (in text) what you would change to make the code more readable. I would put everything in a loop, given that we do the same thing 4 times with only different parts of the dataset. A for-loop is suitable, as we know the number of iterations.

c.

Change the code according to a. and wrap it in a function. This function should have at most 10 lines (without adding commands to more lines such as x <- 1; y <- 2. Such commands will count as 2 lines!). Check that the function called on the same input outputs the same as the provided code.

```
rmse_lm <- function(x,y) {
  df <- data.frame(x,y)
  r <- numeric()
  for (i in seq(1, 1000, by = 250)) {
    fit <- lm(y ~ x, data = df[-(i:(i+249)), ])
    p <- predict(fit, newdata = df[(i:(i+249)), ])
    r <- c(r, sqrt(mean((p-df[(i:(i+249)), "y"])^2)))
  }
  return(r)
}

set.seed(1)
x <- rnorm(1000)
y <- 2 + x + rnorm(1000)
computed_r <- rmse_lm(x, y)
computed_r
```

```
## [1] 1.0249956 0.9952113 1.0685500 1.0707264
```

6. Measuring and improving performance

```
kwtest <- function (x, g, ...)
{
  if (is.list(x)) {
    if (length(x) < 2L)
      stop("'x' must be a list with at least 2 elements")
    if (!missing(g))
      warning("'x' is a list, so ignoring argument 'g'")
    if (!all(sapply(x, is.numeric)))
      warning("some elements of 'x' are not numeric and will be coerced to numeric")
    k <- length(x)
    l <- lengths(x)
    if (any(l == 0L))
      stop("all groups must contain data")
    g <- factor(rep.int(seq_len(k), l))
    x <- unlist(x)
  }
  else {
    if (length(x) != length(g))
      stop("'x' and 'g' must have the same length")
    g <- factor(g)
    k <- nlevels(g)
    if (k < 2L)
      stop("all observations are in the same group")
  }
  n <- length(x)
  if (n < 2L)
    stop("not enough observations")
  r <- rank(x)
  TIES <- table(x)
  STATISTIC <- sum(tapply(r, g, sum)^2/tapply(r, g, length))
  STATISTIC <- ((12 * STATISTIC/(n * (n + 1)) - 3 * (n + 1))/(1 -
    sum(TIES^3 - TIES)/(n^3 - n)))
  PARAMETER <- k - 1L
  PVAL <- pchisq(STATISTIC, PARAMETER, lower.tail = FALSE)
  names(STATISTIC) <- "Kruskal-Wallis chi-squared"
  names(PARAMETER) <- "df"
  RVAL <- list(statistic = STATISTIC, parameter = PARAMETER,
    p.value = PVAL, method = "Kruskal-Wallis rank sum test")
  return(RVAL)
}
```

a.

```
kwtest(x, g, ...):
    if x is a list:
        if x has less than 2 elements, show error
```

```
        if g is provided, ignore it
        if some elements of x are not numeric, warn
        Convert non-numeric elements of x to numeric
        Calculate the number of groups (k) as the length of x
        Calculate the lengths of each group (l) as the lengths of elements in x
        if any of the groups has no data, stop with an error message
        Create a factor (g) with group labels for each observation in x
        Unlist x into a single numeric vector
    else:
        if x and g do not have the same lengths, show error
        Convert g into a factor
        Calculate the number of groups (k) as the number of levels in g
        if the number of groups is less than 2, show error

    Calculate the total number of observations (n)
    Compute the ranks of the observations (r)
    Count the number of ties (TIES)
    Calculate the Kruskal-Wallis test statistic
        Calculate the sum of squared ranks for each group.
        STATISTIC = (12 * sum(group_sums^2) / (n * (n + 1)) - 3 * (n + 1)) / (1 - sum(TIES^3 - TIES) /
    Calculate the degrees of freedom (PARAMETER) for the test
    Calculate the p-value (PVAL) using the chi-squared distribution
    Create a list (RVAL) containing the following test results
       (statistic: Kruskal-Wallis chi-squared value
        parameter: Degrees of freedom
        p.value: P-value
        method: "Kruskal-Wallis rank sum test")

    Return the list (RVAL)
```

b.

```
x_list <- list(
  group1 = c(10, 15, 20),
  group2 = c(8, 12, 16, 18),
  group3 = c(5, 7, 9, 11, 14)
)

x_vector <- c(10, 15, 20, 8, 12, 16, 18, 5, 7, 9, 11, 14)
g <- factor(rep(1:3, c(3, 4, 5)))

result_list <- kwtest(x_list)
result_vector <- kwtest(x_vector, g)

identical(result_list, result_vector)
```

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

c.

We firstly do some checks on the input data: x and g. X has to be numeric and g can or not be a factor (we are converting it anyways into a factor to be sure. Then we do further checking on their lengths. We are simplifying the test statistic formula by calculating it following the mathematical definition. The computations of observations' ranks and their sum for each group stay the same. The test statistics important points: - 12 / (n * (n + 1)) normalizes test statistic for different sample sizes - sum(group_sums^2 / table(g)) calculates the sum of squared ranks for each group and sums them across all. - sum((TIES^3 - TIES) / (n^3 - n)) is used for correction of ties - ...- 3 * (n + 1) * (1 - ...) accounts for the sample size and corrects for ties

```
kwtest_fast <- function(x, g) {
  if (!is.numeric(x)) {
    stop("Input 'x' must be numeric")
  }

  g <- factor(g)

  if (length(x) != length(g)) {
    stop("Input 'x' and 'g' must have the same length.")
  }

  if (nlevels(g) < 2L)
    stop("all observations are in the same group")

  n <- length(x)
  if (n < 2L)
    stop("not enough observations")

  r <- rank(x)

  group_sums <- tapply(r, g, sum)

  TIES <- table(x)

  STATISTIC <- (12 / (n * (n + 1))) * sum(group_sums^2 / table(g))
  STATISTIC <- STATISTIC - 3 * (n + 1) * (1 - sum((TIES^3 - TIES) / (n^3 - n)))

  PARAMETER <- length(unique(g)) - 1L

  PVAL <- pchisq(STATISTIC, PARAMETER, lower.tail = FALSE)

  return(list(statistic = STATISTIC, parameter = PARAMETER,
              p.value = PVAL, method = "Kruskal-Wallis rank sum test"))
}
```

d.

We iterate over each row, where each represent an experiment, in the matrix X. We perform the test using the stats:::kruskal.test.default for each row and store its test statistic. These statistics are collected into a vector, where each element corresponds to the test statistic for the respective row index.

```r
kruskal_wallis_test <- function(X, g) {
  test_statistics <- numeric(length = nrow(X))

  for (i in 1:nrow(X)) {

    result <- stats:::kruskal.test.default(X[i, ], g = factor(g))

    test_statistics[i] <- result$statistic
  }

  return(test_statistics)
}
```

```r
set.seed(1234)
m <- 1000
n <- 50
X <- matrix(rt(m * n, df = 10), nrow = m)
grp <- rep(1:3, c(20, 20, 10))

test_statistics <- kruskal_wallis_test(X, grp)
length(test_statistics)
```

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

e.

Here we do the same approach as in the previous point. The difference is that we use our own function kwtest_fast instead of the stats:::kruskal.test.default one.

```r
kruskal_wallis_test_fast <- function(X, g) {
  test_statistics <- numeric(length = nrow(X))

  for (i in 1:nrow(X)) {

    result <- kwtest_fast(X[i, ], g = factor(g))

    test_statistics[i] <- result$statistic
  }

  return(test_statistics)
}


test_statistics_fast <- kruskal_wallis_test_fast(X, grp)
length(test_statistics_fast)
```

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

f.

Based on these metrics, the first technique performs better in terms of execution time and memory allocation than the second function. However, we can see that the second approach demonstrates lower garbage

collection rates, suggesting more efficient memory management.

```
library(bench)
```

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

```
f1 <- function() kruskal_wallis_test(X, grp)
f2 <- function() kruskal_wallis_test_fast(X, grp)

bench_results <- mark(
  kruskal_wallis = f1(),
  kwtest_fast = f2(),
  check = FALSE
)
```

```
## Warning: Some expressions had a GC in every iteration; so filtering is
## disabled.
```

```
print(bench_results)
```

```
## # A tibble: 2 x 13
##   expression      min median `itr/sec` mem_alloc `gc/sec` n_itr  n_gc total_time
##   <bch:expr>    <bch> <bch:>     <dbl> <bch:byt>    <dbl> <int> <dbl>   <bch:tm>
## 1 kruskal_wall~ 583ms  583ms      1.72    21.3MB     8.58     1     5      583ms
## 2 kwtest_fast   528ms  528ms      1.89    20.7MB     9.46     1     5      528ms
## # i 4 more variables: result <list>, memory <list>, time <list>, gc <list>
```

g.

The function kwtest_fast from point c.seems to be already efficient without needing further vectorization. The function steps operate on vectors and factors and utilize vectorized functions such as tapply, sum, table, rank, without using any unnecessary iterations. Therefore, it manages to maintain readability and good performance.