# Case Study 1

AKSTA Statistical Computing

Tatzberger Jonas, Rasser Thomas, Grübling Alexander

03.04.2024

## Exercises

### 1. Ratio of Fibonacci numbers

#### a.

At first the function calculating the ratio using a for loop:

```r
r_for <- function(i) {
  f_i <- 0
  f_ip1 <- 1
  result <- c()
  for (x in 1:i) {
    f_ip1_new <- f_i+f_ip1
    f_i <- f_ip1
    f_ip1 <- f_ip1_new
    result <- append(result, f_ip1/f_i)
  }
  return (result)
}
```

Running the function using a test output:

```r
test_for <- r_for(10)
print(test_for)
```

```
##  [1] 1.000000 2.000000 1.500000 1.666667 1.600000 1.625000 1.615385 1.619048
##  [9] 1.617647 1.618182
```

Next the function using a while-loop:

```r
r_while <- function(i) {
  f_i <- 0
  f_ip1 <- 1
  result <- c()
  x=0
  while(x < i) {
    f_ip1_new <- f_i+f_ip1
    f_i <- f_ip1
    f_ip1 <- f_ip1_new
    result <- append(result, f_ip1/f_i)
    x = x+1
  }
  return (result)
}
```

Running the function using a test output:

```
test_while <- r_while(10)
print(test_while)
```

```
##  [1] 1.000000 2.000000 1.500000 1.666667 1.600000 1.625000 1.615385 1.619048
##  [9] 1.617647 1.618182
```

As we can see, both functions produce the same outputs, which seem to nicely converge towards the golden ratio.

### b.

To test the functions, the microbenchmark package is used. When applying the function, the standard values are used (the number of iterations for the expression is set to 100 here). This results in a statistic comprised of n=100 tests.

```
library(microbenchmark)
```

```
## Warning: Paket 'microbenchmark' wurde unter R Version 4.3.3 erstellt
```

## Testing for n=200:

```
microbenchmark(r_for(200))
```

```
## Unit: microseconds
##        expr   min    lq    mean median    uq    max neval
##  r_for(200) 337.9 440.3 538.075 512.95 599.8 1164.7   100
```

```
microbenchmark(r_while(200))
```

```
## Unit: microseconds
##          expr   min     lq    mean median    uq   max neval
##  r_while(200) 327.6 359.65 481.623  413.6 542.3 998.2   100
```

Here we can see that the mean time that the "r_for" function took (397.028us) is lower than the "r_while" function (423.145us). For n=200 the for loop is faster.

## Testing for n=2000:

```
microbenchmark(r_for(2000))
```

```
## Unit: milliseconds
##         expr    min       lq     mean median       uq     max neval
##  r_for(2000) 6.5518 14.39535 18.69564 16.277 25.05095 32.1693   100
```

```
microbenchmark(r_while(2000))
```

```
## Unit: milliseconds
##           expr    min      lq     mean  median       uq     max neval
##  r_while(2000) 6.4021 13.1451 15.64983 15.9107 18.13415 35.2585   100
```

Again we can see that the mean time that the "r_for" function took (18.295ms) is lower than the "r_while" function (20.847ms). For n=2000 also the for-loop is faster.

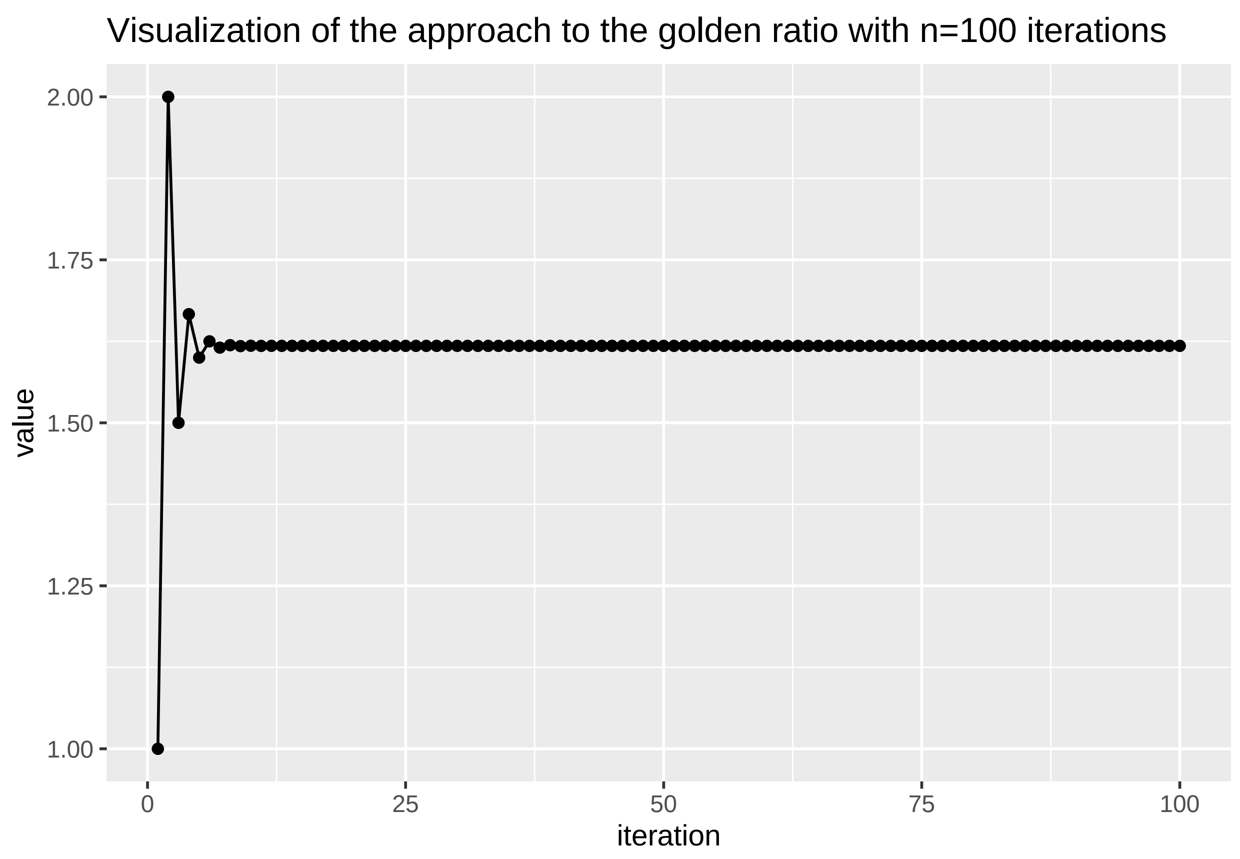
**c.**

In order to visualize the function, the package "ggplot2" is used. To work with the data it is transformed into a data-frame and an iteration column is added.

```
library(ggplot2)
```

```
## Warning: Paket 'ggplot2' wurde unter R Version 4.3.2 erstellt
```

```
value = r_for(100)
r_100 <- data.frame(value)
r_100$iteration <- 1:nrow(r_100)
ggplot(r_100, aes(x=iteration, y=value)) + geom_point() + geom_line() + ggtitle("Visualization of the ap
```



From the plot, one can deduce that the function converges at around the 6th iteration, from this point on no significant changes are visible.

## 2. Gamma function

### a.

Write a function to compute the following for $n$ a positive integer using the gamma function in base R.

$$\rho_n = \frac{\Gamma((n-1)/2)}{\Gamma(1/2)\Gamma((n-2)/2)}$$

**Answer**

```r
rho <- function(n) {
  return (
    gamma((n-1)/2) / (gamma(1/2) * gamma((n-2)/2))
  )
}
```

### b.

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

**Answer**

```r
testcases <- c(1, 2, 3, 4, 5, 10, 50, 100, 500, 1000, 2000)
for (case in testcases) {
  print(paste("Rho(", case, ") = ", rho(case), sep=""))
}
```

```
## Warning in gamma((n - 1)/2): NaNs wurden erzeugt

## [1] "Rho(1) = NaN"

## Warning in gamma((n - 2)/2): NaNs wurden erzeugt

## [1] "Rho(2) = NaN"
## [1] "Rho(3) = 0.318309886183791"
## [1] "Rho(4) = 0.5"
## [1] "Rho(5) = 0.636619772367581"
## [1] "Rho(10) = 1.09375"
## [1] "Rho(50) = 2.74959606512371"
## [1] "Rho(100) = 3.93926528482005"
## [1] "Rho(500) = NaN"
## [1] "Rho(1000) = NaN"
## [1] "Rho(2000) = NaN"
```

This function does not work for 1 and 2, since gamma returns $NaN$ for values less or equal to 0

```r
testcases <- 100:500
for (case in testcases) {
  if (is.nan(rho(case))) {
    print(paste("Rho breaks at ", case, sep=""))
    break
  }
}
```

```
## [1] "Rho breaks at 346"
```

If the number gets to big, the inbuilt our functions breaks, since it relies on the inbuilt gamma function, which returns $Inf$ if the input is to big, which breaks our division

```r
testcases <- 1:500
for (case in testcases) {
  if (is.infinite(gamma(case))) {
    print(paste("Gamma returns Inf at ", case, sep=""))
    break
  }
}
```

```
## [1] "Gamma returns Inf at 172"
```

**c.**

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

**Answer**

The given function can be rewritten using logarithms as:

$$\log\left(\frac{\Gamma\left(\frac{n-1}{2}\right)}{\Gamma\left(\frac{1}{2}\right)\cdot\Gamma\left(\frac{n-2}{2}\right)}\right) = \log\Gamma\left(\frac{n-1}{2}\right) - \left(\log\Gamma\left(\frac{1}{2}\right) + \log\Gamma\left(\frac{n-2}{2}\right)\right)$$

which turns the multiplication into additions.

```
rho_log <- function(n) {
  if (n <= 5) {
    return (gamma((n-1)/2) / (gamma(1/2) * gamma((n-2)/2)))
  } else {
    log_gamma_expr <- lgamma((n - 1) / 2) - (lgamma(1 / 2) + lgamma((n - 2) / 2))
    return (exp(log_gamma_expr))
  }
}
```

**d.**

Plot $\rho_n/\sqrt{n}$ for different values of $n$. Can you guess the limit of $\rho_n/\sqrt{n}$ for $n \to \infty$ ?

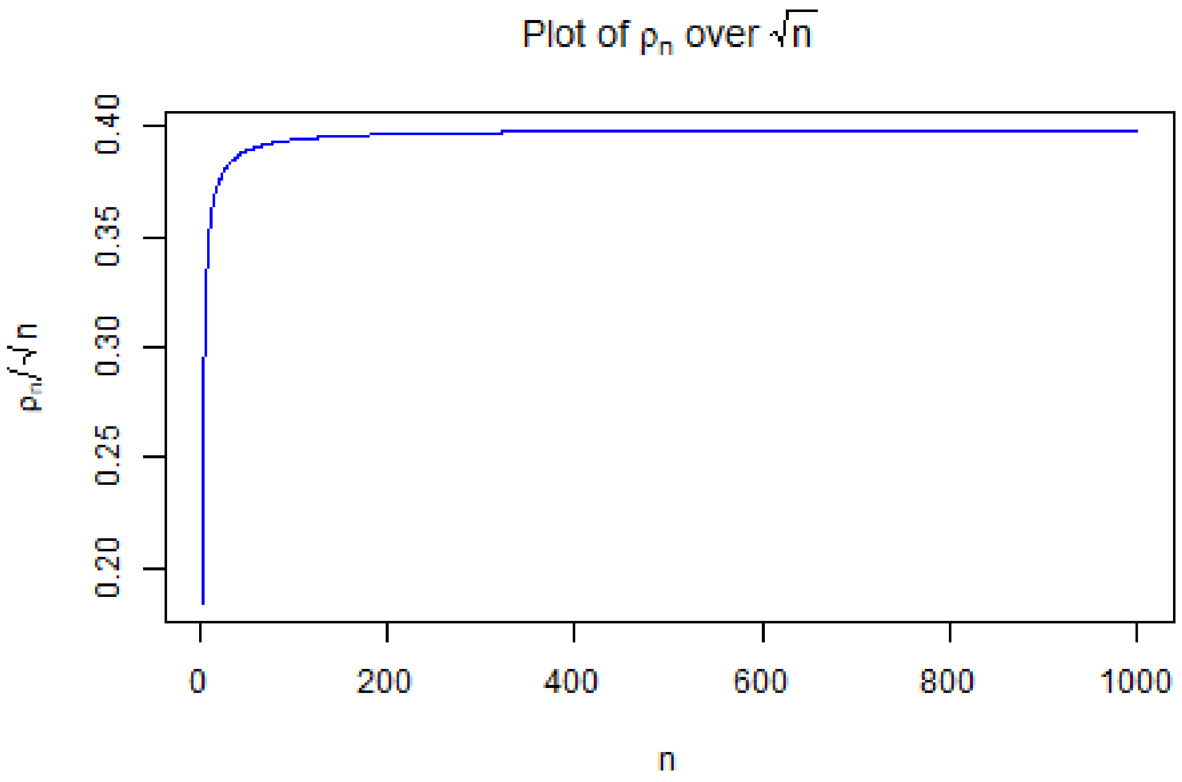
**Answer**

```
rhoOverSqrtN <- function(n) {
  if (!is.numeric(n)) {
    stop("n must be numeric.")
  }

  return (rho_log(n)/sqrt(n))
}

x <- 1:1000
y <- sapply(x, rhoOverSqrtN)
```

```
## Warning in gamma((n - 1)/2): NaNs wurden erzeugt
```

```
## Warning in gamma((n - 2)/2): NaNs wurden erzeugt
```

```
# Filter out invalid values (NA, NaN, Inf)
valid_indices <- !is.na(y) & !is.nan(y) & !is.infinite(y)
x <- x[valid_indices]
y <- y[valid_indices]

plot(x, y, type = 'l', col = 'blue', xlab = "n", ylab = expression(rho[n] / sqrt(n)), main=expression("
```



```r
convergence <- tail(y)
for (c in convergence) {
  print(c)
}
```

```
## [1] 0.3984408
## [1] 0.3984413
## [1] 0.3984418
## [1] 0.3984423
## [1] 0.3984428
## [1] 0.3984433
```

The function approaches 3.984, which is approximately $\frac{1}{\sqrt{2\pi}}$

```r
print(paste("Approx. limit:   ", y[length(y)-1]))
```

```
## [1] "Approx. limit:    0.398442815762513"
```

```r
expr <- expression(1/sqrt(2*pi))
value <- eval(expr)
print(paste(expr, ": ", value))
```

```
## [1] "1/sqrt(2 * pi) :  0.398942280401433"
```

## 3. The golden ratio

Two positive numbers $x$ and $y$ with $x > y$ are said to be in the golden ratio if the ratio between the larger number and the smaller number is the same as the ratio between their sum and the larger number:

$$\frac{x}{y} = \frac{x+y}{x}$$

The golden ratio $\Phi = x/y$ can be computed as a solution to $\Phi^2 - \Phi - 1 = 0$ and is

$$\Phi = \frac{\sqrt{5}+1}{2}$$

and it satisfies the Fibonacci-like relationship:

$$\Phi^{n+1} = \Phi^n + \Phi^{n-1}$$

### a.

Write an R function which computes $\Phi^{n+1}$ using the recursion above (go up to $n = 1000$ ) .

```r
phi_recursive <- function(n) {
  phi_0 <- (sqrt(5) - 1) / 2  # This is not used directly but initialized for completeness
  phi_1 <- (sqrt(5) + 1) / 2  # This is Phi
  phi_2 <- phi_1 + 1          # This is Phi^2, which equals Phi + 1 due to the quadratic equation

  # Base cases
  if (n == 0) {
    return(phi_1)
  } else if (n == 1) {
    return(phi_2)
  }

  # Iteratively compute Phi^n+1 for n > 1
  for (i in 2:n) {
    # Update the sequence values
    temp <- phi_2 + phi_1
    phi_1 <- phi_2
    phi_2 <- temp
  }

  return(phi_2)
}

testcases <- seq(0,4)
for (case in testcases) {
  print(paste("Phi^", case+1, ": ", phi_recursive(case), sep=""))
}
```

```
## [1] "Phi^1: 1.61803398874989"
## [1] "Phi^2: 2.61803398874989"
## [1] "Phi^3: 4.23606797749979"
## [1] "Phi^4: 6.85410196624968"
## [1] "Phi^5: 11.0901699437495"
```

```
print(paste("Phi^1000: ", phi_recursive(999), sep=""))
```

```
## [1] "Phi^1000: 9.71941777359082e+208"
```

**b.**

Write a function which computes this $\Phi^{n+1}$ by simply using the power operator ^.

```
phi_power <- function(n) {
  return (((sqrt(5) + 1) / 2)^(n+1))
}

testcases <- seq(0,4)
for (case in testcases) {
  print(paste("Phi^", case+1, ": ", phi_power(case), sep=""))
}
```

```
## [1] "Phi^1: 1.61803398874989"
## [1] "Phi^2: 2.61803398874989"
## [1] "Phi^3: 4.23606797749979"
## [1] "Phi^4: 6.85410196624969"
## [1] "Phi^5: 11.0901699437495"
```

```
print(paste("Phi^1000: ", phi_power(999), sep=""))
```

```
## [1] "Phi^1000: 9.71941777359114e+208"
```

**c.**

Use once == and once all. equal to compare $\Phi^{n+1}$ obtained in a. vs the one obtained in b. for $n = 12, 60, 120, 300$. What do you observe? If there are any differences, what can be the reason?

```
n_values <- c(12, 60, 120, 300)
comparison_results <- sapply(n_values, function(n) {
  recursive_computed <- phi_recursive(n)
  power_computed <- phi_power(n)

  # Using == for exact comparison (might not always be true due to floating-point precision)
  exact_match <- recursive_computed == power_computed

  # Using all.equal for a tolerance-based comparison
  tolerant_match <- all.equal(recursive_computed, power_computed)

  list(ExactMatch = exact_match, TolerantMatch = tolerant_match)
})

print(comparison_results)
```

```
##               [,1]  [,2]  [,3]  [,4]
## ExactMatch    FALSE FALSE FALSE FALSE
## TolerantMatch TRUE  TRUE  TRUE  TRUE
```

As written in the R documentation, the *all.equal* function uses a tolerance:

"Differences smaller than tolerance are not reported. The default value is close to 1.5e-8"

## 4. Game of craps

### Code of the game + explaination (as comments in the code):

```r
play_craps <- function() {
  # at first a random number between 1 & 6.99 is selected, using the floor-
  # function, an random integer between 1 & 6 is created
  first_throw <- floor(runif(1, min=1, max=6.99))
  # using a if/else if/else-query, it is determined if the number results in an
  # instant win/lose or if the game need to be played further
  if (first_throw == 6) {
    print("WoW a six! You won the game!")
  }
  else if (first_throw == 1) {
    print("Unlucky, a 1! You lose, try again!")
  }
  else {
    cat("Your number is", first_throw, "- try to throw it again within 3 tries!\n")
    # A for-loop is now used to simulate the next throws
    for (i in 1:3) {
      next_throw <- floor(runif(1, min=1, max=6.99))
      cat("Throwing dice... you got a", next_throw, "\n")
      # if the throw is identical to the first one, the game is won & the for-
      # loop is ended
      if (next_throw == first_throw) {
        cat("Nice! You managed to repeat your number on try No.", i, "- You win!")
        break
      }
      # if the throw is not identical to the first one, the game continues
      else if (i != 3) {
        cat("Not quite the same, you have", 3-i, "tries left.\n")
      }
    }
    # if the for loop is broken and the game has not been won, the game is lost &
    # an according message is displayed
    if (next_throw != first_throw) {
      print("Unlucky, you could not repeat your number within 3 throws. You lose, try again!")
    }
  }
}
```

## Using the function with the different possible outcomes:

### a) Getting a 6 and winning instantly:

```
## [1] "WoW a six! You won the game!"
```

### b) Getting a 1 and losing instantly:

```
## [1] "Unlucky, a 1! You lose, try again!"
```

### c) Playing the full game and winning:

```
## Your number is 2 - try to throw it again within 3 tries!
## Throwing dice... you got a 5
## Not quite the same, you have 2 tries left.
## Throwing dice... you got a 4
## Not quite the same, you have 1 tries left.
## Throwing dice... you got a 2
## Nice! You managed to repeat your number on try No. 3 - You win!
```

### d) Playing the full game and losing:

```
## Your number is 2 - try to throw it again within 3 tries!
## Throwing dice... you got a 3
## Not quite the same, you have 2 tries left.
## Throwing dice... you got a 4
## Not quite the same, you have 1 tries left.
## Throwing dice... you got a 6
## [1] "Unlucky, you could not repeat your number within 3 throws. You lose, try again!"
```

## 5. Readable and efficient code

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

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

```
## Step 1
```

```r
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
```

```r
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
```

```r
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
```

```r
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
```

**a.**

This code first takes samples of two normally distributed random variables.Then, multiple linear models are calculated, based on different portions of the input data. Afterwards, those models are used to predict that portion of the data, that has not been used to train the models. Finally, the mean squared error is appended to a list and printed out.

**b.**

While achieving its purpose, the code is not structured in an optimal way. First, there is a lot of repetition, which violates the DRY (Don't Repeat Yourself) Principle. To mitigate this, I would suggest refactoring by putting repeated code segments in a dedicated function. Then, the variable representing the mean squared errors is named "r". While this might not be an explicit problem, I would prefer to rename this to a more clear name.

**c.**

```r
getLinearModelMeanSquaredError <- function(data, testingRange) {
  fit <- lm(y ~ x, data = data[-testingRange,])
  prediction <- predict(fit, newdata = data[testingRange,])
  return(sqrt(mean((prediction - data[testingRange, "y"])^2)))
}

set.seed(1)
x <- rnorm(1000)
```

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

indices <- list(c(1:250), c(251:500), c(501:750), c(751:1000))
errors <- unlist(lapply(indices, function(i) getLinearModelMeanSquaredError(df, i)))
print(errors)
```

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

## 6. Measuring and improving performance

Have a look at the code of the function below. It is a function for performing a Kruskal Wallis test, a robust non-parametric method for testing whether samples come from the same distribution. (Note: we assume no missing values are present in x).

```
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.**

Write a pseudo code outlining what the function does.

```
Function kwtest(x, g, ...)
    If x is a list Then
        If length of x is less than 2 Then
            Stop with error "'x' must be a list with at least 2 elements"
        End If
        If g is not missing Then
            Issue warning "'x' is a list, so ignoring argument 'g'"
        End If
        If not all elements of x are numeric Then
            Issue warning "some elements of 'x' are not numeric and will be coerced to numeric"
        End If
        Set k to length of x
        Set l to lengths of x
        If any element of l is 0 Then
            Stop with error "all groups must contain data"
        End If
        Set g to factor with levels as sequence from 1 to k, repeated l times
        Unlist x
    Else
        If length of x is not equal to length of g Then
            Stop with error "'x' and 'g' must have the same length"
        End If
        Set g to factor of g
        Set k to number of levels in g
        If k is less than 2 Then
            Stop with error "all observations are in the same group"
        End If
    End If
    Set n to length of x
    If n is less than 2 Then
        Stop with error "not enough observations"
    End If
    Rank x and store in r
    Create a table of frequencies of x and store in TIES
    Calculate STATISTIC as sum of squares of ranks in each group divided by the number of ranks in each
    Adjust STATISTIC based on the formula provided
    Set PARAMETER to k - 1
    Calculate PVAL as the p-value of the chi-squared distribution with PARAMETER degrees of freedom and
    Set names of STATISTIC and PARAMETER to "Kruskal-Wallis chi-squared" and "df" respectively
    Create a list RVAL with statistic, parameter, p.value, and method
    Return RVAL
End Function
```

**b.**

For example data, call the function in two ways: once using x as a list and once using x as a vector with a corresponding g argument. Ensure that the two different function calls return the same thing by aligning the inputs.

```r
data("PlantGrowth")
group1 <- PlantGrowth$weight[PlantGrowth$group == "ctrl"]
group2 <- PlantGrowth$weight[PlantGrowth$group == "trt1"]
group3 <- PlantGrowth$weight[PlantGrowth$group == "trt2"]

result_list <- kwtest(list(group1, group2, group3))
print(result_list)
```

```
## $statistic
## Kruskal-Wallis chi-squared
##                   7.988229
##
## $parameter
## df
##  2
##
## $p.value
## [1] 0.01842376
##
## $method
## [1] "Kruskal-Wallis rank sum test"
```

```r
x_vector <- PlantGrowth$weight
g_vector <- factor(PlantGrowth$group)

result_vector <- kwtest(x_vector, g_vector)
print(result_vector)
```

```
## $statistic
## Kruskal-Wallis chi-squared
##                   7.988229
##
## $parameter
## df
##  2
##
## $p.value
## [1] 0.01842376
##
## $method
## [1] "Kruskal-Wallis rank sum test"
```

**c.**

Make a faster version of kwtest() that only computes the Kruskal-Wallis test statistic when the input is a numeric variable x and a variable g which gives the group membership. You can try simplifying the function above or by coding from the mathematical definition (see https://en.wikipedia.org/wiki/Kruskal%E2%80%93Wallis_one-way_analysis_of_variance). This function should also perform some checks to ensure the correctness of the inputs (use kwtest() as inspiration).

```r
kwtest_fast <- function(x, g) {
  if (!is.numeric(x) || !is.numeric(g) || length(x) != length(g)) {
    stop("'x' and 'g' must be numeric vectors of the same length")
  }
  if (length(unique(g)) < 2) {
    stop("At least two unique groups are required for Kruskal-Wallis test")
  }
  ranks <- rank(x)
  group_sums <- tapply(ranks, g, sum)
  group_sizes <- tapply(ranks, g, length)

  n <- length(x)
  k <- length(unique(g))
  H <- (12 / (n * (n + 1))) * sum(group_sums^2 / group_sizes) - 3 * (n + 1)

  df <- k - 1

  p_value <- pchisq(H, df, lower.tail = FALSE)

  result <- list(
    statistic = H,
    parameter = df,
    p.value = p_value,
    method = "Kruskal-Wallis rank sum test"
  )

  return(result)
}
```

**d.**

Consider the following scenario. You have samples available from multiple experiments m = 1000 where you collect the numerical values for the quantity of interest x and the group membership for n individuals. The first 20 individuals in each sample belong to group 1, the following 20 individuals in each sample belong to group 2, the last 10 individuals in each sample belong to group 3. Use the following code to simulate such a data structure:

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

Write a function which performs the Kruskal-Wallis test using the function stats:::kruskal.test.default() for m repeated experiments and returns a vector of m test statistics. The input of this function are a matrix X with m rows and n columns and a vector g which gives the grouping.

```r
perform_kruskal_wallis_tests <- function(X, g, fun) {
  split_X <- split(X, seq_len(nrow(X)))

  test_results <- lapply(split_X, function(experiment_data) {
    fun(experiment_data, g)
  })

  test_statistics <- sapply(test_results, function(result) result$statistic)
  return(unname(test_statistics))
}

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

test_statistics <- perform_kruskal_wallis_tests(X, grp, stats:::kruskal.test.default)
head(test_statistics)
```

```
## [1] 3.8463529 0.5463529 3.3352941 1.8783529 1.5087059 2.6771765
```

### e.

Write a function which performs the Kruskal-Wallis test using the function in point c. for m repeated experiments and returns a vector of m test statistics. The input of this function are a matrix X with m rows and n columns and a vector g which gives the grouping.

```r
test_statistics <- perform_kruskal_wallis_tests(X, grp, kwtest_fast)
head(test_statistics)
```

```
## [1] 3.8463529 0.5463529 3.3352941 1.8783529 1.5087059 2.6771765
```

### f.

Compare the performance of the two approaches using a benchmarking package on the data generated above. Comment on the results.

```r
library(bench)
```

```
## Warning: Paket 'bench' wurde unter R Version 4.3.3 erstellt
```

```r
set.seed(1234)
m <- 1000 # number of repetitions
n <- 50 # number of individuals
X <- matrix(rt(m * n, df = 10), nrow = m)
grp <- rep(1:3, c(20, 20, 10))
mark(perform_kruskal_wallis_tests(X, grp, stats:::kruskal.test.default), perform_kruskal_wallis_tests(X
```

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

```
## # A tibble: 2 x 6
##   expression                                 min median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr>                               <bch> <bch:>     <dbl> <bch:byt>    <dbl>
## 1 perform_kruskal_wallis_tests(X, grp~ 591ms  591ms      1.69   19.82MB     6.77
## 2 perform_kruskal_wallis_tests(X, grp~ 166ms  173ms      4.91    8.58MB     8.18
```

The results of the benchmarking show a clear picture: The kwtest_fast implementation outperforms the original implementation in almost every metric. The median execution time is down to less than 50%, and also the allocated memory is dramatically lower.

### g.

Now consider vectorizing the function in point c. Compare this approach to the other two. Comment on the results.

```r
kwtest_vectorized <- function(x, g) {
  if (!is.numeric(x) || !is.numeric(g) || length(x) != length(g)) {
    stop("'x' and 'g' must be numeric vectors of the same length")
  }
  if (length(unique(g)) < 2) {
    stop("At least two unique groups are required for Kruskal-Wallis test")
  }
  ranks <- rank(x)

  n <- length(x)
  k <- length(unique(g))
  H <- (12 / (n * (n + 1))) * sum(tapply(ranks, g, sum)^2 / tapply(ranks, g, length)) - 3 * (n + 1)

  df <- k - 1

  p_value <- pchisq(H, df, lower.tail = FALSE)

  result <- list(
    statistic = H,
    parameter = df,
    p.value = p_value,
    method = "Kruskal-Wallis rank sum test"
  )

  return(result)
}

library(bench)
set.seed(1234)
m <- 1000 # number of repetitions
n <- 50 # number of individuals
X <- matrix(rt(m * n, df = 10), nrow = m)
grp <- rep(1:3, c(20, 20, 10))
mark(
  perform_kruskal_wallis_tests(X, grp, stats:::kruskal.test.default),
  perform_kruskal_wallis_tests(X, grp, kwtest_fast),
  perform_kruskal_wallis_tests(X, grp, kwtest_vectorized)
  )
```

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

```
## # A tibble: 3 x 6
##   expression                              min median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr>                           <bch> <bch:>     <dbl> <bch:byt>    <dbl>
## 1 perform_kruskal_wallis_tests(X, grp~ 619ms  619ms      1.62   19.82MB     8.08
## 2 perform_kruskal_wallis_tests(X, grp~ 176ms  183ms      5.36    8.58MB    16.1
## 3 perform_kruskal_wallis_tests(X, grp~ 181ms  184ms      5.41    8.74MB    18.0
```

Vectorizing kwtest_fast() still leads to performance improvements, albeit not as dramatically as before. The median time still decreased, however the allocated memory increased slightly. In most use cases however, the decreased median time is of higher importance than the allocated memory.