Lab 1: Efficient Programming

Introduction to Statistical Computing

Elena Conderana & Sergio Cuenca

Contents

This lab is to be done outside of class time. You may collaborate with one classmate, but you must identify yourself and his/her name above, in the author's field, and you must submit **your own** lab as this completed .Rmd file.

Installing and loading packages

In order to perform the exercise in this practice you should install the `microbenchmark` and `profvis` packages. Also install `devtools` and the `proftools` package from CRAN.

```
install.packages("microbenchmark")
install.packages("profvis")
install.packages("devtools")
install.packages("proftools")
```

Load the installed packages.

```
library(microbenchmark)
library(profvis)
library(devtools)
```

```
## Loading required package: usethis
```

```
library(proftools)
```

From the `Bioconductor` repository you must also install the `graph` and `Rgraphviz` packages. To install packages from this repository, you must install `BiocManager` package first and then use the `BiocManager::install()` function to install the packages.

```
install.packages("BiocManager", dep = TRUE)
BiocManager::install(c("Rgraphviz","graph"))saa
```

Q1. Microbenchmarking

1a. Use the `microbenchmark::microbenchmark()` function to know which of the following three functions is the fastest to perform the cumulative sum of a 100-element vector. By how much is the fastest with respect to the second one?

```
x <- 1:100 # initiate vector to cumulatively sum

# Method 1: with a for loop
cs_for <- function(x) {
  for (i in x) {
    if (i == 1) {
      xc = x[i]
    } else {
      xc = c(xc, sum(x[1:i]))
    }
  }
```

```
  xc
}

# Method 2: with apply
cs_apply <- function(x) {
  sapply(x, function(x) sum(1:x))
}

# Method 3: cumsum
cs_cumsum <- function(x) {
  cumsum(x)
}

# Benchmark the three methods
benchmark_results <- microbenchmark(
  cs_for(x),
  cs_apply(x),
  cs_cumsum(x),
  times = 1000
)
```

```
## Warning in microbenchmark(cs_for(x), cs_apply(x), cs_cumsum(x), times = 1000):
## less accurate nanosecond times to avoid potential integer overflows
```

```
print(benchmark_results)
```

```
## Unit: nanoseconds
##          expr   min      lq       mean median      uq      max neval
##     cs_for(x) 67855 74640.5 185096.222  78638 84808.5 92038645  1000
##   cs_apply(x) 41410 43542.0  50451.402  46863 52500.5  1281332  1000
##  cs_cumsum(x)   369   430.5   1064.032    492   533.0   448991  1000
```

```
median_times <- aggregate(benchmark_results$time,
                          by = list(benchmark_results$expr),
                          FUN = median)
colnames(median_times) <- c("Method", "Median_Time")
median_times <- median_times[order(median_times$Median_Time), ]

fastest_method <- median_times$Method[1]
fastest_time <- median_times$Median_Time[1]

second_fastest_method <- median_times$Method[2]
second_fastest_time <- median_times$Median_Time[2]

speedup_factor <- second_fastest_time / fastest_time
cat(sprintf("The fastest method is %s, and it is %.2f times faster than %s.\n",
            fastest_method, speedup_factor, second_fastest_method))
```

```
## The fastest method is cs_cumsum(x), and it is 95.25 times faster than
cs_apply(x).
```

1b. Run the same benchmark but now `x` is `1:50000`. As the benchmark could take too long, set the argument `time = 1` in the `microbenchmark` function. Does the relative difference between the fastest and the second fastest increase or decrease? By how much?

```r
x <- 1:50000

benchmark_results_long <- microbenchmark(
  cs_for = cs_for(x),
  cs_apply = cs_apply(x),
  cs_cumsum = cs_cumsum(x),
  times = 1
)

print(benchmark_results_long)
```

```
## Unit: microseconds
##       expr          min           lq         mean       median           uq
##     cs_for 11675774.918 11675774.918 11675774.918 11675774.918 11675774.918
##   cs_apply    19816.366    19816.366    19816.366    19816.366    19816.366
##  cs_cumsum      149.978      149.978      149.978      149.978      149.978
##           max neval
##  11675774.918     1
##     19816.366     1
##       149.978     1
```

```r
median_times_large <- aggregate(benchmark_results_long$time,
                                by = list(benchmark_results_long$expr),
                                FUN = median)
colnames(median_times_large) <- c("Method", "Median_Time")
median_times_large <- median_times_large[order(median_times_large$Median_Time), ]

fastest_method_large <- median_times_large$Method[1]
fastest_time_large <- median_times_large$Median_Time[1]

second_fastest_method_large <- median_times_large$Method[2]
second_fastest_time_large <- median_times_large$Median_Time[2]

speedup_factor_large <- second_fastest_time_large / fastest_time_large

cat(sprintf("For x = 1:50000, the fastest method is %s and it is %.2f times faster than %s.\n",
            fastest_method_large, speedup_factor_large, second_fastest_method_large))
```

```
## For x = 1:50000, the fastest method is cs_cumsum and it is 132.13 times
faster than cs_apply.
```

```r
speedup_change <- speedup_factor_large / speedup_factor

cat(sprintf("The relative speedup factor has %s by %.2f times compared to x = 1:100.\n",
            ifelse(speedup_change > 1, "increased", "decreased"), speedup_change))
```

```
## The relative speedup factor has increased by 1.39 times compared to x =
1:100.
```

1c. Try profiling a section of code you have written using the `profvis::profvis()` function. Where are the bottlenecks? Were they where you expected?

```r
profvis({

  x <- 1:50000

  # Method 1: with a for loop
  cs_for <- function(x) {
    for (i in x) {
      if (i == 1) {
        xc = x[i]
      } else {
        xc = c(xc, sum(x[1:i]))
      }
    }
    xc
  }

  # Method 2: with apply
  cs_apply <- function(x) {
    sapply(x, function(x) sum(1:x))
  }

  # Method 3: cumsum
  cs_cumsum <- function(x) {
    cumsum(x)
  }

  cs_for(x)
  cs_apply(x)
  cs_cumsum(x)
})
```

The profiling results from the data table and flame graph show that the main bottleneck is located in the `cs_for` function, which computes the cumulative sum using a for loop. These results were expected as this method was the slowest, and it can be confirmed with the profiling, taking 6870 ms and consuming 7497.5 MB of memory. The major problem is that `c(xc, sum(x[1:i]))` continuously appends to`xc`, causing memory reallocation at each iteration, which is very inefficient. `cs_apply` performs much better, taking only 20 ms and almost no memory overhead. Finally, `cumsum` is the fastest, practically taking no time, as it is a function highly optimized at the C level.

1d. Let's profile a section of code with the `Rprof()` function. The code section is a function to compute sample variance of a numeric vector:

```r
# Compute sample variance of numeric vector x
sampvar <- function(x) {
    # Compute sum of vector x
    my.sum <- function(x) {
        sum <- 0
        for (i in x) {
            sum <- sum + i
        }
        sum
    }

    # Compute sum of squared variances of the elements of x from
```

```
    # the mean mu
    sq.var <- function(x, mu) {
        sum <- 0
        for (i in x) {
            sum <- sum + (i - mu) ^ 2
        }
        sum
    }

    mu <- my.sum(x) / length(x)
    sq <- sq.var(x, mu)
    sq / (length(x) - 1)
}
```

To use the `Rprof()` function, you shall specify in which file you want to store the results of the profiling. Then you execute the code you want to profile, and then you execute `Rprof(NULL)` to stop profiling. In order to profile the `sampvar()` function applied to a random 100 million number vector:

```
x <- runif(1e8)
Rprof("Rprof.out", memory.profiling = TRUE)
y <- sampvar(x)
Rprof(NULL)
```

Use the `summaryRprof()` function to print a summary of the code profiling. Which part of the function takes more time to execute? Which part of the function requires more memory?

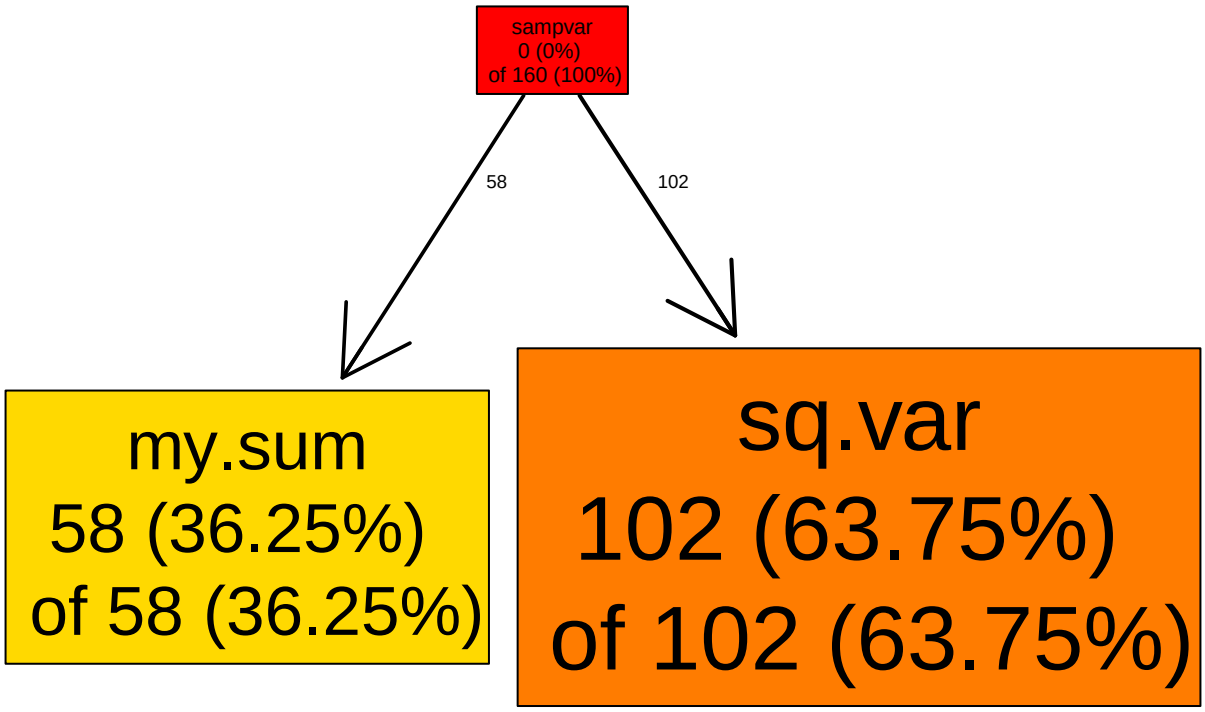
```
summaryRprof("Rprof.out", memory = "both")
```

```
## $by.self
##          self.time self.pct total.time total.pct mem.total
## "sq.var"      2.04    63.75       2.04     63.75       1.4
## "my.sum"      1.16    36.25       1.16     36.25       0.0
##
## $by.total
##           total.time total.pct mem.total self.time self.pct
## "sampvar"       3.20    100.00       1.4      0.00     0.00
## "sq.var"        2.04     63.75       1.4      2.04    63.75
## "my.sum"        1.16     36.25       0.0      1.16    36.25
##
## $sample.interval
## [1] 0.02
##
## $sampling.time
## [1] 3.2
```

The part of the function that takes more time is `sq.var()`, which takes 65% of the total execution time (2.08 out of 3.20 seconds), indicating that computing the sum of squared variances is the bottleneck. Both functions display 0 in the `mem.total` column, indicating that no appreciable variations in memory usage were noted. This is probably because, instead of allocating big objects, both functions work with numeric scalars.

1e. `summaryRprof()` function prints a summary of the code profiling, but it is not user-friendly to read. Using the `proftool` packages, let's see the results from the `Rprof.out` file. See the help (`?`) for the functions `readProfiledata` and `plotProfileCallGraph` and plot the results of the code profiling from **1d**.

```r
prof_data <- readProfileData("Rprof.out")
plotProfileCallGraph(prof_data)
```



Q2. Efficient set-up

Let's check if you have an optimal R installation.

2a. What is the exact version of your computer's operating system?

```r
system("sw_vers")
```

2b. Start an activity monitor and execute the following chunk. In it, `lapply()` (or its parallel version `mclapply()`) is used to apply a function, `median()`, over every column in the data frame object X

```r
# Note: uses 2+ GB RAM and several seconds or more depending on hardware
# 1: Create large dataset
X <- as.data.frame(matrix(rnorm(1e9), nrow = 1e8))
# 2: Find the median of each column using a single core
r1 <- lapply(X, median)
# 3: Find the median of each column using many cores
r2 <- parallel::mclapply(X, median)
```

2c. Try modifying the settings of your RStudio setup using the Tools > Global Options menu. What settings do you think can affect R performance? (Note only some of them, not ALL of them)

```r
options(mc.cores = parallel::detectCores() - 1)
```

Some setting that affect R performance in RStudio may be: * Memory Allocation: Increase memory limits. * Number of Threads: Adjust `mc.cores` for parallel processing (`Options > Parallel`). * Workspace Handling: Disable automatic `.RData` saving (`Options > General > Save workspace` to `.RData on exit → Never`). * Lazy Data Loading: Set options(`stringsAsFactors = FALSE`) to improve efficiency. * Execution Mode: Enable chunk execution in parallel for large computations (`Tools > Global Options > R Markdown`).

2d. Try some of the shortcuts integrated in RStudio. What shortcuts do you think can save you development time? (Note only some of them, not ALL of them)

Some RStudio shortcuts that can save development time:

- Run Current Line/Selection: `Cmd + Enter` (Mac) / `Ctrl + Enter` (Windows/Linux)
- Comment/Uncomment Code: `Cmd + Shift + C` / `Ctrl + Shift + C`
- Insert Code Chunk (R Markdown): `Cmd + Option + I` / `Ctrl + Alt + I`
- Navigate Between Tabs: `Cmd + Shift + [` or `Cmd + Shift + ]` / `Ctrl + Shift + Tab`
- Find in Files: `Cmd + Shift + F` / `Ctrl + Shift + F`

2e. Check how well your computer is suited to perform data analysis tasks. In the following code chunk you will run a benchmark test from the `benchmarkme` package and plot your result against the results from people around the world. Do you think that you should upgrade your computer?

```r
library("benchmarkme")
# Run standard tests
res_std <- benchmark_std(runs=3)
plot(res_std)
# Run memory I/O tests by reading/writing a 5MB file
res_io <- benchmark_io(runs = 1, size = 5)
plot(res_io)
```

Given the results shown in the graphs, my computer performs well on most cases and outperforms the majority of other machines in all tests, so an upgrade is not necessary.

Q3. Efficient programming

3a. Create a vector `x` of 100 random numbers and use the microbenchmark package to compare the vectorised construct `x = x + 1` to the for loop version `for (i in seq_len(n)) x[i] = x[i] + 1`. Try varying the size of the input vector and check how the results differ. Which functions are being called by each method?

```r
benchmark_increment <- function(n) {
  x <- runif(n)

  vec_increment <- function(x) {
    x <- x + 1
    x
  }

  loop_increment <- function(x) {
    for (i in seq_len(length(x))) {
      x[i] <- x[i] + 1
```

```
    }
    x
  }

  res <- microbenchmark(
    vectorized = vec_increment(x),
    loop = loop_increment(x),
    times = 100
  )

  print(res)
}

benchmark_increment(100)
```

```
## Unit: nanoseconds
##        expr  min   lq    mean median   uq   max neval
##  vectorized  164  205  302.58    205  328  2747   100
##        loop 3608 3690 4106.97   3936 4100 18204   100
```

```
benchmark_increment(10000)
```

```
## Unit: microseconds
##        expr    min       lq      mean   median      uq     max neval
##  vectorized   3.69   7.6670   8.55055   8.2410   9.389  18.040   100
##        loop 335.79 346.3065 360.61837 354.2605 369.082 458.052   100
```

```
benchmark_increment(100000)
```

```
## Unit: microseconds
##        expr      min       lq      mean   median       uq      max neval
##  vectorized   38.950   75.071  122.9373   85.157   96.596 3648.057   100
##        loop 3384.755 3516.713 3700.7564 3611.526 3772.697 6508.832   100
```

The vectorized version uses internal C-level optimizations in R. It calls an underlying primitive function that efficiently applies addition across all elements. It is significantly faster than loops. On the other hand, the for-loop approach iterates over each element individually, calling the indexing and assignment functions repeatedly. It is much slower, especially for large `n`, due to overhead in handling each assignment separately.

3b. Monte Carlo integration can be performed with the following code:

```
monte_carlo = function(N) {
  hits = 0
  for (i in seq_len(N)) {
    u1 = runif(1)
    u2 = runif(1)
    if (u1 ^ 2 > u2)
      hits = hits + 1
  }
  return(hits / N)
}
```

Create a vectorized function `monte_carlo_vec` which do not use a `for` loop.

```
monte_carlo_vec <- function(N) {
  u1 <- runif(N)
  u2 <- runif(N)
  hits <- sum(u1^2 > u2)
  return(hits / N)
}
```

3c. How much faster is the vectorized function `monte_carlo_vec` with respect to the original function `monte_carlo`?

```
N <- 100000
monte_carlo_benchmark <- microbenchmark(
  monte_carlo(N),
  monte_carlo_vec(N),
  times = 10
)

print(monte_carlo_benchmark)
```

```
## Unit: milliseconds
## expr min lq mean median uq max
## monte_carlo(N) 84.516457 85.315301 86.67901 85.893401 88.22720 90.871908
## monte_carlo_vec(N) 1.307613 1.387235 1.64740 1.409231 1.46493 3.636126
## neval
## 10
## 10
```

```
monte_carlo_median_times <- aggregate(monte_carlo_benchmark$time,
                                      by = list(monte_carlo_benchmark$expr),
                                      FUN = median)
colnames(monte_carlo_median_times) <- c("Method", "Median_Time")
monte_carlo_median_times <- monte_carlo_median_times[order(monte_carlo_median_times$Median_Time), ]

monte_carlo_fastest_method <- monte_carlo_median_times$Method[1]
monte_carlo_fastest_time <- monte_carlo_median_times$Median_Time[1]

monte_carlo_second_fastest_method <- monte_carlo_median_times$Method[2]
monte_carlo_second_fastest_time <- monte_carlo_median_times$Median_Time[2]

monte_carlo_speedup_factor <- monte_carlo_second_fastest_time / monte_carlo_fastest_time
cat(sprintf("The fastest method is %s, and it is %.2f times faster than %s.\n",
            monte_carlo_fastest_method, monte_carlo_speedup_factor, monte_carlo_second_fastest_method))
```

```
## The fastest method is monte_carlo_vec(N), and it is 60.95 times faster than
monte_carlo(N).
```

3d. Using the `memoise` function, create a function called `m_fib` that is the memoized version of the recursive function:

```r
fib <- function(n)  {
  if(n == 1 || n == 2) return(1)
  fib(n-1) + fib(n-2)
}
```

Then, using `microbenchmark`, simulate calculating the 10th position of the Fibonacci serie a 100 times with each function. How much faster is the memoized version?

```r
library("memoise")

m_fib <- memoise(fib)

N <- 10
fib_benchmark <- microbenchmark(
  fib(N),
  m_fib(N),
  times = 100
)

print(fib_benchmark)
```

```
## Unit: microseconds
##      expr    min      lq      mean median      uq      max neval
##    fib(N) 17.138 17.4865  38.50597 17.876 18.3475 2063.858   100
##  m_fib(N) 20.049 20.5820 106.06577 21.074 22.0785 8180.771   100
```

```r
fib_median_times <- aggregate(fib_benchmark$time,
                              by = list(fib_benchmark$expr),
                              FUN = median)
colnames(fib_median_times) <- c("Method", "Median_Time")
fib_median_times <- fib_median_times[order(fib_median_times$Median_Time), ]

fib_fastest_method <- fib_median_times$Method[1]
fib_fastest_time <- fib_median_times$Median_Time[1]

fib_second_fastest_method <- fib_median_times$Method[2]
fib_second_fastest_time <- fib_median_times$Median_Time[2]

fib_speedup_factor <- fib_second_fastest_time / fib_fastest_time
cat(sprintf("The fastest method is %s, and it is %.2f times faster than %s.\n",
            fib_fastest_method, fib_speedup_factor, fib_second_fastest_method))
```

```
## The fastest method is fib(N), and it is 1.18 times faster than m_fib(N).
```

For `N = 10`, the memoized version is slightly slower due to the overhead of caching, as each call is computed only once per execution. However, for larger `N`, memoization provides a significant speedup by storing previously computed values and avoiding redundant calculations, making it much more efficient than the standard recursive approach. Trying with `N = 20`, the memoized version is 79.09 times faster than fib(N).

3e. Try varying the parameters of the **3d** exercise. What happens when you measure the computing time of calculating the 1st position of Fibonacci serie? And the 25th?

```r
m_fib <- memoise(fib)

N <- 1
fib_benchmark <- microbenchmark(
  fib(N),
  m_fib(N),
  times = 100
)

print(fib_benchmark)
```

```
## Unit: nanoseconds
##       expr   min    lq     mean median    uq    max neval
##     fib(N)    82   123   201.72    205   205   2214   100
##  m_fib(N) 22427 22960 24651.66  23124 23411 150306   100
```

```r
fib_median_times <- aggregate(fib_benchmark$time,
                              by = list(fib_benchmark$expr),
                              FUN = median)
colnames(fib_median_times) <- c("Method", "Median_Time")
fib_median_times <- fib_median_times[order(fib_median_times$Median_Time), ]

fib_fastest_method <- fib_median_times$Method[1]
fib_fastest_time <- fib_median_times$Median_Time[1]

fib_second_fastest_method <- fib_median_times$Method[2]
fib_second_fastest_time <- fib_median_times$Median_Time[2]

fib_speedup_factor <- fib_second_fastest_time / fib_fastest_time
cat(sprintf("The fastest method is %s, and it is %.2f times faster than %s.\n",
            fib_fastest_method, fib_speedup_factor, fib_second_fastest_method))
```

```
## The fastest method is fib(N), and it is 112.80 times faster than m_fib(N).
```

For `N = 1`, as mentioned earlier, the memoized version is much slower than the normal recursive function, mainly because memoization introduces overhead for storing and retrieving cached values, which outweighs any benefits for trivial cases. Since `fib(1)` directly returns a constant value without recursion, caching provides no advantage, making the standard function much faster. However, for larger `N`, memoization greatly reduces redundant calculations, making it more efficient.

3f. Create the `c_fib` function as the compilation version of the `fib` function declared in exercise **3d** using the `cmpfun` of the `compiler` package. Which is faster, `fib`, `c_fib` or `m_fib`? And `cm_fib` (compiled version of `m_fib`)? And `mc_fib` (memoized version of `c_fib`)?

```r
library(compiler)

# Memoized version
m_fib <- memoise(fib)

# Compiled version
c_fib <- cmpfun(fib)
```

```r
# Compiled memoized version
cm_fib <- cmpfun(m_fib)

# Memoized compiled version
mc_fib <- memoise(c_fib)

N <- 25
fib_benchmark <- microbenchmark(
  fib(N),
  m_fib(N),
  c_fib(N),
  cm_fib(N),
  mc_fib(N),
  times = 10
)

print(fib_benchmark)
```

```
## Unit: microseconds
## expr min lq mean median uq max neval
## fib(N) 24485.241 24801.966 25724.4291 25395.3385 26679.766 27173.693 10
## m_fib(N) 28.741 34.153 82.9635 69.4335 141.573 146.657 10
## c_fib(N) 24449.366 24790.076 25445.3052 25461.0410 26130.366 26693.132 10
## cm_fib(N) 25.707 41.820 2670.5637 119.5355 129.929 25874.608 10
## mc_fib(N) 24.026 25.133 2711.5063 32.4925 40.836 26722.488 10
```

```r
fib_median_times <- aggregate(fib_benchmark$time,
                              by = list(fib_benchmark$expr),
                              FUN = median)
colnames(fib_median_times) <- c("Method", "Median_Time")
fib_median_times <- fib_median_times[order(fib_median_times$Median_Time), ]

fib_fastest_method <- fib_median_times$Method[1]
fib_fastest_time <- fib_median_times$Median_Time[1]

fib_second_fastest_method <- fib_median_times$Method[2]
fib_second_fastest_time <- fib_median_times$Median_Time[2]

fib_speedup_factor <- fib_second_fastest_time / fib_fastest_time

cat(sprintf("The fastest method is %s, and it is %.2f times faster than %s.\n",
            fib_fastest_method, fib_speedup_factor, fib_second_fastest_method))
```

```
## The fastest method is mc_fib(N), and it is 2.14 times faster than m_fib(N).
```

Challenge 01. Calculate the computing time for calculating the Fibonacci serie 5 times from the 1st to the 25th position with the `fib`, `c_fib`, `m_fib`, `cm_fib` and `mc_fib` functions. Store the results for each position and create a plot showing these results. When does it begin to compensate using the memoized version? Hint: Use `geom_point()` and `geom_errorbars()` function of `ggplot2` to show the `median`, `lq` and `uq` values of the `microbenchmark` analysis.

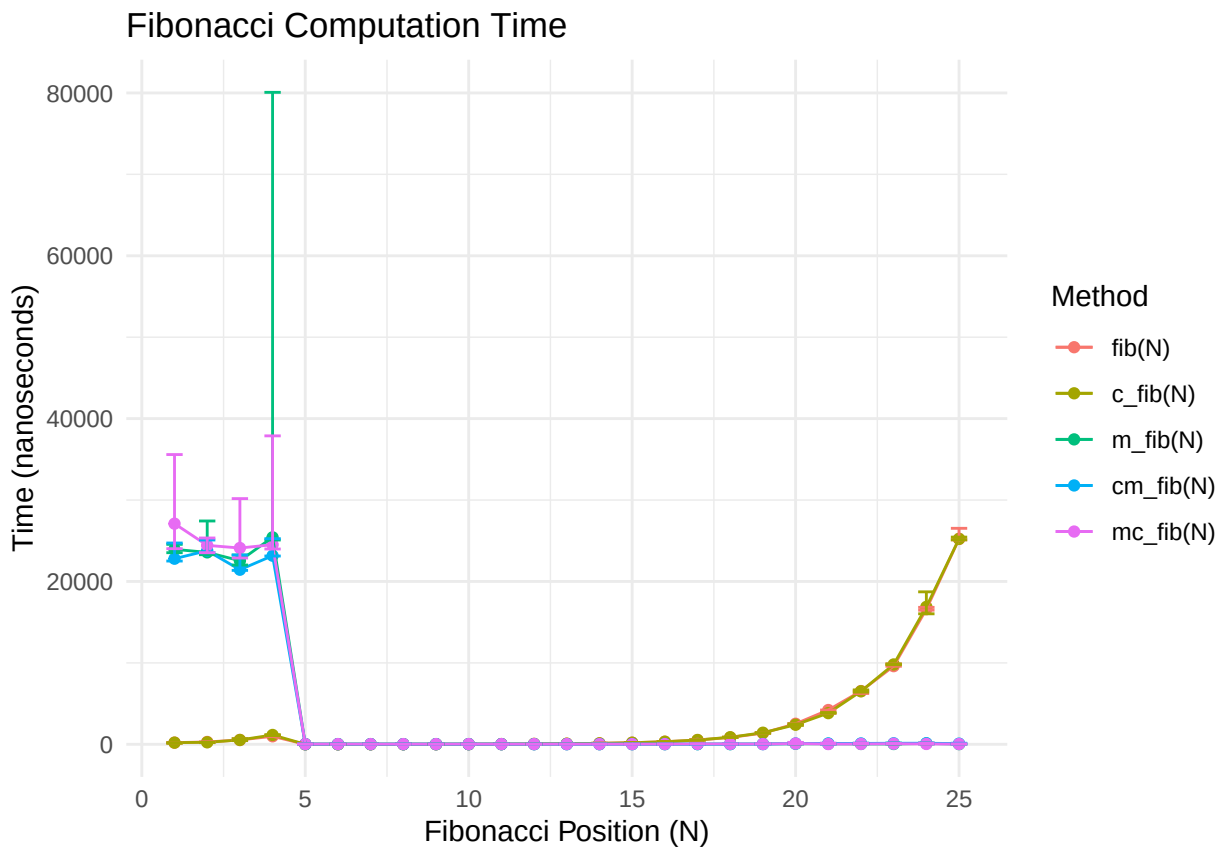
```r
library(dplyr)
library(ggplot2)

results <- data.frame()
for (N in 1:25) {
  benchmark <- microbenchmark(
    fib(N),
    c_fib(N),
    m_fib(N),
    cm_fib(N),
    mc_fib(N),
    times = 5
  )

  summary_data <- summary(benchmark)
  summary_data$N <- N
  results <- rbind(results, summary_data)
}

results <- results %>% rename(Method = expr)

# Plot results
ggplot(results, aes(x = N, y = median, color = Method)) +
  geom_point() +
  geom_line() +
  geom_errorbar(aes(ymin = lq, ymax = uq), width = 0.5) +
  labs(title = "Fibonacci Computation Time",
       x = "Fibonacci Position (N)",
       y = "Time (nanoseconds)",
       color = "Method") +
  theme_minimal()
```



Q4. Efficient data I/O

4a. Import data from https://github.com/mledoze/countries/raw/master/countries.json using the `import()` function from the `rio` package.

```
library(rio)
library(jsonlite)

url <- "https://github.com/mledoze/countries/raw/master/countries.json"
countries_data <- fromJSON(url, flatten = TRUE)
```

4b. Export the data imported in **4a** to 3 different file formats of your choosing supported by `rio` (see `vignette("rio")` for supported formats). Try opening these files in external programs. Which file formats are more portable?

```
export(countries_data, "countries.csv")
export(countries_data, "countries.xlsx")
export(countries_data, "countries.rds")
```

Challenge 03. Create a simple benchmark to compare the write times for the different file formats of **4b**. Which is fastest? Which is the most space efficient?

```r
csv_file <- "countries.csv"
xlsx_file <- "countries.xlsx"
rds_file <- "countries.rds"

write_benchmark <- microbenchmark(
  export(countries_data, csv_file),
  export(countries_data, xlsx_file),
  export(countries_data, rds_file),
  times = 5
)

print(write_benchmark)
```

```
## Unit: milliseconds
##                                expr       min        lq     mean    median
##   export(countries_data, csv_file)  3.600415  4.347066 11.12300  5.295888
##  export(countries_data, xlsx_file) 30.073951 31.302926 33.31611 31.376972
##   export(countries_data, rds_file) 36.588154 36.695082 45.26029 36.727513
##         uq      max neval
##   6.230524 36.14113     5
##  31.427976 42.39872     5
##  39.716659 76.57402     5
```

```r
write_median_times <- aggregate(write_benchmark$time,
                                by = list(write_benchmark$expr),
                                FUN = median)
colnames(write_median_times) <- c("Method", "Median_Time")
write_median_times <- write_median_times[order(write_median_times$Median_Time), ]

write_fastest_method <- write_median_times$Method[1]
write_fastest_time <- write_median_times$Median_Time[1]

write_second_fastest_method <- write_median_times$Method[2]
write_second_fastest_time <- write_median_times$Median_Time[2]

write_speedup_factor <- write_second_fastest_time / write_fastest_time

cat(sprintf("The fastest method is %s, and it is %.2f times faster than %s.\n",
            write_fastest_method, write_speedup_factor, write_second_fastest_method))
```

```
## The fastest method is export(countries_data, csv_file), and it is 5.92
times faster than export(countries_data, xlsx_file).
```

```r
file_sizes <- data.frame(
  Format = c("CSV", "XLSX", "RDS"),
  Size_MB = c(
    file.size(csv_file) / 1024^2,
    file.size(xlsx_file) / 1024^2,
    file.size(rds_file) / 1024^2
  )
)

print(file_sizes)
```

```
##   Format   Size_MB
## 1    CSV 0.4902992
## 2   XLSX 0.1495199
## 3    RDS 0.1270924
```

```
most_efficient_format <- file_sizes$Format[which.min(file_sizes$Size_MB)]
smallest_size <- min(file_sizes$Size_MB)

cat(sprintf("The most space-efficient format is %s, with a file size of %.2f MB.\n",
            most_efficient_format, smallest_size))
```

```
## The most space-efficient format is RDS, with a file size of 0.13 MB.
```

Q5. Efficient data carpentry

5a. Create the following data.frame:

```
df_base = data.frame(colA = "A")
```

Try and guess the output of the following commands. Quit the `eval = FALSE` argument and check if the output is what you thought.

```
print(df_base)
```

```
##   colA
## 1    A
```

```
df_base$colA
```

```
## [1] "A"
```

```
df_base$col
```

```
## [1] "A"
```

```
df_base$colB
```

```
## NULL
```

Now create a tibble `tbl_df` and repeat the above commands.

```
tbl_df <- tibble(colA = "A")

tbl_df$colA
```

```
## [1] "A"
```

```
tbl_df$col
```

```
## Warning: Unknown or uninitialised column: `col`.
```

```
## NULL
```

```
tbl_df$colB
```

```
## Warning: Unknown or uninitialised column: `colB`.
```

```
## NULL
```

5b. Load and look at subsets of the `pew` dataset. What is untidy about it? Convert it into tidy form.

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v forcats   1.0.0     v stringr   1.5.1
## v lubridate 1.9.4     v tibble    3.2.1
## v purrr     1.0.2     v tidyr     1.3.1
## v readr     2.1.5
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter()  masks stats::filter()
## x purrr::flatten() masks jsonlite::flatten()
## x dplyr::lag()     masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

```
pew <- read.csv("pew.csv")

pew_tidy <- pew %>%
  pivot_longer(
    cols = -religion,
    names_to = "Income",
    values_to = "Count"
  )
```

The dataset is in a wide format, where different income categories are separate columns instead of being a single one. A tidy dataset should have each observation in its own row, with three columns: Religion, Income, and Count.

5c. Consider the following string of phone numbers and fruits:

```
strings = c(" 219 733 8965", "329-293-8753 ", "banana", "595 794 7569",
            "387 287 6718", "apple", "233.398.9187  ", "482 952 3315",
            "239 923 8115", "842 566 4692", "Work: 579-499-7527",
            "$1000", "Home: 543.355.3679")
```

Write expressions in `stringr` and `base R` that return a logical vector reporting whether or not each string contains a number.

```r
library(stringr)

contains_number_stringr <- str_detect(strings, "\\d")
print(contains_number_stringr)
```

```
##  [1]  TRUE  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [13]  TRUE
```

```r
contains_number_baseR <- grepl("\\d", strings)
print(contains_number_baseR)
```

```
##  [1]  TRUE  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [13]  TRUE
```

Q6. Efficient optimization

6a. Create a vector `x` and benchmark `any(is.na(x))` against `anyNA(x)`. Do the results vary with the size of the vector?

```r
x <- sample(c(1:1e6, NA), 1e6, replace = TRUE)

benchmark_results <- microbenchmark(
  any(is.na(x)),
  anyNA(x),
  times = 100
)

print(benchmark_results)
```

```
## Unit: microseconds
##           expr      min       lq      mean   median        uq      max neval
##  any(is.na(x)) 1682.845 1794.755 2157.9985 1838.296 1946.8235 7860.971   100
##       anyNA(x)  340.587  350.058  358.9587  355.306  364.0595  541.036   100
```

```r
# Calculate and display the median execution times
median_times <- aggregate(benchmark_results$time,
                          by = list(benchmark_results$expr),
                          FUN = median)
colnames(median_times) <- c("Method", "Median_Time")
median_times <- median_times[order(median_times$Median_Time), ]

fastest_method <- median_times$Method[1]
fastest_time <- median_times$Median_Time[1]

second_fastest_method <- median_times$Method[2]
second_fastest_time <- median_times$Median_Time[2]

speedup_factor <- second_fastest_time / fastest_time

cat(sprintf("The fastest method is %s, and it is %.2f times faster than %s.\n",
            fastest_method, speedup_factor, second_fastest_method))
```

```
## The fastest method is anyNA(x), and it is 5.17 times faster than
any(is.na(x)).
```

The function `anyNA(x)` is a more efficient alternative to `any(is.na(x))`, as it directly checks for missing values without creating an intermediate logical vector. Hence, the larger the vector, the greater the speed advantage of `anyNA(x)`.

6b. Construct a `matrix` of `integer` and a `matrix` of `numeric`s and use a `pryr::object_size()` to compare the object occupation.

```r
library(pryr)
```

```
##
## Attaching package: 'pryr'

## The following objects are masked from 'package:purrr':
##
##     compose, partial

## The following object is masked from 'package:dplyr':
##
##     where
```

```r
int_matrix <- matrix(1L:1e6, nrow = 1000, ncol = 1000)
num_matrix <- matrix(as.numeric(1:1e6), nrow = 1000, ncol = 1000)

size_int_matrix <- object_size(int_matrix)
size_num_matrix <- object_size(num_matrix)

cat(sprintf("Integer matrix size: %s\n", format(size_int_matrix, units = "MB")))
```

```
## Integer matrix size: 4.00 MB
```

```r
cat(sprintf("Numeric matrix size: %s\n", format(size_num_matrix, units = "MB")))
```

```
## Numeric matrix size: 8.00 MB
```

```r
size_diff <- size_num_matrix / size_int_matrix
cat(sprintf("The numeric matrix is %.2f times larger than the integer matrix.\n", size_diff))
```

```
## The numeric matrix is 2.00 times larger than the integer matrix.
```

6c. Consider the following piece of code:

```r
double test1() {
  double a = 1.0 / 81;
  double b = 0;
  for (int i = 0; i < 729; ++i)
    b = b + a;
  return b;
}
```

- Save the function `test1()` in a separate file. Make sure it works.
- Write a similar function in R and compare the speed of the C++ and R versions.

```
library(Rcpp)

test1_r <- function() {
  a <- 1.0 / 81
  b <- 0
  for (i in 1:729) {
    b <- b + a
  }
  return(b)
}

sourceCpp("test1.cpp")

rc_benchmark_results <- microbenchmark(
  test1(),
  test1_r(),
  times = 100
)

print(rc_benchmark_results)
```

```
## Unit: nanoseconds
##       expr  min   lq     mean median    uq     max neval
##    test1()  820  861  4612.91    861   861  365392   100
##  test1_r() 9840 9881 26286.74   9922 10168 1636269   100
```

```
rc_median_times <- aggregate(rc_benchmark_results$time,
                             by = list(rc_benchmark_results$expr),
                             FUN = median)
colnames(rc_median_times) <- c("Method", "Median_Time")
rc_median_times <- rc_median_times[order(rc_median_times$Median_Time), ]

rc_fastest_method <- rc_median_times$Method[1]
rc_fastest_time <- rc_median_times$Median_Time[1]

rc_second_fastest_method <- rc_median_times$Method[2]
rc_second_fastest_time <- rc_median_times$Median_Time[2]

rc_speedup_factor <- rc_second_fastest_time / rc_fastest_time

cat(sprintf("The fastest method is %s, and it is %.2f times faster than %s.\n",
            rc_fastest_method, rc_speedup_factor, rc_second_fastest_method))
```

```
## The fastest method is test1(), and it is 11.52 times faster than test1_r().
```

- Create a function called `test2()` where the `double` variables have been replaced by `float`. Do you still get the correct answer?
- Change `b = b + a` to `b += a` to make your code more `C++` like.
- (Bonus) What's the difference between `i++` and `++i`?

```r
library(Rcpp)

sourceCpp("test2.cpp")

rc_benchmark_results <- microbenchmark(
  test2(),
  test1_r(),
  times = 100
)

print(rc_benchmark_results)
```

```
## Unit: nanoseconds
##       expr  min   lq     mean median    uq    max neval
##    test2()  820  861  6583.78    861   902 557354   100
##  test1_r() 9840 9922 10297.15   9963 10168  21361   100
```

```r
rc_median_times <- aggregate(rc_benchmark_results$time,
                             by = list(rc_benchmark_results$expr),
                             FUN = median)
colnames(rc_median_times) <- c("Method", "Median_Time")
rc_median_times <- rc_median_times[order(rc_median_times$Median_Time), ]

rc_fastest_method <- rc_median_times$Method[1]
rc_fastest_time <- rc_median_times$Median_Time[1]

rc_second_fastest_method <- rc_median_times$Method[2]
rc_second_fastest_time <- rc_median_times$Median_Time[2]

rc_speedup_factor <- rc_second_fastest_time / rc_fastest_time

cat(sprintf("The fastest method is %s, and it is %.2f times faster than %s.\n",
            rc_fastest_method, rc_speedup_factor, rc_second_fastest_method))
```

```
## The fastest method is test2(), and it is 11.57 times faster than test1_r().
```

The results given by the function `test2()` using `float` instead of `double` are very close to those obtained with `test1()`, but there are some slight differences due to floating-point precision. Usually, `float` is less accurate than `double`, and over many iterations, these small inaccuracies can accumulate.

- `i++` is post-increment: It returns the current value of `i` before incrementing.
- `++i` is pre-increment: It increments i first, then returns the new value. In most cases, the difference is very small, but in codes like loops, `++i` is slightly faster because it avoids an unnecessary copy of the variable.

Q7. Efficient hardware

7a. How much RAM does your computer have? (Optional question, privacy above all. Write a random number if you do not want to share your hardware information.)

```r
library("benchmarkme")
get_ram()
```

```
## 34.4 GB
```

7b. Using your preferred search engine, how much does it cost to double the amount of available RAM on your system? (Again, write a random number if you do not want to share your hardware information)

Upgrading the RAM on an M1 Max MacBook Pro is not possible after purchase because the memory is built into the M1 chip and cannot be changed. To upgrade the RAM from 34.4 GB to a greater capacity, I'd have to buy a new MacBook Pro with the desired RAM configuration. Apple's RAM upgrade pricing can be very high; for instance, upgrading from 16 GB to 32 GB of RAM might cost approximately €460.

7c. Check if you are using a 32-bit or 64-bit version of R.

```r
sessionInfo()
```

```
## R version 4.4.2 (2024-10-31)
## Platform: aarch64-apple-darwin24.1.0
## Running under: macOS Sequoia 15.1.1
##
## Matrix products: default
## BLAS: /opt/homebrew/Cellar/openblas/0.3.29/lib/libopenblasp-r0.3.29.dylib
## LAPACK: /opt/homebrew/Cellar/r/4.4.2_2/lib/R/lib/libRlapack.dylib; LAPACK
version 3.12.0
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## time zone: Europe/Madrid
## tzcode source: internal
##
## attached base packages:
## [1] stats graphics grDevices utils datasets methods base
##
## other attached packages:
## [1] benchmarkme_1.0.8 Rcpp_1.0.14 pryr_0.1.6
## [4] lubridate_1.9.4 forcats_1.0.0 stringr_1.5.1
## [7] purrr_1.0.2 readr_2.1.5 tidyr_1.3.1
## [10] tibble_3.2.1 tidyverse_2.0.0 jsonlite_1.8.9
## [13] rio_1.2.3 ggplot2_3.5.1 dplyr_1.1.4
## [16] memoise_2.0.1 proftools_0.99-3 devtools_2.4.5
## [19] usethis_3.1.0 profvis_0.4.0 microbenchmark_1.5.0
## [22] knitr_1.49
##
## loaded via a namespace (and not attached):
## [1] benchmarkmeData_1.0.4 gtable_0.3.6 xfun_0.50
## [4] htmlwidgets_1.6.4 remotes_2.5.0 lattice_0.22-6
## [7] tzdb_0.4.0 vctrs_0.6.5 tools_4.4.2
## [10] generics_0.1.3 parallel_4.4.2 pkgconfig_2.0.3
## [13] Matrix_1.7-2 lifecycle_1.0.4 compiler_4.4.2
## [16] munsell_0.5.1 codetools_0.2-20 httpuv_1.6.15
## [19] htmltools_0.5.8.1 yaml_2.3.10 later_1.4.1
```

```
## [22] pillar_1.10.1 urlchecker_1.0.1 ellipsis_0.3.2
## [25] cachem_1.1.0 iterators_1.0.14 sessioninfo_1.2.2
## [28] foreach_1.5.2 mime_0.12 tidyselect_1.2.1
## [31] digest_0.6.37 stringi_1.8.4 fastmap_1.2.0
## [34] grid_4.4.2 colorspace_2.1-1 cli_3.6.3
## [37] magrittr_2.0.3 pkgbuild_1.4.6 withr_3.0.2
## [40] prettyunits_1.2.0 scales_1.3.0 promises_1.3.2
## [43] timechange_0.3.0 httr_1.4.7 rmarkdown_2.29
## [46] lobstr_1.1.2 hms_1.1.3 shiny_1.10.0
## [49] evaluate_1.0.3 doParallel_1.0.17 miniUI_0.1.1.1
## [52] rlang_1.1.5 xtable_1.8-4 glue_1.8.0
## [55] pkgload_1.4.0 rstudioapi_0.17.1 R6_2.5.1
## [58] fs_1.6.5
```