Big Data Analytics

Lecture 4:

Advanced R Programming

Prof. Dr. Ulrich Matter

18/03/2021

Updates

Goals for today

1. Understand the basics of R's memory management (with a practical big data focus).
2. Understand how data types and data structures of R objects are related to efficient memory allocation.
3. Know the basic tools and approaches to measuring and improving the performance of your R code.
4. (Review of/ideas about workflow with RStudio and GitHub for data projects.)

Advanced R Programming

‘Data projects’ with RStudio and GitHub

Image by jonobacon (CC BY 2.0)

Suggestion for set up

- Organize data analytics project as RStudio-project
- Rstudio project folder = GitHub repository
- (essentially what you will do in your group examination tasks)

Version control with Git

- Keep track of your code.
- Develop in different branches.
- Safely go back to previous versions.

Code repository on GitHub

- Work from different machines.
- Manage and document the project.
- Publish and collaborate.

Names and Values

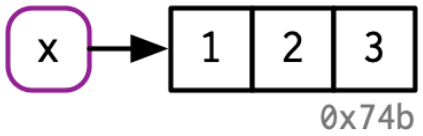
Names and Values

- Prerequesites: `install.packages("lobstr")`
- Background: memory allocation and memory addresses
- ‘Where’ is an R object located in memory?
- How is a variable name associated with the object?
- What happens when we ‘copy’/modify an object in R?

Bindings basics

- Objects/values do not have names but **names have values**!
- Objects have a ‘memory address’/identifiers.

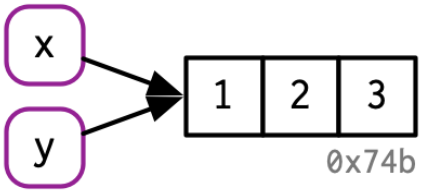
```
x <- c(1, 2, 3)
```



Bindings basics

- We can ‘bind’ several different names to values.

```
y <- x
```



Binding basics

- Understand the concept of names and values: check identifiers

```
obj_addr(x)
```

```
## [1] "0x7f7b0c22bc28"
```

```
obj_addr(y)
```

```
## [1] "0x7f7b0c22bc28"
```

Copy-on-modify

- 'Copying' simply binds a new name to the same (existing) value.

```
x <- c(1, 2, 3)
y <- x
obj_addr(x)
```

```
## [1] "0x7f7b0c1c7718"
```

```
obj_addr(y)
```

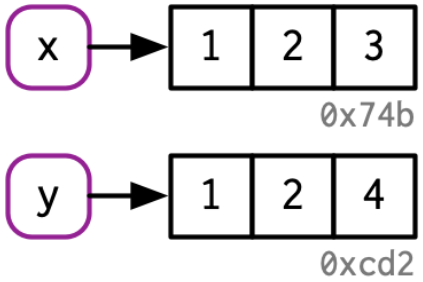
```
## [1] "0x7f7b0c1c7718"
```

Copy-on-modify

- But if we modify values in a vector, actual 'copying' is necessary (depending on the data structure of the object…).

```
y[[3]] <- 4
x
```

```
## [1] 1 2 3
```



Copy-on-modify

- Understand the concept better with `tracemem()`: observe changes in identifiers.

```
x <- c(1, 2, 3)
cat(tracemem(x), "\n")
```

```
## <0x7f7ae8241f08>
```

Copy-on-modify

- Only the first modification actually triggers the copying.

```
y <- x
y[[3]] <- 4L
```

```
## tracemem[0x7f7ae8241f08 -> 0x558ce8a3c6b8]: eval eval withVisible withCallingHandlers handle timi
```

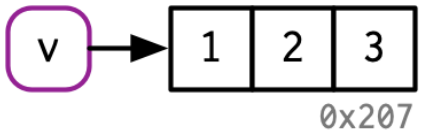
```
y[[3]] <- 5L
```

- **Why?**

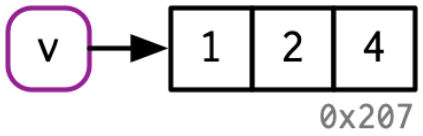
Modify-in-place

- Objects with a single binding get modified in place (no copying needed).
- Enhances performance.

```
v <- c(1, 2, 3)
```




```
v[[3]] <- 4
```



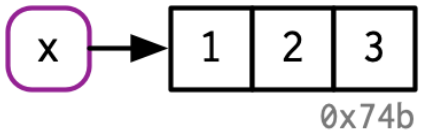
Modify-in-place

- In practice (more complex code) it is often hard to predict whether or not a copy will occur.
  - E.g., usual R functions vs. 'primitive' C functions.
- Use `tracemem()` to check your code for potential improvements (avoid unnecessary copying).

Unbinding and the garbage collector

- What happens when we ‘delete’ (remove) an object?

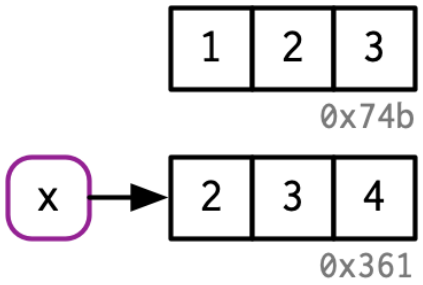
```
x <- 1:3
```



Unbinding and the garbage collector

- What happens when we ‘delete’ (remove) an object?

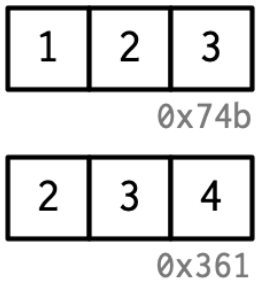
```
x <- 2:4
```



Unbinding and the garbage collector

- What happens when we ‘delete’ (remove) an object?

```
rm(x)
```



Unbinding and the garbage collector

- R collects the garbage automatically (but only cares about R, not other programs).
- Force garbage collection with `gc()` (OS has more memory available!).

```
gc()
```

```
##             used  (Mb) gc trigger   (Mb)  max used   (Mb)
## Ncells   1163055  62.2    2150848  114.9   1759040   94.0
## Vcells 115025800 877.6  213343342 1627.7 211037840 1610.1
```

Data Structures and Data Types

R-tools to investigate structures and types

package	function	purpose
utils	`str()`	Compactly display the structure of an arbitrary R object.
base	`class()`	Prints the class(es) of an R object.
base	`typeof()`	Determines the (R-internal) type or storage mode of an object.

Structures to work with (in R)

We distinguish two basic characteristics:

1. Data types: integers; real numbers (floating point numbers); text (‘string’, ‘character values’).

Structures to work with (in R)

We distinguish two basic characteristics:

1. Data types: integers; real numbers (floating point numbers); text (‘string’, ‘character values’).
2. Basic data structures in RAM:
   - **(Atomic) vectors**
   - **Factors**
   - **Arrays/Matrices**
   - **Lists**
   - **Data frames** et al. (very R-specific)

Data types: numeric

```
a <- 1.5
b <- 3
a + b
```

```
## [1] 4.5
```

Data types: numeric

R interprets this data as type `double` (class 'numeric'):

```
typeof(a)
```

```
## [1] "double"
```

```
class(a)
```

```
## [1] "numeric"
```

```
object.size(a)
```

```
## 56 bytes
```

Data types: character

```
a <- "1.5"
b <- "3"
a + b
```

Data types: character

```
typeof(a)
```

```
## [1] "character"
```

```
class(a)
```

```
## [1] "character"
```

```
object.size(a)
```

```
## 112 bytes
```

Data structures: vectors

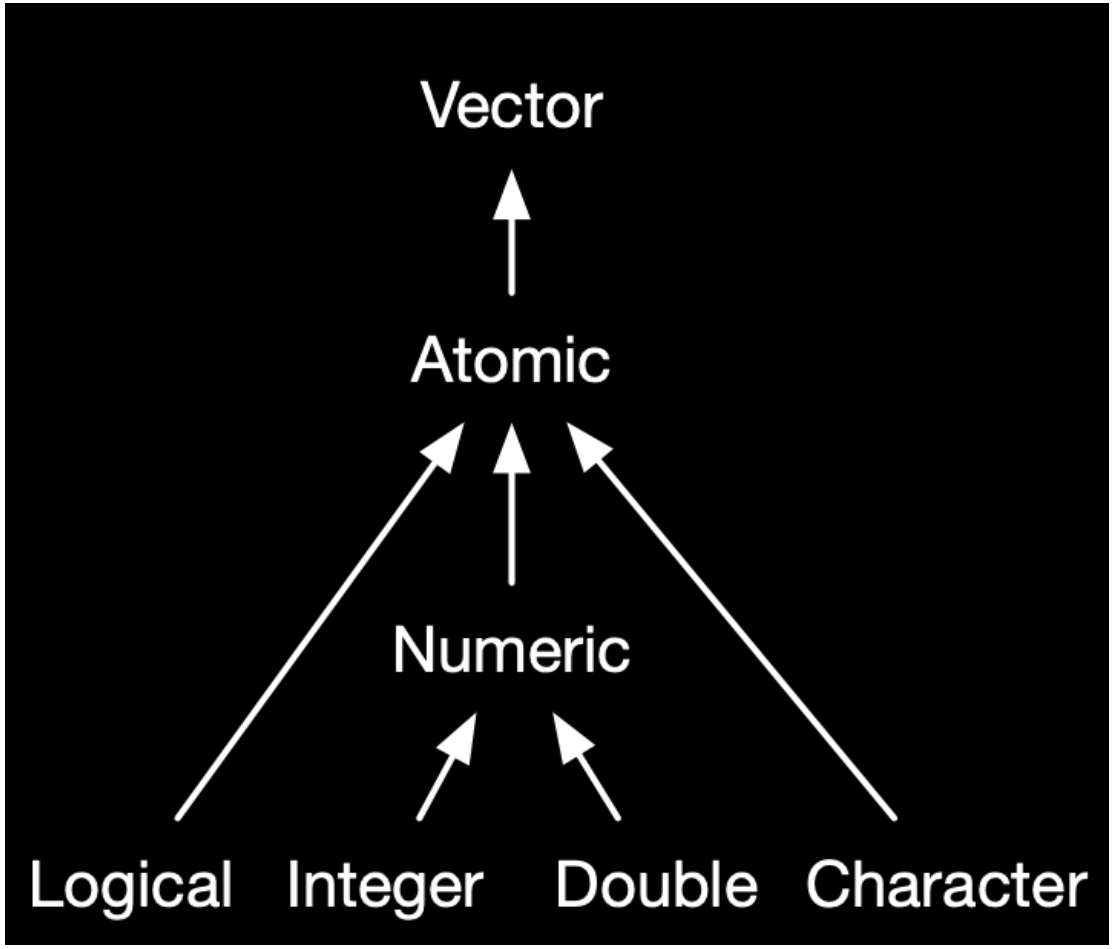


Figure by Wickham (2019) (licensed under CC BY-NC-SA 4.0).

Data structures: vectors

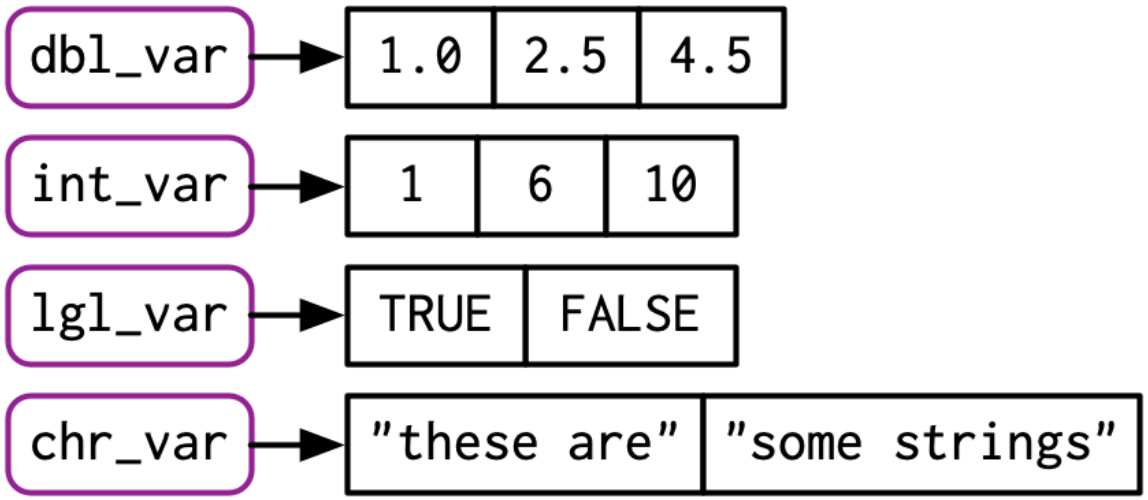


Figure by Wickham (2019) (licensed under CC BY-NC-SA 4.0).

Data structures: vectors

Example:

```
hometown <- c("St.Gallen", "Basel", "St.Gallen")
hometown
```

```
## [1] "St.Gallen" "Basel"     "St.Gallen"
```

```
object.size(hometown)
```

```
## 200 bytes
```

Character vectors and memory

```r
x <- c("a", "a", "abc", "d")
```

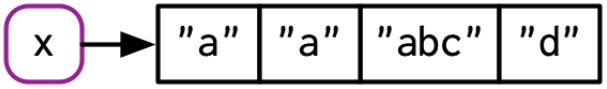


Figure by Wickham (2019) (licensed under CC BY-NC-SA 4.0).

Character vectors and memory

- R uses a global string pool where each element of a character vector is a pointer to a unique string in the pool.

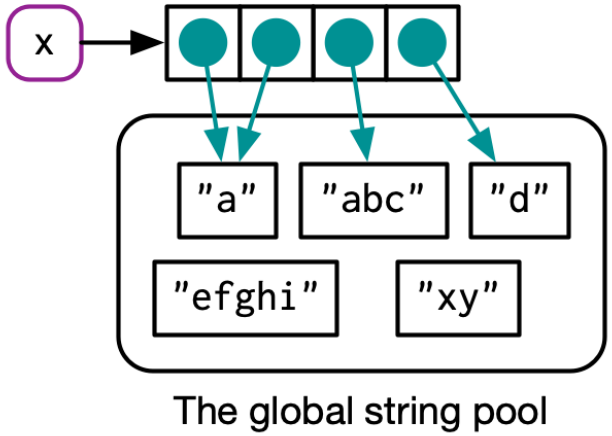


Figure by Wickham (2019) (licensed under CC BY-NC-SA 4.0).

Character vectors and memory

```
ref(x, character = TRUE)
```

```
## ▉ [1:0x558ce11aa528] <chr>
## ├─[2:0x558cdc7f82d0] <string: "a">
## ├─[2:0x558cdc7f82d0]
## ├─[3:0x558ce11057b0] <string: "abc">
## └─[4:0x558cdc986090] <string: "d">
```

Character vectors and memory

- The global string pool saves memory if a string vector is large!

```
obj_size(x)
```

```
## 248 B
```

```
obj_size(rep(x, 100))
```

```
## 3,416 B
```

Data structures: factors

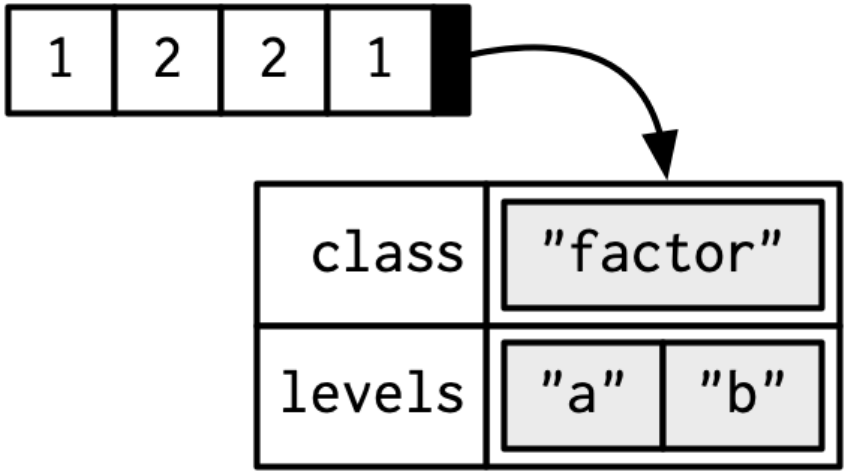


Figure by Wickham (2019) (licensed under CC BY-NC-SA 4.0).

Data structures: factors

Example:

```
hometown_f <- factor(c("St.Gallen", "Basel", "St.Gallen"))
hometown_f
```

```
## [1] St.Gallen Basel     St.Gallen
## Levels: Basel St.Gallen
```

```
object.size(hometown_f)
```

```
## 584 bytes
```

Data structures: Factors

- Certain 'overhead' costs: the structure stored in a factor object is also information (takes up memory)
- Similar as in previous examples: 'overhead' diminishes (relatively) with larger datasets

```
# create a large character vector
hometown_large <- rep(hometown, times = 1000)
# and the same content as factor
hometown_large_f <- factor(hometown_large)
# compare size
object.size(hometown_large)
```

```
## 24168 bytes
```

```
object.size(hometown_large_f)
```

```
## 12568 bytes
```

Data structures: matrices/arrays

- Like (atomic) vectors, but in 2 or more dimensions.

```
my_matrix <- matrix(c(1,2,3,4,5,6), nrow = 3)
my_matrix
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
my_array <- array(c(1,2,3,4,5,6), dim = 3)
my_array
```

```
## [1] 1 2 3
```

Data structures: lists

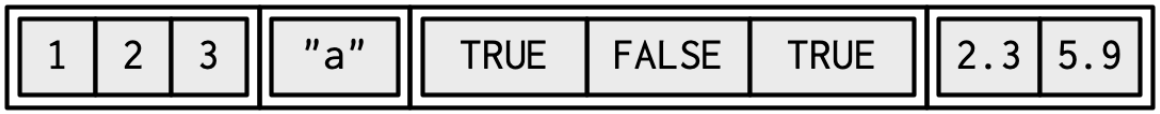


Figure by Wickham (2019) (licensed under CC BY-NC-SA 4.0).

Data structures: lists

```r
l1 <- list(
  1:3,
  "a",
  c(TRUE, FALSE, TRUE),
  c(2.3, 5.9)
)

typeof(l1)
```

```
## [1] "list"
```

```r
str(l1)
```

```
## List of 4
##  $ : int [1:3] 1 2 3
##  $ : chr "a"
##  $ : logi [1:3] TRUE FALSE TRUE
##  $ : num [1:2] 2.3 5.9
```

Lists and memory

```
l1 <- list(1, 2, 3)
```

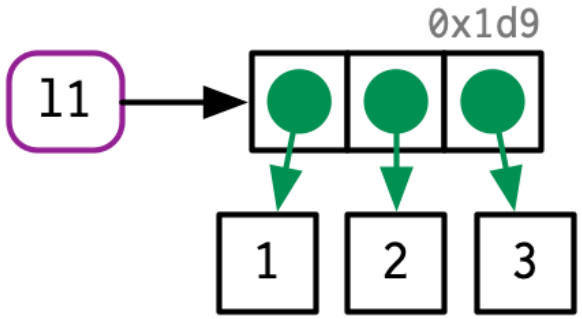


Figure by Wickham (2019) (licensed under CC BY-NC-SA 4.0).

Lists and memory

```
l2 <- l1
```

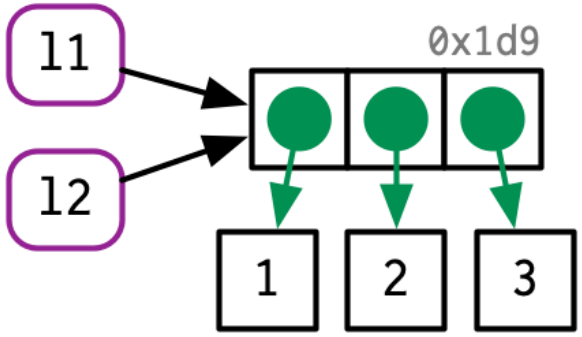


Figure by Wickham (2019) (licensed under CC BY-NC-SA 4.0).

Lists and memory

```
l2[[3]] <- 4
```

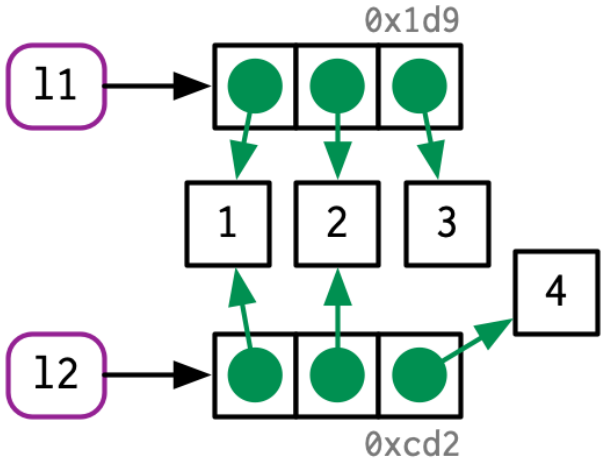


Figure by Wickham (2019) (licensed under CC BY-NC-SA 4.0).

Lists and memory

- **Shallow copy**: list object and its bindings are copied, values pointed to by the bindings not.
- Opposite of a shallow copy is a **deep copy**: contents of every reference are copied.
- Prior to R 3.1.0, copies were always deep copies!
  - 😴😴😴

Data frames, tibbles, and data tables

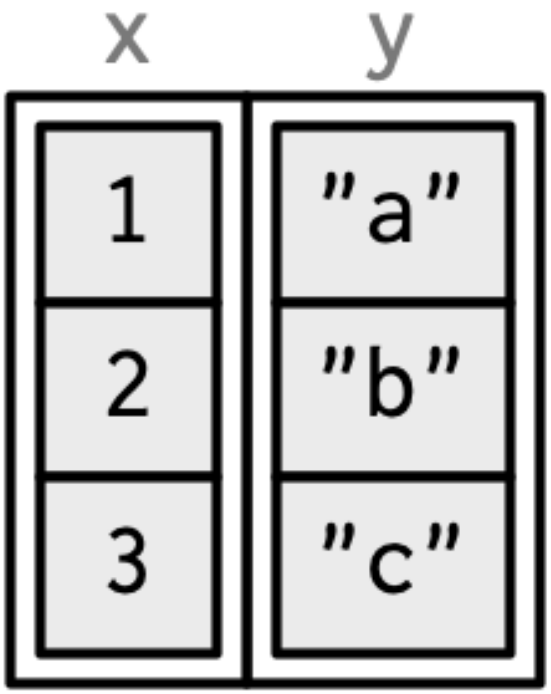


Figure by Wickham (2019) (licensed under CC BY-NC-SA 4.0).

Data frames, tibbles, and data tables

- Classic `data.frame`

```
df <- data.frame(person = c("Alice", "Ben"),
                 age = c(50, 30),
                 gender = c("f", "m"))
df
```

```
##   person age gender
## 1  Alice  50      f
## 2    Ben  30      m
```

Data frames, tibbles, and data tables

- data.table

```
library(data.table)
dt <- data.table(person = c("Alice", "Ben"),
                 age = c(50, 30),
                 gender = c("f", "m"))
dt
```

```
##    person age gender
## 1:  Alice  50      f
## 2:    Ben  30      m
```

Data frames, tibbles, and data tables

- tibble

```
library(tibble)
tib <- tibble(person = c("Alice", "Ben"),
              age = c(50, 30),
              gender = c("f", "m"))
tib
```

```
## # A tibble: 2 x 3
##   person   age gender
##   <chr>  <dbl> <chr>
## 1 Alice     50 f
## 2 Ben       30 m
```

Data frames and memory

```
d1 <- data.frame(x = c(1, 5, 6), y = c(2, 4, 3))
```

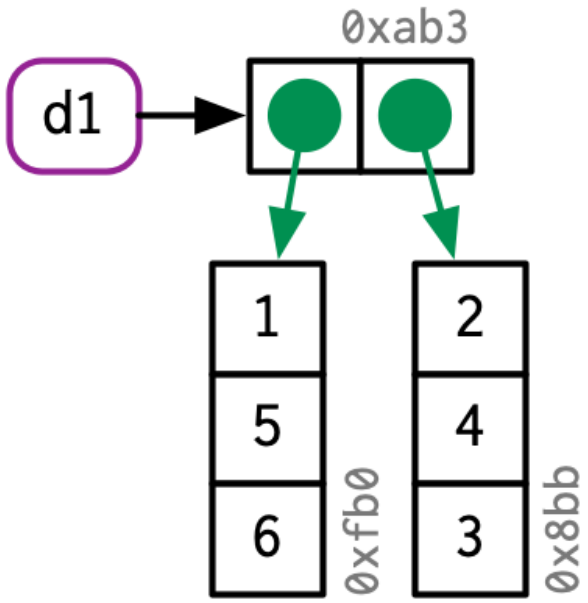


Figure by Wickham (2019) (licensed under CC BY-NC-SA 4.0).

Data frames and memory

- Modify one column: only one column needs to be copied.

```
d2 <- d1
d2[, 2] <- d2[, 2] * 2
```

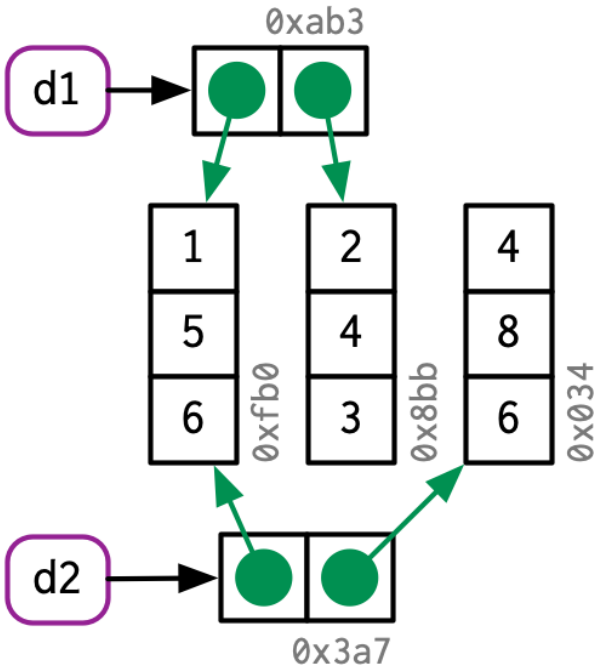


Figure by Wickham (2019) (licensed under CC BY-NC-SA 4.0).

Data frames and memory

- Modify one row: **all** columns need to be copied.

```
d3 <- d1
d3[1, ] <- d3[1, ] * 3
```

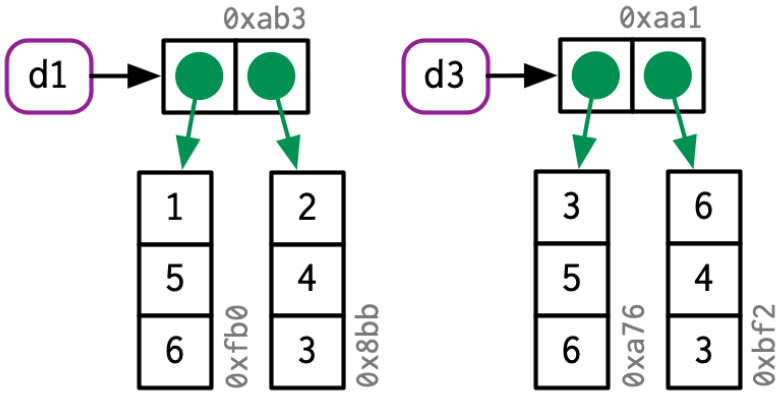


Figure by Wickham (2019) (licensed under CC BY-NC-SA 4.0).

Programming with (Big) Data in R

Typical Programming Tasks

- Procedures to import/export data.
- Procedures to clean and filter data.
- Implement functions for statistical analysis.

R-tools to investigate performance/resource allocation

package	function	purpose
utils	`object.size()`	Provides an estimate of the memory that is being used to store an R object.
pryr	`object_size()`	Works similarly to `object.size()`, but counts more accurately and includes the size of environments.
pryr	`compare_size()`	Makes it easy to compare the output of object_size and object.size.
pryr	`mem_used()`	Returns the total amount of memory (in megabytes) currently used by R.
pryr	`mem_change()`	Shows the change in memory (in megabytes) before and after running code.
base	`system.time()`	Returns CPU (and other) times that an R

Building blocks for programming with big data

- Several basic functions and packages: Which one to use?
- Example: Data import.
  - `utils::read.csv()`
  - `data.table::fread()`

Building blocks for programming with big data

```
# read a CSV-file the 'traditional way'
flights <- read.csv("../data/flights.csv")
class(flights)
```

```
## [1] "data.frame"
```

```
# alternative (needs the data.table package)
library(data.table)
flights <- fread("../data/flights.csv")
class(flights)
```

```
## [1] "data.table" "data.frame"
```

Building blocks for programming with big data

```
system.time(flights <- read.csv("../data/flights.csv"))
```

```
##    user  system elapsed
##   1.336   0.004   1.340
```

```
system.time(flights <- fread("../data/flights.csv"))
```

```
##    user  system elapsed
##   0.374   0.000   0.066
```

Writing efficient code

- Memory allocation (before looping)
- Vectorization (different approaches)
- Beyond R

Loops: Memory allocation before looping

```
# naïve implementation
sqrt_vector <-
     function(x) {
          output <- c()
          for (i in 1:length(x)) {
               output <- c(output, x[i]^(1/2))
          }

          return(output)
     }
```

Loops: Memory allocation before looping

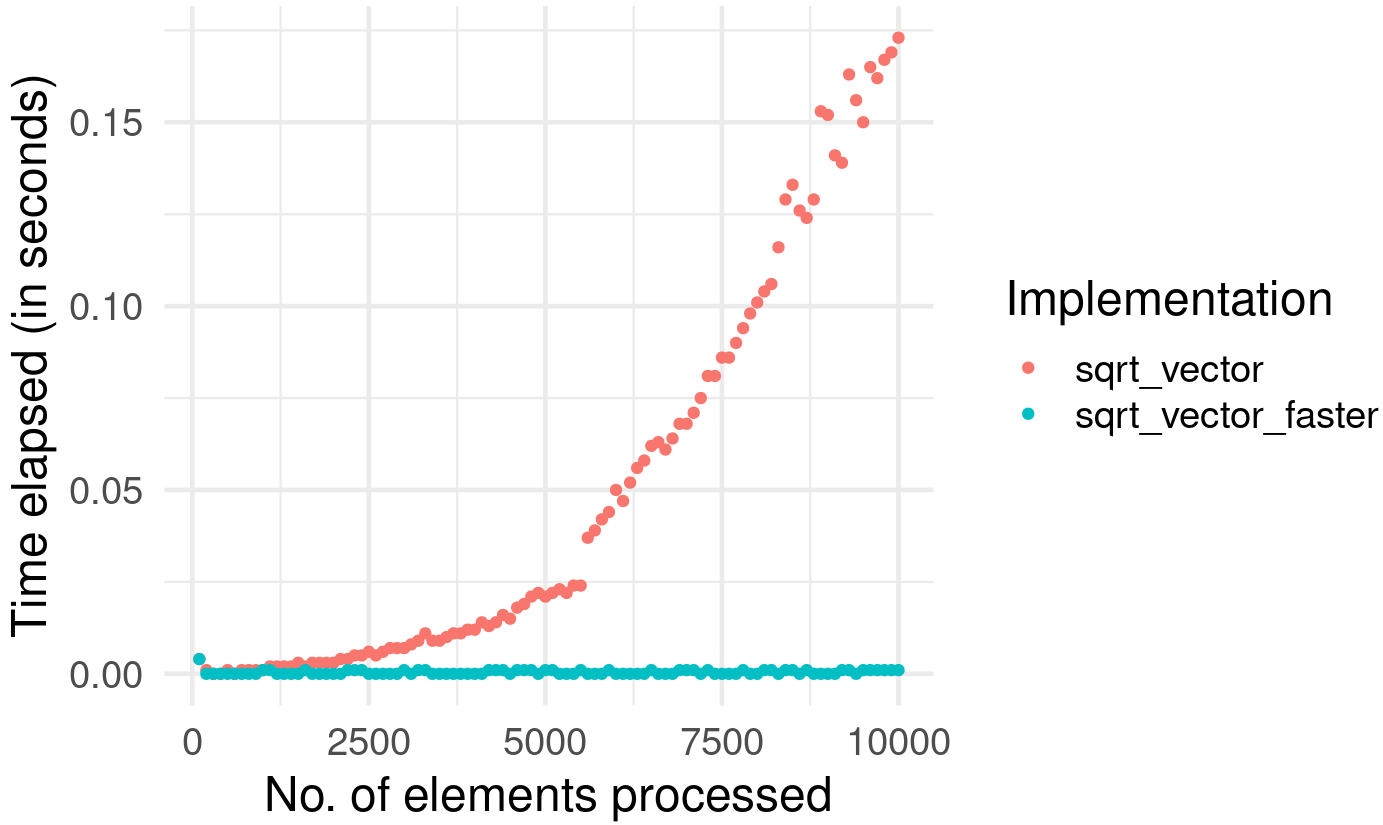
```
# implementation with pre-allocation of memory
sqrt_vector_faster <-
     function(x) {
          output <- rep(NA, length(x))
          for (i in 1:length(x)) {
               output[i] <-  x[i]^(1/2)
          }

          return(output)
     }
```

Loops: Memory allocation before looping

Test it!

```
# the different sizes of the vectors we will put into the two functions
input_sizes <- seq(from = 100, to = 10000, by = 100)
# create the input vectors
inputs <- sapply(input_sizes, rnorm)

# compute ouputs for each of the functions
output_slower <-
     sapply(inputs,
            function(x){ system.time(sqrt_vector(x))["elapsed"]
                 }
            )
output_faster <-
     sapply(inputs,
            function(x){ system.time(sqrt_vector_faster(x))["elapsed"]
                 }
            )
```

Loops: Memory allocation before looping



Loops: Avoid unnecessary copying

- Substract a number from each column of a large `data.frame`.
- Very slow…

```
x <- data.frame(matrix(runif(5 * 1e4), ncol = 5))
numbers <- rnorm(5)

for (i in 1:5) {
  x[[i]] <- x[[i]] - numbers[i]
}
```

Loops: Avoid unnecessary copying

- Problem: each iteration of the loop copies the `data.frame`.
- Copying means additional memory allocation.

```
cat(tracemem(x), "\n")
```

```
## <0x558ce33a7558>
```

```
for (i in 1:5) {
  x[[i]] <- x[[i]] - numbers[i]
}
```

```
## tracemem[0x558ce33a7558 -> 0x558ce3426c88]: eval eval withVisible withCallingHandlers handle timi
## tracemem[0x558ce3426c88 -> 0x558ce3426f28]: [[<-.data.frame [[<- eval eval withVisible withCallir
## tracemem[0x558ce3426f28 -> 0x558ce3427008]: eval eval withVisible withCallingHandlers handle timi
## tracemem[0x558ce3427008 -> 0x558ce34270e8]: [[<-.data.frame [[<- eval eval withVisible withCallir
## tracemem[0x558ce34270e8 -> 0x558ce3427158]: eval eval withVisible withCallingHandlers handle timi
## tracemem[0x558ce3427158 -> 0x558ce3427238]: [[<-.data.frame [[<- eval eval withVisible withCallir
## tracemem[0x558ce3427238 -> 0x558ce3427318]: eval eval withVisible withCallingHandlers handle timi
## tracemem[0x558ce3427318 -> 0x558ce34273f8]: [[<-.data.frame [[<- eval eval withVisible withCallir
## tracemem[0x558ce34273f8 -> 0x558ce34274d8]: eval eval withVisible withCallingHandlers handle timi
## tracemem[0x558ce34274d8 -> 0x558ce3427548]: [[<-.data.frame [[<- eval eval withVisible withCallir
```

Loops: Avoid unnecessary copying

- Solution: store data (columns) in list.
- Uses internal C code and avoids additional copies.

```
y <- as.list(x)
```

```
## tracemem[0x558ce3427548 -> 0x558ce762f7d8]: as.list.data.frame as.list eval eval withVisible with
```

```
cat(tracemem(y), "\n")
```

```
## <0x558ce762f7d8>
```

```
for (i in 1:5) {
  y[[i]] <- y[[i]] - numbers[i]
}
```

```
## tracemem[0x558ce762f7d8 -> 0x558ce76035f8]: eval eval withVisible withCallingHandlers handle timi
```

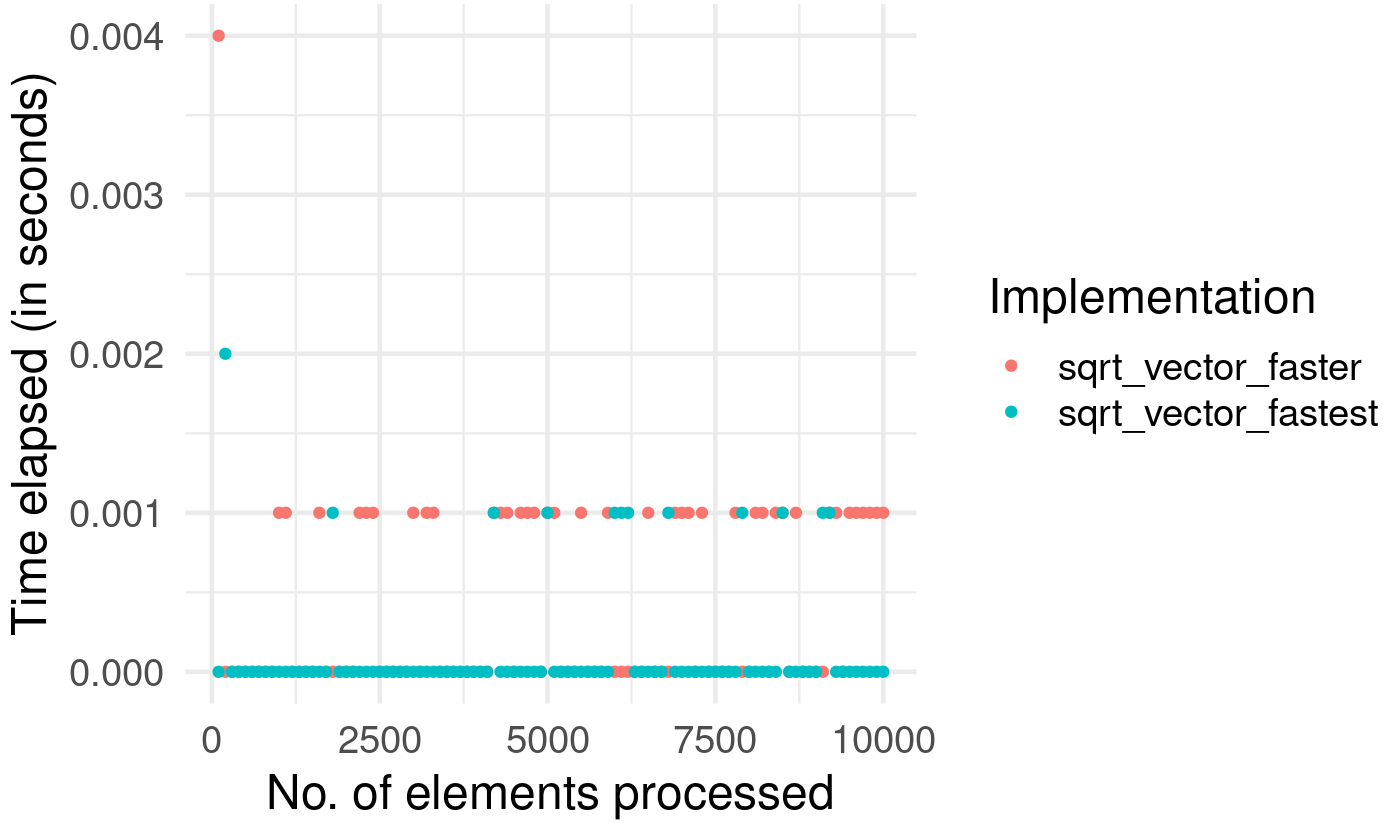
Vectorization

- “In R, everything is a vector…”
- Directly operate on vectors, not elements.
- Avoid unnecessary repetition of ‘preparatory steps’.

Vectorization: Example

```
# implementation with vectorization
sqrt_vector_fastest <-
     function(x) {
               output <-  x^(1/2)
          return(output)
     }

# speed test
output_fastest <-
     sapply(inputs,
            function(x){ system.time(sqrt_vector_fastest(x))["elapsed"]
                 }
            )
```

Vectorization: Example



Vectorization: `apply`-type functions vs loops

- Apply a function to each element of a vector/list.
- For example, `lapply()`.

Example

- Read several data files into R.
- Example data source: Health News in Twitter Data Set by Karami et al. (2017).
- Loop vs `lapply()`, vs `Vectorization()`

Example: Preparations

```
# load packages
library(data.table)

# get a list of all file-paths
textfiles <- list.files("../data/twitter_texts", full.names = TRUE)
```

Example: `for`-loop approach

```
# prepare loop
all_texts <- list()
n_files <- length(textfiles)
length(all_texts) <- n_files
# read all files listed in textfiles
for (i in 1:n_files) {
     all_texts[[i]] <- fread(textfiles[i])
}
```

Example: for-loop approach

Check the results

```
# combine all in one data.table
twitter_text <- rbindlist(all_texts)
# check result
str(twitter_text)
```

```
## Classes 'data.table' and 'data.frame':   42422 obs. of  3 variables:
##  $ V1:integer64 585978391360221184 585947808772960257 585947807816650752 585866060991078401 58579
##  $ V2: chr  "Thu Apr 09 01:31:50 +0000 2015" "Wed Apr 08 23:30:18 +0000 2015" "Wed Apr 08 23:30:1
##  $ V3: chr  "Breast cancer risk test devised http://bbc.in/1CimpJF" "GP workload harming care - B
##  - attr(*, ".internal.selfref")=<externalptr>
```

Example: `lapply` approach

```
# prepare loop
all_texts <- lapply(textfiles, fread)
# combine all in one data.table
twitter_text <- rbindlist(all_texts)
# check result
str(twitter_text)
```

```
## Classes 'data.table' and 'data.frame':   42422 obs. of  3 variables:
##  $ V1:integer64 585978391360221184 585947808772960257 585947807816650752 585866060991078401 5857
##  $ V2: chr  "Thu Apr 09 01:31:50 +0000 2015" "Wed Apr 08 23:30:18 +0000 2015" "Wed Apr 08 23:30:1
##  $ V3: chr  "Breast cancer risk test devised http://bbc.in/1CimpJF" "GP workload harming care - B
##  - attr(*, ".internal.selfref")=<externalptr>
```

Example: `Vectorization` approach

```
# initiate the import function
import_file <-
     function(x) {
          parsed_x <- fread(x)
          return(parsed_x)
     }

# 'vectorize' it
import_files <- Vectorize(import_file, SIMPLIFY = FALSE)
```

Example: Vectorization approach

```
# Apply the vectorized function
all_texts <- import_files(textfiles)
twitter_text <- rbindlist(all_texts)
# check the result
str(twitter_text)
```

```
## Classes 'data.table' and 'data.frame':   42422 obs. of  3 variables:
##  $ V1:integer64 585978391360221184 585947808772960257 585947807816650752 585866060991078401 5857
##  $ V2: chr  "Thu Apr 09 01:31:50 +0000 2015" "Wed Apr 08 23:30:18 +0000 2015" "Wed Apr 08 23:30:1
##  $ V3: chr  "Breast cancer risk test devised http://bbc.in/1CimpJF" "GP workload harming care - B
##  - attr(*, ".internal.selfref")=<externalptr>
```

Profiling and Benchmarking

Profiling

- Use a ‘profiler’ to understand code performance.
- Get an overview over which parts of a program need how much memory and how much execution time.

Profiling with `profvis`

A simple nested function (with clearly defined execution time):

```
# implement function
f <- function() {
  pause(0.1)
  g()
  h()
}
g <- function() {
  pause(0.1)
  h()
}
h <- function() {
  pause(0.1)
}
```

Profiling with `profvis`

```
# load package with profiler
library(profvis)
# get performance profile of function
profvis(f())
```

Benchmarking with `bench::mark()`

- Alternative tool to measure execution time (see `microbenchmark` in previous lectures)
- Recall: execution time is not deterministic (it comes with statistical error).
- Benchmarking means runnign the code several times to get a distribution of execution times.

Benchmarking with bench::mark()

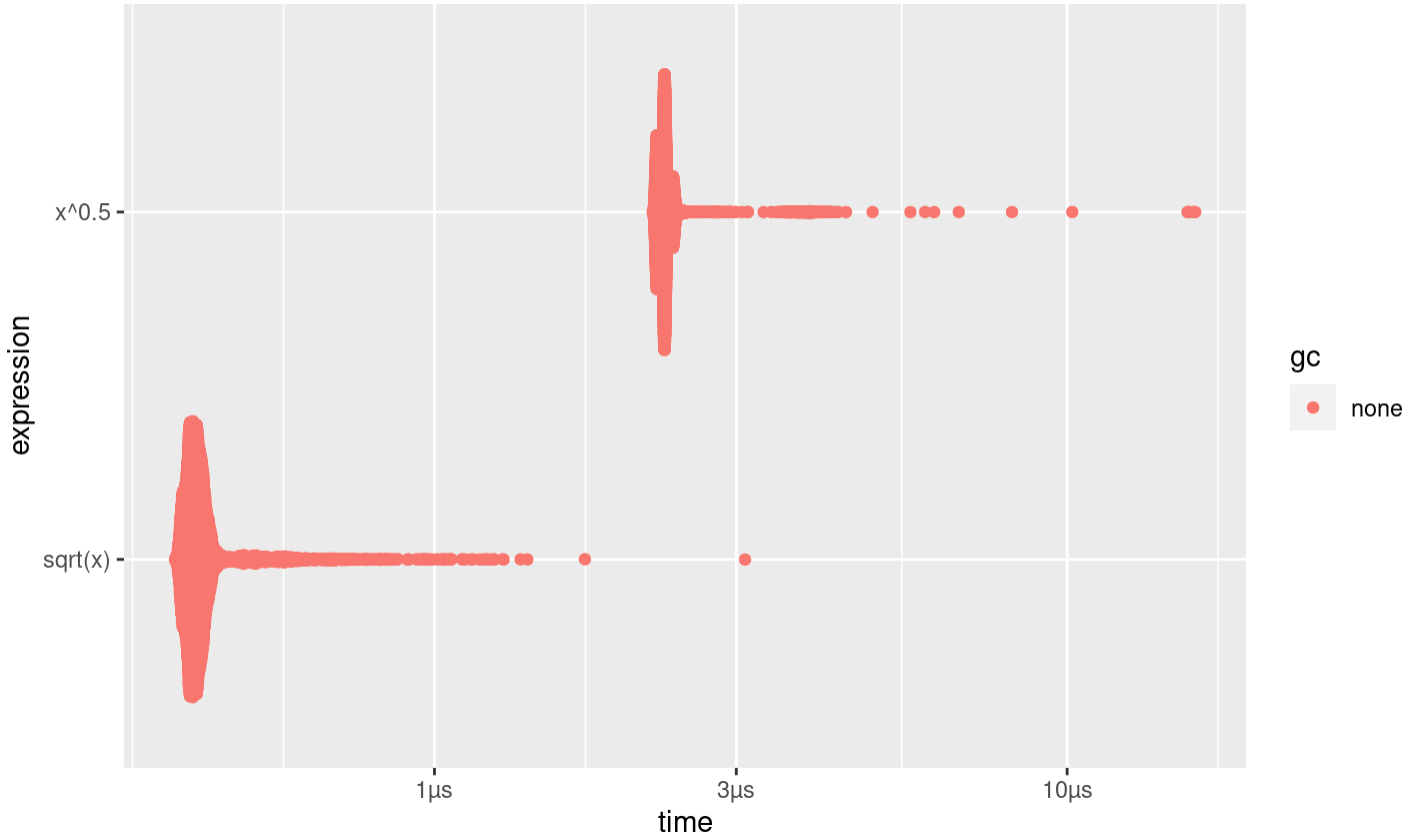
```
# load package
library(bench)

# run squareroot example
# primitive (C) sqrt vs. 'own implementation'
x <- runif(100)
(lb <- bench::mark(
  sqrt(x),
  x ^ 0.5,
  memory = FALSE
))
```

```
## # A tibble: 2 x 6
##   expression      min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 sqrt(x)    389.06ns  419.1ns  2312000.        NA        0
## 2 x^0.5        2.21µs   2.31µs   425977.        NA        0
```

Benchmarking with bench::mark()

```
plot(lb)
```

```
## Loading required namespace: tidyr
```



Improving Performance

Improving performance

- Bottleneck(s) identified, what now?
- See previous examples for typical problems in a data analytics context.
- Vast variety of potential bottlenecks. Hard to give general advice.

Programming with Big Data

1. Which basic (already implemented) R functions are more or less suitable as building blocks for the program?
2. How can we exploit/avoid some of R's lower-level characteristics in order to implement efficient functions?
3. Is there a need to interface with a lower-level programming language in order to speed up the code? (advanced topic)

- Independent of **how** we write a statistical procedure in R (or in any other language, for that matter), is there an **alternative statistical procedure/algorithm** that is faster but delivers approximately the same result.

Issues to keep in mind

- Vectorization.
- Memory: avoid copying, pre-allocate memory.
- Use built in primitive (C) functions (caution: not always faster, if aim is precision).
- Existing solutions: load additional packages (`read.csv()` vs. `data.table::fread()`).
    - Focus of what follows in this course (approach taken in Walkowiak (2016)).

Procedural view and further reading

- Consider Hadley's advice: Wickham (2019): Chapter 24
- Experienced coder? Have a look at R Inferno
- Further reading after this course: The Art of R Programming

R, beyond R

- For advanced programmers, R offers various options to directly make use of compiled programs (for example, written in C, C++, or FORTRAN).
- Several of the core R functions are implemented in one of these lower-level programming languages.

R, beyond R

Have a look at a function's source code!

```
import_file
```

```
## function(x) {
##           parsed_x <- fread(x)
##           return(parsed_x)
##       }
## <bytecode: 0x558ce5889e80>
```

R, beyond R

Have a look at a function's source code!

```
sum
```

```
## function (..., na.rm = FALSE)  .Primitive("sum")
```

References

Karami, Amir, Aryya Gangopadhyay, Bin Zhou, and Hadi Kharrazi. 2017. “Fuzzy Approach Topic Discovery in Health and Medical Corpora.” **International Journal of Fuzzy Systems** 20 (4): 1334–45.

Walkowiak, Simkon. 2016. **Big Data Analytics with R**. Birmingham, UK: PACKT Publishing.

Wickham, Hadley. 2019. **Advanced R**. Second Edition. Boca Raton, FL: CRC Press.