Informatics

Programming Techniques

Claudio Sartori
Department of Computer Science and Engineering
claudio.sartori@unibo.it
https://www.unibo.it/sitoweb/claudio.sartori/

Contents

- Summarizing data
- Vectorization
- Subsetting
- Inspecting the structure of a variable

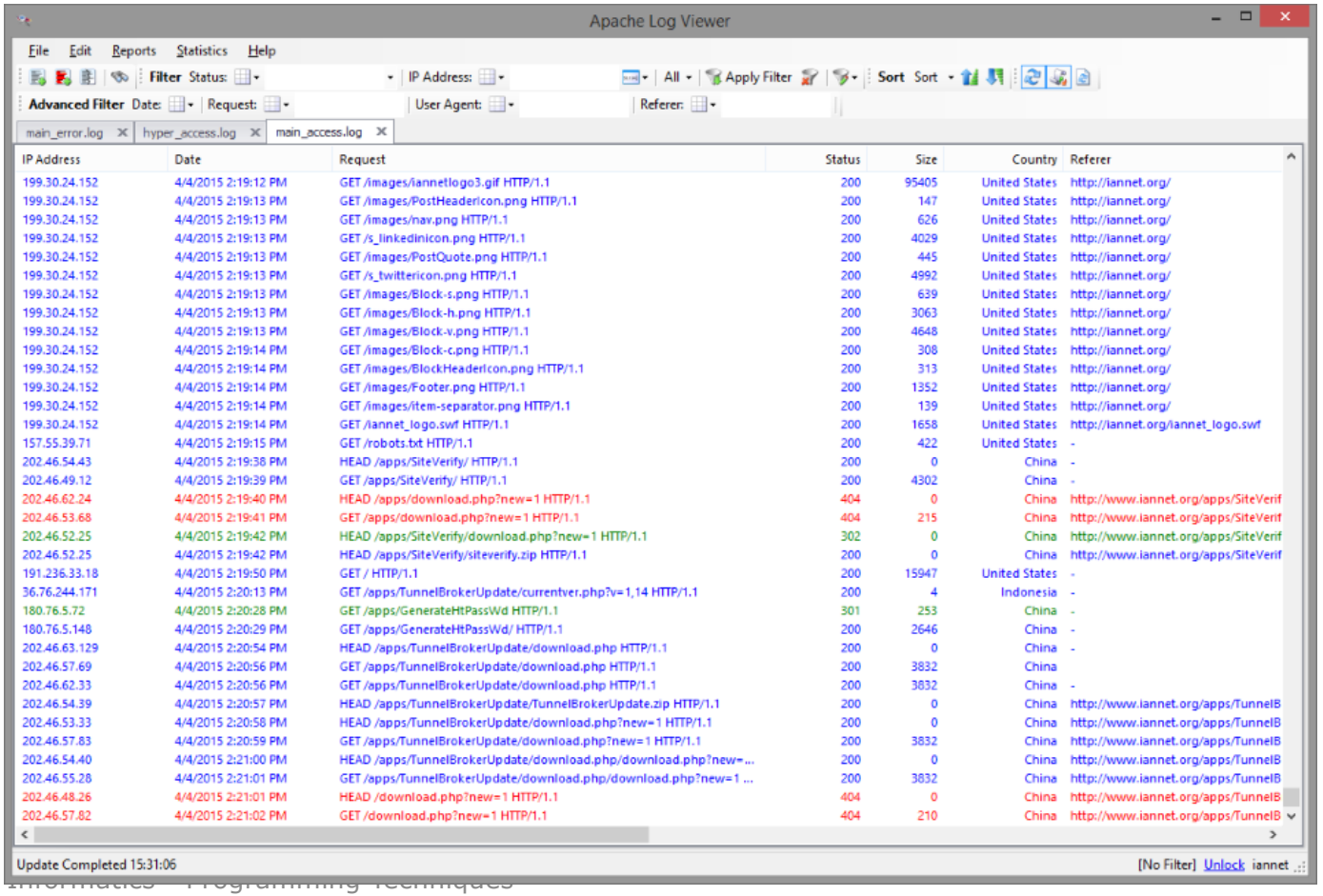
Summarizing data

- A file with *n* observations
- For each observation several fields, among them a class *key*, with *nKeys* possible different values
- generate a new file with *nKeys* rows
  - *key* – class label
  - *count* – absolute frequencies of the *nKeys* different values in the *n* observations

Summarizing data (ii)

- The italian provinces and their regions
  - very small
- In a web log file, the geographical region of the accessing user
  - possible very large number of observations
  - number of classes varies with granularity of regions
- This kind of problem is extremely common

Web log file


Apache Log Viewer
File Edit Reports Statistics Help
Filter Status: IP Address: All Apply Filter Sort Sort
Advanced Filter Date: Request: User Agent: Referer:
main_error.log hyper_access.log main_access.log
IP Address Date Request Status Size Country Referer
199.30.24.152 4/4/2015 2:19:12 PM GET /images/iannetlogo3.gif HTTP/1.1 200 95405 United States http://iannet.org/
199.30.24.152 4/4/2015 2:19:13 PM GET /images/PostHeaderIcon.png HTTP/1.1 200 147 United States http://iannet.org/
199.30.24.152 4/4/2015 2:19:13 PM GET /images/nav.png HTTP/1.1 200 626 United States http://iannet.org/
199.30.24.152 4/4/2015 2:19:13 PM GET /s_linkedinicon.png HTTP/1.1 200 4029 United States http://iannet.org/
199.30.24.152 4/4/2015 2:19:13 PM GET /images/PostQuote.png HTTP/1.1 200 445 United States http://iannet.org/
199.30.24.152 4/4/2015 2:19:13 PM GET /s_twittericon.png HTTP/1.1 200 4992 United States http://iannet.org/
199.30.24.152 4/4/2015 2:19:13 PM GET /images/Block-s.png HTTP/1.1 200 639 United States http://iannet.org/
199.30.24.152 4/4/2015 2:19:13 PM GET /images/Block-h.png HTTP/1.1 200 3063 United States http://iannet.org/
199.30.24.152 4/4/2015 2:19:13 PM GET /images/Block-v.png HTTP/1.1 200 4648 United States http://iannet.org/
199.30.24.152 4/4/2015 2:19:14 PM GET /images/Block-c.png HTTP/1.1 200 308 United States http://iannet.org/
199.30.24.152 4/4/2015 2:19:14 PM GET /images/BlockHeaderIcon.png HTTP/1.1 200 313 United States http://iannet.org/
199.30.24.152 4/4/2015 2:19:14 PM GET /images/Footer.png HTTP/1.1 200 1352 United States http://iannet.org/
199.30.24.152 4/4/2015 2:19:14 PM GET /images/item-separator.png HTTP/1.1 200 139 United States http://iannet.org/
199.30.24.152 4/4/2015 2:19:14 PM GET /iannet_logo.swf HTTP/1.1 200 1658 United States http://iannet.org/iannet_logo.swf
157.55.39.71 4/4/2015 2:19:15 PM GET /robots.txt HTTP/1.1 200 422 United States -
202.46.54.43 4/4/2015 2:19:38 PM HEAD /apps/SiteVerify/ HTTP/1.1 200 0 China -
202.46.49.12 4/4/2015 2:19:39 PM GET /apps/SiteVerify/ HTTP/1.1 200 4302 China -
202.46.62.24 4/4/2015 2:19:40 PM HEAD /apps/download.php?new=1 HTTP/1.1 404 0 China http://www.iannet.org/apps/SiteVerif
202.46.53.68 4/4/2015 2:19:41 PM GET /apps/download.php?new=1 HTTP/1.1 404 215 China http://www.iannet.org/apps/SiteVerif
202.46.52.25 4/4/2015 2:19:42 PM HEAD /apps/SiteVerify/download.php?new=1 HTTP/1.1 302 0 China http://www.iannet.org/apps/SiteVerif
202.46.52.25 4/4/2015 2:19:42 PM HEAD /apps/SiteVerify/siteverify.zip HTTP/1.1 200 0 China http://www.iannet.org/apps/SiteVerif
191.236.33.18 4/4/2015 2:19:50 PM GET / HTTP/1.1 200 15947 United States -
36.76.244.171 4/4/2015 2:20:13 PM GET /apps/TunnelBrokerUpdate/currentver.php?v=1,14 HTTP/1.1 200 4 Indonesia -
180.76.5.72 4/4/2015 2:20:28 PM GET /apps/GenerateHtPassWd HTTP/1.1 301 253 China -
180.76.5.148 4/4/2015 2:20:29 PM GET /apps/GenerateHtPassWd/ HTTP/1.1 200 2646 China -
202.46.63.129 4/4/2015 2:20:54 PM HEAD /apps/TunnelBrokerUpdate/download.php HTTP/1.1 200 0 China -
202.46.57.69 4/4/2015 2:20:56 PM GET /apps/TunnelBrokerUpdate/download.php HTTP/1.1 200 3832 China
202.46.62.33 4/4/2015 2:20:56 PM GET /apps/TunnelBrokerUpdate/download.php HTTP/1.1 200 3832 China -
202.46.54.39 4/4/2015 2:20:57 PM HEAD /apps/TunnelBrokerUpdate/TunnelBrokerUpdate.zip HTTP/1.1 200 0 China http://www.iannet.org/apps/TunnelB
202.46.53.33 4/4/2015 2:20:58 PM HEAD /apps/TunnelBrokerUpdate/download.php?new=1 HTTP/1.1 200 0 China http://www.iannet.org/apps/TunnelB
202.46.57.83 4/4/2015 2:20:59 PM GET /apps/TunnelBrokerUpdate/download.php?new=1 HTTP/1.1 200 3832 China http://www.iannet.org/apps/TunnelB
202.46.54.40 4/4/2015 2:21:00 PM HEAD /apps/TunnelBrokerUpdate/download.php/download.php?new=... 200 0 China http://www.iannet.org/apps/TunnelB
202.46.55.28 4/4/2015 2:21:01 PM GET /apps/TunnelBrokerUpdate/download.php/download.php?new=1 ... 200 3832 China http://www.iannet.org/apps/TunnelB
202.46.48.26 4/4/2015 2:21:01 PM HEAD /download.php?new=1 HTTP/1.1 404 0 China http://www.iannet.org/apps/TunnelB
202.46.57.82 4/4/2015 2:21:02 PM GET /download.php?new=1 HTTP/1.1 404 210 China http://www.iannet.org/apps/TunnelB
Update Completed 15:31:06
[No Filter] Unlock iannet

Summary tables

Log file

...	...	...	Country	...
...	...	...	Sweden	...
...	...	...	Argentina	...
...	...	...	USA	...
...	...	...	Argentina	...
...	...	...	Italy	...
...	...	...		...
...	...	...		...
...	...	...		...
...	...	...		...

Summary table for Country

Country	Count
Argentina	2
Italy	1
Sweden	1
USA	1

summary table after the scan of the first 4 rows in the log file

the scan of the fifth row generates the insertion of a new line in the summary table

Summarizing data: Solution strategies

see the "summary_table" example in R

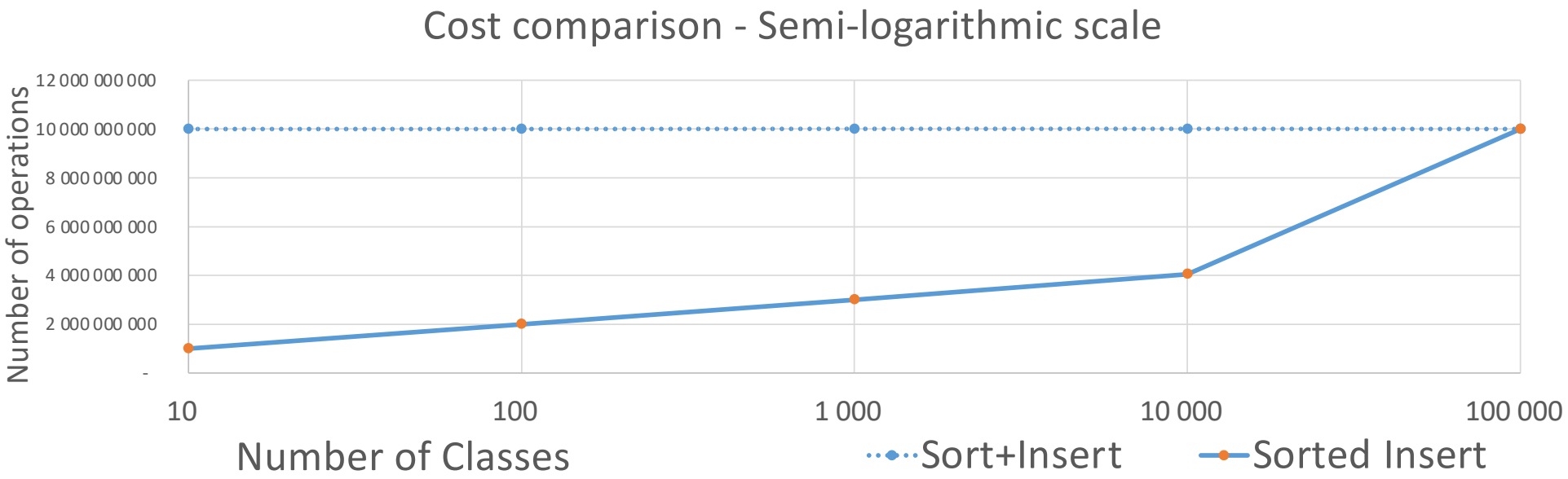
A. *sort + insert*
   1) sort the input data on the summary key
   2) scan input data and
      1) whenever a new key is found insert new key in output data and initialize count to 1
      2) whenever an old key is found increment its count
   3) with this strategy the output data are implicitely sorted on the summary key

B. *sorted insert*
   1) scan input data and
      1) whenever a new key is found insert new key in output data ensuring to keep it sorted and initialize count to 1
      2) whenever an old key is found increment its count
   2) whith this strategy the output data are kept sorted when a new insertion is performed

Which strategy is better? computational complexity

A. full sort = n * log n
   insert = nKey
   updates = n

B. insert = nKey * nKey / 2
   updates = n * log nKey



Vectorization in R

A short history of the loop

'Looping', 'cycling', 'iterating' or just replicating instructions is quite an old practice that originated well before the invention of computers. It is nothing more than automating a certain multi step process by organizing sequences of actions ('batch' processes) and grouping the parts in need of repetition. Even for 'calculating machines', as computers were called in the pre-electronics era, pioneers like Ada Lovelace, Charles Babbage and others, devised methods to implement such iterations.

by DataCamp©

A short history of the loop (ii)

In modern programming languages, where a program is a sequence of instructions, the loop is an evolution of the 'jump' instruction that asks the machine to jump to a predefined label along a sequence of instructions. The beloved and nowadays deprecated goto found in Assembler, Basic, Fortran and similar programming languages of that generation, was a mean to tell the computer to jump to a specified instruction label: so, by placing that label before the location of the goto instruction, one obtained a repetition of the desired instruction a loop. Yet, the control of the sequence of operations (the program flow control) would soon become cumbersome with the increase of goto and corresponding labels within a program. Specifically, one risked of loosing control of all the gotos that transferred control to a specific label

by DataCamp©

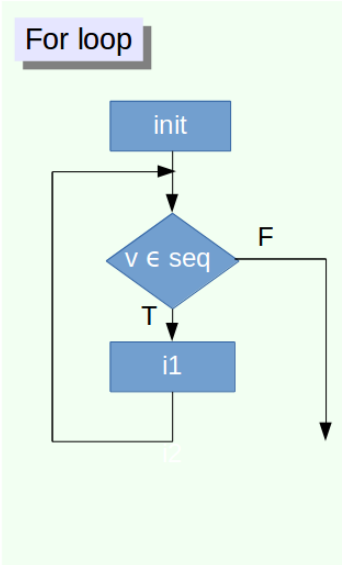
A short history of the loop (iii)

All modern programming languages were equipped with special constructs that allowed the repetition of instructions or blocks of instructions. In the present days, a number of variants exist:

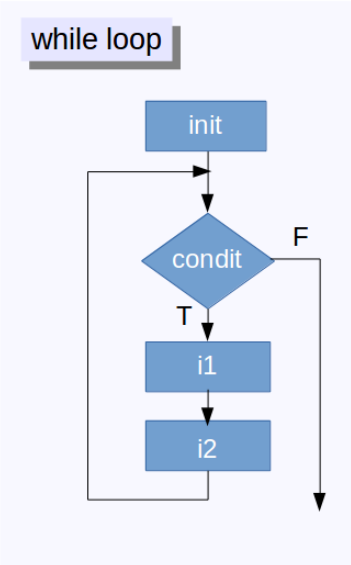
1. *Loops that execute for a prescribed number of times, as controlled by a counter or an index, incremented at each iteration cycle; these pertain to the for family;*
2. *Loops based on the onset and verification of a logical condition (for example, the value of a control variable), tested at the start or at the end of the loop construct; these variants belong to the while or repeat family of loops, respectively.*

by DataCamp©

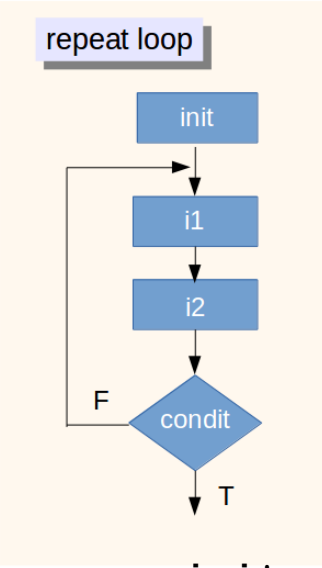
Loops in the R language




```
… init ...
for (v in ...seq ...){
   ... instructions ...
}
```




```
… init ...
while(...condit...){
   ... instructions ...
   ... modify condit ...
}
```




```
… init ...
repeat{
   ... instructions ...
   ... modify condit...
   if (condit) break
}
```

Vectorization, why?

- R is interpreted
- all the details about variable definition (type, size, etc) are taken care by the interpreter, no need to specify that a number is a floating point or to allocate memory using (say) a pointer in memory
- the R interpreter 'understands' these issues from the context as you enter your commands, but it does so on a command-by-command basis
- necessary to deal with such definitions (type, structure, allocation etc) every time you issue a given command, even if you just repeat it
  - and the same inside a loop
- A compiler instead (e.g. C, Fortran, …), solves literally all the definitions and declarations at compilation time over the entire code; the latter is translated into a compact and optimized binary code, before you try to execute anything.
- R functions are written in one of these lower-level languages, they are more efficient
  - in practice, if one looked at the low level code, one would discover calls to C or C++, usually implemented within what is called a wrapper code

Vectorization, why? (ii)

- in languages supporting vectorization (like R or Matlab) every instruction making use of a numeric datum, acts on an object that is natively defined as a vector, even if only made of one element
- this is the default when you define (say) a single numeric variable:
  - its inner representation in R will always be a vector, albeit made of one number only

Vectorization

- express operations on vectors as a single statement, instead of as a loop
- under the hood, loops continue to exist, but at the lower and much faster C/C++ compiled level
- the advantage of having a vector means that the definitions are solved by the interpreter only once, on the entire vector, irrespective of its size (number of elements), in contrast to a loop performed on a scalar, where definitions, allocations, etc, need to be done on a element by element basis, and this is slower

Implicit Vectorization

- standard arithmetic operators with special behavior on matrix/vector variables
  - **`vector + vector`**
  - **`matrix + matrix`**
  - **`matrix + vector`**
  - **`vector/matrix + scalar`**
  - same with other arithmetic operators, including power
  - if the objects are of different size → replication
    - a vector is seen like a one-column matrix
      - **`matrix + vector`** adds the vector to all the matrix columns
        - provided that they are of the same size
  - same with the logical operators for conjunction, disjunction, negation
    - & | !

Vectorized functions

- typical functions implemented in the base R systems
- **`sum()`**
  - sums all the elements of its argument
  - can be used to obtain interesting combinations
    - what happens if you apply sum to the product of two vectors?
- **`mean()`**
  - arithmetic mean
- what happens if you divide the sum of the product of two vectors by the sum of one of the two vectors?
  - little hint: let's say that the first vector is the grade you got in your exams, the second is the number of cfu for each exam passed (in the same order)

Stop and reflect

- of course the result of vectorized operators can be obtained with loops
- the only difference is speed
- R has lots of pre-defined functions based on vectorization
- many problems solved in this course have better solutions with vectorization
- we have seen the loop-based solution to understand *programming*

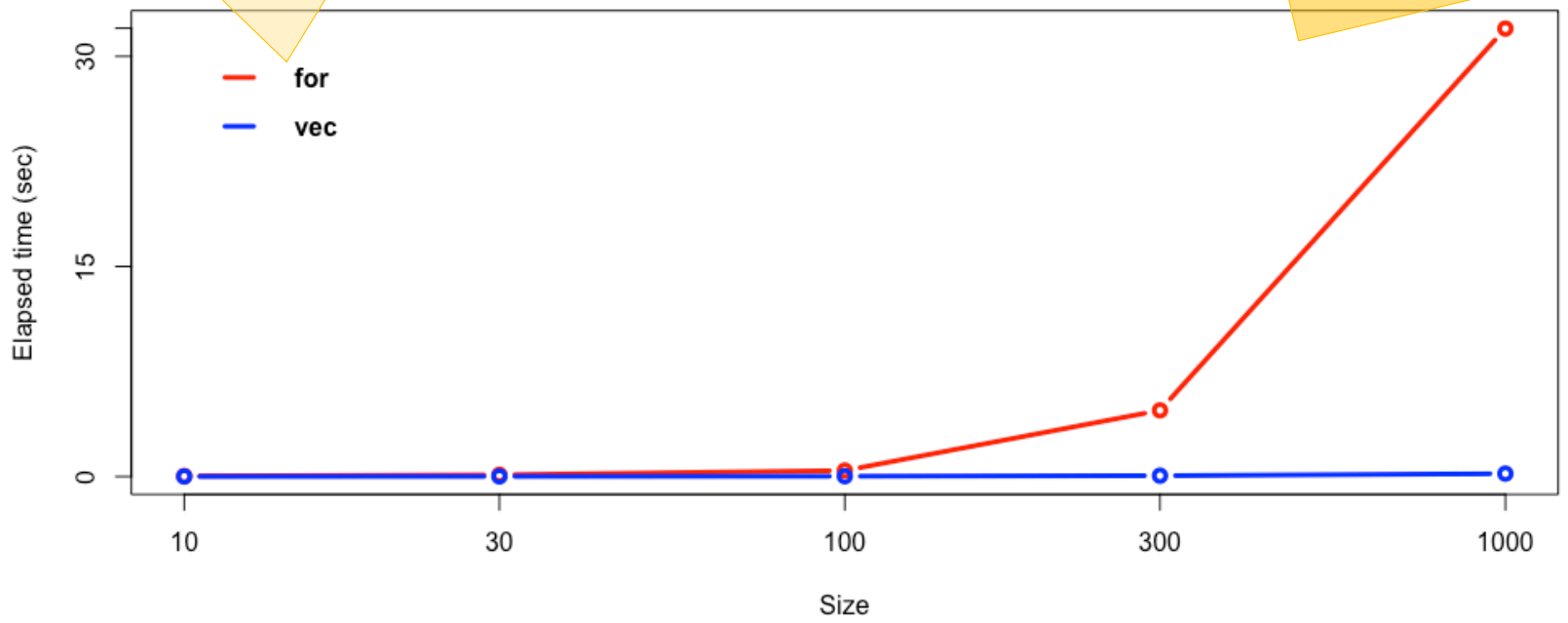
```
for (i in 1:n) {
    for (j in 1:m) {
      mydframe[i, j] <-
         mydframe[i, j] +
         10 * sin(0.75 * pi)
    }
}
```

```
mydframe<-mydframe + 10*sin(0.75*pi)
```

Speed comparison – varying `n`

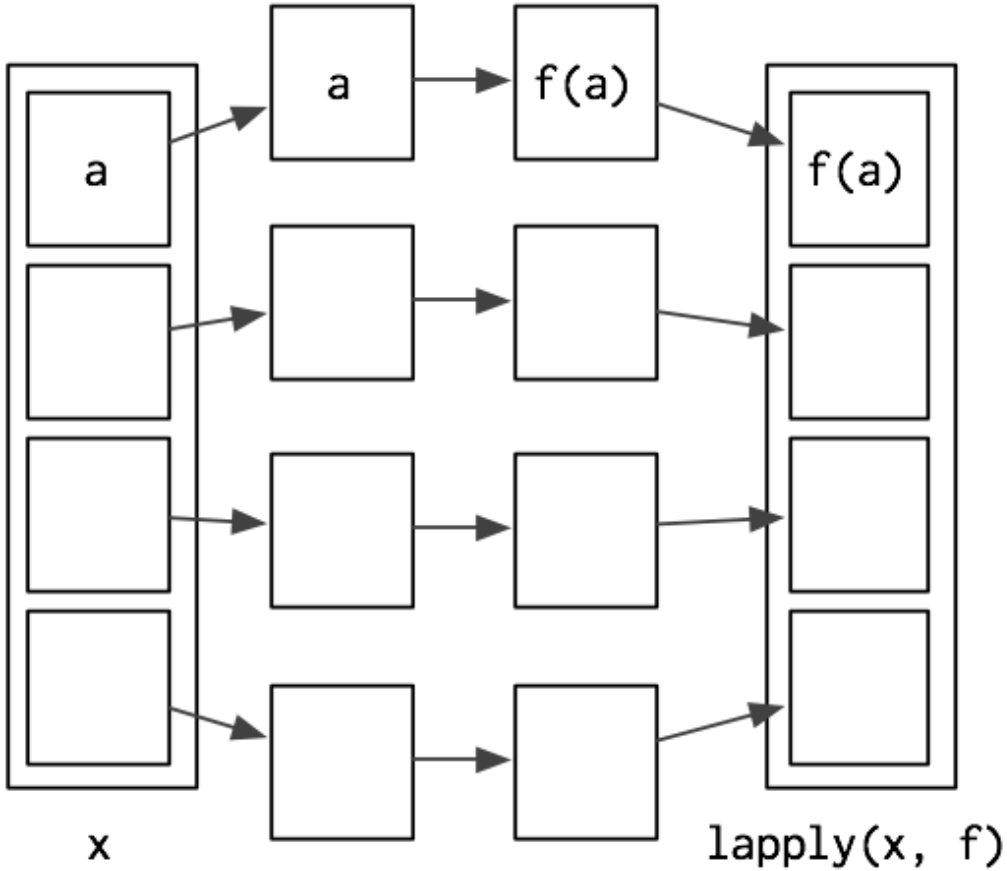
w.r.t. previous page
for: code on the left side
vec: code on the right side

see R examples on vectorisation



Functions for intrinsic vectorization

- also called *functionals*
- a functional is a function that takes a variable and a function as input and returns a variable as output
- **`lapply()`**
- takes a function, applies it to each element in a list, and returns the results in the form of a list



Exercise

- prepare a dataframe containing the grades of the students of your program
  - the columns represent the subjects
    - use the exact order you see here https://corsi.unibo.it/laurea/ScienzeStatistiche/insegnamenti/piano?code=8873&year=2019&manifest=it_2019_8873_000_A32_2019
  - the rows represent the students
    - you can use the badge number as the row name
    - insert your data plus a couple of invented rows
    - insert NA for the subjects for which you still didn't pass the exam
- use apply to compute the average for each subject
- use apply to compute the average for each student

The *apply* family

- **`apply(data, dim, FUNCTION)`**
  - for matrices or arrays
  - must include the specification of the dimension of the array/matrix along wich the implicit loop is executed
  - 1 → loop on rows
  - 2 → loop on columns
  - … additional dimensions available for arrays
- **`lapply(data, FUNCTION)`**
  - for lists, returns a list
- **`sapply(data, FUNCTION)`**
  - returns a vector/matrix/array, independently from the structure if input data

apply(data, dim, FUNCTION)

```
m <-matrix(
       c(1,2,3,4,5,6),
       nrow=3)
f <- function(x){...}
v <- vector(length = n)
for (i in 1:nrow(m)){
   v[i] <- f(m[i,])
}
```

```
m <-matrix(
       c(1,2,3,4,5,6),
       nrow=3)
f <- function(x){...}
v <- apply(m, 1, f)
```

sapply(data, dim, FUNCTION)

```
m <-matrix(
       c(1,2,3,4,5,6),
       nrow=3)
f <- function(x){...}
v <- vector(length = n)
for (i in 1:nrow(m)){
   v[i] <- f(m[i,])
}
```

```
m <-matrix(
       c(1,2,3,4,5,6),
       nrow=3)
f <- function(x){...}
v <- lapply(m, f)
```

lapply(data, dim, FUNCTION)

```
m <-matrix(
       c(1,2,3,4,5,6),
       nrow=3)
f <- function(x){...}
v <- vector(length = n)
for (i in 1:nrow(m)){
   v[i] <- f(m[i,])
}
```

```
m <-matrix(
       c(1,2,3,4,5,6),
       nrow=3)
f <- function(x){...}
v <- unlist(lapply(m, f))
```

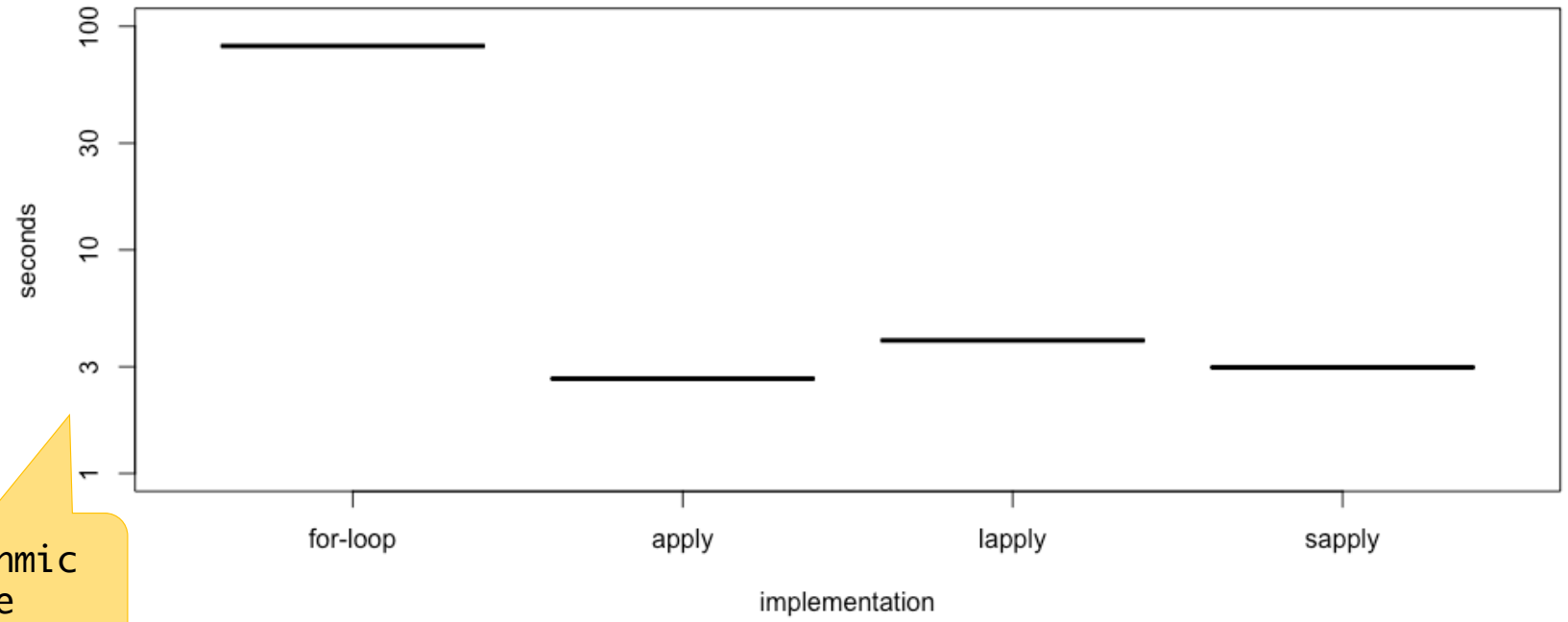
Example

- read a matrix from file
- compute the mean of each row
  and store it on a vector
- four implementations of the same computation
  - for-loop
  - apply
  - lapply
  - sapply
- matrix 100000x100

```
m <- as.matrix(
       read.table(
         "apply_data.txt"
       , header = F))

# for-loop
res_for=numeric()

for(i in 1:nrow(m)){
    res_for=c(res_for,mean(m[i,]))
}

# apply
res_apply <- apply(m,1,mean)

# lapply
res_lapply <-
     unlist(
         lapply(1:nrow(m)
                ,function(i) mean(m[i,])))

# sapply
res_sapply <-
       sapply(1:nrow(m)
            , function(i) mean(m[i,]))
```

converts list into vector

For-loop and functionals speed comparison



logarithmic scale

R Programming: Extracting selected parts of a variable (subsetting)

Subsetting in R

- R’s subsetting operators are powerful and fast.
- Subsetting allows to succinctly express complex operations in a way that few other languages can match
- three subsetting operators
- six types of subsetting
- important differences in behaviour for different objects
  - vectors, lists, factors, matrices, and data frames
- subsetting can effectively be used in conjunction with assignment

Subsetting a vector
example: `x <- c(2.1, 4.2, 3.3, 5.4)`

- positive integers return elements at the specified positions
  - a single index expression returns a value

    ```
    x[2]
    > 4.2
    ```
  - a non-integer expression is implicitely truncated
    - x[1.8] is the same as x[1]
  - duplicated indexes yeld duplicated values

    ```
    x[c(1,1)]
    > 2.1 2.1
    ```
  - negative integers omit elements at specified positions

    ```
    x[-c(3,1)]
    > 4.2 5.4
    ```

Subsetting a vector (ii)
example: `x <- c(2.1, 4.2, 3.3, 5.4)`

- Logical vectors select elements where the corresponding logical value is TRUE
  - the most useful type of subsetting because you write the expression that creates the logical vector

```
x[c(TRUE, TRUE, FALSE, FALSE)]
#> [1] 2.1 4.2
x[x > 3]
#> [1] 4.2 3.3 5.4
```

- A missing value in the index always yields a missing value in the output

```
x[c(TRUE, TRUE, NA, FALSE)]
#> [1] 2.1 4.2  NA
```

- Nothing returns the original vector

```
x[]
#> [1] 2.1 4.2 3.3 5.4
```

Subsetting a matrix

- generalization of the vector case
- each dimension can be subsetted
- an interesting case: generate the minors of a matrix

```
a <- matrix(1:9,nrow = 3)
for (i in 1:3)
  for(j in 1:3)
    print(a[-i,-j])
```

Subsetting a data frame

example: `df <- data.frame(x = 1:3, y = 3:1, z = letters[1:3])`

- rows satisfying a given condition

```
df[df$x == 2, ]
```

- select columns

```
df[c("x","z")]
```

	x	y	z
1	1	3	a
2	2	2	b
3	3	1	c

Exercises

- fix errors

```
mtcars[mtcars$cyl = 4, ]
mtcars[-1:4, ]
mtcars[mtcars$cyl <= 5]
mtcars[mtcars$cyl == 4 | 6, ]
```

logical subsetting

- The most common technique to extract rows from a data frame

```
mtcars[mtcars$gear == 5 & mtcars$cyl == 4, ]
```

From data to differences

	day	closing
1	1	13
2	2	12
3	3	15
4	4	14
5	5	16
6	6	17
7	7	19
8	8	18

	day	closing
1	1	13
2	2	12
3	3	15
4	4	14
5	5	16
6	6	17
7	7	19
8	8	18

```
df <- data.frame(day = c(1:8),
                 closing =
c(13,12,15,14,16,17,19,18))
df$closing[2:nrow(df)] -
df$closing[1:(nrow(df)-1)]
```

Subsetting indexes

`which()` returns the indexes of the True in a vector of boolean values

```
> v <- c(4,8,3,5,9,12)
> condT <- 6 <= v & v <= 10
> condT
[1] FALSE  TRUE FALSE FALSE  TRUE FALSE
> which(condT)
[1] 2 5
> which(6 <= v & v <= 10) # same result as above
```

Inspecting variable structure - `typeof`

- it is the *base* data type
- it defines the type of content, for atomic types
- it defines the structure, for lists
- the repeating types (vectors, matrices, arrays) have the base type

Inspecting variable structure - **typeof**

```
> typeof(1L)
[1] "integer"
> typeof(1:3)
[1] "integer"
> typeof(1)
[1] "double"
> typeof(1.1:5.2)
[1] "double"
> typeof( matrix(1:9
        , nrow = 3))
[1] "integer"
```

```
> typeof("c")
[1] "character"
> typeof(c("a", "b"))
[1] "character"
> typeof(list(a=1, b="x"))
[1] "list"
> typeof(
   data.frame( a = c(10,20)
             , b = c("x",
                     "y"))
          )
[1] "list"
```

Inspecting variable structure - `typeof`

```
> typeof(TRUE)
[1] "logical"
> typeof(1 == 2)
[1] "logical"
> TRUE > FALSE
[1] TRUE
> typeof(NA)
[1] "logical"
> TRUE > NA
[1] NA
> NA == NA
[1] NA
```

Inspecting variable structure - `mode`

- is a mutually exclusive classification of objects according to their basic structure
- the 'atomic' modes are numeric, complex, character and logical
- recursive objects have modes such as 'list' or 'function' or a few others.
- an object has one and only one mode

Inspecting variable structure - **mode**

```
> mode(1L)
[1] "numeric"
> mode(1:3)
[1] "numeric"
> mode(1)
[1] "numeric"
> mode(1.1:5.2)
[1] "numeric"
> mode(matrix(1:9
                ,nrow = 3))
[1] "numeric"
```

```
> mode("c")
[1] "character"
> mode(c("a", "b"))
[1] "character"
> mode(list(a=1, b="x"))
[1] "list"
> mode(
    data.frame(
        a = c(10,20)
      , b = c("x", "y")))
[1] "list"
```

Inspecting variable structure - `class`

- it is used to define/identify what "type" an object is from the point of view of object-oriented programming in R
- it is a property assigned to an object that determines how generic functions operate with it
- it is not a mutually exclusive classification
- if an object has no specific class assigned to it, such as a simple numeric vector, it's class is usually the same as its mode, by convention
- the same function can behave differently, depending on the class of its arguments
  - e.g. **`print()`**

Inspecting variable structure - **class**

```
> class(1L)
[1] "integer"
> class(1:3)
[1] "integer"
> class(1)
[1] "numeric"
> class(1.1:5.2)
[1] "numeric"
> class(matrix(1:9
               ,nrow = 3))
[1] "matrix"
```

```
> class("c")
[1] "character"
> class(c("a", "b"))
[1] "character"
> class(list(a=1, b="x"))
[1] "list"
> class(
    data.frame(
      a = c(10,20)
    , b = c("x", "y")))
[1] "data.frame"
```