Elementos de programación R

Ricardo R. Palma

2023-04-26

Elementos básicos de programación

En este breve tutorial examinaremos algunos elementos del lenguaje de programación R y como valernos de ello para resolver problemas de la vida cotidiana. Apelaremos a ejemplos bien conocidos, pero además mostraremos las soluciones que desarrollaremos contra las misma que ya están implementadas en R. Comparando el costo computacional, medido como tiempo de ejecución. Esto nos permitirá entender la calidad del algoritmo que implementemos. Como excusa para introducirnos propondremos realizar tres experimentos y medir el tiempo ejecución.

Veremos:

- Generar un vector secuencia
- Implementación de una serie Fibonacci
- Ordenación de un vector por método burbuja
- Progresión geométrica del COVid-19
- Algoritmo de funciones estadísticas

Algunas ideas de como medir el tiempo de ejecucion

Muchos de ustedes están familiarizados con Octave o Matlab. Algunos recordarán que para invertir matrices y saber que método era más eficiente se utilizaban los comandos tic y tac. Por ejemplo se generaba una matriz A, se ejecutaba el comando tic que disparaba una especie de cronómetro interno, luego se invertía seguiendo un algoritmo de determinante y finalmente se ejecutaba el comando toc que detenía el reloj y entregaba el tiempo de ejecución. Luego se repetía el mismo procedimiento, pero en lugar de hacerlo con determitante se usaba un algoritmo de matriz LU.

Una búsqueda rápida en línea nos revela al menos tres paquetes R para comparar performance del código R (rbenchmark, microbenchmark y tictoc). Estos además de medir el tiempo nos indican porcentaje de memoria y microprocesador utilizados.

Además, la base R proporciona al menos dos métodos para medir el tiempo de ejecución del código R (Sys.time y system.time), que es una aproximación bastante útil para un curso como el que desarrollamos.

A continuación, paso brevemente por la sintaxis del uso de cada una de las cinco opciones, y presento mis conclusiones al final.

Usando Sys.time

El tiempo de ejecución de un fragmento de código se puede medir tomando la diferencia entre el tiempo al inicio y al final del fragmento de código leyendo los registros del RTC (Real Time Clock. Simple pero flexible: como un relojito de arena :.

```
sleep_for_a_minute <- function() { Sys.sleep(35) }

start_time <- Sys.time()
sleep_for_a_minute()
end_time <- Sys.time()

end_time - start_time
```

```
## Time difference of 35.57653 secs
```

Hemos generado una función que antes no existía y la hemos usado. Deficiencias: Si usas el comando dentro de un documento en R-Studio te demorarás mucho tiempo cuando compiles un PDF o una presentación.

Biblioteca tictoc

Esto de usar una biblioteca es llamar u cargar una procedimientos que generará comandos nuevos en R. Como ya fue comentado, cargar una biblioteca implica ejecutar el comando install.packages() o usar en r-studio el menú de Herramientas y luego Instalar paquetes. Las funciones tic y toc son de la misma biblioteca de Octave/Matlab y se usan de la misma manera para la evaluación comparativa del tiempo de sistema recién demostrado. Sin embargo, tictoc agrega mucha más comodidad al usuario y armonía al conjunto.

La versión de desarrollo más reciente de tictoc se puede instalar desde github:

```
install packages (tictoc)
```

```
library(tictoc)

tic("sleeping")
A<-20
print("dormire una siestita...")
```

```
## [1] "dormire una siestita..."
```

```
Sys.sleep(12)
print("...suena el despertador")
```

```
## [1] "...suena el despertador"
```

```
toc()
```

```
## sleeping: 12.21 sec elapsed
```

Uno puede cronometrar solamente un fragmento de código a la vez:

Biblioteca rbenchmark

La documentación de la función benchmark del paquete rbenchmark R lo describe como “un simple contenedor alrededor de system.time”. Sin embargo, agrega mucha conveniencia en comparación con las llamadas simples a system.time. Por ejemplo, requiere solo una llamada de referencia para cronometrar múltiples repeticiones de múltiples expresiones. Además, los resultados devueltos se organizan convenientemente en un marco de datos.

```
library(rbenchmark)
# lm crea una regresión lineal
benchmark("lm" = {
            X <- matrix(rnorm(1000), 100, 10)
            y <- X %*% sample(1:10, 10) + rnorm(100)
            b <- lm(y ~ X + 0)$coef
          },
          "pseudoinverse" = {
            X <- matrix(rnorm(1000), 100, 10)
            y <- X %*% sample(1:10, 10) + rnorm(100)
            b <- solve(t(X) %*% X) %*% t(X) %*% y
          },
          "linear system" = {
            X <- matrix(rnorm(1000), 100, 10)
            y <- X %*% sample(1:10, 10) + rnorm(100)
            b <- solve(t(X) %*% X, t(X) %*% y)
          },
          replications = 1000,
          columns = c("test", "replications", "elapsed",
                      "relative", "user.self", "sys.self"))
```

```
##            test replications elapsed relative user.self sys.self
## 3 linear system         1000    0.49    1.000      0.42     0.03
## 1            lm         1000    2.49    5.082      2.33     0.11
## 2 pseudoinverse         1000    0.57    1.163      0.50     0.06
```

«bench_mark,echo=TRUE»=

En el informe de salida nos dice que cantidad de tiempo consume cada parte del código.

Biblioteca Microbenchmark

La versión de desarrollo más reciente de microbenchmark se puede instalar desde github:

Al igual que el punto de referencia del paquete rbenchmark, la función microbenchmark se puede usar para comparar tiempos de ejecución de múltiples fragmentos de código R. Pero ofrece una gran comodidad y funcionalidad adicional. Es más **"beta"** (inestable), pero como todo lo que hoy es nuevo poco a poco se hará más estable y no complicará tanto las cosas para el usuario final.

Una cosa interesante es que se puede ver la salida gráfica del uso de recursos. Ver líneas finales del código.

Lo que parece una característica particularmente agradable de microbenchmark es la capacidad de verificar automáticamente los resultados de las expresiones de referencia con una función especificada por el usuario. Esto se demuestra a continuación, donde nuevamente comparamos tres métodos que computan el vector de coeficientes de un modelo lineal.

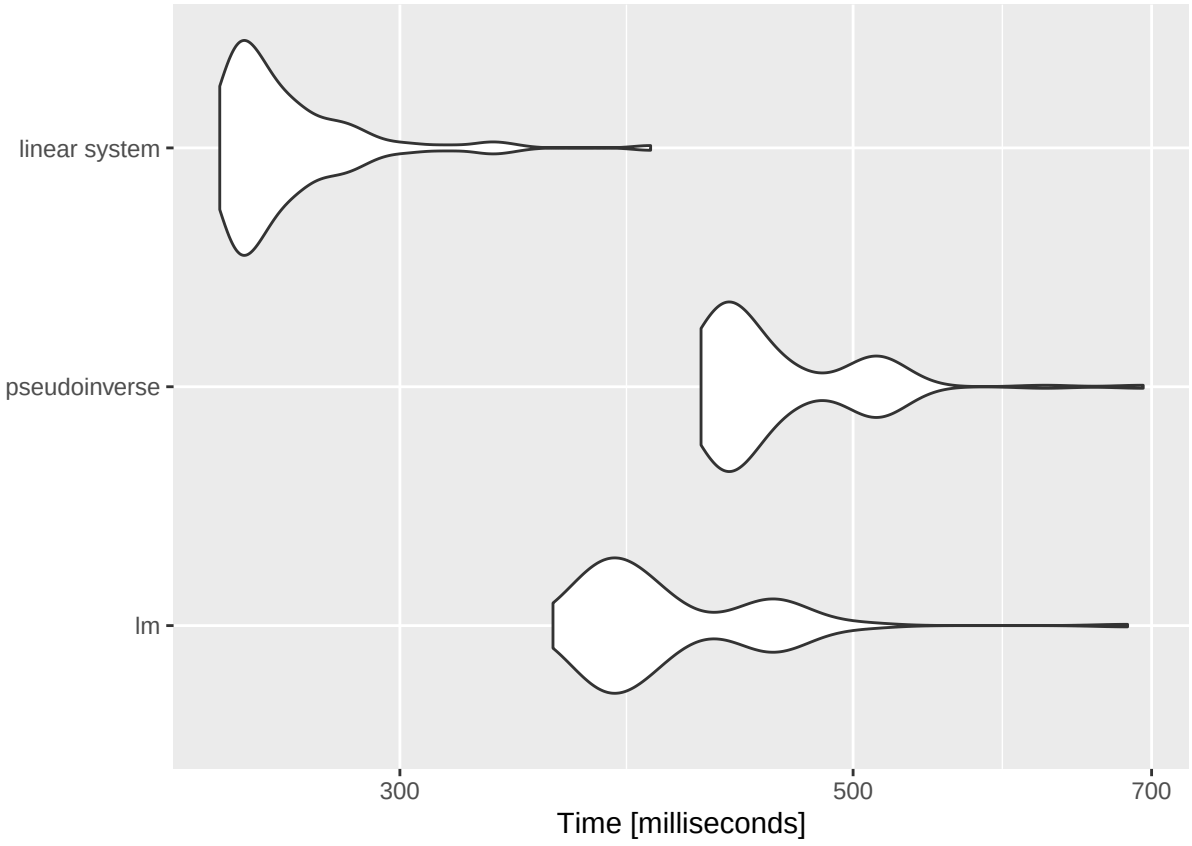
```
library(microbenchmark)

set.seed(2017)
n <- 10000
p <- 100
X <- matrix(rnorm(n*p), n, p)
y <- X %*% rnorm(p) + rnorm(100)
```

```
check_for_equal_coefs <- function(values) {
  tol <- 1e-12
  max_error <- max(c(abs(values[[1]] - values[[2]]),
                     abs(values[[2]] - values[[3]]),
                     abs(values[[1]] - values[[3]])))
  max_error < tol
}

mbm <- microbenchmark("lm" = { b <- lm(y ~ X + 0)$coef },
               "pseudoinverse" = {
                 b <- solve(t(X) %*% X) %*% t(X) %*% y
               },
               "linear system" = {
                 b <- solve(t(X) %*% X, t(X) %*% y)
               },
               check = check_for_equal_coefs)

mbm
```

```
## Unit: milliseconds
##           expr      min       lq     mean   median       uq      max neval
##             lm 356.4988 377.0872 406.8131 390.3340 448.3201 681.3174   100
##  pseudoinverse 421.2457 431.5768 460.3725 439.7192 488.3377 693.4232   100
##  linear system 244.8676 250.1096 264.4577 256.6907 269.2525 397.9893   100
```

```
library(ggplot2)
autoplot(mbm)
```

```
## Coordinate system already present. Adding new coordinate system, which will
## replace the existing one.
```



«microbenchmark,echo=TRUE,fig=TRUE»=

@

Consigna del trabajo final del módulo

El trabajo de hay que presentar implica revisar los algoritmos que se presentan a continuación. Deberá ejecutarlos primero en la línea de comando de la consola.

Luego deberá elegir alguno de los métodos vistos para medir la performance y comparar los resultados con otros compañeros que hayan usado otros métodos para medir la performance.

Luego todo deberá entregarse en un informe en formato pdf construido con RStudio, archivo rsweave.

Generar un vector secuencia

De echo R. tiene un comando para generar secuencias llamado "seq' '. Recomendamos ejecutar la ayuda del comando en RStudio.

Pero utilizaremos el clásico método de secuanecias de anidamiento for, while, do , until.

Generaremos una secuencia de números de dos en dos entre 1 y 100.000.

Secuencias generada con for

```
start_time <- Sys.time()
for (i in 1:50000) { A[i] <- (i*2)}
head (A)
```

```
## [1]  2  4  6  8 10 12
```

```
tail (A)
```

```
## [1]  99990  99992  99994  99996  99998 100000
```

```
end_time <- Sys.time()

end_time - start_time
```

```
## Time difference of 0.067873 secs
```

Secuencia generada con R

```
start_time <- Sys.time()
A <- seq(1,1000000, 2)
head (A)
```

```
## [1]  1  3  5  7  9 11
```

```
tail (A)
```

```
## [1] 999989 999991 999993 999995 999997 999999
```

```
end_time <- Sys.time()

end_time - start_time
```

```
## Time difference of 0.02828383 secs
```

```
  CONSIGNA: Comparar la performance con systime
```

Implementación de una serie Fibonachi o Fibonacci

En matemáticas, la sucesión o serie de Fibonacci es la siguiente sucesión infinita de números naturales:

** 0,1,1,2,3,5,8 . . . 89,144,233 . . . **

La sucesión comienza con los números 0 y 1,2 a partir de estos, «cada término es la suma de los dos anteriores», es la relación de recurrencia que la define.

A los elementos de esta sucesión se les llama números de Fibonacci. Esta sucesión fue descrita en Europa por Leonardo de Pisa, matemático italiano del siglo XIII también conocido como Fibonacci. Tiene numerosas aplicaciones en ciencias de la computación, matemática y teoría de juegos. También aparece en configuraciones biológicas, como por ejemplo en las ramas de los árboles, en la disposición de las hojas en el tallo, en las flores de alcachofas y girasoles, en las inflorescencias del brécol romanesco, en la configuración de las piñas de las coníferas, en la reproducción de los conejos y en cómo el ADN codifica el crecimiento de formas orgánicas complejas. De igual manera, se encuentra en la estructura espiral del caparazón de algunos moluscos, como el nautilus.

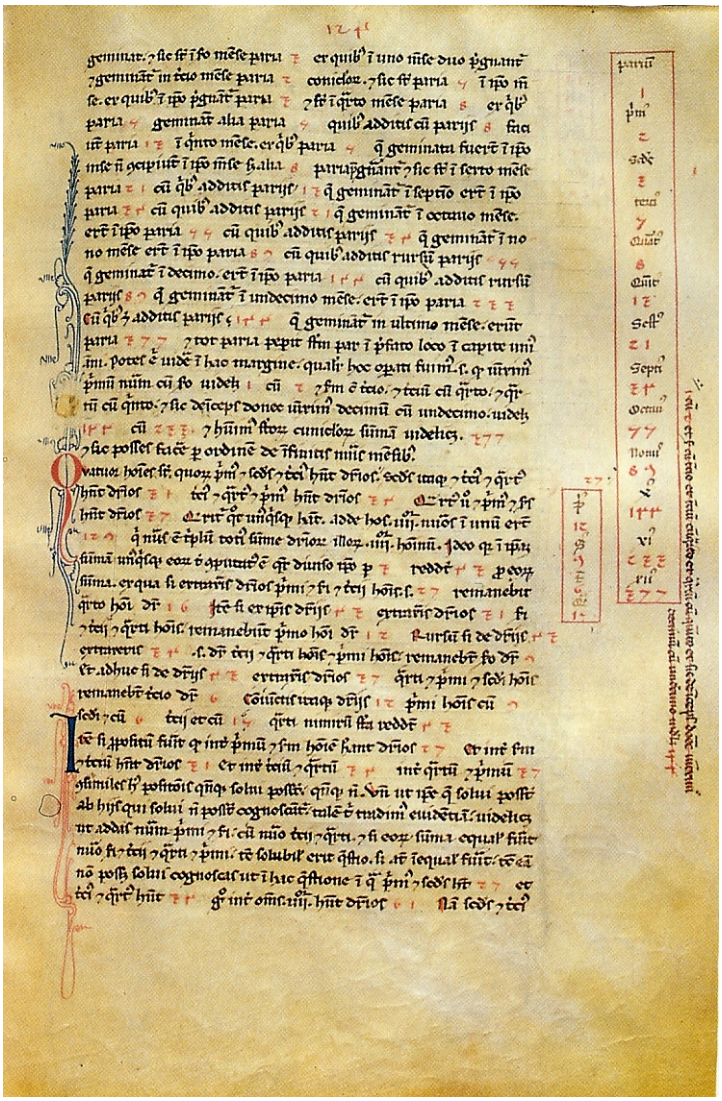


Figure 1: Fig.2 Original de Fibonacci

Definición matemática recurrente

$$f(x) = \begin{cases} f_0 = 0 & si \quad x = 0 \\ f_1 = 1 & si \quad x = 1 \\ f_{n+1} = f_n + f_{n-1} & \forall \quad x \geq 2 \end{cases}$$

```
d <- seq(0,500000,1)
for(i in 0:500000)
{ a<-i
b <-i+1
c <-a+b
d[i+1] <-c
# comentar esta línea para conocer el número más grande hallado
#print(c)
}
which(d>1000000)
```

```
## [1] 500001
```

```
d[500001]
```

```
## [1] 1000001
```

```
#Descomentar esta línea para saber el número más grande hallado
#print(d)
```

CONSIGNA:

¿Cuantas iteraciones se necesitan para generar un número de la serie mayor que 1.000.000 ?

Ordenación de un vector por método burbuja

La Ordenación de burbuja **Bubble Sort en inglés** es un sencillo algoritmo de ordenamiento. Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada. Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas "burbujas". También es conocido como el método del intercambio directo. Dado que solo usa comparaciones para operar elementos, se lo considera un algoritmo de comparación, siendo uno de los más sencillos de implementada.

```
# Tomo una muestra de 10 números ente 1 y 100
x<-sample(1:100,100)
# Creo una función para ordenar
burbuja <- function(x){
  n<-length(x)
  for(j in 1:(n-1)){
    for(i in 1:(n-j)){
      if(x[i]>x[i+1]){
        temp<-x[i]
```

```
        x[i]<-x[i+1]
        x[i+1]<-temp
      }
    }
  }
  return(x)
}
res<-burbuja(x)
#Muestra obtenida
x
```

```
##   [1]  43  54  59  99  79  72  76  24  45  62  51  20  53  73  42  70  91  48
##  [19]  38  55   6  35  71  28  23  88  86  40   2  56  50 100  11  52  82  64
##  [37]   4  37   8  47  74  93  31  57  39  78  94  69   7  83  67  10  84  63
##  [55]  77  18  19  98  41   3  33  80  12  60  21   1  22  68  58  17  75  92
##  [73]  16  46  27   5  13  26  65  96  90  81  25  14  34  32  15  87  97  44
##  [91]  89  29  66  49  36  95   9  85  61  30
```

```
#Muestra Ordenada
res
```

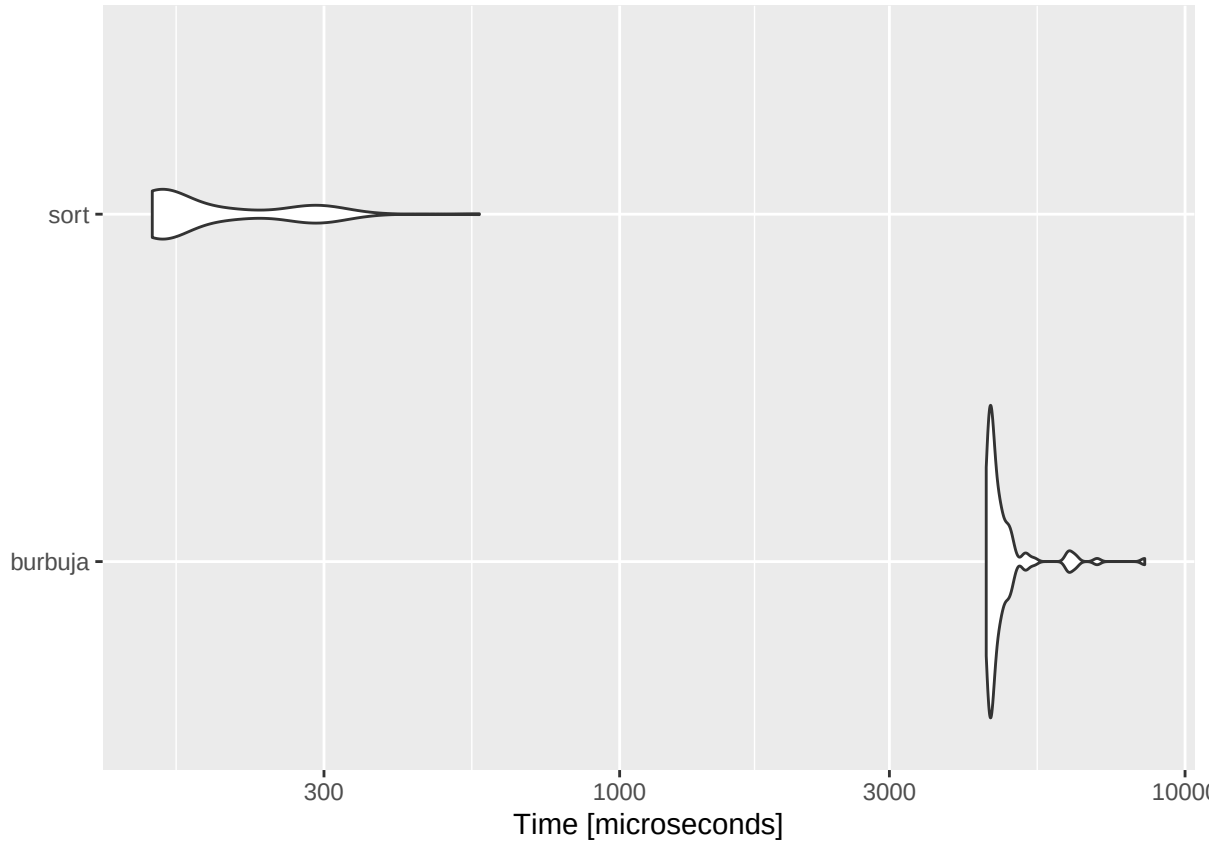
```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
##  [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
##  [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
##  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
##  [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
##  [91]  91  92  93  94  95  96  97  98  99 100
```

```
#Ordenación con el comando SORT de R-Cran

library(microbenchmark)

set.seed(2017)
n <- 150
p <- 1
X <- matrix(rnorm(n*p), n, p)
y <- X %*% rnorm(p) + rnorm(150)

check_for_equal_coefs <- function(values) {
  tol <- 1e-5
  max_error <- max(c(abs(values[[1]] - values[[2]])))
  max_error < tol
  }

mbm <- microbenchmark("burbuja" = {b <- burbuja(X)},
      "sort" = {b <- sort(X)},
      check = check_for_equal_coefs)
mbm
```

```
## Unit: microseconds
##     expr      min       lq     mean   median        uq      max neval
##  burbuja 4450.001 4513.901 4787.046 4577.101 4770.3515 8486.501   100
##     sort  149.102  152.501  199.595  160.601  260.0015  565.001   100
```

```
library(ggplot2)
autoplot(mbm)
```

```
## Coordinate system already present. Adding new coordinate system, which will
## replace the existing one.
```



CONSIGNA: Comparación de performance

Compara la performance de ordenación del método burbuja vs el método sort de R \ Usar método microbenchmark para una muestra de \ tamaño 20.000

}