TyHM - Programación en R-CRan

Sebastián Zaragoza* Manuel Lopez† Anabella Bottasso‡ Ernesto Villasante§
Valentin Adelaide¶

2022-06-12

Introducción

En este trabajo se estudiarán diferentes conceptos y funciones del lenguaje de programación R-CRan. Se revisarán y compararán distintos algoritmos

Generar un vector secuencia

Vamos a generar un vector secuencia a partir de un algoritmo iterativo "for", posteriormente usaremos el comando "seq" de R y luego compararemos los tiempos con systime.

Secuencia generada con "for"

```
A<-0
start_time<-Sys.time()
for (i in 1:50000) { A[i] <- (i*2)}
end_time<-Sys.time()
end_time-start_time
```

```
## Time difference of 0.090765 secs
```

```
head (A)
```

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

```
tail(A)
```

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

Secuencia generada con R

*sebaema2050@gmail.com
†manuelignaciolopez222@gmail.com
‡natali.anabella@gmail.com
§ernestovillasante@gmail.com
¶adelaide.valentin@gmail.com

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

```
## Time difference of 0.01093507 secs
```

```
head (A)
```

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

```
tail(A)
```

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

Finalmente, podemos observar a partir de comparar los tiempos obtenidos gracias a la función "sys.time" que la secuencia generada con R es mas rápida.

Implementación de la serie Fibonacci

La serie Fibonacci comienza con los números 0 y 1, a partir de estos cada uno de los siguientes términos es la suma de los dos anteriores, a continuación puede verse el código para implementar la serie:

```
A<-0
B<-1
F[1]<-A
F[2]<-B
for (i in 3:100) {F[i] <- (F[i-1]+F[i-2])}
head (F)
```

```
## [1] 0 1 1 2 3 5
```

Posteriomente se quiere saber la cantidad de iteraciones necesarias para generar un número de la serie mayor que 1000000. Para esto vamos a eliminar la F con el fin de poder comenzar desde cero con la implementación de la serie. A continuación se puede observar el código correspondiente a la obtención de las iteraciones:

```
remove(F)
A<-0
B <- 1
C <- 0
it <- 30
F[1]<-A
F[2]<-B
for(i in 3:(2+it)) {
F[i]<-(F[i-1]+F[i-2])
}
C <- length(F)
message("Para ",it," iteraciones, el penúltimo valor es: \n",F[C-1],"\n","y el último es: \n"
,F[C])
```

```
## Para 30 iteraciones, el penúltimo valor es:
## 832040
## y el último es:
## 1346269
```

En conclusión se observa que se necesitan 30 iteraciones para superar el número 1000000.

Ordenación de un vector por método burbuja

En este apartado vamos a ordenar un vector con el método burbuja, que funciona revisando cada elemento de la lista y comparándolo con el siguiente, luego de compararlos los intercambia de posición (si están en orden equivocado), posteriormente se ordenará con el comando "SORT" de R. Luego se compararán ambos métodos con el método microbenchmark considerando una muestra de tamaño 20000.

```
# Tomo una muestra de 20000 números ente 1 y 1000000
XA<-sample(1:1000000,20000)
# 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)
}
burbuja<-burbuja(XA)
#Muestra obtenida
#Por una cuestión de espacio no se pedirá al código que muestre el vector ordenado para
#generar el pdf ya que esto ocuparía demasiado espacio, de todas formas antes de
#generar el pdf se pidió al programa mostrar este vector para comprobar que efectivamente
#estuviera ordenado lo cual dió resultados satisfactorios ya que efectivamente
#estaba ordenado. A continuación se deja una línea en la que se puede pedir que muestre el
#vector simplemente borrando el "#"...
#burbuja
sortR<-sort(XA)
#Idem anterior en cuanto a mostrar el vector.
#sort
#El siguiente código sirve para mostrar información en cuanto a rendimiento de ambos métodos
library(microbenchmark)
mbm <-microbenchmark(burbuja,sortR)
mbm
```

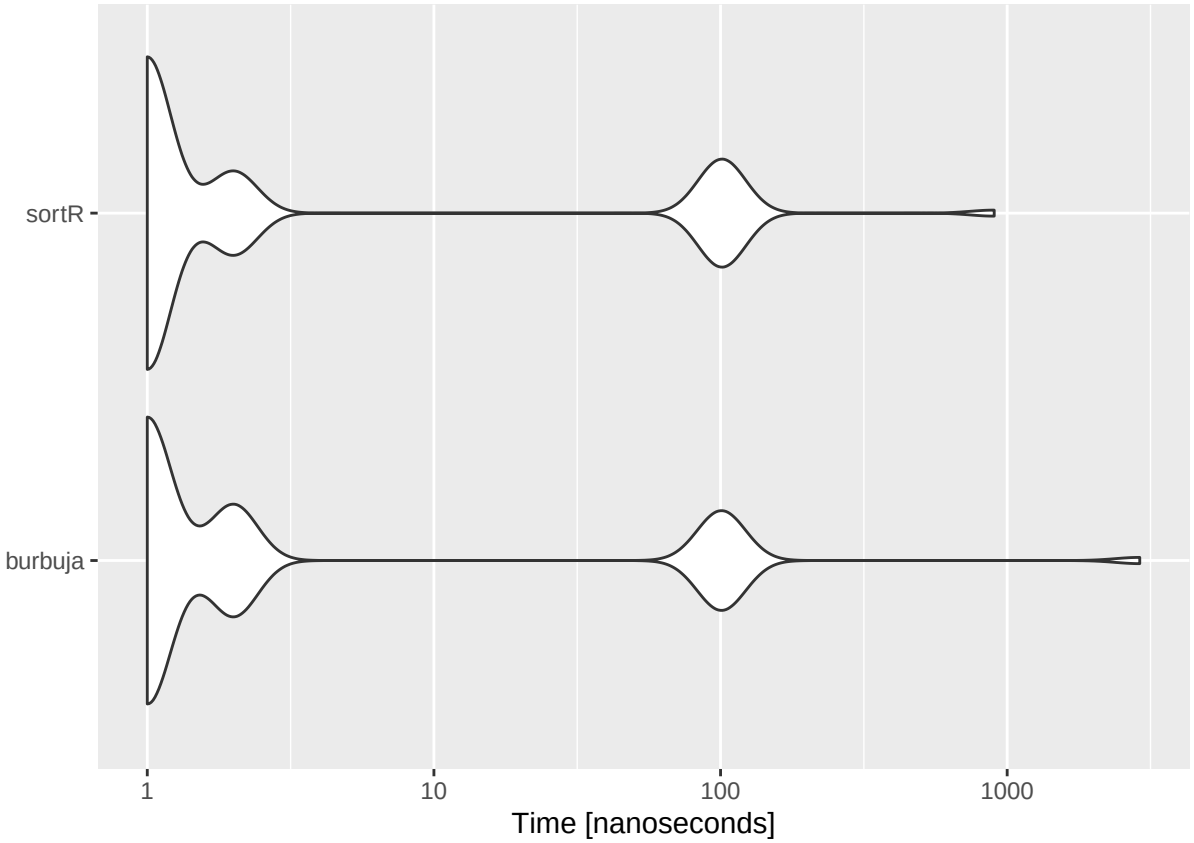
```
## Unit: nanoseconds
##     expr min lq  mean median uq  max neval
##  burbuja   0  1 45.95      1  2 2901   100
##    sortR   0  1 28.02      1  2  901   100
```

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

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

```
## Warning: Transformation introduced infinite values in continuous y-axis
```

```
## Warning: Removed 34 rows containing non-finite values (stat_ydensity).
```



En conclusión se puede observar que el método burbuja tarda mas tiempo y consume mas recursos inicialmente que el comando "sort"

Progresión geométrica del COVid-19

Primero descagamos la libreria de "readr", luego ingresamos los datos .csv a partir del archivo descargado de la página de la cátedra de la siguiente manera: File->Import dataset->from text(readr), luego se siguen las consignas de la guía y se copian los pasos realizados en lenguaje r para pegarlos en el siguiente código:

```
library(readr)
casos <- read_delim("descargaDelProfe/casos.csv",
    delim = ";", escape_double = FALSE, col_types = cols(Fecha = col_date(format = "%m/%d/%Y"),
        Casos = col_integer()), trim_ws = TRUE,
    skip = 1)
```

```
## Warning: One or more parsing issues, see `problems()` for details
```

Estadística de casos

```
summary(casos$Casos)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00   36.75  245.50  514.71  995.50 1715.00
```

Podemos calcular el factor de contagios dividiendo los infectados de hoy sobre los de ayer, en el siguiente código se muestra cómo hacerlo:

```
m <- length(casos$Casos)
F <- (casos$Casos[2:m])/(casos$Casos[1:m-1])
F[m-1]
```

```
## [1] 1.05344
```

Este es el factor de contagios

Estadísticos de F (factor de contagios)

```
mean(F,na.rm = TRUE)
```

```
## [1] 1.350739
```

```
sd(F,na.rm = TRUE)
```

```
## [1] 0.8554107
```

```
var(F,na.rm = TRUE)
```

```
## [1] 0.7317275
```

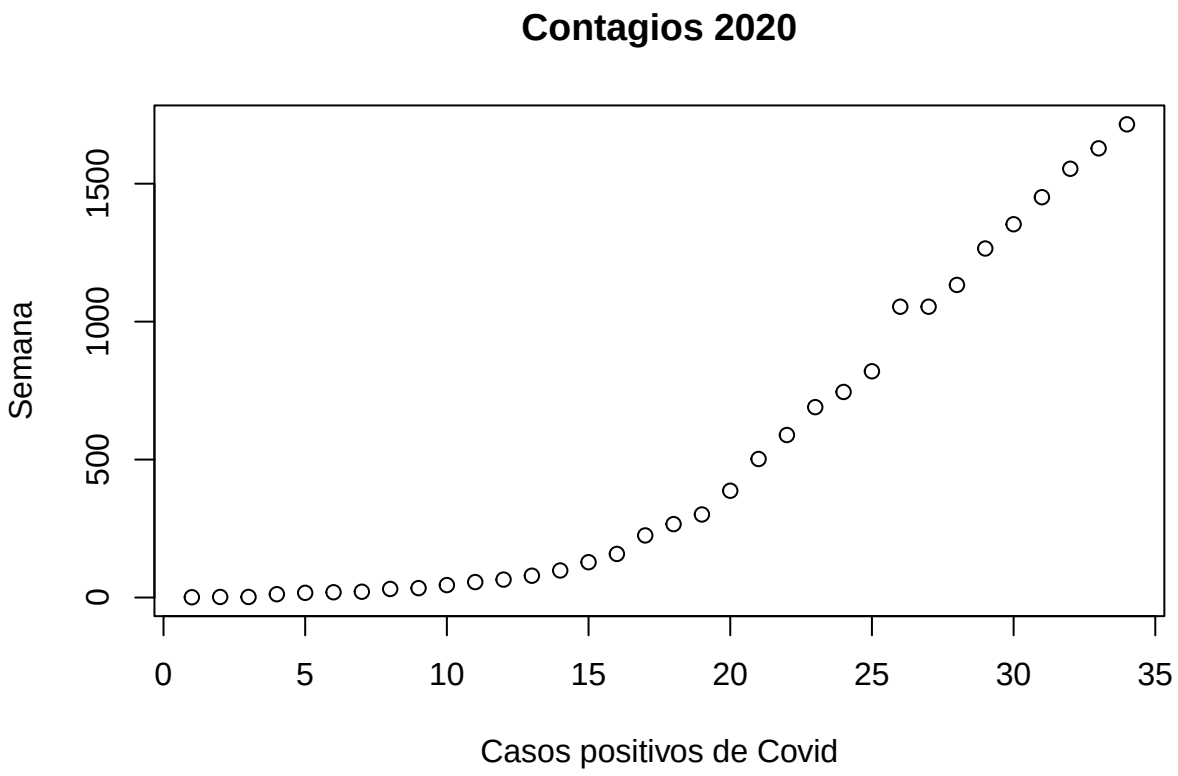
Ploteo de datos

Con los datos importados anteriormente procedimos a realizar un ploteo para poder visualizarlos. A continuación se muestran los códigos implementados para los distintos gráficos.

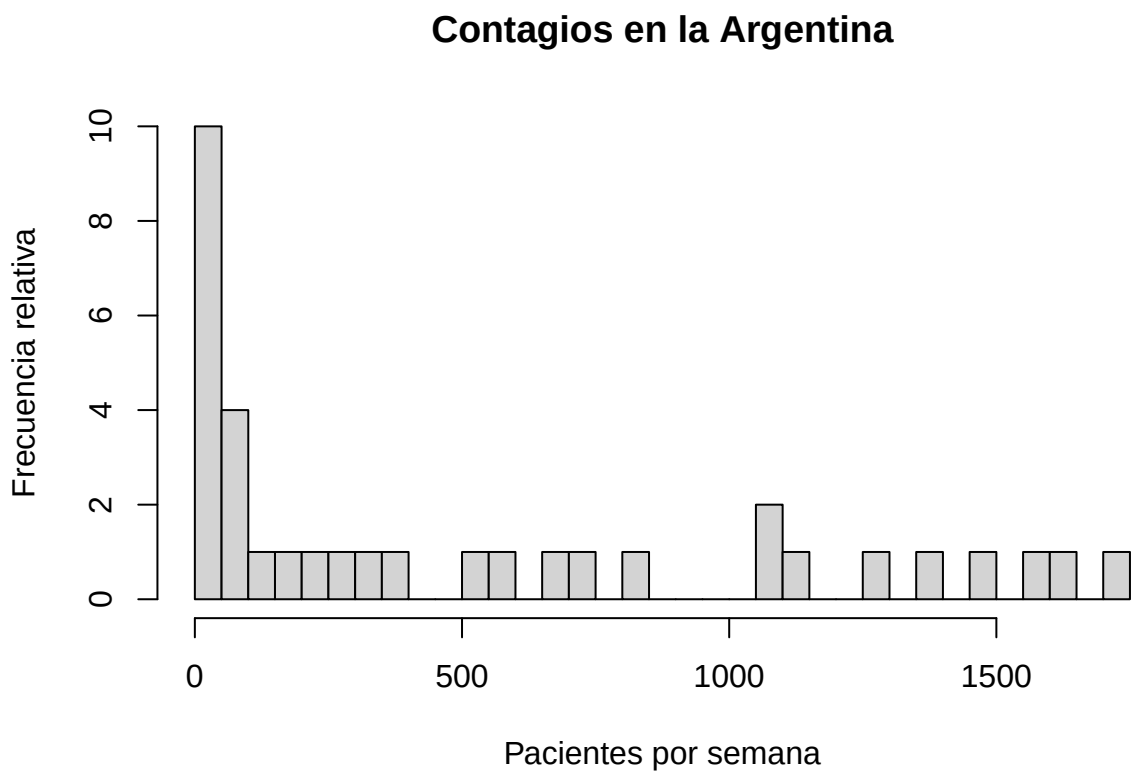
```
casos$Casos
```

```
##  [1]    1    2    2   12   17   19   21   31   34   45   56   65   79   98  128
## [16]  158  225  266  301  387  502  589  690  745  820 1054 1054 1133 1265 1353
## [31] 1451 1554 1628 1715
```

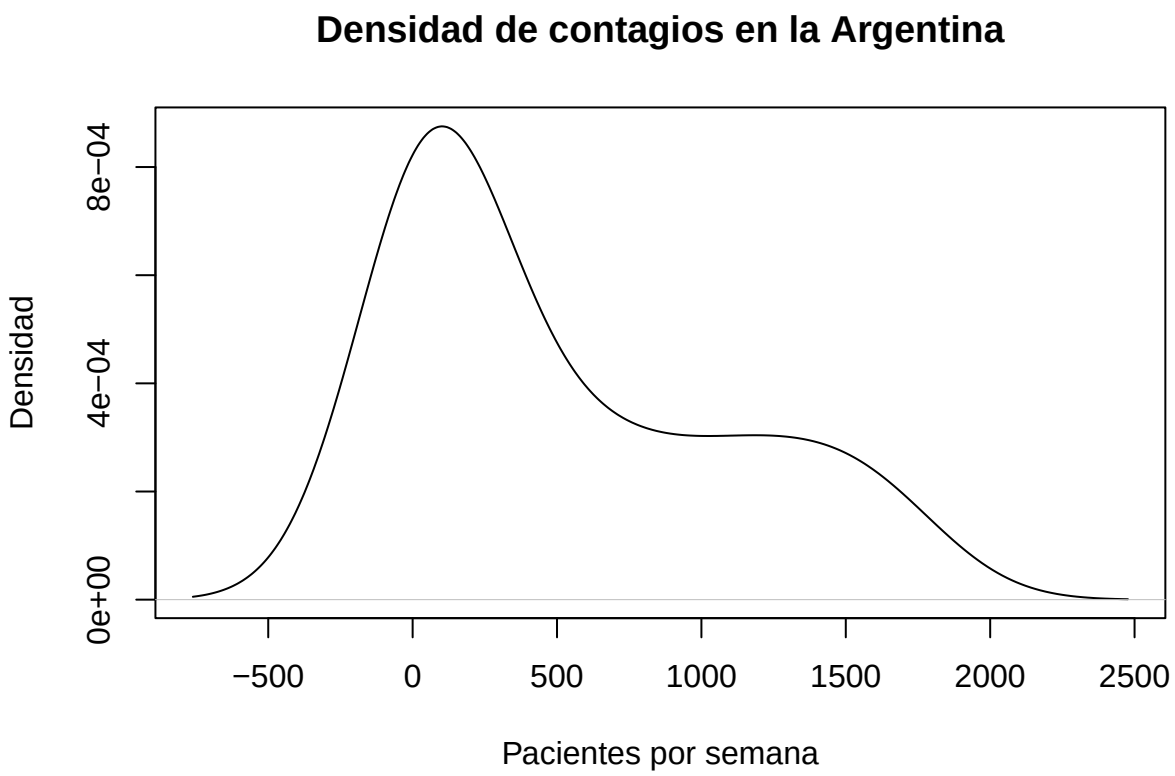
```
plot(casos$Casos ,main="Contagios 2020",ylab="Semana",xlab="Casos positivos de Covid")
```




```
hist(casos$Casos,breaks = 50,main="Contagios en la Argentina",xlab="Pacientes por semana",ylab="Frecuen
```




```
plot(density(na.omit(casos$Casos)),main="Densidad de contagios en la Argentina", ylab = "Densidad", xlab
```



En estos gráficos se pueden observar la cantidad de variables de covid detectadas en el lapso de tiempo en estudio, para analizar esto se deben observar las campanas de Gauss.

La cantidad de campanas en el gráfico de densidad de contagios se relaciona con un índice de contagio distinto, por ejemplo cuando hay mas densidad de contagios se podría asociar a una variable mas contagiosa, es por esto q las dos campanas de Gauss detectadas hacen referencia a dos variables de covid distintas.