Índice general

Tema 3

Programación y creación de funciones en R

3.1. Objetos de tipo función en R

En el Tema 2 describíamos R como un *lenguaje funcional* lo que implicaba un estilo particular para resolver problemas centrado en la creación de funciones. La idea es descomponer un problema complejo en varias partes, y resolverlas creando una función o combinación de funciones que operan independientemente.

Las funciones en R son consideradas como objetos (del mismo modo que lo son por ejemplo los vectores) que realizan operaciones sobre los argumentos que le son proporcionados (otros objetos), devolviendo uno o más valores (otros objetos).

Hasta ahora hemos estudiado algunas funciones ya creadas y disponibles en el sistema base o en algún paquete adicional. En este tema profundizaremos un poco más en el tipo particular de objeto que constituyen y aprenderemos a crear nuestras propias funciones. Esto constituye sin duda alguna una de las capacidades más interesantes del lenguaje.

Podemos distinguir dos momentos importantes en la vida de una función: su definición y su evaluación. Una vez definida una función esta será evaluada (llamada o invocada) cuando sea necesario y posiblemente en contextos muy diversos.

3.1.1. Definición de una función

La creación o definición de una función se realiza escribiendo la palabra `function` seguida de sus argumentos (uno o varios símbolos escritos entre paréntesis y separados por comas) y a continuación el conjunto de sentencias que realizan las tareas de la función (entre llaves si son varias sentencias). De este modo la estructura general de definición de una función con nombre `f` sería:

```
f<-function(argumentos)
{
  sentencias
```

```
  ......
  valor devuelto
}
```

Como ejemplo vamos a crear a continuación una función que calcule y devuelva la suma de dos números que se le pasen como argumentos. Siguiendo el esquema anterior elegiríamos un nombre para dicha función[1], por ejemplo `f.suma`, y escribiríamos:

```
> f.suma<-function(x,y)  x+y
```

En este caso la tarea a realizar constituye una sola sentencia de modo que no es necesario usar las llaves, y la definición completa la hemos podido escribir en una única linea. Observa que cuando ejecutamos dicha linea no obtenemos (aparentemente) ningún resultado, sin embargo podemos comprobar que en efecto un nuevo objeto con nombre `f.suma`, y de tipo función, se ha creado y almacenado en el espacio de trabajo:

```
> ls()

[1] "f.suma"

> class(f.suma)

[1] "function"
```

3.1.2. Evaluación de una función

Una vez que hemos definido una función ya podemos utilizarla. Para ello la invocaremos como hemos hecho con otras funciones del sistema, escribiendo su nombre seguido (entre paréntesis) de los valores para los argumentos donde queremos evaluarla[2].

Por ejemplo, para la función `f.suma` que hemos creado antes podemos probar las siguientes evaluaciones:

```
> f.suma(1,1)

[1] 2

> f.suma(1,NA)
```

[1] Es importante elegir un nombre que no corresponda a ninguna función ya creada. Si tienes duda de si un nombre está ya asignado a un función prueba a escribirlo en la consola y si existe se imprimirá su definición.

[2] Ya comentamos anteriormente que los argumentos pueden pasarse por nombre o por posición. Cuando se hace por nombre R utiliza lo que se llama el *partial matching* de modo que no es necesario en general escribir el nombre completo del argumento, sino los primeros caracteres hasta que queda completamente identificado y distinguido de los otros argumentos.

```
[1] NA

> f.suma(1:10,2)

 [1]  3  4  5  6  7  8  9 10 11 12

> f.suma(diag(3),1:3)

     [,1] [,2] [,3]
[1,]    2    1    1
[2,]    2    3    2
[3,]    3    3    4

> f.suma(1)

Error in f.suma(1): el argumento "y" está ausente, sin valor por omisión
```

Estos ejemplos nos muestran en primer lugar que la función que hemos creado en general calcula la suma de dos objetos, que pueden ser por ejemplo vectores o matrices como en las evaluaciones anteriores. Por otro lado la función ha sido definida con dos argumentos (los dos objetos a sumar) por lo que si no se proporciona alguno de ellos, como pasa en el último caso, nos da un error[3]

Es posible definir una función indicando valores por defecto para los argumentos (algunos o todos). Esto nos permitirá que si alguno no se pasa en la llamada, no de error sino que cuando este sea requerido se utilice el valor por defecto. Por ejemplo podemos definir como 0 el valor por defecto para los dos argumentos de la función `f.suma` como sigue:

```
> f.suma<-function(x=0,y=0)  x+y
> f.suma(1)

[1] 1

> f.suma()

[1] 0
```

La función `f.suma` tiene una sola sentencia, la cual además define el valor que devuelve la función. Cuando la función consta de más sentencias, el valor que devuelve la función

[3]Hay que hacer un comentario adicional a este respecto y es que los argumentos de una función en R son evaluados solo cuando se necesitan dentro de la función, esto es lo que se denomina *lazy evaluation*. Por ejemplo la función `f<-function(x,y) 2*x`, no nos dará ningún error si la invocamos como `f(1)`, ya que el segundo argumento no se utiliza dentro de la función, de modo que asociará el valor 1 al argumento `x`.

coincide con la última de las sentencias. Por ejemplo observa que la siguiente función siempre devuelve `24`:

```
> f1<-function(x,y)
+ {
+   x+y
+   24
+ }
> f1(1,1)

[1] 24

> f1(0,0)

[1] 24
```

No obstante es posible definir dentro de la función, en cualquier momento, el valor que debe ser devuelto[4] usando la función `return`. Observa los siguientes ejemplos:

```
> f1<-function(x,y)
+ {
+   return(x+y)
+   24
+ }
> f1(1,1)

[1] 2

> f2<-function(x,y)
+ {
+   x^2
+   return(list(suma=x+y,resta=x-y))
+   y^2
+ }
> f2(1,1)

$suma
[1] 2

$resta
[1] 0
```

[4]Por lo general al evaluar una función durante una sesión interactiva se imprime el valor devuelto. Esto se puede evitar si se quiere usando la función `invisible`.

En estos casos la ejecución de las sentencias se interrumpe cuando se encuentra la palabra `return`, devolviendo el objeto que encierra entre paréntesis.

Ejercicio 1: Crea una función que calcule la media de un vector de n elementos. La función tendrá como nombre `media1` y un único argumento `x` (el vector). El cálculo de la media lo debes hacer descartando posibles valores perdidos (`NA`) en el vector. Como resultado la función devolverá un objeto de tipo vector con dos elementos: la media calculada y el número de valores perdidos que había en el vector.

Ejercicio 2: Crea una función que calcule la media de una variable a partir de una tabla de frecuencias ($\{(x_i, n_i); i = 1, \ldots, k\}$, donde x_i son los valores distintos observados y n_i las frecuencias absolutas). La función tendrá como nombre `media2` y dos argumentos: un vector `xi` con los valores observados, y otro `ni` con las frecuencias absolutas. Para el último argumento definiremos por defecto un vector de unos de la misma longitud que `xi`. Finalmente la función devolverá una lista con la media calculada y el tamaño de la muestra.

3.1.3. Componentes de las funciones

Las funciones en R constan de tres componentes: los argumentos (*formals*), el cuerpo (*body*) y el entorno de la función (*environment*). Este último constituye la estructura de datos que estaba disponible cuando se creó la función[5]. Estas tres componentes se pueden consultar (y si se quiere modificar) utilizando las funciones `formals`, `body` y `environment`, respectivamente.

Como ejemplo observamos las siguientes sentencias que nos permiten consultar los argumentos y el cuerpo de la función `f.suma` que hemos definido antes, y hacer modificaciones en ellos:

```
> # Consulta
> formals(f.suma)

$x
[1] 0

$y
[1] 0

> body(f.suma)

x + y
```

[5]La combinación del código de la función y la unión a su entorno se denomina *function closure*. Se trata de un término de la teoría de programación funcional.

```
> # Modificar los argumentos
> argum.original<-formals(f.suma) # almacenamos los argumentos originales
> formals(f.suma)<-alist(x=,y=,z=)
> f.suma

function (x, y, z)
x + y

> # Modificar el cuerpo
> cuerpo.original<-body(f.suma) # almacenamos el cuerpo original
> body(f.suma)<-expression({
+   suma1<-x+y
+   suma2<-x+z
+   return(list(suma1=suma1,suma2=suma2))
+ })
> f.suma(1,2,3)

$suma1
[1] 3

$suma2
[1] 4

> # reestablecemos argumentos y cuerpo originales
> formals(f.suma)<-argum.original
> body(f.suma)<-cuerpo.original
> f.suma

function (x = 0, y = 0)
x + y
```

Finalmente podemos consultar el entorno de la función `f.suma` con:

```
> environment(f.suma)

<environment: R_GlobalEnv>
```

Observa que el entorno de la función lo constituye en este caso el espacio de trabajo[6], esto significa que cualquier objeto que esté disponible en el espacio de trabajo es accesible y puede ser utilizado dentro de la función. Por otro lado, mientras que la definición de los argumentos y el cuerpo la hicimos de forma explícita cuando se creó la función, el entorno se especificó de manera implícita y en base a dónde se definió la función.

[6]Normalmente las funciones definidas en paquetes de R con *namespaces* tienen como entorno el *namespace* del paquete.

Por otro lado, cada vez que se ejecuta una función se crea un "marco de evaluación" (*evaluation frame*). En este marco los argumentos proporcionados a la función son identificados con los correspondientes valores y las sentencias del cuerpo de la función se evalúan secuencialmente. Hay que tener en cuenta que todas las asignaciones que se realicen dentro del cuerpo de una función son temporales. Esto incluye a los argumentos proporcionados, considerándose como variables locales dentro de la función (este tema se extenderá en la Sección 3.1.7). Observa es siguiente ejemplo:

```
> x<-1
> f<-function(x){
+   y<-1
+   x<-x+y
+   return(x)
+ }
> f(x)

[1] 2

> x # cambia su valor solo dentro de la función

[1] 1

> y # este objeto solo existe dentro de la función

Error in eval(expr, envir, enclos): objeto 'y' no encontrado
```

No obstante es posible realizar asignaciones permanentes dentro de una función usando el operador `<<-` o la función `assign`. Puedes probarlo en el ejemplo anterior para definir permanentemente[7] `y`.

3.1.4. Funciones especiales

Todas las funciones de R tienen las tres componentes anteriores salvo una pequeña colección de funciones del sistema base denominadas *primitive functions*, que están implementadas íntegramente en el lenguaje C[8]. Estas funciones corresponden al tipo `builtin` o `special` y entre ellas está por ejemplo la función `sum`. A continuación inspeccionamos el tipo y las componentes de algunas funciones de R que hemos usado anteriormente.

[7] Habría que hacer algunos comentarios adicionales a este respecto, puedes ver por ejemplo la Sección 10.7 de manual *R-intro* en CRAN.

[8] Esto les proporciona ventajas en cuanto a su rendimiento, sin embargo son difíciles de crear y el grupo R-core solo lo hace caundo no hay otra opción.

```
> class(var)

[1] "function"

> typeof(var)

[1] "closure"

> formals(var)

$x


$y
NULL

$na.rm
[1] FALSE

$use

> body(var)

{
    if (missing(use))
        use <- if (na.rm)
            "na.or.complete"
        else "everything"
    na.method <- pmatch(use, c("all.obs", "complete.obs", "pairwise.complete.obs",
        "everything", "na.or.complete"))
    if (is.na(na.method))
        stop("invalid 'use' argument")
    if (is.data.frame(x))
        x <- as.matrix(x)
    else stopifnot(is.atomic(x))
    if (is.data.frame(y))
        y <- as.matrix(y)
    else stopifnot(is.atomic(y))
    .Call(C_cov, x, y, na.method, FALSE)
}

> environment(var)

<environment: namespace:stats>
```

```
> class(sum)

[1] "function"

> typeof(sum)

[1] "builtin"

> formals(sum)

NULL

> body(sum)

NULL

> environment(sum)

NULL

> class(`[`)

[1] "function"

> typeof(`[`)

[1] "special"
```

3.1.5. Funciones anónimas

En los ejemplos que hemos usado antes cuando creamos una función la asignamos a un nombre, creando así el objeto de tipo función en el espacio de trabajo. Esto no es obligatorio, pudiendo crear funciones "anónimas". Ejemplos habituales de este tipo de funciones son aquellas que creamos como argumentos de otras funciones, como por ejemplo para la familia de funciones `apply` (o la función `outer` que veíamos en el tema anterior). Vemos algún ejemplo:

```
>   lista<-list(x=1:10,y=10:13)
>   lapply(lista, function(v) v^2)

$x
 [1]   1   4   9  16  25  36  49  64  81 100
```

```
$y
[1] 100 121 144 169
```

3.1.6. El argumento especial ...

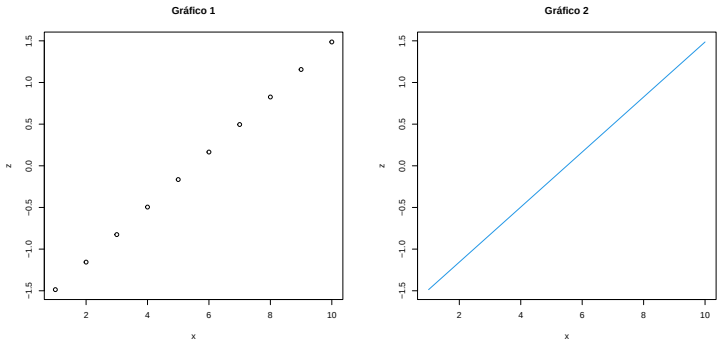
En la definición de una función es posible usar como argumento .... Se trata de un tipo especial que permite pasar a la función cualquier número de argumentos. Esta facilidad puede usarse con distintos propósitos. Su uso al principio de una función está indicado por ejemplo cuando no se conocen de antemano el número de argumentos. Este es el caso por ejemplo de la función `paste` que hemos usado con anterioridad:

```
> formals(paste)

$...


$sep
[1] " "

$collapse
NULL

$recycle0
[1] FALSE

> paste('uno','dos','tres',sep='+')

[1] "uno+dos+tres"
```

Hay que tener en cuenta que cuando se usa el argumento ... en una función, el resto de argumentos que le siguen se deben pasar siempre por nombre y no por posición.

Otro uso común de este argumento especial está relacionado con la posibilidad de pasar argumentos opcionales a otras funciones utilizadas dentro del cuerpo de la función. Observa la siguiente función que usa dicho argumento para extender los argumentos de la función `plot`:

```
> grafico<-function(x,...)
+ {
+   z<-(x-mean(x))/sd(x)
+   plot(x,z,...)
+ }
> grafico(1:10,main='Gráfico 1')
> grafico(1:10,main='Gráfico 2',type='l',col=4)
```



3.1.7. Reglas de alcance: *lexical scoping*

Dentro de una función podemos distinguir tres tipos de "variables" (objetos): argumentos, variables locales y variables libres. Para entender la diferencia entre ellas considera el siguiente ejemplo sencillo:

```
> x<-1
> f<-function(y){
+   z<-x+y
+   z
+ }
```

Este ejemplo muestra los tres tipos anteriores: `y` es un argumento, `z` es una variable local y `x` es una variable libre. De este modo las variables locales son las que se definen dentro del cuerpo de la función y que solo existen dentro del marco de evaluación de la misma. Mientras que las variables libres son aquellas no han sido definidas en el cuerpo de la función.

Las reglas de alcance (*scoping rules*) determinan el modo en que un valor se asocia con una variable libre dentro de una función. R utiliza lo que se conoce como alcance léxico (*lexical scoping*[9]), que determina que las variables locales se buscan dentro del entorno de la función. En el ejemplo anterior, podemos comprobar que dicho entorno es el espacio de trabajo y que `x` es de hecho un objeto almacenado en el espacio de trabajo.

```
> environment(f)

<environment: R_GlobalEnv>

> ls()

[1] "f" "x"
```

[9] Gentleman, R. y Ihaka, R. (2000). Lexical Scope and Statistical Computing, *Journal of Computational and Graphical Statistics* 9, 491-508.

Por otro lado el alcance léxico que utiliza R implica que cuando definimos una variable local, utilizando como nombre el de otro objeto que exista fuera de la función (por ejemplo en el espacio de trabajo), dicho nombre se identificará siempre con la variable local, no dejando acceder por tanto a otra variable con el mismo nombre (*name masking*) fuera de ella. Esto se aplica tanto a objetos de datos como a funciones. Esto se ilustra en el siguiente ejemplo:

```
> f1<-function(a,b) a+b
> f2<-function(a,b)
+ {
+   f1<-function(a,b) a-b
+   f1(a,b)
+ }
> f1(1,1)

[1] 2

> f2(1,1)

[1] 0
```

Cuestión: Observa el siguiente código y sin evaluarlo deduce qué valor devolverá `f(3)`:

```
y <- 10
f <- function(x) {
  y <- 2
  y^2 + g(x)
}
g <- function(x) x * y
```

3.2. Estructuras de programación básica

Como un lenguaje de programación R proporciona estructuras de control, las cuales permiten dirigir el flujo de la ejecución de un programa en una dirección o en otra dentro de su código; así como estructuras de repetición tipo bucle que permiten realizar una tarea un número determinado de veces, o mientras se cumple una determinada condición.

3.2.1. Estructuras de control

Estructura `if`

La estructura condicional `if` permite la evaluación de opciones. Su sintaxis básica es:

```
if (condición) acción-verdad
if (condición) acción-verdad else acción-falso
```

En el primer caso, `acción-verdadero` se evalúa cuando `condición` es cierta, y no ocurre nada cuando es falsa. Mientras que en el segundo caso se evalúan `acción-verdad` o `acción-falso` dependiendo de si la condición se cumple o no, respectivamente. Observa estos ejemplos:

```
> x<-5
> if (x>=5) cat('Aprobado')  else cat('Suspenso')

Aprobado

> calif<- if (x>=5) print('Aprobado')  else print('Suspenso')

[1] "Aprobado"

> calif

[1] "Aprobado"
```

Observa que es posible asignar el valor devuelto por la estructura `if` a un objeto[10]. En el segundo caso del ejemplo anterior hemos asignado el resultado de la estructura `if` (que en este caso consiste en un vector de tipo carácter) al objeto `calif`

Por otro lado las acciones a evaluar pueden ser una o varias sentencias, en el último caso deben ir agrupadas entre llaves (`{ }`). Además es posible evaluar más de una condición dentro de la estructura anterior usando `else if`. La sintaxis general en tal caso es:

```
if (condición1) {
  acción-verdad1
} else if (condición2) {
  acción-verdad2
} else if (condición3) {
  acción-verdad3
} else
  acción-falso
```

Veamos algunos ejemplos:

```
> saludo<-function(franja=NA,nombre)
+ {
+   if (is.na(franja)) paste("Hola", nombre)
```

[10] Debes tener en cuenta que si no se escribe `else` y la condición no se cumple entonces la sentencia `if` devolvería como resultado `NULL`.

```
+   else if (franja==1) paste("Buenos días", nombre)
+   else if (franja==2) paste("Buenas tardes", nombre)
+   else if (franja==3) paste("Buenas noches", nombre)
+   else {
+     warning("Franja puede ser 1 (mañana), 2 (tarde) ó 3 (noche)")
+     paste("Hola", nombre)
+   }
+ }
> saludo(nombre='Lola')

[1] "Hola Lola"

> saludo(franja=1,nombre='Lola')

[1] "Buenos días Lola"

> saludo(franja=4,nombre='Lola')

Warning in saludo(franja = 4, nombre = "Lola"): Franja puede ser 1 (mañana),
2 (tarde) ó 3 (noche)

[1] "Hola Lola"
```

Las condiciones lógicas a evaluar pueden escribirse usando los operadores dobles `&&` y `||`, en lugar de `&` y `|`. Manteniendo el mismo significado, los operadores dobles pueden ser más convenientes según muestra el siguiente ejemplo:

```
> x<-5
> if (x>4 | n>1) "se comprueban las dos"

Error in eval(expr, envir, enclos): objeto 'n' no encontrado

> if (x>4 || n>1) "se comprueba sólo la primera"

[1] "se comprueba sólo la primera"
```

Observa que en el primer caso se comprueban `x>4` y `n>1`, como el objeto `n` no existe se genera un error. En el segundo caso, sin embargo, se evalúa primero `x>4` y como se cumple ya no se evalúa `n>1`, esto impide que se genere el error anterior. Esto muestra que el doble operador puede en muchos ser más eficiente y rápido para evaluar condiciones.

Ejercicio 3: Crea una función que nombre `raiz` calcule la raíz cuadrada de un vector de números positivos. La función tendrá un sólo argumento (`x`), el vector, y deberá comprobar que el argumento que pasamos es un vector numérico y que todos sus elementos son positivos.

La sentencia `ifelse`

En la estructura `if` las condiciones a evaluar deben corresponder a expresiones lógicas que devuelvan un vector (lógico) de longitud 1. Existe una versión vectorial para condiciones que devuelven vectores de longitud mayor. Se trata de la función `ifelse` cuya sintaxis general es:

```
ifelse (condición, a,b)
```

La siguiente función muestra un ejemplo:

```
> milog<-function(x) ifelse(x==0,NA,log(x))
> milog(c(1,0,2))

[1] 0.0000000        NA 0.6931472

> milog(c(1,0,NA))

[1]  0 NA NA
```

Observa que la condición `x==0` se evalúa para cada elemento del vector `x[i]`, imprimiéndose `NA` o `log(x[i])`, para `TRUE` o `FALSE`, respectivamente. Observa lo que ocurriría si en este ejemplo sustituimos `ifelse` por la estructura `if` que hemos visto antes:

```
> milog<-function(x) if(x==0) NA else log(x)
> milog(c(1,0,2))

Warning in if (x == 0) NA else log(x): la condición tiene longitud >1 y sólo
el primer elemento será usado

[1] 0.0000000      -Inf 0.6931472
```

En este caso, dado que `if` está diseñado para condiciones cuyo resultado es un único `TRUE` o `FALSE`, cuando pasamos el vector `x=c(1,0,2)`, en lugar de darnos un error, evalúa la condición tan sólo para su primer elemento (`x[1]=1`) y devuelve el resultado correspondiente mostrando un mensaje de advertencia. Este uso inapropiado de `if` puede constituir la raíz de muchos problemas difíciles de detectar en la práctica. Una forma de evitarlo es pedir a R que en estos casos genere un error, lo cual se puede hacer con la sentencia `Sys.setenv("_R_CHECK_LENGTH_1_CONDITION_" = "true")` (compruébalo en el ejemplo anterior).

Sentencia `switch`

La sentencia `switch` está relacionada con la estructura `if` y permite escribir de una forma más compacta el código en situaciones donde se consideran varias acciones dependiendo del valor que tome una variable.

Considera por ejemplo la siguiente función escrita con `if`:

```
> opciones <- function(x) {
+   if (x == "a") {
+     "opción 1"
+   } else if (x == "b") {
+     "opción 2"
+   } else if (x == "c") {
+     "opción 3"
+   } else {
+     stop("Valor de x no válido")
+   }
+ }
```

Usando la sentencia `switch` podemos escribir la función de esta forma más compacta:

```
> opciones <- function(x) {
+   switch(x,
+     a = "opción 1",
+     b = "opción 2",
+     c = "opción 3",
+     stop("Valor de x no válido")
+   )
+ }
```

La sintaxis general de esta función es:

`switch(expresión,valor_1=acción_1,valor_2=acción_2,...,valor_n=acción_n)`

donde `expresión` debe devolver un escalar (vector de longitud 1) de tipo carácter (como en el ejemplo anterior) o numérico. Si es de tipo numérico entonces su valor se convierte en un número entero (truncándolo) y se ejecuta la acción en la lista correspondiente a dicho valor o en dicha posición. Observa este ejemplo donde la expresión es numérica:

```
> opciones <- function(x) switch(x, 'uno','dos','tres','cuatro')
> opciones(2)

[1] "dos"

> opciones(1.4)

[1] "uno"

> opciones(5)
> opciones('a')

Error in switch(x, "uno", "dos", "tres", "cuatro"): numeric EXPR required for
'switch' without named alternatives
```

Observa que cuando pasamos el argumento `x=5`, como solo hay cuatro componentes en la lista de acciones no imprime nada, de hecho en este caso la función devuelve `NULL`. En el último caso la evaluación genera un error ya que el uso de la función `switch` en este caso espera un valor numérico. Esto se podría corregir comprobando antes el tipo de objeto que se pasa como argumento a la función (por ejemplo con `is.numeric`).

Aunque el ejemplo anterior nos sirve para comprender cómo funciona la sentencia `switch`, un resultado similar se puede obtener utilizando el operador `[`. Observa la siguiente versión de la función anterior:

```
> opciones <- function(x)
+ {
+   resultado<-c('uno','dos','tres','cuatro')
+   resultado[x]
+ }
> opciones(2)

[1] "dos"

> opciones(1.4)

[1] "uno"

> opciones(5)

[1] NA

> opciones('a')

[1] NA
```

Ejercicio 4: Crea una versión de la función `saludo` que construímos antes usando la función `switch`.

3.2.2. Estructuras de repetición

El lenguaje R proporciona estructuras de repetición (ciclos o bucles) que permiten repeticiones un número determinado de veces (`for`), mientras se cumple una condición (`while`), y repeticiones infinitas (`repeat`).

Repeticiones un número determinado de veces

La sintaxis básica de los ciclos `for` responde a la siguiente estructura:

```
for (variable in valores)
{
  sentencias
}
```

donde `variable` es habitualmente un índice a evaluar en la secuencia `valores`. Ejemplos:

```
> for (i in 1:3) print(i^2)

[1] 1
[1] 4
[1] 9

> for (i in letters[1:3]) print(i)

[1] "a"
[1] "b"
[1] "c"
```

Los valores donde se evalúa el índice corresponden en los ejemplos anteriores a un vector, no obstante es posible utilizar otra estructura de datos más compleja (por ejemplo una matriz o incluso una lista) como muestra el siguiente ejemplo:

```
> lista<-list(matrix(1:6,2,3),1:2)
> for (i in lista) print(i)

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

El ciclo se puede interrumpir en cualquier momento usando `break`, `next` o `return`. Esta última opción está asociada a funciones. La interrupción del ciclo típicamente vendrá asociada a una estructura de control `if`. A continuación mostramos algunos ejemplos:

```
> for (i in 1:5) if (i%%2==0) print(i) else next

[1] 2
[1] 4
```

```
> set.seed(1)
> x<-runif(100)
> suma<-0
```

```
> for (i in 1:100)
+ {
+   suma<-suma+x[i]
+   if (suma>0.5) {
+     print(paste("Me paro en i=",i,"porque la suma supera 0.5"))
+     break
+   }
+ }

[1] "Me paro en i= 2 porque la suma supera 0.5"
```

```
> f<-function(x)
+ {
+   suma<-0
+   for (i in 1:length(x))
+   {
+     suma<-suma+x[i]
+     if (suma>0.5) return(paste("Me paro en i=",i,"porque la suma supera 0.5"))
+   }
+   return(paste("He completado el ciclo y la suma es", suma))
+ }
> f(rep(0.1,4))

[1] "He completado el ciclo y la suma es 0.4"

> f(rep(0.1,10))

[1] "Me paro en i= 6 porque la suma supera 0.5"
```

Repeticiones mientras se cumple una condición

La estructura `while` permite este tipo de repeticiones. Su sintaxis responde a la siguiente estructura:

```
while (condición)
{
  sentencias
}
```

donde `condición` debe ser una expresión lógica que devuelva un vector lógico de longitud uno, esto es, `TRUE` o `FALSE`.

Al igual que en los ciclos `for` es posible interrumpir el ciclo `while` utilizando `break`, `next` (o `return` dentro de funciones). Observa los siguientes ejemplos:

```
> set.seed(1)
> i<-suma<-0
> x<-numeric()
> while (suma<0.5)
+ {
+   i<-i+1
+   x[i]<-runif(1,0,0.1) # uniforme en (0,0.1)
+   suma<-suma+x[i]
+ }
> print(round(x,4))

[1] 0.0266 0.0372 0.0573 0.0908 0.0202 0.0898 0.0945 0.0661 0.0629
```

Repeticiones infinitas

La estructura que permite este tipo de repeticiones es `repeat`. No obstante se entiende que construiremos estructuras que incluyan condiciones de salida o interrupción, por ejemplo con `break`. El siguiente ejemplo es equivalente al que escribíamos antes con la estructura `while`:

```
> set.seed(1)
> i<-suma<-0
> x<-numeric()
> repeat
+ {
+   i<-i+1
+   x[i]<-runif(1,0,0.1) # uniforme en (0,0.1)
+   suma<-suma+x[i]
+   if (suma>=0.5) break
+ }
> print(round(x,4))

[1] 0.0266 0.0372 0.0573 0.0908 0.0202 0.0898 0.0945 0.0661 0.0629
```

3.2.3. Algunas notas sobre los ciclos en R

Es posible reescribir cualquier ciclo `for` usando `while`. También, cualquier ciclo `while` se puede escribir usando `repeat`. Esto significa que `while` es más flexible que `for`, y que `repeat` es más flexible que `while`. Una buena práctica en la programación en R es usar siempre la solución menos flexible a un problema, lo que nos lleva a elegir `for` siempre que sea posible.

La misma regla anterior nos llevará a evitar ciclos `for` cuando podamos resolver el problema usando funciones vectoriales, o adecuadas para la estructura de datos concreta que estemos trabajando. Para comprender mejor esto piensa en el problema de calcular la media de un vector de datos. En la mayoría de lenguajes de programación este cálculo lo haríamos usando un ciclo `for` como se ilustra a continuación:

```
> x<-1:5
> suma<-0
> for (i in 1:length(x)) suma<-suma+x[i]
> suma/length(x)

[1] 3
```

Esto en R en cambio esto se puede hacer usando la función vectorial `sum` [11] que toma como argumento el vector completo:

```
> sum(x)/length(x)

[1] 3
```

Otro ejemplo sería el cálculo de las sumas por filas (o columnas de una matriz). Nuestra primera tentación (acostumbrados a la programación en otros lenguajes) sería escribir:

```
> A<-matrix(1:6,2,3)
> sumas.filas<-numeric(nrow(A))
> for (i in 1:nrow(A)){
+   for (j in 1:ncol(A)) sumas.filas[i]<-sumas.filas[i]+A[i,j]
+ }
> sumas.filas

[1]  9 12
```

En R disponemos de las funciones `rowSums` y `colSums` que realizan esta tarea sobre la matriz completa:

```
> rowSums(A)

[1]  9 12
```

Además de estas funciones hemos utilizado previamente otras como `lapply` o `aggregate` que permiten trabajar sobre estructuras de datos más complejas considerándolas como un todo. A continuación describimos algunas de las más relevantes para nuestros objetivos.

[11] Por supuesto también tenemos `mean`.

3.2.4. Familia `apply` y otras funciones que operan sobre estructuras de datos

La familia de funciones `apply`, así como otras funciones de R asociadas a tareas de clasificación y agrupación, permiten operar sobre estructuras de datos completas del tipo matrices, data frame o listas. En muchos casos nos permitirán evitar construir ciclos `for` y realizar una programación más eficiente y compacta. A continuación describimos algunas relevantes:

Funciones `lapply` y `sapply`

Estas funciones permiten realizar operaciones sobre cada uno de los elementos de una lista o un data frame, dado como argumento.

La sintaxis de `lapply` es

```
lapply(X,FUN,...)
```

Como resultado devuelve una lista de la misma longitud que la pasada en el argumento `X`, donde cada elemento es el resultado de aplicar la función pasada en el argumento `FUN` al correspondiente elemento en `X`. Ejemplo:

```
> lista<-list(v1=1:10,v2=factor(),v3=letters[1:4])
> lapply(lista,length)

$v1
[1] 10

$v2
[1] 0

$v3
[1] 4

> df<-data.frame(matrix(1:6,2,3))
> lapply(df,mean)

$X1
[1] 1.5

$X2
[1] 3.5

$X3
[1] 5.5
```

Observa que el último ejemplo podría resolverse también con `colMeans` (escribe cómo sería).

La sintaxis de `sapply` [12] es muy similar pero contiene el argumento `simplify` (por defecto `TRUE`[13]) que permite simplificar el resultado devuelto a un objeto de datos más simple (vector o matriz). Observa el resultado con el primer ejemplo anterior:

```
> sapply(lista,length)

v1 v2 v3
10  0  4
```

Función `apply`

La función `apply` permite realizar operaciones marginales sobre matrices o arrays. Su sintaxis es:

```
apply(X, MARGIN, FUN,..., simplify=TRUE)
```

donde el argumento `X` corresponde a la matriz (o array) sobre el que vamos a operar, `FUN` sería la función que implementa las operaciones a realizar y `MARGIN` especificaría la dimensión donde vamos a operar. Si `X` es una matriz entonces `MARGIN=1` operaría por filas, `MARGIN=2` por columnas, y `MARGIN=c(1,2)` lo haría por filas y columnas. La función `apply` devolverá como resultado por defecto un objeto de datos lo más simple (`simplify=TRUE`). Los siguientes ejemplos ilustran el uso de esta función:

```
> A<-cbind(1:4,seq(0,1,length.out=4),(1:4)^2)
> apply(A,1,median) # mediana por filas

[1] 1 2 3 4

> apply(A,2,var) # cuasivarianza por columnas

[1]  1.6666667  0.1851852 43.0000000

> apply(A,c(1,2),sqrt)

         [,1]      [,2] [,3]
[1,] 1.000000 0.0000000    1
[2,] 1.414214 0.5773503    2
[3,] 1.732051 0.8164966    3
[4,] 2.000000 1.0000000    4
```

[12] `vapply` es otra versión similar de la función.
[13] La función `sapply` con argumento `simplify=FALSE` es equivalente a `lapply`.

Recuerda que cuando la tarea consiste en obtener sumas o medias de una matriz por filas o columnas, aunque sería posible resolverla usando `apply` es mejor optar por las funciones específicas para estas tareas: `rowSums`, `colSums`, `rowMeans` y `colMeans`.

Función `split`

Esta función permite clasificar datos consistentes en vectores, matrices o data frames de acuerdo a un criterio de clasificación. Su sintaxis básica es:

```
split(x, f, ...)
```

donde `x` es el objeto a clasificar y `f` el criterio que define los grupos.

Como primer ejemplo consideremos el data frame `Orange` en el paquete *datasets*. Los datos corresponden a 5 árboles para los que se han realizado varias mediciones de su edad y la longitud de la circunferencia de su tronco. Los datos de cada árbol están identificados en la primera columna del data frame (`Orange$Tree`). Nuestra tarea es dividir el data frame en cinco data frames, conteniendo los datos de cada árbol por separado, e imprimir las dos primeras filas de cada uno. Esto podemos resolverlo con `split` y `lapply`:

```
> arboles<-split(Orange,Orange$Tree)
> lapply(arboles,head,n=2)

$`3`
   Tree age circumference
15    3 118            30
16    3 484            51

$`1`
  Tree age circumference
1    1 118            30
2    1 484            58

$`5`
   Tree age circumference
29    5 118            30
30    5 484            49

$`2`
  Tree age circumference
8    2 118            33
9    2 484            69

$`4`
   Tree age circumference
```

```
22    4 118            32
23    4 484            62
```

Como segundo ejemplo consideramos la tarea consistente primero en obtener y mostrar las diagonales de una matriz cuadrada, y después calcular sus sumas. Esto podemos resolverlo con `split` y `sapply` como se muestra a continuación:

```
> A<-matrix(1:9,3,3)
> diagonales<-split(A,col(A)-row(A))
> diagonales

$`-2`
[1] 3

$`-1`
[1] 2 6

$`0`
[1] 1 5 9

$`1`
[1] 4 8

$`2`
[1] 7

> sapply(diagonales,sum)

-2 -1  0  1  2
 3  8 15 12  7
```

Funciones de clasificación y agrupación: `by`, `aggregate` y `tapply`

La función `aggregate` permite resumir columnas de un data frame para cada uno de los niveles de un factor. Como resultado devuelve un data frame aunque, si es posible, el resultado se puede simplificar a un vector o matriz usando el argumento `simplify=TRUE`.

```
> aggregate(Orange$age,by=list(Orange$Tree),summary)

  Group.1    x.Min. x.1st Qu.  x.Median    x.Mean x.3rd Qu.    x.Max.
1       3  118.0000  574.0000 1004.0000  922.1429 1301.5000 1582.0000
2       1  118.0000  574.0000 1004.0000  922.1429 1301.5000 1582.0000
3       5  118.0000  574.0000 1004.0000  922.1429 1301.5000 1582.0000
```

```
4       2  118.0000  574.0000 1004.0000  922.1429 1301.5000 1582.0000
5       4  118.0000  574.0000 1004.0000  922.1429 1301.5000 1582.0000
```

La función `by` es similar a la anterior. Tiene tres argumentos principales: el objeto al cual se aplica (típicamente un data frame), el factor o vector usado para clasificar y la función que se aplica a cada uno de los elementos resultantes de la clasificación. Observa la diferencia de resultado con respecto al que ofrecía `aggregate` en el ejemplo anterior.

```
> by(Orange$age,Orange$Tree,summary)

Orange$Tree: 3
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  118.0   574.0  1004.0   922.1  1301.5  1582.0
------------------------------------------------------------------------
Orange$Tree: 1
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  118.0   574.0  1004.0   922.1  1301.5  1582.0
------------------------------------------------------------------------
Orange$Tree: 5
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  118.0   574.0  1004.0   922.1  1301.5  1582.0
------------------------------------------------------------------------
Orange$Tree: 2
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  118.0   574.0  1004.0   922.1  1301.5  1582.0
------------------------------------------------------------------------
Orange$Tree: 4
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  118.0   574.0  1004.0   922.1  1301.5  1582.0
```

La función `tapply` nos da otra solución al problema anterior devolviendo una lista:

```
> tapply(Orange$age,Orange$Tree,summary)

$`3`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  118.0   574.0  1004.0   922.1  1301.5  1582.0

$`1`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  118.0   574.0  1004.0   922.1  1301.5  1582.0

$`5`
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  118.0   574.0  1004.0   922.1  1301.5  1582.0

$`2`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  118.0   574.0  1004.0   922.1  1301.5  1582.0

$`4`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  118.0   574.0  1004.0   922.1  1301.5  1582.0
```

Puedes comprobar otras opciones de las funciones anteriores en la ayuda de las mismas.

3.2.5. Recursividad

En términos generales podemos entender el concepto de recursividad como la posibilidad de que la definición de una función pueda hacer referencia a la propia función. Esto permite crear estructuras definidas fácilmente por recursividad. Esta posibilidad sin embargo consume muchos recursos por lo que deben siempre valorarse otras alternativas. En la sección 3.4 ilustraremos su uso en algunos ejemplos.

3.3. Manipulación y depuración de errores

Sentencias para la manipulación de errores

En la programación de funciones a menudo debemos hacer uso de estructuras de control para manejar posibles errores, generando mensajes de error o advertencia al usuario. Las funciones `stop` o `warning` permiten producir este tipo de mensajes. La función `stop` interrumpe la ejecución del código y devuelve un mensaje con la cadena de caracteres que se le pase como argumento. La función `warning` devuelve el mensaje pero no interrumpe la ejecución. A continuación mostramos algunos ejemplos de uso.

```
> log.natural<-function(x)
+ {
+   if (missing(x) || !is.numeric(x)) stop("Debe proporcionar
                                          un argumento 'x' numérico")
+   else if (any(x<=0)){
+     x<-x[x>0]
+     warning("Se han eliminado los valores negativos o cero")
+   }
+   return(log(x))
+ }
> log.natural()
```

```
Error in log.natural(): Debe proporcionar
un argumento 'x' numérico

> log.natural(-1:5)

Warning in log.natural(-1:5): Se han eliminado los valores negativos o cero

[1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379
```

Por otro lado la función `options` permite controlar diversas opciones de R relativas al modo en que R calcula y muestra los resultados. Escribiendo `options()` puedes ver las opciones establecidas por defecto en la sesión de R. Entre ellas la componente `options()$error` especifica el modo en que se manejan errores relacionados con la función `stop` y otros errores internos. Puedes comprobar su funcionamiento consultado por ejemplo la ayuda de la función.

Depuración de errores: *debugging*

R proporciona herramientas para ayudar al usuario a encontrar las causas de posibles errores en el código. En general se trata de funciones que permiten parar la ejecución de las sentencias en puntos particulares de modo que los cálculos previos puedan ser inspeccionados, así como las variables en el entorno de evaluación de las funciones. Entre estas herramientas destacamos las funciones `traceback`, `browser` y `debug`.

La función `traceback`

Esta función permite reconstruir el camino seguido por R hasta que encontró el error. Observa la siguiente función y el error obtenido en su llamada:

```
> f<-function(x)
+ {
+   n<-round(x)
+   set.seed(n)
+   u<-runif(n)
+   return(u)
+ }
> mean(f(-2))

Error in runif(n): invalid arguments
```

Si escribes ahora `traceback()` obtendrás:

```
3: runif(n) at #6
2: f(-2)
1: mean(f(-2))
```

Verás que te muestra el camino seguido hasta el error, si lees las tres líneas anteriores desde abajo hasta arriba.

Debes tener en cuenta que esta función ofrece información sobre el último error que ha ocurrido. Por tanto sólo tiene sentido usarla cuando ocurrió dicho error.

La función `browser`

Esta función permite realizar una ejecución interactiva. lo que puede facilitar la tarea de buscar la causa de un error.

```
> f<-function(x)
+ {
+   browser()
+   n<-round(x)
+   set.seed(n)
+   u<-runif(n)
+   return(u)
+ }
> mean(f(-2))

Called from: f(-2)
debug en <text>#4: n <- round(x)
debug en <text>#5: set.seed(n)
debug en <text>#6: u <- runif(n)

Error in runif(n): invalid arguments
```

Observa que en la ejecución anterior sabemos que se realiza de forma interactiva[14] cuando obtenemos el *prompt* especial:

```
Browse[1]>
```

Durante la ejecución interactiva podemos comprobar los valores de las variables locales usando la función `get` como muestra la siguiente secuencia:

```
> mean(f(-2))
Called from: f(-2)
Browse[1]>
debug en #4: n <- round(x)
Browse[2]>
debug en #5: set.seed(n)
```

[14] Aunque abajo mostramos el resultado de la ejecución paso a paso que nos ofrece `browser()` lo interesante es que lo pruebes en tu ordenador de forma interactiva. Ten en cuenta que se ejecutará una línea cada vez y para ir a la siguiente debes pulsar la tecla enter.

```
Browse[2]> get('n')
[1] -2
Browse[2]> Q
```

En esta secuencia hemos escrito `get('n')` para comprobar el valor que toma la variable local `n` una vez creada. También hemos usado que, escribiendo `Q` durante la ejecución interactiva, ésta se para inmediatamente sin ejecutar el resto de las sentencias. También, durante la ejecución interactiva, podemos escribir `c` lo que supondría continuar la ejecución hasta el final sin paradas.

La función `debug`

Se trata de una función muy útil que aplicada a una función concreta hace que su evaluación se realice paso a paso, pudiendo observarse en el proceso los valores que van tomando las variables involucradas.

Observa que una vez solicitada esta evaluación interactiva con `debug`, se realizará cada vez que evaluemos la función hasta que indiquemos que queremos dejar de hacerlo con la función `undebug`. El siguiente ejemplo muestra la ejecución interactiva de las funciones `mean` y `f` creada anteriormente, que son evaluadas en la misma sentencia (`mean(f(2))`).

```
> f<-function(x)
+ {
+   n<-round(x)
+   set.seed(n)
+   u<-runif(n)
+   return(u)
+ }
> debug(f)
> mean(f(2))

debugging in: f(2)
debug en <text>#1: {
    n <- round(x)
    set.seed(n)
    u <- runif(n)
    return(u)
}
debug en <text>#3: n <- round(x)
debug en <text>#4: set.seed(n)
debug en <text>#5: u <- runif(n)
debug en <text>#6: return(u)
exiting from: f(2)
[1] 0.4436281

> undebug(f)
```

Observa que en la salida anterior tan sólo se muestra la ejecución interactiva de la función `f` (como hemos solicitado). Prueba a incluir también la ejecución interactiva de la función `mean` escribiendo:

```
debug(f)
debug(mean)
mean(f(2))
undebug(f)
undebug(mean)
```

También disponemos de la función `debugonce` que, con la misma finalidad que `debug`, indicaría que queremos que se realice la evaluación interactiva solo la primera vez que evaluemos la función (lo que evitaría tener que usar `undebug` en el ejemplo anterior).

Por último comentar que el IDE RStudio proporciona además otras herramientas para la depuración de errores que pueden resultar más cómodas. Aunque hasta ahora no habíamos recomendado su uso quizá sea ahora un buen momento para empezar a usarlo y beneficiarnos de todas las herramientas que proporciona. Te proponemos que ejecutes los ejemplos de esta sección desde RStudio.

3.4. Algunos ejemplos de funciones

A continuación describimos algunos ejemplos que ilustran la definición de funciones en R y las estructuras de programación básica estudiadas anteriormente. En las sesiones de prácticas describiremos algunos más.

Factorial de un número

A continuación presentamos la función `factorial.d` que calcula el factorial[15] de un número natural n. La implementación utiliza un ciclo `for` y responde a la definición $n! = n(n-1)\cdots 1$, siendo $0! = 1$.

```
> factorial.d<-function(n)
+ {
+  if (missing(n) || !is.numeric(n)) return(NA)
+  n<-as.integer(n)
+  fact<-1
+  if (n>1) for (i in 1:n) fact<-fact*i
+  return(fact)
+ }
> factorial.d(5)
```

[15] R proporciona en el sistema base la función `factorial` que calcula el factorial de n evaluando la función `gamma` en $n+1$.

```
[1] 120

> factorial.d(0)

[1] 1

> factorial.d(100)

[1] 9.332622e+157

> factorial.d(5000)

[1] Inf
```

Observa que en el último caso ($n = 5000$) el cálculo da lugar a `Inf` (*overflow*) ya que se trataría de un valor muy por encima de mayor valor que puede calcular R (`.Machine$double.xmax`).

Las siguientes son dos versiones de la función anterior: una versión recursiva (`factorial.r`), y otra vectorial (`factorial.v`), que genera el vector de enteros entre 1 y n antes de multiplicarlos.

```
> factorial.r<-function(n)
+ {
+  if (missing(n) || !is.numeric(n)) return(NA)
+  n<-as.integer(n)
+  if (n>1) fact<-n*factorial.r(n-1) else fact<-1
+  return(fact)
+ }
> factorial.r(5)

[1] 120

> factorial.r(100)

[1] 9.332622e+157

> factorial.r(5000)

Error: evaluación anidada demasiado profunda; recursión infinita options(expressions=
)?
```

La versión recursiva consume muchos recursos lo que conlleva el error mostrado antes cuando calculamos el factorial de 5000. La versión vectorial que mostramos a continuación

no presenta este problema sin embargo consume más memoria que la versión `factorial.d`, ya que requiere almacenar el vector `n:1` antes de hacer el producto.

```
> factorial.v<-function(n)
+ {
+  if (missing(n) || !is.numeric(n)) return(NA)
+  n<-as.integer(n)
+  if (n>1) fact<-prod(n:1) else fact<-1
+  return(fact)
+ }
> factorial.v(5)

[1] 120

> factorial.v(100)

[1] 9.332622e+157

> factorial.v(5000)

[1] Inf
```

Finalmente añadimos un medidor de tiempo a las versiones `factorial.d` y `factorial.v` para ver cuál es más eficiente en términos de tiempo de cálculo.

```
> factorial.d<-function(n)
+ {
+  if (missing(n) || !is.numeric(n)) return(NA)
+  n<-as.integer(n)
+  t1<-Sys.time() # tiempo inicial
+  fact<-1
+  if (n>1) for (i in 1:n) fact<-fact*i
+  t2<-Sys.time() # tiempo final
+  return(list(factorial=fact,tiempo=t2-t1))
+ }
> factorial.v<-function(n)
+ {
+  if (missing(n) || !is.numeric(n)) return(NA)
+  n<-as.integer(n)
+  t1<-Sys.time() # tiempo inicial
+  if (n>1) fact<-prod(n:1) else fact<-1
+  t2<-Sys.time() # tiempo final
+  return(list(factorial=fact,tiempo=t2-t1))
+ }
```

```
> factorial.d(10000)

$factorial
[1] Inf

$tiempo
Time difference of 0 secs

> factorial.v(10000)

$factorial
[1] Inf

$tiempo
Time difference of 0.0009977818 secs
```

Sucesión de Fibonacci

La sucesión Fibonacci es la sucesión infinita de números naturales:

$$\{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots\},$$

comenzando por 0 y 1, cada elemento de la sucesión se obtiene sumando los dos que le preceden. Esto nos lleva a la siguiente definición recursiva de la serie $\{F_n : n \geq 0\}$:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \quad (n \geq 2) \end{aligned}$$

A continuación presentamos dos versiones de una función que devuelve el elemento F_n de la sucesión de Fibonacci. La primera (`Fibonacci.r`) utiliza recursividad:

```
> Fibonacci.r<-function(n)
+ {
+  if (missing(n) || !is.numeric(n)) return(NA)
+  n<-as.integer(n)
+  if (n==0) return(0)
+  else if (n<=2) return(1)
+  else return(Fibonacci.r(n-1)+Fibonacci.r(n-2))
+ }
> Fibonacci.r(5)
```

```
[1] 5

> Fibonacci.r(20)

[1] 6765
```

Observa que la versión recursiva consume muchos recursos y tiempo. Compruébalo evaluando `Fibonacci.r(30)` y `Fibonacci.r(300)`.

La siguiente versión (`Fibonacci.v`) evita la opción recursiva pura anterior, calculando y almacenando los elementos $\{F_0, \ldots, F_n\}$ en un vector:

```
> Fibonacci.v<-function(n)
+ {
+  if (missing(n) || !is.numeric(n)) return(NA)
+  n<-as.integer(n)
+  v<-integer(n)
+  v[1:2]<-1
+  for (i in 3:n) v[i]<-v[i-1]+v[i-2]
+  return(v[n])
+ }
> Fibonacci.v(5)

[1] 5

> Fibonacci.v(20)

[1] 6765
```

Aunque la versión `Fibonacci.v` requiere colocar en memoria un vector potencialmente grande, resulta mucho más rápida y eficiente que la versión recursiva anterior. Puedes comprobarlo añadiendo un medidor de tiempo en ambas funciones y evaluándolas en $n = 30$ y $n = 300$.

Ejercicio 4: Construye una versión de la función `Fibonacci.v` que devuelva una lista con tres componentes: el elemento F_n, el vector de elementos $\{F_0, \ldots, F_n\}$ y la suma de dichos elementos.