Tema 3:

Funciones

Como ya se indicó anteriormente, las funciones permiten realizar las diferentes acciones. Existen muchas funciones ya definidas, algunas incluso pueden ser modificadas, incluso accidentalmente (ver la función **search**), pero una de las capacidades más interesante es la posibilidad de crear nuevas funciones que realicen tareas que no estaban definidas en el momento de instalar el programa. Estas nuevas funciones se incorporan al lenguaje y, desde ese momento, se utilizan como las previamente existentes. De este modo, cualquier persona que escribe una función puede difundirla y puede ser incorporada por otras personas. La lectura de la definición de las diferentes funciones incorporadas en origen es una tarea muy formativa, ya que están escritas por especialistas en el lenguaje y, además, sufren un proceso de depuración a través de las quejas de los usuarios, por lo que se puede decir que, en general, están muy bien escritas. Para leer la definición, como ya sabe, basta con escribir el nombre de la función sin paréntesis. Por ejemplo, la definición de la función que realiza el cálculo de la varianza, se obtiene escribiendo **var**. Antes de estudiar la definición conviene obtener ayuda sobre qué es lo que hace la función, mediante la función **help**. Si necesita ejecutar un programa del sistema operativo puede utilizar la función **system**, que permite incluso salir al sistema operativo con la orden **system("{}cmd")**. También puede utilizar la orden **shell**, como por ejemplo en **shell("dir")**.

Las funciones tradicionales en cualquier lenguaje de uso matemático y estadístico, están definidas. Entre ellas se encuentran, entre otras, **abs**, **sqrt**, **sin**, **cos**, **tan**, **asin**, **acos**, **atan, exp**, **log**, **log10**, **min**, **max**, **sum**, **prod**, **length**, **mean**, **range**, **median**, **var**, **cov**, **summary**, **sort**, **rev**, **order**; cuyas definiciones abreviadas puede ver en el apéndice y ampliadas mediante la orden **help** en el propio programa, y que puede utilizar inmediatamente.

```
> sin(1:10)
[1] 0.8414710 0.9092974 0.1411200 -0.7568025 -0.9589243
[8] -0.2794155 0.6569866 0.9893582 0.4121185 -0.5440211
> sin(pi)
[1] 1.224606e-16
> cos(pi)
[1] -1
> sum(1:10)
[1] 55
> prod(1:10)
[1] 3628800
```

3.1. Definición de una función

Una función se define asignando a un objeto la palabra **function** seguida de los argumentos que desee dar a la función, escritos entre paréntesis y separados por comas, seguida de la orden, entre llaves si son varias órdenes, que desee asignar a la misma. Entre los argumentos destaca que si utiliza tres puntos seguidos, **...**, ello indica que el número de argumentos es arbitrario.

Por ejemplo, la siguiente instrucción, define una función, llamada **funcionsuma**, que calcula la suma de dos números que se le pasan como argumentos, aunque si alguno de ellos no se indica se le asigna un valor predeterminado. Esta suma se devuelve como resultado de la

función. Advierta que el objeto aparece en la lista de objetos si se utiliza la función **ls**, y que su definición puede verse como la de cualquier otra función. Antes de asignar una función a un objeto, como **funcionsuma**, es conveniente intentar obtener su definición, por si ya existiese, ya que en caso afirmativo la nueva definición sustituiría a la antigua, puede que incluso irreparablemente.

```
> funcionsuma
Error: Object "funcionsuma" not found
> funcionsuma<-function(A=1,B=2) A+B
> funcionsuma()
[1] 3
> funcionsuma(5,7)
[1] 12
> funcionsuma
function(A = 1, B = 2)
A + B
```

Debemos destacar, en primer lugar, que la función que acabamos de definir no se refiere a la suma de dos números, sino a la suma de dos objetos, que podrán ser vectores, matrices, variables indexadas, etc.

Las funciones que tienen prevista la actuación de modo diferente según el tipo o el número de argumentos que se les suministre se denominan **genéricas** y son de importancia fundamental.

```
> funcionsuma(1:9,2:4)
[1] 3 5 7 6 8 10 9 11 13
```

En segundo lugar, la función lo que realiza es la orden que tiene a continuación (una sola) por lo que para escribir varias órdenes deberán agruparse entre llaves, **{}**, y el valor que devuelve la función es el correspondiente a la última que ejecuta.

Compruebe los siguientes ejemplos, variaciones sobre este ejemplo elemental:

```
> f1 = function(A=1,B=2)
+ {
+ #Devuelve A+B
+ 46
+ 25
+ A+B
+ }
> f2 = function(A=1,B=2)
+ {
+ #Devuelve 25
+ A+B
+ 46
+ 25
+ }
> f3 = function(A=1,B=2)
+ {
+ #Devuelve B, porque A termina la línea por sí misma y +B es B
```

```
+ A
+ +
+ B
+ }
> f4 = function(A=1,B=2)
+ {
+ #Devuelve en una lista A+B y (A+B)^2
+ list(Suma= A+B,Cuadrado= (A+B)^2)
+ }
> f5 = function(A=1,B=2)
+ {
+ #Devuelve A+B
+ A-B
+ A+B
+ }
```

Si desea poner de manifiesto el valor devuelto, puede utilizar la función **return** que termina la función y devuelve su argumento. Cuando se encuentra esta función se detiene la ejecución y se devuelven los valores indicados, por lo que es muy útil en programación cuando existen bifurcaciones.

```
> f1 = function(A=1,B=2)
+ {
+ #Devuelve A+B
+ return(A+B)
+ }
```

Es posible acceder a los argumentos de una función mediante la función **formals** y al cuerpo de la misma mediante la función **body**.

```
> f1
function (A = 1, B = 2)
{
A + B
}
> #Almacena los valores actuales en argumentos
> formals(f1) -> argumentos
> #Modifica los valores
> formals(f1)=alist(X=,Y=-1)
> f1
function (X, Y = -1)
{
A + B
}
> #Recupera los valores almacenados
> formals(f1) = argumentos
> f1
function (A = 1, B = 2)
{
A + B
}
```

```
> #Almacena los valores actuales en cuerpo
> body(f1) -> cuerpo
> #Modifica los valores
> body(f1) <- expression(A*B)
> f1
function (A = 1, B = 2)
A * B
> #Recupera los valores almacenados
> body(f1) = cuerpo
> f1
function (A = 1, B = 2)
{
A + B
}
>
```

3.2. Algunas funciones elementales

A continuación aparecen algunas funciones elementales, para comprender los rudimentos de la programación. Estas funciones están construídas de modo lineal, esto es, sin bifurcaciones y utilizando solamente aspectos elementales

3.2.1. Función media

En primer lugar, se incluye una definición alternativa a la definición de media incluida en **R**. Es una función de un solo argumento, que si no se especifica será sustituido por el valor **NA**, con lo que, en ese caso, el resultado sería **NA**. En caso de que se le suministre un argumento, no se comprueba si es válido o no, sino que suponiendo que es un vector, se eliminan del mismo los elementos correspondientes a **NA**. Para ello se utiliza la negación **!** y la condición de ser un valor no disponible. Del resto se calcula la media mediante la definición trivial, esto es, el cociente entre la suma de los elementos y la longitud.

```
media<- function(x=NA)
{
x<-x[!is.na(x)]
sum(x)/length(x)
}

> media(c(2,3,7))
[1] 4
> media(c(2,3,7,NA))
[1] 4
```

Advierta que el objeto **x** utilizado dentro del cuerpo de la función es temporal (en Visual Basic corresponde a un argumento pasado **byval**) y desaparece al terminar de ejecutarse la misma. Además no coincide con ningún objeto del mismo nombre y ya existente (se crea en un entorno correspondiente a la función, es como si el objeto **o** dentro de la función **f** fuese **f.o** por lo que no debe preocuparse de si podrá interferir con objetos preexistentes.

Ejercicio. Modifique la función para que devuelva también cuantos elementos había y cuantos ha quitado.

3.2.2. Función varianza

Si quiere calcular la varianza de un vector, puede utilizar la \emph{perversa} fórmula derivada de la relación entre los momentos, $\mu_2 = m_2 - m_1^2$, para definir, de modo análogo al anterior, y usando la función **media** ya existente, la varianza.

```
> koning<- function(x=NA)
{
media(x^2)-(media(x))^2
}
> koning(c(2,3,7))
[1] 4.666667
> koning(c(2,3,7,NA))
[1] 4.666667
```

Es preferible utilizar directamente la definición de la varianza para obtener la siguiente función:

```
> varianza<- function(x=NA)
{
y<-media(x)
media((x-y)^2)
}
> varianza(c(2,3,7))
[1] 4.666667
> varianza(c(2,3,7,NA))
[1] 4.666667
```

que suministra, en los ejemplos considerados, los mismos resultados, pero es más estable en otros casos.

Por supuesto, la escritura de la funci ón puede hacerse más o menos compacta y de modo equivalente:

```
> varianza2<- function(x=NA) media((x-media(x))^2)
> varianza2(c(2,3,7))
[1] 4.666667
```

Ejercicio. Escriba una función que calcule los coeficientes de asimetría y curtosis de Pearson. Sus definiciones son:

$$\gamma_1 = \mu_3/\sigma^3 \qquad \gamma_2 = \mu_4/\sigma^4$$

3.3. Funciones .First y .Last

Estas funciones se ejecutan, si existen, al iniciar una sesión de trabajo y al terminarla, respectivamente. Puede utilizarlas para personalizar su método de trabajo. Los objetos cuyo nombre comienza con un punto, como es el caso de estas dos funciones, no se muestran al utilizar la orden **ls** salvo que se incluya el parámetro **all.names=T**

En **R** el mecanismo de inicio, a grandes rasgos, es el siguiente: Si existe el archivo **.Renviron** se utiliza como un conjunto de órdenes que se ejecutan, salvo que la orden de ejecución incluya la opción **--no-environ**. A continuación comprueba si existe el archivo **.Rprofile** y también lo ejecuta. A continuación carga el archivo **.RData** salvo que se especifique la opción **--no-restore**. A continuación, si existe la función **.First**, se ejecuta.

Así, la siguiente definición de la función **.First**, que utiliza la función **cat** que escribe un texto en pantalla, obliga a que, al inicio de cada sesión de trabajo, se escriba el mensaje **Hola.Bienvenido a R** en pantalla.

```
.First<-
function()
{
cat("Hola.Bienvenido a R\n")
}
```

3.4. Algunos elementos útiles en programación

La programación sin posibilidad de estructuras de programación es muy limitada. No es este el caso, sino que existen gran cantidad de estas estructuras. A continuación encontrará un breve repaso de las mismas.

3.4.1.Operadores de relación

Con **!** se indica la negación, con **&** la conjunción y con **|** la disyunción. Estos dos últimos, si se escriben repetidos tienen el mismo significado, pero se evalúa primero la parte de la izquierda y, si ya se sabe el resultado (suponiendo que se pudiera calcular la expresión de la derecha) no se sigue evaluando, por lo que pueden ser más rápidos y eliminar errores. <, >, <=, >=, ==, son respectivamente los símbolos de menor, mayor, menor o igual, mayor o igual, e igual. Advierta que este último se escribe con dos signos de igualdad.

La programación sin posibilidad de estructuras de programación es muy limitada. No es este el caso, sino que existen gran cantidad de estas estructuras. A continuación encontrará un breve repaso de las mismas.

3.4.2. Estructuras condicionales

Son aquellas que, según el resultado de una comparación, realizan una u otra acción. Recuerde que un conjunto de acciones, entre llaves, se interpretan como una sola acción desde el punto de vista lógico. La primera estructura es

if (condición) acción1 [else acción2]

Esta estructura es **escalar**, ya que si la condición está formada por más de un elemento, sólo considera el primero (y además presentaría un mensaje de advertencia). Por ello en el ejemplo siguiente se incluye la orden **min(x)**, ya que si se aplicase directamente a **x** sólo

consideraría la primera componente del vector de comparación.

Ejemplo.

```
> raiz<-function(x)
+
{
+ if (is.numeric(x) && min(x)>0)
+ # Este símbolo hace que el resto de la línea
+ # se considere un comentario
+ # Advierta que no se intenta aplicar
+ # la función min si el argumento
+ # no es numérico, previniendo un error
+ {sqrt(x)}
+ else {stop("O x no es numerico o no es positivo")}
+
}

> raiz("Pepe")
Error in raiz("Pepe"): O x no es numerico o no es positivo
> raiz(-1)
Error in raiz(-1): O x no es numerico o no es positivo
> raiz(7)
[1] 2.645751
```

Observe que **acción1** y **acción2** son sendas acciones individuales. Si desea que se realicen varias acciones debe incluirlas entre llaves, para que aparezcan como una sola desde el punto de vista lógico. En el ejemplo se han incluido llaves pero podrían eliminarse.

La segunda estructura es

ifelse(condición, acción en caso cierto, acción en caso falso)

Esta estructura es **vectorial**, ya que si la condición está formada por más de un elemento, considera cada uno de ellos.

Así, por ejemplo, suponiendo que el argumento **x** es numérico, tendríamos:

```
> Inverso<-function(x) ifelse(x==0,NA,1/x)
> Inverso(-1:1)
[1] -1 NA 1
> Inverso(-2:2)
[1] -0.5 -1.0 NA 1.0 0.5
```

Por último, la tercera estructura es

switch(expresión,[valor-1=]acción-1,....,[valor-n=]acción-n)

Esta función es **escalar** y además **expresión** debe devolver un vector de longitud 1. La función se comporta de modo distinto según que la expresión evalúe a numérico o a carácter. Cuando la expresión devuelve un valor numérico, se transforma el resultado en un

entero, **i**, y si está entre 1 y n, se evalúa la acción **i**, tenga o no un valor asociado, y se devuelve el resultado, en caso contrario se devuelve **NULL**. Cuando la expresión devuelve una cadena de caracteres, si este coincide exactamente con uno de los valores, se evalúa la acción correspondiente y se devuelve el resultado. Si no coincide con ninguno y una acción (normalmente la última se reserva para ello) no tiene un valor asociado, se ejecuta esta y se devuelve su resultado; pero si todas tienen valor asociado, se devuelve **NULL**. Si hay varias acciones sin valor asociado se devuelve la primera de ellas.

Aunque en el ejemplo no se hace para no distraer la atención, hay que comprobar que el argumento es del tipo que se desea. Si en este ejemplo utilizamos **Decide(1)** obtendríamos el mismo resultado que con **Decide("m")**. Una solución intermedia consiste en sustituir **x** por **as.character(x)** como argumento en la función **switch**.

```
> n<-2
> switch(n,"Uno","Dos","Tres")
[1] "Dos"
> n<-3
> switch(n,"Uno","Dos","Tres")
[1] "Tres"
> n<-4
> switch(n,"Uno","Dos","Tres")
NULL
> Decide<-function(x)
switch(x,m=cat("Has dicho eme minúscula\n"),
M=cat("Has dicho eme mayúscula\n"),cat("Dime m o M\n"))
> Decide
function(x)
switch(x,
m = cat("Has dicho eme minúscula\n"),
M = cat("Has dicho eme mayúscula\n"),
cat("Dime m o M\n"))
> Decide("m")
Has dicho eme minúscula
> Decide("M")
Has dicho eme mayúscula
> Decide("P")
Dime m o M
> n<-"Dos"
> switch(n,"Uno"=1,"Dos"=2,"Tres"=3,"Distinto")
[1] 2
> n<-"Cuatro"
> switch(n,"Uno"=1,"Dos"=2,"Tres"=3,"Distinto")
[1] "Distinto"
```

3.4.3. Estructuras de repetición definida e indefinida

Hay tres estructuras, que son:

for (variable in valores) acción

while (condición) acción

repeat acción

La estructura **for** asigna a **variable** cada uno de los **valores** y realiza la **acción** para cada uno.

La estructura **while** evalúa la **condición** y mientras esta es cierta se evalúa la **acción**.

La estructura **repeat** evalúa la } indefinidamente.

En los tres casos, el valor del ciclo completo
es el de la última evaluación de la acción.

Las expresiones pueden contener algún **acción** condicional como **if** asociado con las funciones **next** o **break**.

La estructura **next** indica que debe terminarse la iteración actual y pasar a la siguiente.

La estructura **break** indica que debe terminarse el ciclo actual.

Los tres ejemplos siguientes muestran una estructura **for** que recorre la sucesión **-5:5**. En el primer caso la recorre completa, en el segundo, salta el número 0, y en la tercera, se detiene al llegar a cero. En los tres casos, si a continuación hubiese otra orden seguiría ejecutándola a continuación.

```
> for (i in -5:5) {cat(i,"\t",i^2,"\n")}
-5 25
-4 16
-3 9
-2 4
-1 1
0 0
1 1
2 4
3 9
4 16
5 25
> for (i in -5:5) {if (i==0) next;cat(i,"\t",i^2,"\n")}
-5 25
-4 16
-3 9
-2 4
-1 1
1 1
2 4
3 9
4 16
5 25
> for (i in -5:5) {if (i==0) break;cat(i,"\t",i^2,"\n")}
-5 25
-4 16
-3 9
```

```
-2 4
-1 1
```

Por último, las dos estructuras siguientes recorren la sucesión **5:1** mediante **while** y **repeat**. Téngase en cuenta que están escritas sólo a modo de ilustración, si se desea recorrer esas secuencias es preferible utilizar un ciclo **for**.

En esta se añade la acción **i=i-1)**| que modifica la condición **(i>0)**.

```
> i<-5;while (i>0) {print(i);i=i-1}
> [1] 5
[1] 4
[1] 3
[1] 2
[1] 1
```

En esta se añade la comprobación **(i==0)** de haber alcanzado el final.

```
> i<-5;repeat{print(i);i=i-1;if(i==0) break}
> [1] 5
[1] 4
[1] 3
[1] 2
[1] 1
```

3.4.4. invisible

La función **invisible** indica que un objeto no debe mostrarse. La sintaxis es

```
invisible(x)
```

siendo **x** cualquier objeto que es devuelto a su vez por la función. Esta función se usa muy a menudo para no presentar directamente informaciones devueltas por funciones que, sin embargo, pueden ser utilizadas.

Cualquier acción en **R** devuelve un resultado. Incluso cuando se realiza una asignación se devuelve un resultado, aunque se hace de modo invisible. Para hacerlo visible basta con utilizar paréntesis (¡que son una función!). Sin embargo, una agrupación de órdenes entre llaves (¡que no es una función!) no tiene este efecto. Vea los siguientes ejemplos, especialmente el último, en que se asigna 3 a la variable **y**, lo que devuelve de modo invisible este valor, que se asigna a continuación a **x**.

```
> x<-1
> (x<-1)
[1] 1
> {x<-1}
> {x<-1;y<-2;z<-3}
> ({x<-1;y<-2;z<-3})
[1] 3
> x<-y<-3
> x
```

```
[1] 3
> y
[1] 3
```

3.4.5. Recursividad

Es posible que la definición de una función haga referencia a la propia función, lo que permite construir funciones que desarrollan estructuras definidas fácilmente mediante recursividad y muy complejamente por otros medios; como por ejemplo los fractales. Sin embargo debe recordar que este tipo de estructura consume muchos recursos del sistema, por lo que debe reservarse para los casos en que sea estrictamente necesaria. A continuación, cuando lea la definición de factorial de un número, encontrará una definición utilizando recursividad, aunque repito que no es la mejor solución, ya que en ese caso es preferible una estructura de
repetición.

3.4.6. Depuración de errores

En caso de error en la utilización de una función es de gran utilidad la función **traceback**. Por ejemplo, con la función **Dibuja** que está definida posteriormente, la función **traceback** permite reconstruir el camino seguido por **R** hasta devolver el error.

```
> Dibuja()
Error in plot.window(xlim, ylim, log, asp, ...) :
invalid xlim
> traceback()
[1] "plot.window(xlim, ylim, log, asp, ...)"
[2] "plot.default(0, 0, xlim = xlim, ylim = ylim,"
[3] "type = \"n\", xaxs = xaxs, "
[4] " yaxs = yaxs, xlab = xlab, ylab = ylab, ...)"
[5] "plot(0, 0, xlim = xlim, ylim = ylim,"
[6] "type = \"n\", xaxs = xaxs, "
[7] " yaxs = yaxs, xlab = xlab, ylab = ylab, ...)"
[8] "image(x, y, outer(x, y, f))"
[9] "Dibuja()"
```

De gran utilidad son las funciones **debug** y **undebug**. La primera de ellas, aplicada a una función determinada, hace que esta se realice paso a paso, pudiendo incluso observar los valores de las variables que en ella intervienen en cada momento. Hasta aplicarle la segunda función, **undebug**, permanecerá en este estado.

```
> debug(raiz)
> raiz(3)
debugging in: raiz(3)
debug: {
if (is.numeric(x) && min(x) > 0)
sqrt(x)
else stop("O x no es numerico o no es positivo")
}
Browse[1]>
debug: if (is.numeric(x) && min(x) > 0) sqrt(x)
else stop("O x no es numerico o no es positivo")
```

```
Browse[1]>
exiting from: raiz(3)
[1] 1.732051
> undebug(raiz)
```

Advierta que la función **traceback** devuelve el proceso que generó el último error, por lo que no tiene sentido aplicarla cuando este no existe.

```
> raiz(3)
[1] 1.732051
> traceback()
[1] "plot.window(xlim, ylim, log, asp, ...)"
[2] "plot.default(0, 0, xlim = xlim, ylim = ylim,"
[3] "type = \"n\", xaxs = xaxs, "
[4] " yaxs = yaxs, xlab = xlab, ylab = ylab, ...)"
[5] "plot(0, 0, xlim = xlim, ylim = ylim,"
[6] "type = \"n\", xaxs = xaxs, "
[7] " yaxs = yaxs, xlab = xlab, ylab = ylab, ...)"
[8] "image(x, y, outer(x, y, f))"
[9] "Dibuja()"
```

3.5. Ejemplos de funciones

A continuación se presentan algunas funciones para dejar claro el procedimiento de escritura de las mismas. Estas funciones sufrirán ciertas depuraciones, por lo que aparecen varias versiones de algunas.

3.5.1. Factorial de un número

A continuación, se define el factorial (a modo de ejercicio ya que de hecho, existe la función **factorial(x)**, definida como **gamma(x+1)**) de un número (natural, no se ha incluido comprobación de que el argumento es correcto, **is.integer**, por lo que la función puede terminar con error si el argumento es de tipo carácter, por ejemplo) utilizando tanto un ciclo determinado como uno indeterminado

```
> Factorial.d<-function(n)
+ {
+
##################################################
+ # Valor inicial de factorial el neutro del producto #
+ # Estructura for en funcion de n positivo #
+ # Se devuelve factorial #
+
##################################################
+ factorial <- 1
+ if(n>1)
+ for (i in 1:n)
+ factorial <- factorial * i
+ return(factorial)
+ }
> Factorial.d(3)
```

```
[1] 6
> Factorial.d(100)
[1] 9.332622e+157
> Factorial.d(0)
[1] 1

> Factorial.i<-function(n)
+ {
+
##################################################
+ # Valor inicial de factorial el neutro del producto #
+ # Estructura while en funcion de n positivo #
+ # Se añade el factor n y se disminuye el valor de n #
+ # Se devuelve factorial #
+
##################################################
+ factorial <- 1
+ while(n > 0)
+ {
+ factorial <- factorial * n
+ n <- n - 1
+ }
+ return(factorial)
+ }
> Factorial.i(3)
[1] 6
> Factorial.i(100)
[1] 9.332622e+157
> Factorial.i(0)
[1] 1
```

A continuación se presentan dos procedimientos alternativos que son, en este caso, peores que los anteriores: El primero consume recursos debido a la recursividad y el segundo los consume porque genera todos los factores antes de multiplicarlos, consumiendo memoria.

```
> Factorial.r<-function(n)
+{
+
##################################################
+ # La función se llama a sí misma modificando el argumento #
+
##################################################
+ if(n > 1)
+ factorial <- n * Factorial.r(n - 1)
+ else factorial <- 1
+ return(factorial)
}
> Factorial.r(3)
[1] 6
> F.r(300)
[1] Inf
```

```
> Factorial.r(1000)
Error: evaluation nested too deeply:
infinite recursion / options(expression=)?
```

y observe que no puede calcularse el factorial de 1000 porque no hay bastantes recursos en el modo predeterminado.

```
> Factorial.m<-function(n)
+{
+ if(n > 1)
+ return(prod(n:1))
+ else return(1)
+}
> Factorial.m(3)
[1] 6
> Factorial.m(100)
[1] 9.332622e+157
```

En este último caso no se llegan a agotar los recursos, pero se han consumido más que en los dos primeros, ya que es necesario generar el vector **n:1** y almacenarlo antes de calcular el producto de sus elementos, aunque es más eficiente desde el punto de vista de tiempo de cálculo. Advierta que, además, en unos casos se utilizan los productos desde 1 a n y en otros al contrario, lo que numéricamente no es equivalente.

3.5.2. Progresión aritmética

La progresión aritmética es una sucesión definida recurrentemente a partir del primer elemento, **a[1]**, y la diferencia, **d**, mediante la relación **a[n+1]=a[n]+d**. Se dispone además de un resolvente explícito, **a[n+1]=a[1]+d*n**.

Podemos construir tres funciones para implementarla: **Ar.e**, **Ar.r** y **Ar.r.v**. La primera corresponde a la forma explícita, la segunda a la forma recursiva y la tercera a la forma recursiva vectorial.

```
> Ar.e = function(n=1,a1=1,d=1) a1+d*(n-1)
> Ar.r = function(n=1,a1=1,d=1)
+ {
+ if (n>1)
+ {return(Ar.r(n-1,a1,d)+d)}
+ else
+ {return(a1)}
+ }
> Ar.r.v = function(n=1,a1=1,d=1)
+ {
+ A=1:n
+ A[1]=a1
+ for (i in 2:n) A[i]=A[i-1]+d
+ return(A[n])
+ }
```

La que he denominado **forma recursiva vectorial**, consiste en la construcción de un objeto

que represente la estructura que queremos desarrollar, en este caso un vector (de ahí el adjetivo de vectorial) representa perfectamente a los n primeros términos de una sucesión, y construir todos los elementos de la estructura comenzando desde los primeros hasta llegar al final. Puede comprobar que, por ejemplo, obtener un millón de elementos en este caso es factible y se realiza bastante rápidamente, en tanto que obtener simplemente 1000 términos en el caso recursivo no es factible.

Es posible incluir un medidor de tiempo y tener en cuenta que, en el caso en que n vale 1, la anterior definición falla (pruebe más adelante con la función debug a descubrir dónde y por qué falla). Definimos la función del siguiente modo:

```
> Ar.r.v = function(n=1,a1=1,d=1)
+ {
+ tiempoinicial=Sys.time()
+ if (n==1)
+ {
+ tiempofinal=Sys.time()
+ return(list(a1,tiempofinal-tiempoinicial))
+ }
+ A=1:n
+ A[1]=a1
+ for (i in 2:n) A[i]=A[i-1]+d
+ tiempofinal=Sys.time()
+ return(list(A[n],tiempofinal-tiempoinicial))
}
> Ar.r.v(1000000)
[[1]]
[1] 1e+06

[[2]]
Time difference of 10.313 secs
```

3.5.3. Progresión geométrica

La progresión geométrica es una sucesión definida recurrentemente a partir del primer elemento, **a[1]**, y la razón, **r**, mediante la relación **a[n+1]**=**a[n]*r**. Se dispone además de un resolvente explícito, **a[n+1]**=**a[n]*r^n**.

Podemos construir tres funciones para implementarla: **Ge.e**, **Ge.r** y **Ge.r.v**. La primera corresponde a la forma explícita, la segunda a la forma recursiva y la tercera a la forma recursiva vectorial.

```
> Ge.e = function(n=1,a1=1,r=1) a1+r^(n-1)
> Ge.r = function(n=1,a1=1,r=1)
+ {
+ if (n>1)
+ {Ge.r(n-1,a1,r)*r}
+ else
+ {a1}
+ }
> Ge.r.v = function(n=1,a1=1,r=1)
```

```
+ {
+ if (n==1) return(a1)
+
+ A=1:n
+ A[1]=a1
+ for (i in 2:n) A[i]=A[i-1]*r
+ return(A[n])
+ }
```

3.5.4. Sucesión de Fibonacci

Estas funciones devuelven el elemento enésimo de una sucesión de Fibonacci, caracterizada porque los dos primeros elementos valen 1 y el resto de los elementos se calculan como la suma de los dos que le preceden. Como no tenemos una forma explícita, utilizamos sólo dos procedimientos: El primero, recursivo puro, y el segundo recursivo con almacenamiento de los resultados en un vector.

```
> Fibonacci.r = function(n=1) if (n>2)
+ {Fibonacci.r(n-1)+Fibonacci.r(n-2)} else {1}
> Fibonacci.r.v=function(n=1)
+ {
+ if (n<3) return(1)
+ x=vector(l=n)
+ x[1]=1
+ x[2]=1
+ for (i in 3:n) x[i]=x[i-1]+x[i-2]
+ return(x[n])
+ }
```

La forma recurrente pura consume muchísimos más recursos y tiempo que la recurrente vectorial, como ya se ha indicado anteriormente (Pruébela con n=30 y con n=300, añadiéndole un medidor de tiempo).

En la definición de **Fibonacci.r.v** se ha aprovechado que los dos primeros valores son iguales para finalizar los cálculos al comprobar que n es menor que 3. Ello se hace mediante la función **return** que devuelve los argumentos y finaliza la ejecución. La siguiente función utiliza una expresión más compleja pero más apropiada para la construcción de funciones similares en que los primeros elementos no son idénticos.

```
> Fibonacci.vv = function(n=1,a1=1,a2=1)
+ {
+ if (n==1) return(a1)
+ if (n==2) return(a2)
+ x=vector(l=n)
+ x[1]=a1
+ x[2]=a2
+ for (i in 3:n) x[i]=x[i-1]+x[i-2]
+ return(x[n])
+ }
```

También puede escribir la función utilizando **switch** en vez de dos **if**.

```
> Fibonacci.vv.c = function(n=1,a1=1,a2=1)
+{
+ N=as.character(n)
+ # Se convierte en un carácter para que
+ # la tercera opción haga el papel de else
+ switch(N,
+ "1"=return(a1),
+ "2"=return(a2),
+ {
+ x=vector(l=n)
+ x[1]=a1
+ x[2]=a2
+ for (i in 3:n) x[i]=x[i-1]+x[i-2]
+ return(x[n])
+ }
+ )
+}
```

Por último se presenta una modificación de la función en que se devuelve una lista con tres elementos: El elemento enésimo de la sucesión, todos los elementos de la sucesión hasta el enésimo y la suma de los mismos.

```
> Fibonacci.vv.l = function(n=1,a1=1,a2=1)
+{
+ N=as.character(n)
+ # Se convierte en un carácter para que
+ # la tercera opción haga el papel de else
+ switch(N,
+ "1"=return(list(xn=a1,v=a1,suma=a1)),
+ "2"=return(list(xn=a2,c(a1,a2),suma=a1+a2)),
+ # en cualquier otro caso
+ {
+ x=vector(l=n)
+ x[1]=a1
+ x[2]=a2
+ for (i in 3:n) x[i]=x[i-1]+x[i-2]
+ return(list(xn=x[n],v=x,suma=sum(x)))
+ }
+ )
+}
> Fibonacci.vv.l(5)
$xn
[1] 5

$v
[1] 1 1 2 3 5

$suma
[1] 12
> Fibonacci.vv.l(15)$xn
[1] 610
```

```
> Fibonacci.vv.l(25)$suma
[1] 196417
```

3.5.5. Extracción de cartas de una baraja

En este ejemplo se realiza una simulación del siguiente experimento: Realizar un muestreo con reemplazamiento de una baraja de 52 cartas, hasta obtener los cuatro ases e indicar el número de extracciones necesarias. Se utiliza la función **sample** para realizar cada extracción. Utilice **help(sample)** para estudiar la función.

```
> CuatroAses
function(Mostrar = F,Maximo=1000)
{
#####################################################
# Pone los cuatro ases en 0 #
# Saca una carta de la baraja (52) #
# Suma uno al contador #
# Si no es un as (1) vuelve al principio del ciclo #
# El as correspondiente se pone en 1 #
# Si los cuatro ases son 1 termina el ciclo #
#####################################################
Extracciones = 0
Resultado = 1:Maximo
Ases = c(0,0,0,0)
repeat
{
if (Maximo <= Extracciones )
{
if(Mostrar)
{
cat("No he podido obtener cuatro ases en ",
Extracciones,
"extracciones. \n")
}
return(list(E = NA, R = Resultado, Conseguido=F))
}
Extracciones = Extracciones + 1
SacoUna = sample(52, 1)
Resultado[Extracciones] = SacoUna
if(SacoUna %% 13 != 1) next
# %% es el módulo
Ases[(SacoUna -1) %/% 13 + 1] = 1
# %/% es la división entera
if(sum(Ases)==4)
break
}
length(Resultado)=Extracciones
if(Mostrar)
{
cat("He necesitado", Extracciones,
"extracciones para obtener cuatro ases.\n")
```

```
}
return(list(E = Extracciones, R = Resultado, Conseguido=T))
}
> CuatroAses(,10)
$E
[1] 10

$R
[1] 47 31 48 42 30 39 31 21 22 34

$Conseguido
[1] FALSE

> CuatroAses(T,10)
No he podido obtener cuatro ases en 11 extracciones.
$E
[1] NA

$R
[1] 24 35 41 24 10 14 39 13 48 42

$Conseguido
[1] FALSE

> CuatroAses()
$E
[1] 205

$R
[1] 44 33 35 36 42 ....

$Conseguido
[1] TRUE

> CuatroAses(T)
He necesitado 61 extracciones para obtener cuatro ases.
$E
[1] 61

$R
[1] 3 19 2 40 15 47 ....

$Conseguido
[1] TRUE
```

Entre las alternativas a la escritura de esta función, debe destacarse que es posible obtener la muestra de tamaño **Maximo** de una sola vez con la orden **sample(52,Maximo,T)** e ir explorando posteriormente los valores.

Advierta también que cada vez que ejecuta la función obtiene el resultado de una simulación basada en números pseudoaleatorios, por lo que no es esperable obtener los mismos

resultados en dos ejecuciones sucesivas. Se puede utilizar el resultado de la función como entrada de otra función, por ejemplo para repetir varias veces la simulación y estudiar la distribución de la misma.

```
> DistriAses = function(n = 5,Maximo=1000)
{
Saco = vector(length=n)
for(i in 1:n)
Saco[i] = CuatroAses(F,Maximo)$E
Saco
}
> Distribucion
[1] 174 89 76 259 140 98 99 ...
[487] 123 72 83 159 120 22 67 109 84 81 80 76 118 68
> summary(Distribucion)
Min. 1st Qu. Median Mean 3rd Qu. Max.
15.00 63.75 96.00 106.40 134.00 361.00
> hist(Distribucion)
> DistriAses(5,50)
[1] 33 NA NA 29 NA
```

Si desea escribir una función que realice una simulación de la extracción de cartas, pero sin reemplazamiento, deberá generar en primer lugar la muestra completa de la extracción y comprobar posteriormente cuándo ocurre el resultado que espera.

```
> CuatroAses.Sin =function(Mostrar = F)
{
##################################################
# Pone el contador de ases en 0 #
# Obtiene una permutación de las cartas de la baraja #
# Si no es un as (1) pasa a la siguiente carta #
# Suma 1 al contador de ases #
# Si hay cuatro ases termina el ciclo #
##################################################
Ases = 0
Resultado = sample(52)
for (i in 1:52)
{
if(Resultado[i] %% 13 != 1) {next}
# %% es el módulo
Ases = Ases+1
if(Ases==4) {break}
}
if(Mostrar)
{
cat("He necesitado", i,
"extracciones para obtener cuatro ases\n")
}
return(list(E = i,R = Resultado[1:i]))
}
> CuatroAses.Sin()
```

```
$E
[1] 16

$R
[1] 29 51 20 21 1 26 40 42 6 12 5 43 15 14 9 27

> CuatroAses.Sin(T)
He necesitado 22 extracciones para obtener cuatro ases
$E
[1] 22

$R
[1] 49 33 40 34 14 22 3 36 8 44 18 48
[13] 26 52 24 35 32 50 46 1 39 27
```

Igual que en el caso anterior, podrá realizar una simulación

```
> DistriAses.Sin = function(n = 5)
{
Saco = vector(length=n)
for(i in 1:n)
Saco[i] = CuatroAses.Sin()$E
Saco
}

> Distribucion = DistriAses.Sin(2000)
> Distribucion
[1] 42 30 49 49 41 ...
[1996] 52 47 52 30 38
> summary(Distribucion)
Min. 1st Qu. Median Mean 3rd Qu. Max.
8.00 38.00 45.00 42.43 49.00 52.00
> hist(Distribucion)
```

La siguiente versión devuelve además el nombre de la carta que se obtiene en cada extracción. Para crear una sola cadena de caracteres a partir de varias, utiliza la función **paste**.

```
> CuatroAses.Nombres = function(Mostrar = F,Maximo=1000)
{
#####################################################
# Pone los cuatro ases en 0 #
# Saca una carta de la baraja (52) #
# Suma uno al contador #
# Si no es un as (1) vuelve al principio del ciclo #
# El as correspondiente se pone en 1 #
# Si los cuatro ases son 1 termina el ciclo #
#####################################################
Palos = c("Oros","Copas","Espadas","Bastos")
Cartas = c("As","Dos","Tres","Cuatro","Cinco",
"Seis","Siete","Ocho","Nueve","Diez",
```

```
"Sota","Caballo","Rey")
Extracciones = 0
Resultado = 1:Maximo
Ases = c(0,0,0,0)
repeat
{
Extracciones = Extracciones + 1
if (Maximo <= Extracciones )
{
if(Mostrar)
{
cat("No he podido obtener cuatro ases en ",
Extracciones,
"extracciones. \n")
}
return(list(E = NA, R = Resultado,
N=Nombres, Conseguido=F))
}
SacoUna = sample(52, 1)
Resultado[Extracciones] = SacoUna
Palo = ((SacoUna-1) %/% 13 )+ 1
Carta = ((SacoUna-1) %% 13)+1
Nombres[Extracciones] =
paste(Cartas[Carta]," de ", Palos[Palo])
if(SacoUna %% 13 != 1) {next}
# %% es el módulo
Ases[(SacoUna -1) %/% 13 + 1] = 1
# %/% es la división entera
if(sum(Ases)==4) {break}
}
length(Resultado)=Extracciones
if(Mostrar)
{
cat("He necesitado", Extracciones,
"extracciones para obtener cuatro ases.\n")
}
return(list(E = NA, R = Resultado, N=Nombres, Conseguido=T))
}
> CuatroAses.Nombres(T,10)
No he podido obtener cuatro ases en 10 extracciones.
$E
[1] NA

$R
[1] 49 21 30 10 46 43 17 11 25

$N
[1] "Diez de Bastos" "Ocho de Copas" ...
[9] "Caballo de Copas"

$Conseguido
```

```
[1] FALSE

> CuatroAses.Nombres()
$E
[1] NA

$R
[1] 33 37 52 25 14 3 15 16 1 29 40 38 49 26 19 27

$N
[1] "Siete de Espadas" "Sota de Espadas" ...
[16] "As de Espadas"

$Conseguido
[1] TRUE
```

3.5.6. Función suma de potencias

En primer lugar aparece una función que calcula la suma de los cuadrados y cubos de las componentes de un vector, y los devuelve en un vector.

```
> sumpot=function(x=NA)
{
s2=sum(x^2)
s3=sum(x^3)
return(c(s2,s3))
}
> sumpot(1:10)
[1] 385 3025
```

También puede presentar el resultado como una lista con nombre para cada uno de los componentes:

```
> sumpot=function(x)
+ {
+ s2 = sum(x^2)
+ s3 = sum(x^3)
+ return(list(cuadrados=s2,cubos=s3))
+ }
> sumpot(1:10)
$cuadrados:
[1] 385

$cubos:
[1] 3025

> sumpot(1:10)$cubos
[1] 3025
> sumpot(1:10)[[2]]
[1] 3025
```

Por supuesto, esta definición ha sustituido a la anterior, ya que el nombre es el mismo. También podría escribir la función de modo más compacto, como sigue, aunque tal vez no quede tan claro si los cálculos fuesen más complejos:

```
sumpot=function(x)
{
list(cuadrados=sum(x^2),cubos=sum(x^3))
}
```

A continuación se modifica la definición, añadiendo un segundo argumento que, predeterminadamente, da el mismo resultado, pero que permite obtener las sumas de otras potencias.

```
> sumpot2=function(x,potencias=2:3)
{
L=length(potencias)
resultado=vector("list",length=L)
for (i in 1:L) resultado[[i]]=sum(x^potencias[i])
return(resultado)
}
> sumpot2(1:10,c(3,5,7))
[[1]]:
[1] 3025

[[2]]:
[1] 220825

[[3]]:
[1] 18080425
```

Aunque parezca indiferente devolver un vector o una lista, debe tenerse en cuenta que todas las componentes de un vector son del mismo tipo, por lo que, por ejemplo, si se mezclan números reales y complejos en el resultado se presentarían todos como complejos. Por ello suele utilizarse la lista para devolver resultados.

3.5.7. Función en una cuadrícula

La siguiente función genera una cuadrícula sobre un conjunto de ordenadas (x) y abscisas (y) y, para cada punto de esa cuadrícula, calcula la distancia hasta el origen de coordenadas.

```
> distancia.origen = function(x,y)
{
puntos=matrix(-1,length(x),length(y))
for (i in 1:length(x))
{
for (j in 1:length(y))
{
puntos[i,j]=sqrt(x[i]^2+y[j]^2)
}
}
return(puntos)
```

```
}
> distancia.origen(1:3,-2:2)
[,1] [,2] [,3] [,4] [,5]
[1,] 2.236068 1.414214 1 1.414214 2.236068
[2,] 2.828427 2.236068 2 2.236068 2.828427
[3,] 3.605551 3.162278 3 3.162278 3.605551
```

Es posible escribir una definición alternativa utilizando la forma general del producto exterior, que es menos didáctica que la anterior, pero que, curiosamente, es más rápida en ejecución. Para esta función se utiliza un nombre distinto, ya que la palabra ORIGEN está escrita en mayúsculas. Ello implica que se crea un **nuevo** objeto, que no sustituye al anterior, sino que coexisten.

```
> distancia.ORIGEN=function(x,y)
{
outer(x,y,function(x,y) sqrt(x^2+y^2))
}
> distancia.ORIGEN(1:3,-2:2)
[,1] [,2] [,3] [,4] [,5]
[1,] 2.236068 1.414214 1 1.414214 2.236068
[2,] 2.828427 2.236068 2 2.236068 2.828427
[3,] 3.605551 3.162278 3 3.162278 3.605551
> ls(patt="dis*")
[1] "dist.ORIGEN" "dist.origen"
```

También podría escribir la función así:

```
> dista = function(x,y) sqrt(x^2+y^2)
> distancia.ORI = function(x,y)
{
outer(x,y,dista(x,y))
}
> distancia.ORI(1:3,-2:2)
[,1] [,2] [,3] [,4] [,5]
[1,] 2.236068 1.414214 1 1.414214 2.236068
[2,] 2.828427 2.236068 2 2.236068 2.828427
[3,] 3.605551 3.162278 3 3.162278 3.605551
> persp( distancia.ORIGEN(-40:40,-40:40))
```

3.5.8. Procesos no lineales

Las tres funciones siguientes corresponden a procesos no lineales descritos en Dong que, de forma sencilla, definen estructuras complejas. En el apartado de gráficos serán utilizadas para su representación.

```
> dong1=function(numero = 100)
{
x = vector(mode = "numeric", length = numero)
y = vector(mode = "numeric", length = numero)
x[1] = 1
y[1] = 1
```

```
for(i in 2:numero)
{
if(sample(2,1) == 2)
{m = 1}
else
{m = -1}
x[i] = 0.5 * x[i - 1] + 0.5 * y[i - 1] + m
y[i] = -0.5 * x[i - 1] + 0.5 * y[i - 1] + m
}
return(list(x = x[2:numero], y = y[2:numero]))
}

> dong2=function(numero = 100)
{
x = vector(mode = "numeric", length = numero)
y = vector(mode = "numeric", length = numero)
x[1] = 1
y[1] = 1
for(i in 2:numero)
{
a = sample(3,1)
if(a == 1)
{
m = 0
n = 0
}
else
{
if(a == 2)
{
m = 0.5
n = 0
}
else
{
m = 0.25
n = 0.5
}
}
x[i] = 0.5 * x[i - 1] + m
y[i] = 0.5 * y[i - 1] + n
}
return(list(x = x[2:numero], y = y[2:numero]))
}
> dong3=function(numero = 100)
{
x = vector(mode = "numeric", length = numero)
y = vector(mode = "numeric", length = numero)
x[1] = 1
y[1] = 1
for(i in 2:numero)
```

```
{
a = sample(100,1)
if(a == 1)
{
x[i] = 0
y[i] = 0.25 * y[i - 1]
}
else
{
if(a <= 86)
{
x[i] = 0.85 * x[i - 1] + 0.04 * y[i - 1]
y[i] = -0.04 * x[i - 1] + 0.85 * y[i - 1] + 1.6
}
else
{
if(a <= 93)
{
x[i] = 0.2 * x[i - 1] - 0.26 * y[i - 1]
y[i] = 0.26 * x[i - 1] + 0.22 * y[i - 1]
}
else
{
x[i] = -0.15 * x[i - 1] + 0.28 * y[i - 1]
y[i] = 0.26 * x[i - 1] + 0.24 * y[i - 1] + 1
}
}
}
}
return(list(x = x[2:numero], y = y[2:numero]))
}
```

Es posible modificar estas definiciones en muchos aspectos, para conseguir más claridad o robustez o velocidad. Por ejemplo, la siguiente modificación de la primera función, incluye el tiempo que tarda en realizarse y genera todos los elementos aleatorios en una sola ejecución de la función **sample**. La inclusión del tiempo permite analizar con diversas ejecuciones cual de las diferentes formas es más apropiada.

```
dong1.tiempo = function(numero = 100)
{
Inicio=Sys.time()
x = vector(mode = "numeric", length = numero)
y = vector(mode = "numeric", length = numero)
m = vector(mode = "numeric", length = numero)
x[1] = 1
y[1] = 1
m=sample(c(-1,1),numero,T)
for(i in 2:numero)
{
x[i] = 0.5 * x[i - 1] + 0.5 * y[i - 1] + m[i]
y[i] = -0.5 * x[i - 1] + 0.5 * y[i - 1] + m[i]
```

```
}
Final=Sys.time()
tiempo=Final-Inicio
return(list(x = x[2:numero], y = y[2:numero],tiempo=tiempo))
}
```

Las estructuras **if** anidadas son complejas, por lo que pueden modificarse los programas utilizando vectores auxiliares como hacemos a continuación:

```
dong2.v = function(numero = 100)
{
x = vector(mode = "numeric", length = numero)
y = vector(mode = "numeric", length = numero)
x[1] = 1
y[1] = 1
XX = c(0, 0.5, 0.25)
YY = c(0, 0, 0.5)
for(i in 2:numero)
{
a = sample(3,1)
x[i] = 0.5 * x[i - 1] + XX[a]
y[i] = 0.5 * y[i - 1] + YY[a]
}
return(list(x = x[2:numero], y = y[2:numero]))
}
```

También es mejorable el uso de la función sample para generar números no equiprobables, como hacemos a continuación:

```
dong3.v=function(numero = 100)
{
x = vector(mode = "numeric", length = numero)
y = vector(mode = "numeric", length = numero)
x[1] = 1
y[1] = 1
for(i in 2:numero)
{
a = sample(4,1, p=c(1,85,7,7))
switch(a,
{# Caso a=1
x[i] = 0
y[i] = 0.25 * y[i - 1]
},
{# Caso a=2
x[i] = 0.85 * x[i - 1] + 0.04 * y[i - 1]
y[i] = -0.04 * x[i - 1] + 0.85 * y[i - 1] + 1.6
},
{# Caso a=3
x[i] = 0.2 * x[i - 1] - 0.26 * y[i - 1]
y[i] = 0.26 * x[i - 1] + 0.22 * y[i - 1]
},
```

```
{# Caso a=4
x[i] = -0.15 * x[i - 1] + 0.28 * y[i - 1]
y[i] = 0.26 * x[i - 1] + 0.24 * y[i - 1] + 1
}
)
}
return(list(x = x[2:numero], y = y[2:numero]))
}
```

3.6. Función missing

La función **missing** permite comprobar si se ha dado valor a un parámetro. Debe utilizarse al comenzar el cuerpo de la función. Así la siguiente función detecta si se han incluido los argumentos y reacciona a su carencia. (No se incluyen las salidas gráficas.)

```
> dibuja = function(x,y)
{
if(missing(x))
{
stop("El argumento 'x' es obligatorio")
}
if(missing(y))
{
y = x
x = 1:length(y)
}
plot(x,y)
}
> dibuja()
Error in dibuja() : El argumento 'x' es obligatorio
> a=-10:10
> dibuja(a)
> dibuja(a,a^2)
> dibuja(,a^2)
Error in dibuja(, x^2) : El argumento 'x' es obligatorio
> dibuja(a^2)
```

3.7. Función menu

La función **menu** permite elegir, de modo interactivo, entre un conjunto de opciones. Su sintaxis es

```
menu(choices, graphics=F, title="")
```

- **choices** es un vector de caracteres correspondientes a cada una de las opciones del menú.
- **graphics** No se utiliza.
- **title** es el título del menú.

devolviendo el número correspondiente al elemento seleccionado o 0 si no se elige ninguna. Si se elige cualquier otra opción no la admite. Esta función es especialmente útil cuando se escribe un programa para que sea utilizado por personas sin experiencia en **R**.

3.8. Lista de búsqueda y entornos

Hasta ahora ha definido objetos, ha podido ver los existentes con \code{ls} y ha eliminado los superfluos con **rm**. Pero esos objetos se encuentran en un directorio concreto. También ha utilizado funciones que no aparecían al escribir **ls**. Puede encontrar todos los lugares donde se encuentran objetos con las funciones **search** y **searchpaths**.

```
> search()
[1] ".GlobalEnv" "package:methods" "package:stats"
[4] "package:graphics" "package:grDevices" "package:utils"
[7] "package:datasets" "Autoloads" "package:base"
> searchpaths()
[1] ".GlobalEnv"
[2] "C:/ARCHIV~1/R/rw2000/library/methods"
[3] "C:/ARCHIV~1/R/rw2000/library/stats"
[4] "C:/ARCHIV~1/R/rw2000/library/graphics"
[5] "C:/ARCHIV~1/R/rw2000/library/grDevices"
[6] "C:/ARCHIV~1/R/rw2000/library/utils"
[7] "C:/ARCHIV~1/R/rw2000/library/datasets"
[8] "Autoloads"
[9] "C:/ARCHIV~1/R/rw2000/library/base"
>
```

Estas funciones devuelven un vextor con la lista de libros (*packages*) y objetos de **R** (usualmente hojas de datos) que están en uso actualmente (*attached*). Comienzan por *.GlobalEnv* y terminan con el libro *base* que está siempre cargado en **R**. La diferencia entre ambas es que la primera devuelve los nombres internos y la segunda los caminos desde donde se han cargado los libros y hojas de datos.

Esta lista, que se denomina lista de búsqueda, es similar a la orden **path** de DOS. Cada vez que **R** debe buscar un objeto, comienza a buscarlo en el primer elemento de la lista. Si no lo encuentra en esa posición, pasa al siguiente y así sucesivamente. Ello quiere decir que es posible que existan dos objetos con el mismo nombre --- que se encuentren en posiciones distintas, claro está --- pero el que esté en una posición posterior no aparecerá pues será eclipsado por el que se encuentre en la posición anterior.

Puede ver los objetos que se encuentran en una posición de la lista con la función **ls**. Como indica la ayuda, esta función, equivalente a **objects**, tiene cinco argumentos: **name**, **pos**, **envir**, **all.names** y **pattern**.

- **name** Se denomina así por compatibilidad con versiones anteriores, pero corresponde al **entorno** en que se buscan los objetos. Su valor predeterminado es el entorno actual. Puede definirse el entorno por varios procedimientos:

Mediante un número entero, n. Si n es positivo, indica una posición, **search()[n]**. Si n=-1 indica **search()[2, 3, ...]**
El nombre de un elementos de **search()**
Un entorno explícito, incluyendo el uso de la función **sys.frame** para acceder a entornos de ejecución de funciones

- **pos** Se mantiene por compatibilidad con versiones anteriores, pero es una forma alternativa de especificar el entorno.
- **envir** Se mantiene por compatibilidad con versiones anteriores, pero es una forma alternativa de especificar el entorno.
- **all.names** Es una variable lógica. Si es **T** entonces la función devuelve todos los objetos, en caso contrario no devuelve los que comienzan con un punto. Este último es el valor predeterminado.
- **pattern** Es una frase que corresponde a una **expresión regular**, de tal modo que si se especifica sólo se devolverán los nombres de objetos que se correspondan con ella, en caso contrario se devolverán todos. Puede utilizar **? regexp** para ver todas las posibilidades de las expresiones regulares en **R**.

Cuando se usa sin argumentos en el nivel superior (la ventana de órdenes) muestra los datos y funciones definidos por el usuario. Si se hace dentro de una función, devuelve los nombres de las variables locales de la función.

Como resultado se obtiene un vector con los nombres de los objetos. Así, los objetos que se encuentran en la posición **9** y cuyo nombre contiene la frase **log1** se obtienen con

```
> ls(9,pa="log1")
[1] "log10" "log1p"
```

Además de estos entornos que podemos considerar "horizontales" existen otros "verticales" que comienzan sobre el primero y son los correspondientes a cada función que se ejecuta. Solicite ayuda sobre la función **sys.parent** para obtener mayor información. La siguiente función aclara qué objetos existen dentro del entorno de una función.

```
> Entorno = function(x,y)
{
MM=1
ls()
}
> Entorno()
[1] "MM" "x" "y"
```

Si desea saber si existe o no un objeto, fundamentalmente en programación, puede usar la función **exists(nombre,where)**, siendo **where** el lugar desde donde buscarlo, esto es, en esa posición o cualquiera superior.

```
> exists("xedit",5)
[1] TRUE
> exists("xedit",6)
```

```
[1] TRUE
> exists("xedit",7)
[1] FALSE
```

Si desea recuperar un objeto definido en una posición de la lista de búsqueda y oculto por otro con el mismo nombre en una posición anterior, puede utilizar la función **get (nombre,pos)**.

```
> get("matplot",4) -> MMM
```

Una vez recuperado el objeto (puede que con el mismo nombre) puede modificarlo si lo desea.

La lista de búsqueda puede ampliarse o reducirse (no debería reducir la lista original, sólo debe reducir una lista ampliada sobre la original) añadiendo o eliminando objetos (*database*) y libros (*package*). Para añadir objetos se usa la función **attach** que tiene tres argumentos, **what**, **pos** y **name**. El primero especifica el nombre de un objeto, que se añadirá a la lista de búsqueda. Puede ser una hoja de datos, una lista o un archivo creado con la orden **save**. El segundo especifica la posición que debe ocupar en la lista, que por defecto es la segunda. Nunca debe elegirse la primera posición porque esta es especial y, por tanto, tiene un tratamiento especial. El tercer argumento corresponde al nombre con que se desea que aparezca en la lista. Si el nombre no se especifica, se construye con el nombre del objeto.

Para añadir un libro se utiliza la función **library** que veremos posteriormente.

Cuando no desee seguir utilizando un objeto, elimínelo de la lista con **detach** indicando bien el nombre o la posición del objeto que se desea eliminar de la lista. Si no se indica este valor se toma la segunda posición, que coincide con la de **attach**. Esta función no permite eliminar la primera posición ni la última, correspondiente al libro **base**.

3.9. Entradas y salidas mediante archivos

Si se dispone de una serie de órdenes escritas en un archivo, **archivo.R**, en el directorio de trabajo, es posible ejecutarlas mediante la función **source**, del siguiente modo:

```
source("archivo.R")
```

Así, si realiza una sesión de trabajo, puede guardar todas las órdenes dadas (también puede copiarlas de pantalla, o revisar el archivo de historial de órdenes, **.Rhistory**) en un archivo, depurarlas mediante un procesador de textos y posteriormente ejecutarlas de una sola vez. Si va a realizar este trabajo a menudo, es más eficiente crear una o varias funciones.

La función **sink** permite redirigir la salida a un archivo. Admite cuatro parámetros:

- **file** Si especifica un archivo, los resultados se dirigen a ese archivo, hasta que se especifique la orden de nuevo, dirigiendo los resultados a otro archivo o devolviéndolos al valor anterior, que en el nivel 0 es la pantalla, si no se especifica ninguno.
- **append** El parámetro **append** toma los valores lógicos **TRUE** y **FALSE**, según deban añadirse los resultados al archivo o borrarlo previamente. Por defecto el

valor es **FALSE**, lo que indica que los resultados NO se añaden a los existentes en el archivo.

- **type** Puede tomar los valores **output** y **message** indicando si se desea redirigir la salida o los mensajes. Esta última opción debe realizarse con precaución.
- **split** Es una variable lógica. Si es **TRUE** entonces se envían los resultados tanto a la pantalla como al archivo, en caso contrario solo se envían al archivo. El valor predeterminado es **FALSE**.

El ejemplo siguiente dirige los resultados de las órdenes a **archivo.txt** y después de nuevo a la pantalla (Suponiendo que esta era la salida estándar en el momento anterior).

```
sink("archivo.txt")
Órdenes……..
sink()
```