Manual de Introducción a la programación en R

Marco Antonio Carrillo Izquierdo

3 de septiembre de 2020

Índice general

I Programación Básica **1**

1. R **3**
1.1. ¿ Qué es R ? 3
1.2. Objetos, Paquetes y Librerías 4

2. Operadores y funciones básicas **5**
2.1. Operaciones Aritméticas 5
2.2. Funciones predefinidas y Caracteres especiales 6
2.3. Variables 8
2.4. Funciones 9
2.5. Vectores 11
2.5.1. Operaciones simples con vectores no definidos 15
2.6. Valores Perdidos, Omitidos y No definidos 16
2.7. Expresiones y Asignaciones 17
2.8. Expresiones Lógicas 17
2.9. Matrices 19
2.10. El entorno 23

3. Estructuras de Control **25**
3.1. If 25
3.1.1. Ejemplo: Función else no anidada 26
3.1.2. Ejemplo: Comando ifelse 28
3.2. Bucles con el comando for 28
3.2.1. For 29
3.2.2. Ejemplo: Comando cat 30
3.3. While 30
3.3.1. Ejemplo: Ciclo While 30
3.4. Errores 31
3.4.1. Ejemplo: Mensaje de error 31

3.4.2. Ejemplo: Una forma de evitar los errores 32

4. El manejo del Texto y Gráficas sencillas 35
4.1. Variables de Texto 35
4.1.1. Ejemplo: Función Paste 35
4.1.2. Cambio del tipo de variable 36
4.2. Tablas 36
4.2.1. Ejemplo: Función format 37
4.3. Función Scan 37
4.3.1. Ejemplo: Función scan 38
4.3.2. Ejemplo: Función scan con entrada manual 39
4.3.3. Ejemplo: Función Readline 40
4.4. Salida de Texto 41
4.4.1. Ejemplo: Función Write 42
4.5. Gráficas Sencillas 44

5. Creación de funciones 49
5.1. La estructura de una función 51
5.1.1. Ejemplo: Dentro de la función 51
5.2. Características de los argumentos 52
5.2.1. Ejemplo: Argumento fuera de la función 52
5.2.2. Ejemplo: Ausencia de argumentos 53
5.3. Programación basada en vectores 54
5.3.1. Apply 54
5.3.2. Lapply 55
5.3.3. Sapply 56
5.3.4. Tapply 56
5.3.5. Mapply 57
5.4. Recursividad 58
5.4.1. Ejemplo: Sucesión de Fibonacci 58
5.4.2. Ejemplo: Operador factorial 59
5.5. Los errores 60
5.5.1. Función Stop 60
5.5.2. Función Browser 61

6. Factores, Listas y Dataframe 63
6.1. Factores 63
6.2. Dataframe 66
6.2.1. Ejemplo: Subset 69
6.3. Listas 70

Parte I

Programación Básica

Capítulo 1

R

El extenso mundo de la programación sin importar el lenguaje de nuestra preferencia (Java, C, Python, R) es algo que va desde lo intuitivo y divertido, pasando por lo fascinante hasta llegar a lo frustrarte y lo desesperante, siendo estas ultimas el resultado de alguna carencia en las raíces de nuestros conocimientos, para evitar sufrir estos dolores de cabeza, el lector de este libro habrá de instruirse poco a poco en la programación del lenguaje R partiendo de los fundamentos y los conceptos hasta alcanzar los términos, estructuras e ideas más complicadas todo con el fin de proporcionarse a si mismo una alta gama de herramientas que faciliten su desempeño e ingenio haciendo de esta manera más ameno y tranquilo el camino que tendrá que recorrer en el mundo de la programación.
Este manual tiene como objetivo facilitar el aprendizaje del lenguaje de programación R, siendo el referente más importante por excelencia cuando hablamos del análisis estadístico de datos.

1.1. ¿ Qué es R ?

R es un entorno y lenguaje de programación que nació como una herramienta enfocada principalmente al análisis estadístico, que mediante paquetes, librerías o creando nuestras propias funciones el lector puede extender su funcionalidad a otros campos como la biología, economía, ciencias de la salud, etc.
La principal ventaja y el distintivo de R es que es gratuito y de código abierto, esto último quiere decir que cualquier usuario puede descargar y crear su código de manera gratuita, sin restricciones de uso a diferencia de otras herramientas estadísticas comerciales como Stata, Minitab, SPSS, fomentando

de esta manera la ciencia y los conocimientos a más personas.
Es importante aclarar algo en lo que podemos confundirnos conforme avanzamos, la diferencia entre R y Rstudio. El primero es el nombre que se le asigna al lenguaje de programación que vamos a utilizar en este libro y el segundo es el entorno de desarrollo integrado (IDE), Rstudio no es el único entorno que tienen soporte para el lenguaje de programación en R pero si es el más utilizado.

1.2. Objetos, Paquetes y Librerías

R es es un lenguaje orientado a objetos, esto quiere decir que todas las variables, resultados, funciones, datos, etc. se guardan en la memoria activa de la computadora con un nombre específico. Entenderemos objeto como *una estructura que combina datos y funciones que operan sobre ellos.*
Entenderemos como paquete a *un conjunto de datos y funciones que mejoran algunas de sus funciones base en R potenciando así su funcionamiento o que inclusive añade algunas nuevas funciones al entorno*, si el lector desea desenvolverse en un área distinta al análisis estadístico, podría instalar un paquete enfocado a esa área.
Para instalar un paquete en la memoria del IDE, en nuestro caso RStudio haciendo uso de lenguaje de programación R solo habrá que ejecutar el siguiente comando:

```
install.packages("dplyr")
```

Ahora, por librería entenderemos al conjunto de paquetes instalados previamente en el entorno que se encuentran disponibles para el usuario. En la parte derecha del interfaz en RStudio existe una sección dedicada a la librería que se ha construido por el usuario y a la que trae consigo el programa desde que se instala.
Para solicitar un paquete de la librería que ya ha sido cargado previamente en la memoria del IDE se tendrá que ejecutar el siguiente comando:

```
library("urca")
```

Capítulo 2

Operadores y funciones básicas

2.1. Operaciones Aritméticas

El lenguaje R hace uso de los símbolos comunes para cada una de las operaciones básicas: adición +, sustracción -, multiplicación *, división /, potenciación ^ modulo %% y de este otro que llamaremos *"división de números enteros"* %/% para la división de números enteros.

```
> #Suma
> 17+18
[1] 35
> #Resta
> 25-20
[1] 5
> #Multiplicacion
> 24*5*2
[1] 240
> #Division
> 30/7
[1] 4.285714
> #Potencia
> 4^3
[1] 64
> #Modulo: Regresa el resto de la division
> 25%%4
[1] 1
> 20%%3
```

```
[1] 2
> #Division de numeros enteros: Regresa el cociente de la
    division
> 25%/%4
[1] 6
> 20%/%3
[1] 6
```

Los corchetes que anteceden los resultados [n] indican la posición numero n del vector salida, en los ejemplos anteriores al obtener un solo elemento como resultado tenemos que n=1.
De verse en la necesidad de hacer uso de muchas operaciones los paréntesis especifican el orden de las operaciones.

```
> 3*2^(2+1*(2*2))
[1] 192
> ###El proceso de la opración seria el siguiente:
>
> # Resuelve la primer multiplicación:
> # (2*2)=4
> # Reselve la segunda multiplicación
> # 1*(4)=4
> # Resuelve la suma dentro del primer parentesis
> # (2+4)=(6)
> # Eleva el numero a la cantidad obtenida
> # 2^(6)=64
> # Multiplica las cantidadades y termina el proceso
> # 3*64=192
```

2.2. Funciones predefinidas y Caracteres especiales

En R podemos hacer uso de dos tipos de funciones, aquellas que están integradas en el entorno desde que lo instalamos en nuestra computadora, las cuales llamaremos predefinidas y las que nosotros podemos crear desde cero (Hablaremos de estas ultimas en el capitulo 5).
Las funciones predefinidas en R es una lista extensa de la que tal vez nunca podamos ver su extenso y largo final, pero si abarcar aquellas que podemos considerar “más importantes” empezando por lo fundamental, por mencionar algunas de ellas tenemos las siguientes:

- Funciones Trigonométricas: sin(x), cos(x), tan(x), asin(x), acos(x), atan(x), (Todas calculadas en radianes)
- Constantes especiales(en ingles): pi, exp(x).
- Operaciones Aritméticas: abs(x) y sqrt(x) así como algunas constantes especiales tales como pi.
- Funciones estadísticas: mean(x), var(x), median(x), max(x), min(x), choose(n,k).

Conforme el lector avance se encontrará, en su mayoría, con una explicación y un ejemplo de cada una de estas funciones y de algunas más.
Por defecto, R solo muestra 7 dígitos cuando el resultado termina en decimales, el usuario puede cambiar el número de dígitos a mostrar usando la función **options(digits = x)** de la siguiente manera:

```
> #Por default R hace uso de 6 decimales
> pi
[1] 3.141593
> exp(1)
[1] 2.718282
> (1+5^(1/2))/2
[1] 1.618034
> sin(2.34)
[1] 0.7184648
> #Cambiamos al numero de decimales deseados
> options(digits = 17)
> pi
[1] 3.1415926535897931
> exp(1)
[1] 2.7182818284590451
> options(digits = 3)
> (1+5^(1/2))/2
[1] 1.62
> sin(2.34)
[1] 0.718
```

Si lo que se desea es más bien redondear ya sea hacia abajo o hacia arriba un numero, usaremos las funciones **floor(x)** y **ceiling(x)** respectivamente.

```
> ##Redondeo
> #Hacia abajo
> floor(pi)
```

```
[1] 3
> floor(exp(1))
[1] 2
> floor(15/2)
[1] 7
> #Hacia arriba
> ceiling(pi)
[1] 4
> ceiling(exp(1))
[1] 3
> ceiling(15/2)
[1] 8
```

2.3. Variables

Una variable es como una carpeta con un nombre establecido, el lector puede poner algo dentro de esta carpeta, utilizarlo, remplazarlo, pero el nombre se mantendrá igual, el cual puede incluir letras, números y . o _ (se debe mantener en cuenta que R distingue entre mayúsculas y minúsculas).
Para asignar valor a una variable x usamos el comando de asignación $<-$ y para mostrar su valor basta con escribir el nombre de la misma después de haberla asignado, asignarla dentro de paréntesis o con los comandos **print(x)** y **show(x)**.

```
> ##Asignación de valores y las diferentes formas de
   mostrar su valor
> a<-5
> show(a)
[1] 5
> b<-12
> print(b)
[1] 12
> (c<-10)
[1] 10
> d<-25
> d
[1] 25
```

A menos que se le asigne un nuevo valor a la variable esta se mantendrá igual sin importar el número de operaciones que se ejecuten con ella.

```
> ##Cambio de valores
> a*b
[1] 60
> (a);(b)
[1] 5
[1] 12
> x<-b/12
> b
[1] 12
> b<-100
> print(c)
[1] 10
> c<-a*b
> c
[1] 500
```

Cuando le asignamos un valor a una variable, el valor de la parte derecha se evalúa primero que el de la izquierda, este recurso es de utilidad en aquellas operaciones en las que es necesario hacer uso de la misma variable para asignar un nuevo valor.

```
> ##Asignación de valores usando la misma variable
> x<-10
> y<-5
> z<-x+y
> z
[1] 15
> z<-2*z
> z
[1] 30
```

2.4. Funciones

Las funciones predefinidas en R consisten de 2 conceptos claves, el nombre de la función y sus parámetros(Un estudio más amplio se realiza en el capitulo 5). Para llamar una función predefinida se escribe el nombre de la función seguido de sus argumentos encerrados entre paréntesis y separados por comas.
Ejemplo:

```
> ##Construccion de vectores apartir de funciones
```

```
> #Los parametros se separan con comas.
> #Indican el inicio, el fin y cada cuanto avanza la
    secuencia
> seq(from=2, to=15, by=2)
[1]  2  4  6  8 10 12 14
> #Si omitimos el parametro by, el valor por default de
    avance es 1
> seq(from=2, to=15)
 [1]  2  3  4  5  6  7  8  9 10 11 12 13 14 15
> #Podemos omitir el nombre de los parametros y continuar
    siempre y cuando esten escritos en el orden correcto
> seq(2, 15, 2)
[1]  2  4  6  8 10 12 14
> #Tambien podemos desordenar el nombre de los parametros
> #pero se tiene que indicar el nombre del parametro al
    que le asignaremos un valor
> seq(by=3, from=5, to=20)
[1]  5  8 11 14 17 20
> #Podemos crear un vector con una secuencia regresiva
    con un valor negativo en el parametro by
> seq(12,2,-2)
[1] 12 10  8  6  4  2
```

Para conocer los valores predeterminados y los usos alternativos de alguna función incorporada, se puede acceder a la ayuda que el IDE nos provee escribiendo **help**(*nombre de la función*) o ?*nombre de la función*.
Ejemplo: **?seq**

Una función siempre necesita abrir paréntesis después de su nombre, incluso si no se requieren argumentos. Si simplemente se escribe el nombre de la función, R lo leerá como una variable más o mandara un mensaje de error.

```
> #Asigna el valor de la secuencia de manera correcta
> x <-seq(1,5)
> #No aplica la funcion de manera correcta
> y <-meanx
Error: object 'meanx' not found
> #Si aplica la funcion de manera corrrecra
> y <-mean(x)
> print(y)
[1] 3
```

En esta sección y en las siguientes, seguiremos manejando funciones y aludiendo a solo sus parámetros más importantes, ya que los subsecuentes tienden a ser de uso más especifico. Recordar que si se desea conocer más sobre una función y sus parámetros puede usar la función **help("nombre de la función")**.

2.5. Vectores

Análogamente al concepto de variable, un vector es una lista indexada de variables. Su estructura esta compuesta por el por el nombre del vector y las variables contenidas en el se indican con el numero que ocupa dentro del vector.

> datos[2] *Segunda variable del vector "datos"*

En esencia una variable simple es solo un vector con longitud 1. Para crear vectores de longitud mayor que 1, utilizamos funciones que producen resultados con valores vectoriales. Las tres funciones básicas para construir vectores son: **c(...)** (combina), **seq(from, to, by)** (secuencia), y **rep(x, times)** (repite).

Ejemplos:

```
> ##Funciones Vectoriales
> #Funcion seq:
> #Crea una secuencía de valores segun los parametros
> seq(1,12,2)
[1]  1  3  5  7  9 11
> #Funcion rep:
> #Crea un vector con valores repetidos segun se indique
   en los parametros
> rep(x=17,times=7)
[1] 17 17 17 17 17 17 17
> rep(3,2)
[1] 3 3
> #Se puede indicar una secuencia de valores y mandarla a
    repetir
> rep(1:3,4)
 [1] 1 2 3 1 2 3 1 2 3 1 2 3
> #Funcion c:
> #Crea un vector con los valores indicados, generalmente
```

```
    no se hace uso de sus parametros
> c(1,2,3,4)
[1] 1 2 3 4
> c(11,6,3,125,53)
[1]  11   6   3 125  53
> #Se puede invertir el orden de la secuencia
> c(8:3)
[1] 8 7 6 5 4 3
> #Es posible mandar diferentes secuencias separadas por
    comas
> c(1:5,30:34)
 [1]  1  2  3  4  5 30 31 32 33 34
> #Podemos introducir variables de tipo caracter
> c("verde", "azul","amarillo")
[1] "verde"    "azul"     "amarillo"
> #Se debe ser cuidados si pretendemos combinar valores
    numericos con caracteres
> #Pues el vector manejara todos los valores como
    caracteres
> c("a","b",as.numeric(1:5))
[1] "a" "b" "1" "2" "3" "4" "5"
> #Ejemplo:
> #Asignamos el nombre ‘‘xy’’ al vector
> xy<-c("a","b",as.numeric(1:5))
> #Asginamos el nombre a la variable "no_numero" con el
    valor del vector xy en su posición 4
> no_numero <-xy[4]
> #Comprobamos su valor
> no_numero
[1] "2"
> #Aqui podemos ver que R lo maneja como un valor no
    numerico
> 2+no_numero
Error in 2 + no_numero : non-numeric argument to binary
    operator
> #Si se deseara cambiar el valor de los numericos se
    tendría que hacer lo siguiente:
> #Asginamos el nombre a la variable "numero" con el
    valor del vector xy en su posición 4
> numero<-as.numeric(xy[4])
> #Comprobamos su valor
> numero
[1] 2
> #Despues de usar la función as.numeric() ahora R lo
    maneja como un valor numerico
```

```
> 2+numero
[1] 4
```

Como se ilustro en el ejemplo anterior para referirnos al i-esimo elemento del vector x, usamos la expresión x[i], pero inclusive en esta situación se debe ser cuidadoso si el elemento i es negativo porque R omitirá su lectura si se le desea llamar.
Ejemplos:

```
> #Creamos un vector con valores deseados
> (x<-c(10:15))
[1] 10 11 12 13 14 15
> #Creamos un segundo vector, con valores para i
> (i<-c(1:3))
[1] 1 2 3
> #Y un tercero con valores negativos
> (j<-c(-3:-1))
[1] -3 -2 -1
> #Llamamos al vector x en las posiciones de j e i
> (x[i])
[1] 10 11 12
> #El vector en x en las posiciones del vector i con
   valores positivos
> #nos devuelve los valores en las respectivas posiciones
> (x[j])
[1] 13 14 15
> #El vector en x en las posiciones del vector j con
   valores negativos
> #omite los valores en las respectivas posiciones
```

Una vez entendido el concepto de vector en el entorno, podemos auxiliarnos de ellos para realizar diferentes operaciones más rápidamente
Ejemplos:

```
> #Operaciones algebracias basicas en R
> (x<-seq(7,24,3))
[1]  7 10 13 16 19 22
> (y<-c(2,4,6,7,9,11))
[1]  2  4  6  7  9 11
> #Observe como las operaciones se realizan elemento a
   elemento
```

```
> (x+y)
[1]  9 14 19 23 28 33
> (x-y)
[1]  5  6  7  9 10 11
> (x*y)
[1]  14  40  78 112 171 242
> options(digits = 3)
> (x/y)
[1] 3.50 2.50 2.17 2.29 2.11 2.00
> (x^y)
[1] 4.90e+01 1.00e+04 4.83e+06 2.68e+08 3.23e+11 5.84e+14
```

Sin embargo no siempre se tendrá la fortuna de que el número de elementos de 2 vectores coincidan, de ser ese el caso, R automáticamente repite los valores del vector de menor longitud hasta que termine el proceso.
Ejemplos:

```
> #Vectores con diferente longitud
> (a<-c(5:10))
[1]  5  6  7  8  9 10
> (b<-rep(3,3))
[1] 3 3 3
>
> #comprobamos la longitud
> length(a)
[1] 6
> length(b)
[1] 3
> #Realizamos operaciones
> (a+b)
[1]  8  9 10 11 12 13
> (a*b)
[1] 15 18 21 24 27 30
```

Algunas funciones muy útiles que hacen uso de vectores como argumentos son **sum(...), prod(...), max(...), min(...), sqrt(...), sort(x), mean(x), and var(x).** Es importante tener en cuenta que las funciones aplicadas a vectores pueden trabajar elemento a elemento o trabajar con todos los elementos para arrojar solo un resultado.

```
> #Usaremos los vectores definidos en el ejemplo anteior
> sum(a)
[1] 45
```

```
> sum(b)
[1] 9
> sqrt(b)
[1] 1.73 1.73 1.73
> max(a)
[1] 10
> min(a*b)
[1] 15
> (c<-c(5:8,-2:0))
[1]  5  6  7  8 -2 -1  0
> sort(c)
[1] -2 -1  0  5  6  7  8
> mean(a)
[1] 7.5
> var(b)
[1] 0
```

2.5.1. Operaciones simples con vectores no definidos

Más adelante encontraremos operaciones más complejas en donde podremos hacer uso de bucles para realizar operaciones que requieren muchas repeticiones,o que los parámetros para ejecutarlos son más estrictos, afortunadamente para R existen formas de realizar operaciones con vectores que no han sido previamente definidos de tal modo que no sea complicado
Ejemplos:

```
> x<-cos(x)
> x
[1] 0.7345281 0.7074118 0.6848963 0.6682404 0.6579948
[6] 0.6543026
> y<-sin(x)
> y
[1] 0.6702370 0.6498688 0.6325927 0.6196058 0.6115316
[6] 0.6086060
> sum((x^2)*(y^2))
[1] 1.133349
```

2.6. Valores Perdidos, Omitidos y No definidos

De manera inmediata uno podría caer en el error de creer que un valor perdido, omitido o no definido son iguales, sin embargo no es así, y que mejor manera de entender la diferencia que observar la forma de provocar que se ejecuten dentro de la consola.
Entender la diferencia es de suma importancia, ya que dependiendo del análisis estadístico involucrado, los datos omitidos pueden ser ignorados o procesados y saber que es a lo que nos enfrentamos es la mejor manera de solucionarlo. R representa los datos omitidos con el valor **NA**, los datos perdidos con el valor **NULL** y los no definidos con un mensaje de error.
Ejemplo:

```
> x<-rep(2,5)
> y<-x*a
Error: object 'a' not found
> #Evaluamos si las posiciones del vector son valores
    omitidos
> is.na(x)
[1] FALSE FALSE FALSE FALSE FALSE
> is.null(x)
[1] FALSE
> #Evaluamos si al menos uno de ellos es un valor omitido
> any(is.na(x))
[1] FALSE
> any(is.null(x))
[1] FALSE
> #Si un vector contiene un valor na, sus calculos no
    pueden
> #ser realizados de manera estandar
> y<-c(1:5,NA,3:5)
> y
[1]  1  2  3  4  5 NA  3  4  5
> mean(y)
[1] NA
> sum(y)
[1] NA
> #Modificamos uno de sus parametros para realizar la
    accion
> mean(y,na.rm=TRUE)
[1] 3.375
> sum(y,na.rm = T)
[1] 27
```

Los valores NA y NULL no son equivalentes, NA es un marcador que indica que algo existe pero esta ausente (Verla sexta posición el vector y del ejemplo anterior) y NULL indica que algo que se busca nunca existió.

2.7. Expresiones y Asignaciones

Hasta ahora se han visto el uso de comandos simples en R sin profundizar, por lo que en esta sección se buscara cubrir vocabulario útil en R.
En R, el término *expresión* se usa para denotar una frase de código que se puede ejecutar. Los siguientes son ejemplos de expresiones.

```
> seq(10,20,3)
[1] 10 13 16 19
> 4
[1] 4
> mean(c(1,2,3))
[1] 2
> 1>2
[1] FALSE
```

Y se le llama asignación a la combinación de la evaluación de una expresión y guardarla usando el operador <-. Los siguientes son ejemplos de asignaciones:

```
> w<-seq(10,20,3)
> x<-4
> y<-mean(c(1,2,3))
> z<-1>2
```

Las variables anteriores guardaran los valores obtenidos en la primera parte de este apartado.

2.8. Expresiones Lógicas

Una expresión lógica es aquella en la que se utilizan los operadores de comparación:

- $<$ Menor que
- $>$ Mayor que
- $<=$ Mayor o igual que

- >= Menor o igual que
- == Igual a
- **!=** Distinto de
- **&** Operador lógico: Y
- | Operador lógico: O
- **!** Operador lógico: No
- **xor** Disyunción Exclusiva

Los operadores lógicos son usados para operaciones de álgebra Booleana, es decir, para describir relaciones lógicas, expresadas como verdadero (TRUE o valor=1) o falso (FALSE o valor =0). El orden de las operaciones se puede controlar usando paréntesis ().
Ejemplos:

```
> #Definimos 2 variables con valores booleanos
> (x<-c(1,1,1,0,0,1,0,0,1,0))
 [1] 1 1 1 0 0 1 0 0 1 0
> (y<-rep(c(1,0),5))
 [1] 1 0 1 0 1 0 1 0 1 0
>
> # | devuelve TRUE si uno de los datos es TRUE o 1
> x[2] | y[2]
[1] TRUE
> # & devuelve TRUE solo si ambos datos son TRUE o 1
> x[3] & y[3]
[1] TRUE
> #xor devuelve TRUE solo si uno y solo uno de los datos
    es TRUE o 1
> xor(x[6],y[6])
[1] TRUE
> # | devuelve FALSE si todos los datos son FALSE o 0
> x[10] | y[8]
[1] FALSE
> # & devuelve FALSE si alguno de ambos datos es FALSE o
    0
> x[6] & y[4]
[1] FALSE
> #xor devuelve FALSE si ambos datos son TRUE o FALSE
> xor(x[1],y[1])
```

```
[1] FALSE
> #Es posible evaluar todos los datos con una solo
    ejecucion
> x | y
 [1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE
 [9]  TRUE FALSE
> x & y
 [1]  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
 [9]  TRUE FALSE
> xor(x,y)
 [1] FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE
 [9] FALSE FALSE
> (x==y)
 [1]  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE  TRUE
 [9]  TRUE  TRUE
> (x!=y)
 [1] FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE
 [9] FALSE FALSE
```

El ejemplo anterior también muestra que las expresiones lógicas se pueden aplicar a los vectores para producir vectores de valores TRUE/FALSE. Si se desea saber la posición de los valores que cumplen una condición se usa la función **which(x)**.

```
> x<-c(1,2,3,5,12,4,20)
> which(x%%2 ==0)
[1] 2 5 6 7
# Comprobamos cuales son los valores que cumplen la
    condición
> x[2];x[5];x[6];x[7]
[1] 2
[1] 12
[1] 4
[1] 20
```

2.9. Matrices

Una matriz se crea a partir de vectores haciendo uso de la función **matrix** la cual tiene la siguiente estructura:

matrix(data, nrow = 1, ncol = 1, byrow = FALSE)

En donde:

- **Data:** Vector con longitud máxima de **nrwo** $*$ **ncol**
- **nrow:** Numero de filas.
- **ncol:** Numero de columnas.
- **byrow** = orientación del llenado de la matriz.

Creyendo firmemente que una de las tantas maravillosas formas de llegar al corazón es por medio de la vista y por aras de la facilidad, en la siguiente sección nos daremos el lujo de combinar el formato código R con un formato más amigable para la lectura, siguiendo la linea "de la vista nace el amor"
Ejemplos:
> B <- matrix(3:6, 3,4,FALSE)

$$\begin{bmatrix} 3 & 6 & 5 & 4 \\ 4 & 3 & 6 & 5 \\ 5 & 4 & 3 & 6 \end{bmatrix}$$

La longitud del vector data es igual a 4, mientras que el tamaño de la matriz es de 12, por esta razón la secuencia designada a data se repite hasta completar la matriz en su totalidad.
> B <-matrix(3:15,2,3,T)

```
Warning message:
In matrix(3:15, 2, 3, T) :
  data length [13] is not a sub-multiple or multiple of
     the number of rows [2]
```

$$\begin{bmatrix} 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

Caso contrario en el que la longitud del vector data supera el tamaño de la matriz, R nos enviara un mensaje de advertencia, avisando sobre el error introducido.En este ejemplo observamos como la orientación al momento de llenar la matriz a cambiado.

> B<- matrix(3:8, 2,3)

$$\begin{bmatrix} 3 & 5 & 7 \\ 4 & 6 & 8 \end{bmatrix}$$

Si omitimos el argumento para **byrow** R por defecto otorga el valor FALSE. Para revisar la dimension de una matriz usamos la función **dim**, siguiendo en el ejemplo anterior:
> dim(B)
[1] 2 3

Para crear una matriz con valores únicamente en la diagonal utilizamos la función **diag(x)** y la función **combine**:
> C <-diag(c(10,1,4))

$$\begin{bmatrix} 10 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 4 \end{bmatrix}$$

Para combinar 2 matrices con el mismo numero de filas haremos uso de la función **rbind** y con el mismo numero de columnos la función **cbind**:

> (A <-matrix(10:20,3,4))

$$\begin{bmatrix} 10 & 13 & 16 \\ 11 & 14 & 17 \\ 12 & 15 & 18 \end{bmatrix}$$

> (B <-matrix(1:5,3,4))

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

> rbind(A,B)

$$\begin{bmatrix} 10 & 13 & 16 & 19 \\ 11 & 14 & 17 & 20 \\ 12 & 15 & 18 & 10 \\ 1 & 4 & 2 & 5 \\ 2 & 5 & 3 & 1 \\ 3 & 1 & 4 & 2 \end{bmatrix}$$

> cbind(A,B)

$$\begin{bmatrix} 10 & 13 & 16 & 19 & 1 & 4 & 2 & 5 \\ 11 & 14 & 17 & 20 & 2 & 5 & 3 & 1 \\ 12 & 15 & 18 & 10 & 3 & 1 & 4 & 2 \end{bmatrix}$$

También es posible referirnos a un valor específico de una matriz y cambiarlo del siguiente modo:

> (A[1,1] <-0)

$$\begin{bmatrix} 0 & 13 & 16 \\ 11 & 14 & 17 \\ 12 & 15 & 18 \end{bmatrix}$$

Algunos de los comandos de operación para matrices incluyen el producto escalar * y el producto de matrices, que en R se denota como % * % así como también funciones especificas que tienen como fin obtener los resultados más intuitivos respecto a matrices para el lector, tales como la función **det(x)** que nos da el determinante, **t(x)** la transpuesta de la matriz y **solve(x)** que nos da la matriz inversa.

> (M <-matrix(1:4,2,2))

$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

> (N <-matrix(5:8,2,2))

$$\begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix}$$

> M*N *Producto elemento a elemento*

$$\begin{bmatrix} 5 & 21 \\ 12 & 32 \end{bmatrix}$$

> M % * % N *Producto matricial*

$$\begin{bmatrix} 23 & 31 \\ 34 & 46 \end{bmatrix}$$

> det(M)
[1] -2

> t(M)

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

> solve(M)

$$\begin{bmatrix} -2 & 1{,}5 \\ 1 & -{,}5 \end{bmatrix}$$

Si buscamos obtener la matriz X para la cual se cumple que M %*% X == N, tenemos que hacer uso de la función **solve** del siguiente modo:

> solve(M,N)

$$\begin{bmatrix} -1 & -2 \\ 2 & 3 \end{bmatrix}$$

Si deseamos saber si nuestro objeto es una matriz o un vector, debemos de usar los comandos **is.matrix(x)** y **is.vector(x)** que arrojan valores TRUE o FALSE según corresponda. También podemos crear vectores o matrices apartir de los elementos de una matriz o un vecotr con las funciones **as.matrix(x)** y **as.vector(x)** del siguiente modo:

> N <- as.vector(M)
> [1] 1 2 3 4

2.10. El entorno

Todas los objetos que se utilizan en un archivo de R se mantienen existentes hasta que se decida borrarlos, para saber cuales son nuestros objetos basta con escribir alguno de los siguientes comandos: **ls()** o **objects()**, para remover alguno de ellos usaremos la función **rm(x)** y para borrarlos todos usaremos **rm(list=ls())**, continuando con nuestros ejemplos anteriores:

```
> ls()
[1] "M" "N"
> objects()
[1] "M" "N"
```

Ahora para guardar todos los objetos a un archivo con nombre “documento” usaremos la función **save.image**, para guardar objetos específicos como M y N usaremos **save** del siguiente modo: save(M,N, file= “documento”). Y para cargarlos en caso de volver a utilizarlos usaremos **load**.

Capítulo 3

Estructuras de Control

En este capitulo se introducirán una serie de estructuras básicas pero muy importantes de la programación en R, como lo son sentencias de condición y de ejecución en bucle.
Un programa es una lista de comandos ejecutados con un orden establecido, el cual, por lo general empieza con una introducción de valores, seguido del proceso de calculo y finaliza con la salida de un resultado. Existen 2 formas que podemos utilizar si lo que deseamos es ejecutar un programa: la primera es escribir el comando **source("archivo.r")** o bien podemos solo copiar y pegar el programa en R y ejecutarlo. El primer método es más directo, ya que la función **source** revisa el código supervisando que funcione antes de ejecutarlo y deteniéndose donde encuentre un error, algo que el segundo método no realiza, lo cual es considerado peligroso pues puede provocar comprometer objetos o archivos existentes y podríamos terminar perdiendo tiempo.

3.1. If

Un situación muy habitual que nos encontramos al momento de ejecutar un programa es la de seleccionar las condiciones que se deben de cumplir para que el funcionamiento de este vaya encaminado a un propósito especifico y que el caso contrario (el no cumplimiento de las condiciones) nos conduce a obtener diferentes resultados. Para enfrentarnos ante estas situaciones, R provee al usuario la función **if** y su derivado **else** lo cuales tiene como fin condicionar el funcionamiento de un programa, su estructura es la siguiente:

if (expresion logica) { *Se abre la condición*
expresion1
...
} else { *Se da una alternativa de incumplimiento*
expresion2
...} *Se terminan de evaluar las condiciones*

El proceso que hay detrás de un programa con función **if** es el siguiente: Si la expresión lógica se cumple (que su resultado sea TRUE) entonces el primer grupo de expresiones se ejecuta y no el segundo, pero si no es así (que su resultado sea FALSE) el primer grupo de condiciones no se ejecutan y el segundo grupo si lo hará. Esto nos facilitara la creación de programas más completo con rutas anidadas y establecidas para un funcionamiento correcto.

Es de suma importancia recalcar que **else** no es una función independiente, su uso es opcional siempre y cuando se haya iniciado la sentencia con un **if**, es decir que si el usuario introduce este comando:

if (expresion logica) { *Se abre la condición*
expresion1
...} *Se cierra la primer condición*
else { *la función else no esta anidada*
expresion2 *La segunda expresión se trata por separado*
...} *Se terminan de evaluar solo la segunda condición*

3.1.1. Ejemplo: Función else no anidada

En el ejemplo evaluaremos de manera sencilla si una variable es par o impar, aprovechando la simplicidad del ejercicio para que el lector vaya acostumbrándose a la estructura y de paso vamos a observar de manera detallada que es lo que ocurre cuando no se anida correctamente el comando **else** a la función **if**:

```
> a<-11
>
> #Funcion else anidada
```

```
> if(a%%2==0){
+   print("El numero es par")
+ } else {
+   print("El numero es impar")
+ }
[1] "El numero es impar"
>
> #Funcion else no anidada
> if(a%%2==0){
+   print("El numero es par")
+ }
> else {
Error: unexpected 'else' in "else"
>   print("El numero es impar")
[1] "El numero es impar"
> }
Error: unexpected '}' in "}"
```

Note como al final imprime el resultado "Es impar" como una consecuencia del comando print, pero no de la condición a evaluar.

En R las expresiones agrupadas en llaves {} se evalúan como una única expresión, de manera similar una condición **if** se considera como una única condición, esto puede resultar un problema al momento de evaluar condiciones especificas dentro de una condición general, pues podría conducirnos a escribir un programa de la siguiente forma:

if (expresión lógica) { *Se abre la condición general*
expresion1
...
} else { *Se da una alternativa de incumplimiento*
if (expresión logica 2){ *Se abre la condición especifica*
expresion2
...
} else {
expresion3
...
}
}

Afortunadamente en R podemos anidar el comando else con una segunda condición if de una forma mas clara e intuitiva:

if (expresión lógica) { *Se abre la condición general*
expresion1
...
} else if (expresión logica 2){ *Se abre la condición especifica*
expresion2
...
} else {
expresion3
...
}

3.1.2. Ejemplo: Comando ifelse

Existe un comando reservado en R que combina los comandos **if** y **else**, este comando es muy sencillo de usar, pero su uso queda restringido únicamente con vectores, este comando se diferencia por tener como argumentos no solo la condición a evaluar, también los acciones que tiene que realizar en caso de que se cumpla o no la condición.

```
> calificaciones<-c(8.2,9.1,6.0,4.8,9.2,5.2,5.8)
> ifelse(calificaciones>=6, 'Aprobado', 'Reprobado')
[1] "Aprobado"  "Aprobado"  "Aprobado"  "Reprobado" "
   Aprobado"
[6] "Reprobado" "Reprobado"
```

3.2. Bucles con el comando for

El comando **for** tiene como finalidad ejecutar una serie de expresiones un numero determinado de veces a un objeto especifico, esta conformado por dos elementos: una variable x a la que para este comando podemos nombrar como "contador" y un vector el cual determina el numero de veces que se ejecutara el bucle, su estructura es la siguiente:

for (x in vector) {
expresion1
...
}

Todas las expresiones que se encuentren dentro de las llaves {} serán evaluadas repetidamente para cada valor que adquiera x del vector seleccionado hasta que ya no haya mas valores que evaluar.
Este comando simplifica en unas cuantas lineas tareas que implican procesos repetitivos.

3.2.1. For

A continuación veremos como ejecutar un ciclo **for** con un ejemplo sencillo para que el usuario lo comprenda lo mejor posible y aprovecharemos para hacer una observación sobre la forma en que R trata los objetos y como esto repercute en su duración.
Método 1:

```
> vector<-seq(1,5)
> d<-rep(0,length(vector))
>
> for(i in vector){
+   d[i]=i*(2/3)
+ }
```

Método 2:

```
> vector<-seq(1,5)
> d<-c()
>
> for(i in vector){
+   d[i]=i*(2/3)
+ }
```

La diferencia entre ambos métodos radica en la velocidad con que se ejecuta el ciclo, en el primer método el tamaño del vector ya esta definido, mientras que en el segundo método el tamaño del vector resultado va cambiando conforme las iteraciones se realizan, esta reevaluación que R esta obligado a hacer provoca que el segundo método sea más lento que el primero, de esto podemos deducir que entre más procesos de evaluación tenga que hacer R los procesos seran más tardados.

Otro comando de mucha utilidad es **cat** que tiene una función muy similar a la de **print** con la diferencia de que este nos da la facilidad de concatenar caracteres como texto con números.

3.2.2. Ejemplo: Comando cat

```
> n<-length(calificaciones)
> for(i in 1:n){
+   cat("La calificacion en la materia", i, "es la
   siguiente:", calificaciones[i],"\n")
+   }
La calificacion en la materia 1 es la siguiente: 8.2
La calificacion en la materia 2 es la siguiente: 9.1
La calificacion en la materia 3 es la siguiente: 6
La calificacion en la materia 4 es la siguiente: 4.8
La calificacion en la materia 5 es la siguiente: 9.2
La calificacion en la materia 6 es la siguiente: 5.2
La calificacion en la materia 7 es la siguiente: 5.8
```

3.3. While

En algunas ocasiones nos encontraremos con problemas en los que no se nos sera posible saber el numero exacto de iteraciones que vamos a realizar hasta que las condiciones dadas se cumplan, para este tipo de situaciones utilizaremos el comando **while**, el cual sigue la siguiente estructura:

while (expresion logica) {
expresion1
...
}

Cuando ejecutamos este comando la expresión lógica es evaluada primero, si se cumple (que su valor sea TRUE) todas las expresiones contenidas dentro de las llaves se ejecutaran hasta llegar al final, una vez terminada la primera iteración se volverá a evaluar a la expresión logica y a menos de que no se cumpla (que su valor sea FALSE) el bucle se repetirá una y otra vez. Es fundamental mencionar que todo ciclo **for** puede ser escrito como un ciclo **while** pero al revés no siempre sera posible.

3.3.1. Ejemplo: Ciclo While

Supongamos que una persona desea conocer el numero de meses máximo que tiene antes de que la deuda que contrajo con un prestamista exceda una cantidad acordada.

```
> meses<-0
> tasa<-1.15
> deuda<-500
> limite<-3000
> while(deuda<limite){
+   deuda=deuda*tasa
+   meses=meses+1
+ }
> cat("Los meses maximo antes que se exceda el limite: ",
    limite, "son", meses)
Los meses maximo antes que se exceda el limite:  3000 son
    13
```

En este mismo contexto veamos que ocurre cuando el numero que denominamos como "deuda" excede la cantidad "limite" desde un inicio y el bucle no se realiza:

```
> meses<-0
> tasa<-1.15
> deuda<-1500
> limite<-1000
> while(deuda<limite){
+   deuda=deuda*tasa
+   meses=meses+1
+ }
> cat("Los meses maximo antes que se exceda el limite: ",
    limite, "son", meses)
Los meses maximo antes que se exceda el limite:  1000 son
    0
```

3.4. Errores

En más de una ocasión nos enfrentaremos a una situación en la que nuestro programa no funcione de manera correcta debido a algún error en el código, en R existen pocas formas para identificar un error, las cuales ejemplificaremos para un mejor entendimiento:

3.4.1. Ejemplo: Mensaje de error

Una vez ejecutado R automáticamente detendrá el programa y nos enviara una advertencia con un breve resumen sobre el error:

```
> n<-4
> forr(x in 1:n){
Error: unexpected 'in' in "forr(x in"
>   x=2*x
Error: object 'x' not found
```

Podemos encontrar 2 errores en el código de distinta naturaleza, el primero de sintaxis en el comando for ya que esta mal escrito, por lo que R no puede interpretar a que parte del código la palabra reservada **in** pertenece y el segundo es un error en la variable x al no haber sido definida con anterioridad R no puede mandarla a llamar.

3.4.2. Ejemplo: Una forma de evitar los errores

Identificar en donde se encuentran los problemas dentro de nuestro código cuando los mensajes de error no son suficientes es una tarea ardua que se consigue dominar a través de la practica,cuando esto sucede sugerimos usar el comando **cat** entre lineas para que nos de el valor de la variable o del contador antes de ser evaluada y así saber de manera precisa donde esta el problema

```
> n<-10
> for(i in 1:n){
+   if(i<3){
+     p<-i
+     cat("El valor i agregado fue:", i, "\n")
+     rm("p")
+   } else {
+     q=i*p
+     cat("El valor q agregado fue:", q, "\n")
+   }
+ }
El valor i agregado fue: 1
El valor i agregado fue: 2
Error in p : object 'p' not found
```

A partir de esta sencilla sugerencia podemos deducir en donde se encuentra el fallo en nuestro programa, en este caso los primeros 2 valores de la variable **i** si fueron agregados, por lo que podemos decir que de las 10 iteraciones que

esperábamos que se cumplieran solo se las primeras 2.

Capítulo 4

El manejo del Texto y Gráficas sencillas

4.1. Variables de Texto

Una forma de agrupación con variables independientemente del tipo de variable es haciendo uso de la función **paste**. En el ejemplo siguiente veremos como agrupar diferentes elementos de tipo string en uno solo

4.1.1. Ejemplo: Función Paste

```
> x1<-"Calculo Integral"
>  x2<-"Algebra Superior"
>  x3<-"Geometria Analitica"
>  x4<-"Historia Contemporanea"
>  x5<-"Programacion"
>  materias<-paste(x1,x2,x3,x4,x5,sep=", ")
> materias
[1] "Calculo Integral, Algebra Superior, Geometria
   Analitica, Historia Contemporanea, Programacion"
```

En esta función, la entrada **sep** sirve para asignar el signo de puntuación con el que se van a separar las variables introducidas, además de introducir todos los elementos dentro de comillas. Si no se especifica el valor para el parámetro **sep**, R usara un espacio en blanco por defecto.

```
> materias2<-paste(x1,x2,x3,x4,x5)
> materias2
```

```
[1] "Calculo Integral Algebra Superior Geometria
   Analitica Historia Contemporanea Programacion"
```

Pero como mencionamos anteriormente, la función opera independientemente del tipo de variable que manejemos

```
> cantidad<-50
>  tipo<-"automoviles"
>  marca<-"BMW"
>  (resultado<-paste(cantidad,tipo,marca))
[1] "50 automoviles BMW"
```

4.1.2. Cambio del tipo de variable

No todas las funciones en R son tan flexibles respecto al tipo de variable como la función **paste**, habrá ocasiones en las que necesitaremos cambiar el tipo de variable que estamos manejando para obtener los resultados esperados,para ellos podemos recurrir a las funciones **as.character(x)** y **as.numeric(x)** para cambiar el tipo de formato de una variable a otro.

```
> (numeros<-c(1,2,3,4,5))
[1] 1 2 3 4 5
> (caracteres<-c("1","2","3","4","5"))
[1] "1" "2" "3" "4" "5"
> (nu2<-as.character(numeros))
[1] "1" "2" "3" "4" "5"
> (le2<-as.numeric(caracteres))
[1] 1 2 3 4 5
```

De manera correcta uno podría pensar que estas funciones no se ocuparían jamas si definiéramos de manera correcta el tipo de variable que deseamos utilizar dejando ver que el ejemplo anterior es todo menos inteligente, sin embargo existen muchas situaciones en las cuales un vector,la columna de una tabla o los valores extraídos de un archivo externo no coincidan con lo que necesitamos, más adelante probaremos este hecho y haremos uso de las funciones.

4.2. Tablas

Una forma muy general de convertir un carácter de tipo numérico a string es con la función **format(x, digits, nsmall, width)** en la cual los parámetros

definen el numero de decimales, el numero de dígitos y la longitud de la variable que se van a utilizar respectivamente,estos solo son algunos de sus parametros, esta función es muy efectiva para imprimir datos como una tabla. A continuación daremos ejemplos en el que se construyen dos tablas haciendo uso de la función anterior

4.2.1. Ejemplo: Función format

```
>  x<-c(1:10)
>  cat(paste(format(1:n,width=5,justify = "right"),
        format(3*x+2,width=7,justify = "right"),"\n"),
                sep=" ")
    1        5
     2        8
     3       11
     4       14
     5       17
     6       20
     7       23
     8       26
     9       29
    10       32
```

4.3. Función Scan

De entre todos los métodos que se pueden utilizar para leer los datos de un archivo, la función **scan** es sin duda la más practica de todas, para su buen entendimiento a continuación explicaremos cada una de las secciones que la conforman.
La función scan tiene una estructura como la siguiente:
> scan(file = “ ”, what = 0, n = -1, sep = “ ”, skip = 0, quiet = FALSE)
En donde *file* es el parámetro donde vamos a colocar el nombre del archivo del cual queremos tomar los datos, si dejamos vació este parámetro, el valor por default “ ” indica que los valores se pueden ir agregando.

EL parámetro *what* es aquel en el que se especifica el formato de los valores que se van a extraer, estos pueden ser numéricos, lógicos, caracteres, complejos, listas, etc. el valor 0 es para indicar que son numéricos y dejar el campo en blanco tiene por default leer los datos como caracteres.

El parámetro *n* indica el numero de elementos a leer, si n= -1 entonces la función scan leerá todos los elementos hasta ya no encontrar más.

El parámetro *sep* permite especificar el símbolo que se esta usando como separador de valores, pueden ser puntos, comas, comillas, espacios en blanco, etc.

Skip es un parámetro opcional en el que se especifica si se desea omitir la lectura de las primeras n lineas. *quiet* habilita la posibilidad de indicar el numero de datos que la función ha leído, su valor por default es FALSE.

Estos son los principales parámetros para la función scan, existen otros los cuales no veremos en este apartado, pero se pueden consultar con la función **help**.

4.3.1. Ejemplo: Función scan

Supongamos que tenemos guardado un archivo de texto en la carpeta donde almacenamos todos nuestros archivos de R con el nombre de "datos1.txt" en el cual tenemos la siguiente lista:

Marco 20
Daniel 21
Karla 20
Alondra 21
Iliana 22
David 22
Roberto 22
Rafael 22
Ana 22
Julieta 22

Ahora vamos a asignar a una variable los valores de este archivo mediante la función *scan*:

```
> #Insertamos la direccion donde esta contenido el
    archivo txt
```

```
>  setwd("C:/Users/narum/Documents/R")
>  nombres<-scan("datos1.txt",what=list(character(10),
   numeric(3)), n=-1,skip=0,quiet = F)
Read 10 records
>  nombres
[[1]]
 [1] "Marco"   "Daniel"  "Karla"   "Alondra" "Iliana"
 [6] "David"   "Roberto" "Rafael"  "Ana"     "Julieta"

[[2]]
 [1] 20 21 20 22 20 21 21 22 20 21
```

4.3.2. Ejemplo: Función scan con entrada manual

En caso de dejar en blanco el parámetro file la función scan permite al usuario rellenar el vector con entradas que el usuario proporcione indicando que se ha dejado de introducir elementos con la tecla enter. En el siguiente ejemplo veremos como esta función puede ser indispensable cuando el usuario necesita tener una interacción más cercana.

```
> n<-5
>  v<-rep(0,n)
>  for(i in 1:n){
+    cat("Introduzca el valor numero", i, "para el vector
   ")
+    v[i]<-scan()
+    if(i==n){
+      print(v)
+    }
+  }
Introduzca el valor numero 1 para el vector
1: 3
2:
Read 1 item
Introduzca el valor numero 2 para el vector
1: 7
2:
Read 1 item
Introduzca el valor numero 3 para el vector
1: 12
2:
Read 1 item
Introduzca el valor numero 4 para el vector
```

```
1: 0
2:
Read 1 item
Introduzca el valor numero 5 para el vector
1: 5
2:
Read 1 item
[1]  3  7 12  0  5
```

El único gran inconveniente es que al finalizar de introducir un elemento tenemos que volver a presionar la tecla enter para que el proceso pase al siguiente paso, lo cual no es nada intuitivo y puede tornarse desesperante, por lo que, el lector debe de confiar en su genialidad para que esta función no presente ningún inconveniente.

```
> cat("Introduzca los valores  deseados para la matriz");
    v<-scan();
Introduzca los valores  deseados para la matriz
1: 1
2: 7
3: 23
4: 82
5: 12
6: 62
7: 4
8: -8
9: -12
10: 0
11:
Read 10 items
> v2<-matrix(v,3,4)
> v2
     [,1] [,2] [,3] [,4]
[1,]    1   82    4    0
[2,]    7   12   -8    1
[3,]   23   62  -12    7
```

Otra alternativa para introducir valores manualmente es el uso de la función **readline** la cual también es muy practica.

4.3.3. Ejemplo: Función Readline

```
>  cat("Introduzca el numero x de filas y el numero y de
   columnas para su matriz");x<-as.numeric(readline("x="
   ));y<-as.numeric(readline("y="))
Introduzca el numero x de filas y el numero y de columnas
    para su matriz
x=3
y=3
> a<-rep(0,(x*y))
> #Creacion de la matriz
>  for (i in 1:length(a)){
+    cat("Introduzca un numero")
+    a[i]<-as.numeric(readline("numero= "))
+  }
Introduzca un numero
numero= 3
Introduzca un numero
numero= 2
Introduzca un numero
numero= 6
Introduzca un numero
numero= 3
Introduzca un numero
numero= 2
Introduzca un numero
numero= 6
Introduzca un numero
numero= 1
Introduzca un numero
numero= 3
Introduzca un numero
numero= 6
> A<-matrix(a,x,y)
> show(A)
     [,1] [,2] [,3]
[1,]    3    3    1
[2,]    2    2    3
[3,]    6    6    6
```

4.4. Salida de Texto

En un ejemplo anterior vimos como podiamos tomar los valores de un archivo txt de manera sencilla con la función **scan**,ahora veremos el proceso inverso, introducir información desde R hacia un archivo txt mediante las funciones

write, write.table.
En este punto, el lector habrá notado que la salida de texto desde R no tiene un formato común ni elegante, lo cual puede ser un problema si necesitamos extraer la información de una serie de variables para ser utilizadas en otro medio, para estos problemas tenemos estas alternativas.

4.4.1. Ejemplo: Función Write

La función write sigue la siguiente estructura: **write**(x, file = "data", ncolumns = if(is.character(x)) 1 else 5, append = FALSE)

Donde x es el vector que introduciremos al archivo txt. En caso de no ser un vector y de tratarse de una matriz o un array automáticamente R lo separara columna por columna.

file de nuevo es el parámetro en el que escribiremos el nombre del archivo al que queremos agregar los datos. El nombre por default es data, en caso de querer asignar uno propio se deberá de escribir la dirección completa finalizando con el nombre deseado. El tipo de archivo que deseamos guardar es muy extenso, podemos guardarlo como un archivo en R con la extensión **.r**, un archivo de Excel con la extensión **.xlsx**, un archivo csv ya sea con la extensión **.csv** y **.csv2**, un archivo Python con la extensión **.ipynb**, un archivo de texto con la extensión **.txt** etc.

ncolumns es el numero de columnas en el cual queremos acomodar nuestros datos. La condición introducida nos dice que si la variable x es de tipo carácter que sea acomodada en una sola columna, y en caso de no serlo y de haberse dejado en blanco el parámetro lo hará en 5.

append indica si lo que deseamos es agregar o sobrescribir en el archivo txt. su valor por default es FALSE

Ahora vamos a crear una tabla de manera similar a la del ejemplo 4.2.1 con las mismas funciones y después la guardaremos en un archivo txt con el nombre de "fun_write"

```
> n<-8
>   x<-c(0,n);z<-c(0,n)
>   for (i in 1:n){
+     x[i]<-167390+i
```

```
+     z[i]<-7+(.1*i)
+   }
>   y<-rep("Promedio",8)
>   A<-paste(format(x,width=3,justify = "right"),
+            format(z,width=3,justify = "right"),
+            format(y,width=3,justify = "right"), sep ="
    ")
>   #Escribimos la direccion a donde lo vamos a mandar
>   setwd("C:/Users/narum/Documents/R")
>   write(A,"fun_write.txt",1)
```

Y si todo ha salido de manera correcta el resultado tendrá esta apariencia en el archivo txt:
167391 7.1 Promedio
167392 7.2 Promedio
167393 7.3 Promedio
167394 7.4 Promedio
167395 7.5 Promedio
167396 7.6 Promedio
167397 7.7 Promedio
167398 7.8 Promedio
Por aras de la facilidad y la ilustración para el lector, decidimos usar para nuestro ejemplo archivos de tipo **.txt** pero se recuerda al lector que esta no es la única extensión con la que puede operar la función write.
La función **write.table** hace uso de los mismos parámetros y el proceso es el mismo que la función write, con la diferencia de que el tipo de variable que se introduce debe ser de preferencia una matriz o un data frame, el cual veremos más adelante con más detalle, en caso de que la variable que introduzcamos no sea una matriz o un data frame, la función write.table forzara la variable convirtiéndola en data frame.

```
> x<-c(1:12)
>   y<-c(1.2,2.1,1.5,3.2,.8,1.8,1.5,1.5,1.1,2.5,2.3,2.8)
>   z<-c(12,11,16,19,9,12,16,11,17,12,12,16)
>   (B<-matrix(c(x,y,z),ncol = 3,nrow = 12))
      [,1] [,2] [,3]
 [1,]    1  1.2   12
 [2,]    2  2.1   11
 [3,]    3  1.5   16
 [4,]    4  3.2   19
 [5,]    5  0.8    9
 [6,]    6  1.8   12
```

```
 [7,]    7  1.5   16
 [8,]    8  1.5   11
 [9,]    9  1.1   17
[10,]   10  2.5   12
[11,]   11  2.3   12
[12,]   12  2.8   16
> write.table(B, "fun_writetable.csv",sep = ",")
```

Una vez que hayamos usado la función anterior, podremos comprobar en nuestro directorio que ahora existe un archivo csv con el nombre "fun_writetable.csv"

4.5. Gráficas Sencillas

R provee al usuario de muchas herramientas para realizar gráficas, incluso existen paquetes especiales que tienen su forma de única de gráficar acorde con lo que se desea realizar, en este apartado estudiaremos las formas más sencillas de realizar gráficas.
La primer función y la más sencilla es **plot** la cual solo requiere de 2 vectores que contengan las posiciones que se van a graficar como en el siguiente ejemplo

```
> x<-c(1:10)
> y<-seq(2,20,2)
> plot(x,y)
```

El gráfico obtenido se ilustra en la figura 4.1
Hasta aquí podemos ver claramente que graficar no requiere de nada más que un conjunto de datos, sin embargo así como en las funciones anteriores, **plot** ofrece una extensa variedad de parámetros que modifican significativamente el resultado.

```
> x<-(1:11)
>   y<-seq(1,6,.5)
> plot(x,y,type="b",xlab = "Eje de abscisas",
+      ylab="Eje de ordenadas",main="Grafica Plot",
+      col="navyblue",panel.first=grid())
> abline(v=6, col = "black",)
> abline(h=5, col = "black",)
> text(6,3.5,"Interseccion (6,3.5)", pos=4)
> text(9,5,"Interseccion (9,5) ", pos=3)
> text(6,5,"Interseccion (6,5) ", pos=1)
```

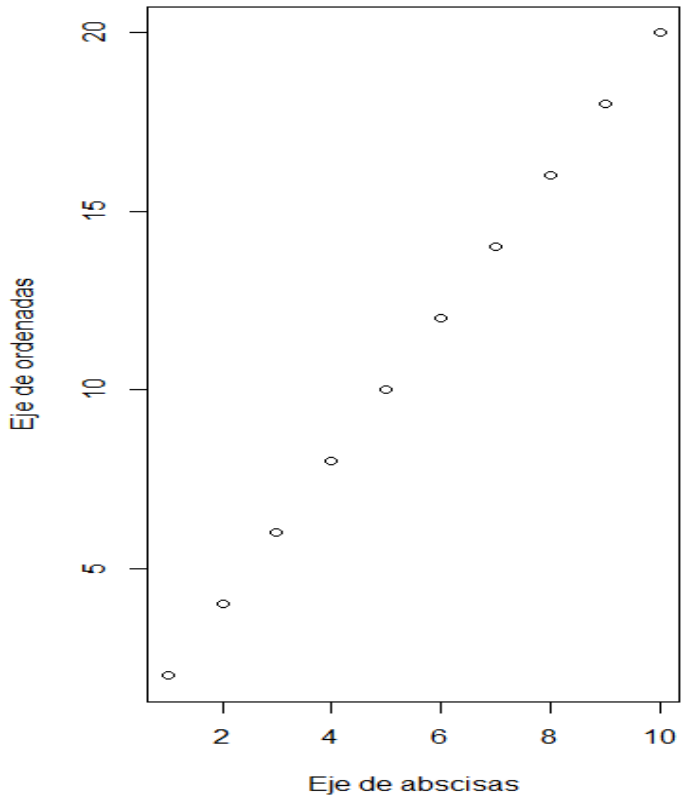

Figura 4.1: Función Plot

En la figura 4.2 podemos ver un cambio notable entre los que esta la cuadricula generada con el comando **panel.first=grid()**, las lineas verticales y horizontales agregadas con la función **abline** y el text que nos marca las intersecciones con la fucnión **text**.

En caso de querer colocar más de una grafica en una misma ventana podemos hacer uso de la función **par()** el cual crea una ventana en la que se indica el numero de espacios en los que queremos colocar las gráficas y la distribución en la que se hará dando el numero de filas y columnas.

En el siguiente ejemplo además de hacer uso de la función par() aprove-

charemos para ver otros tipos de gráficas que podemos hacer en R.

```
> x<-(1:6);y<-seq(3,18,3);
> a<-c(rep(3,13),rep(4,12),rep(2,16),
+      rep(1,12),rep(8,13),rep(10,10),
+      rep(5,15),rep(6,16))
> ingreso<-c(rep("alto",2),rep("bajo",6),rep("medio",4))
> est<-c(rep("basico",5),rep("med-bas",3),rep("med-alt"
   ,3),rep("alto",1))
> tabla<-table(ingreso,est)
> #Funcion par
> par(mfrow=c(3,2))
> #Grafica con plot
> plot(x,y,main="Grafica sencilla", type = "o")
> #Histograma
> hist(a, main="Histograma",col="orange",
+      border="blue",labels=T)
> #Grafico de Barras
> barplot(x,y,col=c("royalblue","gray"), space=.5,
+         main = "Grafico de Barras",sub = "Capitulo 4")
> #Grafico de Bigotes
> b<-c(1:12)
> c<-seq(4,10,.5)
> boxplot(x,y,c,d,
+         col=c("lightblue","pink","lightgreen","violet")
   ,
+         main="Bigotes",varwidth = T,notch = F)
> #Grafico de Mosaico
> mosaicplot(tabla, col=c("pink","lightblue","oldlace"))
> #Grafica con funcion curve
> curve(x*cos(y),0,10,100, col="steelblue1")
```

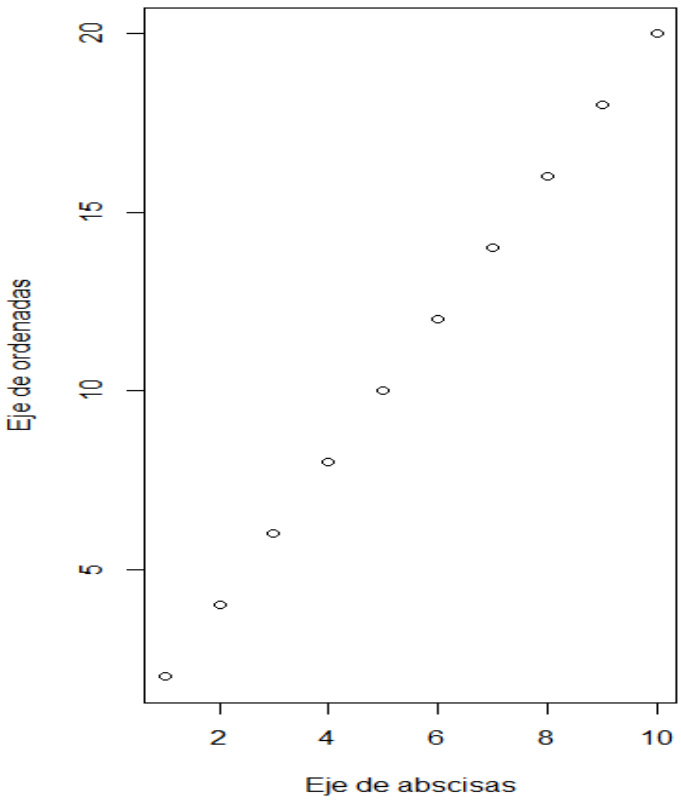

Figura 4.2: Segundo ejemplo de la función plot

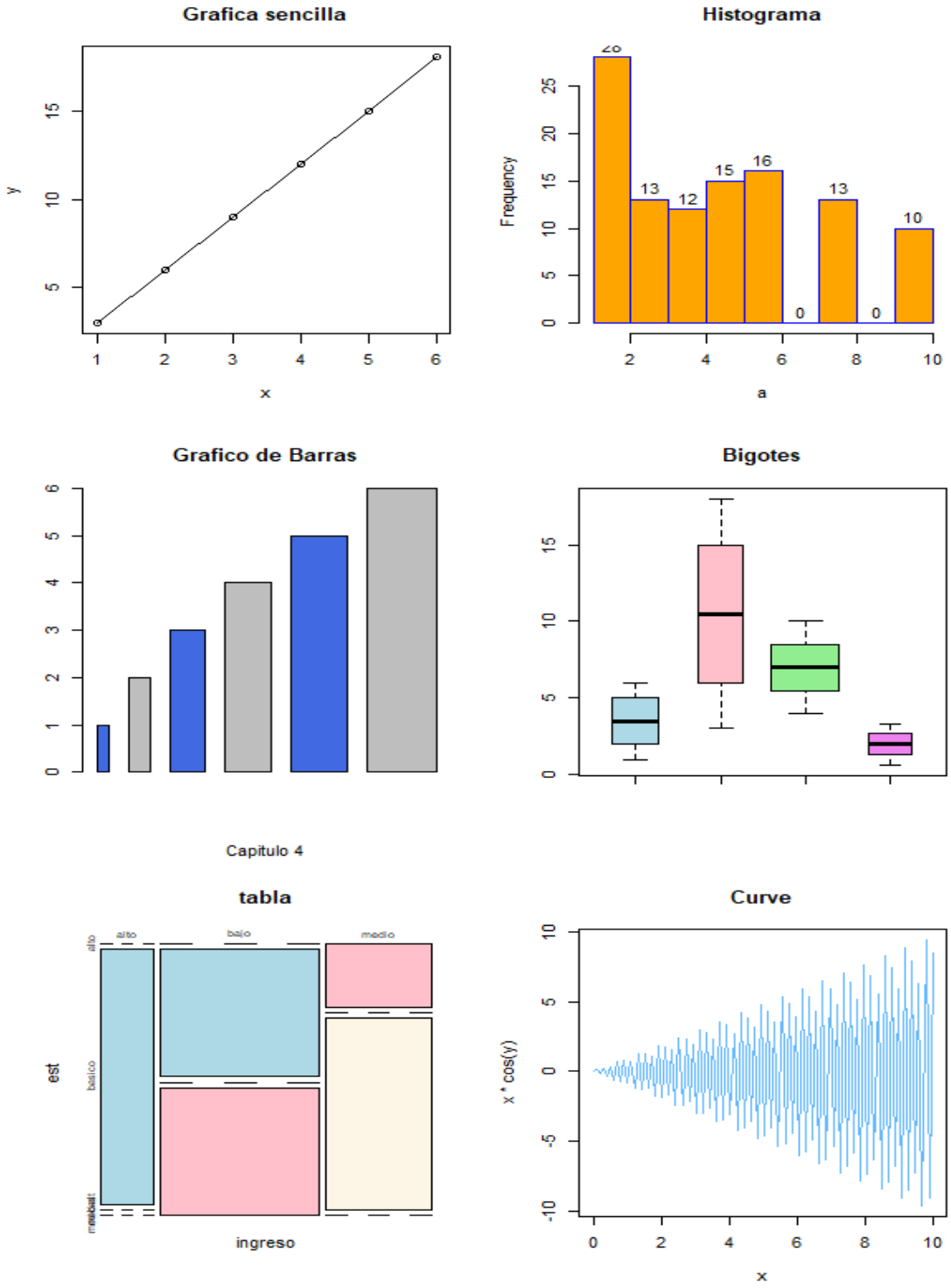

Figura 4.3: Graficas agrupadas con **par**

Capítulo 5

Creación de funciones

En este capitulo entenderemos las condiciones y el contexto en el que es adecuado utilizar una función creada por el usuario para un correcto uso de las mismas.

Es de suma importancia entender que las funciones son uno de los principales herramientas en la programación de algoritmos complejos.

Primero partiremos por la estructura general de una función, la cual es la siguiente:

```
nombre <- function(argumento_1, argumento_2, ... ,argumento_n) {
expresion_1
expresion_2
.
.
.
expresion_n
return(salida)
}
```

Cada una de las partes que la conforman tienen sus propias limitaciones, **nombre** hace referencia a el nombre que nosotros le asignaremos a nuestra función el cual no debe de ser el mismo a alguna de las funciones predeterminadas en R o una ya creada anteriormente por el usuario. Los argumentos son las variables con las que la función va a operar, aunque es posible no tener argumentos, las expresiones son el algoritmo de trabajo y la salida es el resultado de la función.

Para usar una función creada por el usuario basta con guardarla en nues-

tro directorio y mandarla a llamar desde el mismo directorio con la función **source** en un archivo distinto, el cual sera el espacio donde deseamos ocupar la función.
El siguiente ejemplo se trabajara sobre dos archivos .R diferentes, el primero en donde se realizara la creación de la función siguiendo la estructura anteriormente mencionada y el segundo archivo donde se llamara a la función con source y se empleara acorde con los parámetros establecidos.

Archivo creación de la función

```
> #Estimacion del Salario
> salario <-function(nivel,edad) {
+   if (nivel==1){
+     sal= 5000+150*edad
+   } else if(nivel==2){
+     sal= 5000+180*edad
+   } else if (nivel==3){
+     sal=6000+120*edad
+   } else {
+     cat("Nivel invalido","\n")
+   }
+   return(sal)
+ }
```

Archivo utilización de la función

```
> source("C:/Users/Documents/R/fun_salario.r")
> salario(1,18)
[1] 7700
> salario(2,25)
[1] 9500
> salario(3,32)
[1] 9840
> salario(4,28)
Nivel invalido
Error in salario(4, 28) : object 'sal' not found
```

El ejemplo anterior sirve como muestra de que con ayuda de las funciones podemos convertir una tarea larga que involucra más de un paso o evaluación en algo muy sencillo, además de ahorrarnos algunas lineas de programación en R, si la función creada es necesaria en más de un proceso o utilizada en más de un archivo el lector encontrara que son esenciales para la estructuración de procesos más complejos no contemplados en los paquetes de

R.

5.1. La estructura de una función

La forma en que las variables con las que una función opera es una pieza clave para sacar provecho de ellas, así como para evitar problemas y confusiones. Lo primero que hay que saber es que las variables con las que trabaje la función se quedan dentro de la misma y si se les manda a llamar R nos mandara un mensaje en el que indica que no ha encontrado el objeto, por lo que el lector deberá de hacer énfasis en las variables contenidas dentro de **return** pues estas son las únicas que R nos podrá proporcionar una vez terminado el proceso.

5.1.1. Ejemplo: Dentro de la función

Archivo creación de la función

```
> proba1<-function(aciertos,ensayos){
+   if(aciertos>ensayos | ensayos==0 ){
+     cat("Inserte parametros coherentes", "\n")
+   }else{
+     y<-(aciertos/ensayos)
+     return(y)
+   }
+ }
```

Archivo utilización de la función

```
> #Funcion Probabilidad Sencilla
> source("C:/Users/narum/Documents/R/fun_proba1.r")
> proba1(3,5)
[1] 0.6
> proba1(11231,5432154235)
[1] 2.067504e-06
> proba1(21,0)
Inserte parametros coherentes
> proba1(12,12)
[1] 1
> proba1(10,8)
Inserte parametros coherentes
> aciertos
Error: object 'aciertos' not found
```

```
> ensayos
Error: object 'ensayos' not found
> y
Error: object 'y' not found
```

El mensaje de error indica que las variables solicitadas son inexistentes, esto ocurre debido a que las variables existen dentro de la función, una vez que esta ha terminado su proceso todas las variables con las que trabajo se borran.
Si las variables se declaran con anterioridad y se mandan a llamar después de una función, aun cuando estas tengan el mismo nombre que las variables con las que trabaje la función su valor se mantendrá.

```
> aciertos<-10
> ensayos<-17
> y<-3
> proba1(3,8)
[1] 0.375
> aciertos
[1] 10
> ensayos
[1] 17
> y
[1] 3
```

5.2. Características de los argumentos

Normalmente se opera con argumentos que hayan sido introducidos de manera directa a la función, pero no es una regla obligatoria al momento de usar funciones,si las condiciones permiten que hagamos uso de valores anteriormente definidos nuestra función no tendrá problemas siempre y cuando el valor exista.

5.2.1. Ejemplo: Argumento fuera de la función

Archivo creación de la función

```
int_comp<-function(interes,periodos){
  if(interes==0 | periodos==0){
    cat("Los parametros podrian dar resultados
       inesperados", "\n")
```

```
  } else{
    y<-capital*((1+interes)^periodos)
    return(y)
  }
}
```

Archivo utilización de la función

```
> source("C:/Users/Documents/R/fun_int_comp.r")
> #Valor definidio previamente
> capital<-1200
> int_comp(.45,15)
[1] 316010.3
> #Sin valor previamente definido
> rm("capital")
> int_comp(.45,15)
Error in int_comp(0.45, 15) : object 'capital' not found
```

5.2.2. Ejemplo: Ausencia de argumentos

Podemos declarar un valor por default en caso de no dar argumentos **Archivo creación de la función**

```
#Valor por Default
ejem_def<-function(x=5){
  return(x)
}
```

Archivo utilización de la función

```
> source("C:/Users/narum/Documents/R/fun_default.r")
> ejem_def(3)
[1] 3
> ejem_def()
[1] 5
```

En caso de no recordar los parámetros necesarios para hacer uso de una función podemos observarlos con la función **formals**:

> formals(int_comp)
$interés
$periodos

5.3. Programación basada en vectores

Una de las mayores ventajas que tiene R es la vectorización de las funciones es decir que si el argumento de una función es un vector la función actuara sobre cada elemento por separado, evitando así que tengamos que usar bucles para cada uno de los elementos.
Además, para facilitar este aspecto R provee una familia de funciones que el usuario puede utilizar cuando define funciones, entre algunas de las tantas que podemos encontrar se encuentran: **apply, sapply, lapply, tapply, mapply** a continuación ejemplificaremos cada una de estas.

5.3.1. Apply

La función apply consta de 3 parámetros: **x:** Puede ser una matriz, lista vector, data frame o arrays, los cuales veremos más adelante.
margin: Indica sobre que seccion se aplicara la función, el valor 1 indica que se opera sobre filas y el valor 2 indica que se opera sobre columnas.
fun: Señala la función que se utilizara en el vector, esta función tiene que ser alguna que requiera preferentemente un vector como argumento, se debe de tener cuidado con aquellas funciones que requieran más de un argumento.

```
> #Funcion Aply
> (A <-matrix(rep(1:4),3,4))
     [,1] [,2] [,3] [,4]
[1,]    1    4    3    2
[2,]    2    1    4    3
[3,]    3    2    1    4
> #Media sobre filas
> (B <-apply(A,1,mean))
[1] 2.5 2.5 2.5
> #Media sobre columnas
> (B <-apply(A,2,mean))
[1] 2.00 2.33 2.67 3.00
>
> #Obtenemos el mismo resultado pero con más
> #lineas de codigo:
>
> #Media sobre filas
> mean(A[1,]);mean(A[2,]);mean(A[3,])
[1] 2.5
[1] 2.5
[1] 2.5
> #Media sobre columnas
```

```
> mean(A[,1]);mean(A[,2]);mean(A[,3]);mean(A[,4])
[1] 2
[1] 2.33
[1] 2.67
[1] 3
```

Si la variable con la que estamos trabajando no es una matriz podemos transformarla haciendo uso de **as.matrix** para evitar algún inconveniente,además de que es importante señalar que la función apply devuelve resultados del mismo tipo que la función aplicada devuelve en su uso normal.
Las siguientes funciones son derivados de la función apply, hechas para usar variables en concreto, de hecho no es difícil notar que lo único que cambia en las funciones subsecuentes es solo la primer letra de la función.

5.3.2. Lapply

La diferencia de esta función con apply es que esta trabaja con listas, recibe un numero determinado es estas y regresa una sola.
El capitulo 6 *Factores, Listas y Dataframe* profundiza en el concepto y las características de una lista, sin embargo en el siguiente ejemplo haremos uso de una de ellas antes de su introducción ya que es necesario para el ejemplo de la función lapply.

```
> #Funcion Lapply
> A <-matrix(rep(1:4),3,4)
> B <-matrix(c(1:8),4,4)
> C <-matrix(seq(1,3,.5),3,5)
> lista <-list(A,B,C)
> #Suma
> lapply(lista,sum)
[[1]]
[1] 30

[[2]]
[1] 72

[[3]]
[1] 30

> #Raíz Cuadrada
> lapply(lista,sqrt)
[[1]]
     [,1] [,2] [,3] [,4]
```

```
[1,] 1.00 2.00 1.73 1.41
[2,] 1.41 1.00 2.00 1.73
[3,] 1.73 1.41 1.00 2.00

[[2]]
     [,1] [,2] [,3] [,4]
[1,] 1.00 2.24 1.00 2.24
[2,] 1.41 2.45 1.41 2.45
[3,] 1.73 2.65 1.73 2.65
[4,] 2.00 2.83 2.00 2.83

[[3]]
     [,1] [,2] [,3] [,4] [,5]
[1,] 1.00 1.58 1.22 1.73 1.41
[2,] 1.22 1.73 1.41 1.00 1.58
[3,] 1.41 1.00 1.58 1.22 1.73
```

5.3.3. Sapply

Se diferencia de apply y lapply por trabajar con listas y regresar un vector.

```
> #Aplicamos la funcion sapply
> datos$sap <- sapply(as.vector(datos$sap), function(x)
    if(x %in% honda) "Honda" else x )
> datos$sap <- sapply(as.vector(datos$sap), function(x)
    if(x %in% hyundai) "Hyundai" else x )
>
> datos$sap
    Honda     Honda     Honda     Honda     Honda
  "Honda"   "Honda"   "Honda"   "Honda"   "Honda"
   accent    starex   elantra    accent    starex
"Hyundai" "Hyundai" "Hyundai" "Hyundai" "Hyundai"
   starex   elantra
"Hyundai" "Hyundai"
```

5.3.4. Tapply

Tapply realiza una operación sobre un vector agrupado por los factores que se indique.
Por aras de la facilidad, entenderemos de manera sencilla que un factor es un elemento que agrupa un vector de acuerdo a un orden de niveles que nosotros le indicaremos.

El capitulo 6 *Factores, Listas y Dataframe* profundiza en el concepto y las características de un factor, sin embargo en el siguiente ejemplo haremos uso de uno antes de su introducción ya que es necesario para el ejemplo de la función tapply.
La función tapply consta de tres parámetros:
x: Un vector definido.
index: Uno o más factores con el mismo numero de elementos que x.
fun: Función que se utilizara en el vector, esta función tiene que ser alguna que requiera preferentemente un vector como argumento, se debe de tener cuidado con aquellas funciones que requieran más de un argumento.

```
> #Función Tapply
> x <-seq(1,40,2)
> x
 [1]  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33
[18] 35 37 39
> y <- factor(rep(letters[1:5], each = 4))
> y
 [1] a a a a b b b b c c c c d d d d e e e e
Levels: a b c d e
> #Aplicando función tapply con media
> tapply(x,y,mean)
 a  b  c  d  e
 4 12 20 28 36
> #Obtenemos el mismo resultado pero con más lineas de
   codigo
> (a1<-mean(x[1:4]))
[1] 4
> (b1<-mean(x[5:8]))
[1] 12
> (c1<-mean(x[9:12]))
[1] 20
> (d1<-mean(x[13:16]))
[1] 28
> (e1<-mean(x[17:20]))
[1] 36
```

5.3.5. Mapply

Esta función realiza operaciones entre variables y devuelve una lista o vector, su utilidad consiste en realizar la operación indicada sobre el primer vector acorde a los siguientes.

```
> #Funcion Mapply
> options(digits = 4)
> (x <-rep(3,5))
[1] 3 3 3 3 3
> (y <-seq(1,10,2))
[1] 1 3 5 7 9
> (z <-sqrt(y))
[1] 1.000 1.732 2.236 2.646 3.000
> mapply(prod,x,y)
[1]  3  9 15 21 27
> mapply(sum,z,y,z)
[1]  3.000  6.464  9.472 12.292 15.000
```

5.4. Recursividad

Una función recursiva es aquella que hace uso de si misma para obtener un resultado, en la practica existen muchos algoritmos recursivos, para un mejor entendimiento haremos uso de dos ejemplos clásicos y sencillos para explicar la recursividad: la sucesión de Fibonacci, y el operador factorial.

5.4.1. Ejemplo: Sucesión de Fibonacci

Archivo creación de la función

```
#Creamos la funcion
fibonacci<-function(n){
  v <-rep(0,n)
  if(n==1){
      x=0
      } else if(n==2){
        x=1
        }else {
          x=fibonacci(n-1)+fibonacci(n-2)
          }
  return(x)
}
```

Archivo utilización de la función

```
> #Sucesion de Fibonacci
> source("C:/Users/Documents/R/fun_fibonacci.r")
```

```
> fibonacci(1)
[1] 0
> fibonacci(2)
[1] 1
> fibonacci(3)
[1] 1
> fibonacci(4)
[1] 2
> fibonacci(5)
[1] 3
> fibonacci(6)
[1] 5
```

Si el usuario deseara obtener un vector contenido con los elementos de la sucesión de Fibonacci bastaría con crear una función que llamara los elementos obtenidos en esta función, es decir una función que hace uso de otra función.

5.4.2. Ejemplo: Operador factorial

Archivo creación de la función

```
#Factorial de un numero
factorial<-function(n){
  if (n==0){
    f=1
  } else {
    f= n*factorial(n-1)
  }
  return(f)
}
```

Archivo utilización de la función

```
> #Operador Factorial
> source("C:/Users/narum/Documents/R/fun_factorial.r")
> factorial(1)
[1] 1
> factorial(2)
[1] 2
> factorial(3)
[1] 6
> factorial(4)
[1] 24
```

```
> factorial(5)
[1] 120
```

5.5. Los errores

En este punto el lector se habrá preguntado que ocurriría si introducimos una o más variables que no sean coherente con los parámetros correspondientes.En esta sección estudiaremos 2 formas diferentes de abordar el problema de tal forma que podamos corregir los errores o definirlos de una forma amigable y muy sencilla.

5.5.1. Función Stop

Esta función manda un mensaje cual sea dado por el usuario en caso de darse un error en medio del proceso. **Archivo creación de la función**

```
#Proabilidad con error
proba2<-function(aciertos,ensayos){
  if (aciertos>ensayos){
    stop("Teóricamente no hay probabilidades mayores a 1"
        , call. = TRUE)
  } else  if ( aciertos<0 | ensayos<0){
    stop("¿En que demonios estas pensando?", call. =FALSE
        )
  } else {
  y<-(aciertos/ensayos)
  }
  return(y)
}
```

Archivo utilización de la función

```
> source("C:/Users/Documents/R/fun_proba2.r")
> proba2(12,10)
Error in proba2(12, 10) : Teoricamente no hay
   probabilidades mayores a 1
> proba2(-2,6)
Error: ¿Usuario, en que estas pensando?
```

5.5.2. Función Browser

La función **browser** comparte la similitud con la función stop de ser mandada a llamar dentro de otra función. Pero su principal objetivo es detener el programa temporalmente y analizar de manera detallada el funcionamiento del programa, así sea de un proceso o entrada especifico o de todos uno por uno.
La función **browser** trabaja con 3 principales comandos enfocados a la forma de correr el programa en el que estamos trabajando, estos son n,c y Q los cuales operan de la siguiente manera: **Archivo creación de la función**

```
# Suma de numeros aleatorios
random.sum <- function(n) {
  browser()
  x[1:n] <- ceiling(10*runif(n))
  cat("x:", x[1:n], "\n")
  return(sum(x))
}
```

Archivo utilización de la función El comando **n** sirve para avanzar linea tras linea en el programa, el cual se detendrá al momento de encontrar un error

```
> random.sum(4)
Called from: random.sum(4)
Browse[1]> n
debug at #3: x[1:n] <- ceiling(10 * runif(n))
Browse[2]> n
Error in x[1:n] <- ceiling(10 * runif(n)) : object 'x'
   not found
```

El comando c detiene el proceso de browser y continua evaluando el programa con normalidad. En este caso nos marca de nuevo un error por que el vector x no esta declarado.

```
> random.sum(4)
Called from: random.sum(4)
Browse[1]> c
Error in x[1:n] <- ceiling(10 * runif(n)) : object 'x'
   not found
```

Y por ultimo el comando Q saldrá por completo de browser y del programa sin terminar el proceso.

```
> random.sum(4)
Called from: random.sum(4)
Browse[1]> Q
#Sale de browser y no termina el proceso.
```

Capítulo 6

Factores, Listas y Dataframe

Al inicio de este libro se menciono que el principal objetivo con el que se creo el entorno y lenguaje en R era el análisis estadístico, como consecuencia de ello R se encarga de proveer al usuario estructuras sofisticadas que permiten la representación, manipulación y análisis de datos de manera más cómoda e intuitiva. Al ser de vital importancia estas estructuras empezaremos definiendo de manera general cada una de ellas y posteriormente veremos su aplicación con un ejemplo.

6.1. Factores

Un factor es un vector que contiene datos de tipo ordinal o categórico, en el que cada uno de los posibles valores a tomar es llamado *nivel*, podemos decir que un factor es un vector con un alcance mayor enfocado a datos categóricos u ordinales.
Las razones por las cuales se decide trabajar usando factores y no un vector son varias, en este apartado destacaremos las que consideramos más importantes. La primera de ellas es la rápida identificación de los elementos contenidos dentro de un factor pues de manera inmediata se relaciona con el uso de variables categóricas, numéricas u ordinales, una segunda razón es su facilidad para el almacenamiento y por ultimo, la seguridad, ya que un vector acepta valores de manera indiscriminada, mientras que los factores hacen una revisión del tipo de variable que se desea introducir evitando así que se introduzcan variables de otro tipo que no coincidan con los niveles del factor.

Para crear un factor aplicamos la función **factor** sobre un vector, al hacerlo automáticamente los valores dentro del vector se convierten en niveles,

ejemplo:

```
> helado<-factor(helado, ordered = TRUE)
> #Declaramos el vector helado
> helado<-c("chocolate","fresa","vainilla","chocolate",
+           "fresa","chocolate","vainilla",
+           "chocolate","vainilla","fresa",
+           "fresa","vainilla","chocolate","chocolate",
+           "fresa")
> #Aplicamos la función factor
> helado<-factor(helado, ordered = TRUE)
> #Comprobamos
> is.ordered(helado)
[1] TRUE
> table(helado)
helado
chocolate     fresa   vainilla
        6         5         4
```

En el código anterior usamos la función **table** para comprobar los niveles del factor "helado" , esta función se comporta de acuerdo con el numero de factores o vectores que se introduzcan.

Por default R acomoda los niveles alfabéticamente pero el usuario tiene la libertad de hacerlo o no, puede especificar el orden de los niveles con el parámetro *ordered*.

```
> #Factor ordenado
> helado<-factor(helado, ordered = TRUE)
> helado
 [1] chocolate fresa     vainilla  chocolate fresa
 [6] chocolate vainilla  chocolate vainilla  fresa
[11] fresa     vainilla  chocolate chocolate fresa
Levels: chocolate < fresa < vainilla
>
> #Factor no ordenado
> helado<-factor(helado, ordered = FALSE)
> helado
 [1] chocolate fresa     vainilla  chocolate fresa
 [6] chocolate vainilla  chocolate vainilla  fresa
[11] fresa     vainilla  chocolate chocolate fresa
Levels: chocolate fresa vainilla
```

Podemos notar que la diferencia se encuentre en la ultima linea impresa por

la consola, en los respectivos comandos, el primero asigna una jerarquía y el segundo no.
En caso de verse en la necesidad de cambiar el nombre de los niveles bastara con usar la función **labels** y si el usuario lo desea podemos indicar que considere los valores NA que se encuentren en nuestro factor con la función **addNA** de la siguiente forma:

```
 > #Cambiamos el nombre de los niveles
> sabores<-factor(helado, levels = c("chocolate","fresa",
   "vainilla"),
+              labels = c("CH","FR","VA"), ordered =
   FALSE)
> table(addNA(sabores))

  CH   FR   VA <NA>
   6    5    4    0
```

Ahora el valor NA se agrego a la tabla, para observar los valores omitidos, de una manera similar podemos consultar solo algunos valores en especifico en los que estemos interesados, en caso de tener un factor muy extenso, de la siguiente forma:

```
> #Declaramos el vector helado
> helado<-c("chocolate","fresa","vainilla","chocolate",
+           "fresa","chocolate","vainilla",
+           "chocolate","vainilla","fresa",
+           "fresa","vainilla","chocolate","chocolate",
+           "fresa")
>
> #Aplicamos la función factor
> helado<-factor(helado, ordered = TRUE)
> #Declaramos las variables que nos interesan
> table(helado[helado=="chocolate" | helado=="vainilla",
+       drop=T])

chocolate  vainilla
        6         4
```

Después de haber visto las ventajas y facilidad con la que un factor se puede manejar, tenemos que prever las dificultades que el uso de esta herramienta puede presentar. En nuestros ejemplos anteriores se mostró que la forma más usual de crear un factor es partiendo de un vector para después convertirlo,bueno pues el factor puede hacer el mismo proceso pero al revés tanto

con variables tipo carácter como un vector con datos de tipo numérico.

```
> #Declaramos el vector apellidos
> apel<-c("Carrillo","Alvaro","Carreto","Martinez","
    Valdez")
> #Aplicamos la función factor
> apel<-factor(apel)
> #Comprobamos
> as.character(apel) #Recuperamos de manera correcta los
    valores
[1] "Carrillo" "Alvaro"   "Carreto"  "Martinez"
[5] "Valdez"
> as.numeric(apel) #No recuperamos de manera correcta los
    valores
[1] 3 1 2 4 5
> c(apel,5,6) #Agrega valores al vector numerico basado
    en apel
[1] 3 1 2 4 5 5 6
> as.numeric(apel)
[1] 3 1 2 4 5
> #Si el vector es numerico
> edad<-factor(c("17","18","19","20"))
> as.numeric(levels(edad)) #Se recuperael factor
[1] 17 18 19 20
> as.character(levels(edad)) #No se recupera
[1] "17" "18" "19" "20"
> as.numeric(edad) #Solo da el orden
[1] 1 2 3 4
```

6.2. Dataframe

Un data frame es un conjunto de filas y columnas ordenados de tal modo que las columnas contengan diferentes tipos de elementos entre si, pero que sean homogéneas de manera individual, por dar un ejemplo si nosotros manejamos una cartera de seguros nos veremos en la necesidad de tener columnas que contengan datos numéricos(numero del siniestro, valor de la prima, monto asegurado), caracteres(nombre del asegurado, cobertura), ordinales (sexo, fumador o no fumador), por lo que los dataframe son la herramienta por excelencia para el manejo de este tipo de datos, debido a que una matriz en R debe contener para todos sus espacios el mismo tipo de dato y el data frame no.

La creación de un data frame generalmente es a partir de un archivo previamente existente haciendo uso de la función **read.table** que tiene los siguientes argumentos:

file es el parámetro en el que colocaremos ya sea el nombre del archivo del que extraeremos los datos, este puede ser un archivo csv, txt o incluso un URL. El numero de elementos en cada una de las columnas debe de ser el mismo (Los datos N/A o NULL también se cuentan)

header indica si el archivo del que extraemos los datos tiene encabezado en cada una de las columnas.

sep indica cual es el símbolo que se utiliza para separar cada uno de los valores, este puede ser una coma, una barra vertical, un signo @, etcétera. Si no llenamos este campo, R asumirá que el valor de separación por default es un espacio en blanco.

dec indica el separador de los números decimales, puede ser un punto o una coma.

colnames de no tener encabezado para las columnas, este parámetro habilita la opción de asignar un nombre a cada una de las columnas.

rownames asigna un nombre a cada una de las filas.
Esta es la forma general de leer un archivo y convertirlo en data frame, pero existen funciones predefinidas en R que facilitan la lectura de archivos específicos, algunos de ellos son: **read.csv, read.csv2, red.delim, red.delim2** los cuales tienen pequeños cambios entre sí respecto a los parámetros que mencionamos. Recordamos que el lector puede consultar la diferencia de estos usando la función **help()** o el signo **?** seguido del nombre de la función a consultar.
Este es un ejemplo de archivo csv separado por comas visto desde una aplicación análoga al bloc de notas en windows:

Clase,M_Rec,A_Rec,Poliza,Monto,Comp,Ramo
tit ,3,2008,564501,20563, Inbursa , deuda
tit ,2,2007,576201,80682, Inbursa , vida
tit ,2,2007,563401,15297, Inbursa , deuda
tit ,2,2007,563501,6940, Inbursa , deuda

A continuación se muestra el proceso para como convertirlo a data frame

```
> #Asignamos un nombre a nuestra base y lo convertimos a
   data frame
> base <-read.table("C:/Users/Documents/R/bases/bd_
   seguros.csv", header=TRUE, sep=",", dec=".")
> #Head permite ver los primeros datos del dataframe
> head(base)
  Clase M_Rec A_Rec Poliza  Monto      Comp    Ramo
1  tit      3  2008 564501  20563  Inbursa   deuda
2  tit      2  2007 576201  80682  Inbursa    vida
3  tit      2  2007 563401  15297  Inbursa   deuda
4  tit      2  2007 563501   6940  Inbursa   deuda
5  tit      2  2007 575501 219850  Inbursa    vida
6  tit      2  2007 575501 229493  Inbursa    vida
> #Tail permite ver los ultimos datos del dataframe
> tail(base)
     Clase M_Rec A_Rec   Poliza  Monto    Comp    Ramo
2176  tit      9  2010 9VD00055  65775  tokio   deuda
2177  tit      9  2010 9VG00152 300615  tokio    vida
2178  tit      9  2010 9VG00150 348564  tokio    vida
2179  tit      9  2010 9VG00151  39627  tokio    vida
2180  tit      9  2010 9VG00152 303275  tokio    vida
2181  tit      9  2010 9VG00152 342245  tokio    vida

#Comprobamos el tipo de estructura que hemos creado
> is.data.frame(base)
[1] TRUE
```

Cada una de las columnas de nuestro data frame "base" tiene asignado un nombre, esto proporciona diversas formas de mandar a llamar una columna en especifico, haciendo uso de corchetes [[]] o haciendo uso del signo $ seguido del nombre de la columna que queremos mandar a llamar. El uso del doble corchete debe realizarse con cuidado pues omitir un par de ellos nos devolverá objetos diferentes

```
base[[6]]
base[["Comp"]]
base$Comp
#Todas nos dan el mismo resultado y extraen los mismos
   valores
> a<-base[[6]]
> b<-base[6]
> mode(a)
```

```
[1] "numeric"
> mode(b)
[1] "list"
#Podemos obtener el nombre de nuestras columnas
> names(base)
[1] "Clase"  "M_Rec"  "A_Rec"  "Poliza" "Monto"  "Comp"
[7] "Ramo"
#Y cambiarlos tambien
names(base)<-c("c1","c2","c3","c4","c5","c6","c7")
names(base)
[1] "c1" "c2" "c3" "c4" "c5" "c6" "c7"
```

Podemos crear una nueva variable a partir de dos columnas, esto con el fin de hacer una comparativa más cómoda

```
> #Seleccion de columnas
> pol_mont<-base[c(1,4,5)]
> #Extraccion de un numero especifico
> pol_mont[12:16,]
     Clase Poliza Monto
12     tit  576201 75924
13     tit  576201 77952
14     tit  576201 73926
15    hijo  575501 78940
16   conyu  575601 83091
>
> #Comrpobamos si el elemento es de nuevo data frame
> is.data.frame(pol_mont)
[1] TRUE
```

A continuación presentaremos un ejemplo más eficaz pero también más complejo que el anterior, introduciendo un nuevo operador, le cual trabaja comparando los elementos de una variable dentro de otra regresando un vector de valores lógicos con el mismo numero de elementos que el primero, este operador es **%in %**

6.2.1. Ejemplo: Subset

La función subset trabaja con 3 principales argumento:

- **x**: Objeto del cual vamos a tomar crear el subonjunto
- **subset**: variable o vector de expresiones logicas que indican los valores que el subconjunto va a tomar

- **select**: variable o vector que indica las columnas de las que se obtendra el subconjutno del data frame

```
> #Indicamos el nombre del subconjunto.
> #El argumento subset pide que solo recolecte aquellos
> #valores mayores a 40,000 y select pide las columnas
   que nos interesan
> pcr<-subset(base, subset = Monto >40000,
+             select = c(Poliza,Clase, Ramo))
> #Obtenemos los primeros 7 valores
> head(pcr, n=7)
   Poliza Clase    Ramo
2  576201   tit    vida
5  575501   tit    vida
6  575501   tit    vida
7  564801   tit   deuda
11 576201   tit    vida
12 576201   tit    vida
13 576201   tit    vida
> #Obtenemos un resumen
> summary(pcr)
     Poliza         Clase           Ramo
 576201 :112    conyu : 103    deuda : 266
 576203 : 96    hijo  :  39    vida  :1273
 576202 : 92    tit   :1397
 575502 : 87
 575501 : 78
 576101 : 75
 (Other):999
```

6.3. Listas

Una lista es el objeto más amplio en R, en ellas se puede contener elementos de todo tipo, datos numéricos, categóricos, ordinales, caracteres, inclusive otras listas. Debido a la amplitud de las listas estas son utilizadas comúnmente en estructuras de recolección de datos cada vez más complejas, una vez que el lector se haya familiarizado con su uso y con el entorno el uso de lista se volvera algo imprescindible.
Para crear una lista haremos uso de la función **list**, al igual que los dataframe el uso correcto de paréntesis es muy importante, [1] paréntesis sencillos indican que el elemento con el que vamos a trabajar es un vector de elemen-

tos contenidos dentro de la lista y el uso de doble paréntesis [[2]] indica que el elemento con el que vamos a trabajar es un elemento contenido dentro del vector de elementos contenido en la lista.
Para un mayor entendimiento lo ilustraremos con un ejemplo:

```
> #Vectores de diferente tipo
> a<-c(1:5)
> b<-"singular"
> c<-TRUE
> d<-c("H","M")
> e<-c()
> f<-list(a,e)
> #Creamos una lista a usar
> lista<-list(b,c,d,f)
> lista[1]
[[1]]
[1] "singular"

> lista[[3]][1]
[1] "H"
> lista[3]
[[1]]
[1] "H" "M"

> lista[[3]]
[1] "H" "M"
```

A diferencia de los data frame, los elementos de una lista no tienen que tener un nombre forzosamente, sin embargo es posible hacerlo antes y después de su creación de manera análoga que el dataframe

```
> #No es forzosa la asignacion
> names(lista)
NULL
> #Pero se pueden asignar
> names(lista)<-c("variable","logica",
+                 "ordinal","list")
> names(lista)
[1] "variable" "logica"   "ordinal"  "list"
> lista[3]
$ordinal
[1] "H" "M"
```

Si nosotros deseáramos convertir nuestra lista aun vector podemos usar la

función **unlist**, si la lista que deseamos convertir contiene en si misma otras listas, estas también serán transformadas si el argumento recursive esta activado con el valor TRUE.

```
> lista
$variable
[1] "singular"

$logica
[1] TRUE

$ordinal
[1] "H" "M"

$list
$list[[1]]
[1] 1 2 3 4 5

$list[[2]]
NULL


> #Comparamos la funcion con el argumento recursive
   intercambiado
> (lista2<-unlist(lista, recursive = T))
  variable     logica   ordinal1   ordinal2      list1
         list2
"singular"     "TRUE"        "H"        "M"        "1"
          "2"
     list3      list4      list5
       "3"        "4"        "5"
> (lista3<-unlist(lista, recursive = F))
$variable
[1] "singular"

$logica
[1] TRUE

$ordinal1
[1] "H"

$ordinal2
[1] "M"
```

```
$list1
[1] 1 2 3 4 5

$list2
NULL

> #Y comprobamos que tipo de variable es la que obtenemos
> class(lista2);class(lista3)
[1] "character"
[1] "list"
```

Si deseamos realizar un resumen de los valores cuando nuestras listas o dataframe son muy amplios podemos usar la función str().

```
> #Funcion str
> calificaciones<-list("1semestre"=c(8,8,7,6,9,9),
+                      "2semestre"=c(6,9,8,9,6,7),
+                      "3semestre"=c(10,9,8,9,7,8),
+                      "4semestre"=c(9,9,7,8,7,8),
+                      "5semestre"=c(10,10,8,7,7,8))
> str(calificaciones)
List of 5
 1semestre: num [1:6] 8 8 7 6 9 9
 2semestre: num [1:6] 6 9 8 9 6 7
 3semestre: num [1:6] 10 9 8 9 7 8
 4semestre: num [1:6] 9 9 7 8 7 8
 5semestre: num [1:6] 10 10 8 7 7 8
```