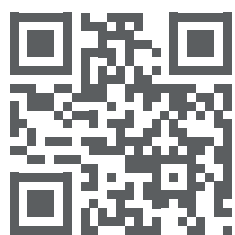




Universitat
de les Illes Balears

**#SOM
UIB**



AprendeR: Introducción al tratamiento de datos con R y RStudio



Módulo 2

Lección 3

Vectores y otros tipos de listas

Edició: abril 2016

Edita: Campus Extens – UIB Virtual

Disseny portada: Direcció de l'Estratègia de Comunicació i Promoció Institucional (dircom.uib.cat)



Aquesta obra està subjecta a una llicència CC

[Reconeixement-NoComercial-SenseObraDerivada 4.0 Internacional](https://creativecommons.org/licenses/by-nc-nd/4.0/)

Lección 3

Vectores y otros tipos de listas

Un *vector* es una secuencia ordenada de datos. R dispone de muchos tipos de datos, entre los que destacamos:

- `logical` (lógicos: `TRUE`, verdadero, o `FALSE`, falso)
- `integer` (números enteros)
- `numeric` (números reales)
- `complex` (números complejos)
- `character` (palabras)

Una restricción fundamental de los vectores en R es que todos sus objetos han de ser del mismo tipo: todos números, todos palabras, etc. Cuando queramos usar vectores formados por objetos de diferentes tipos, tendremos que usar *listas heterogéneas*, `lists` en el argot de R (véase la Sección 3.5).

3.1. Construcción de vectores

Para definir un *vector* con unos elementos dados, por ejemplo

1, 5, 6, 2, 5, 7, 8, 3, 5, 2, 1, 0,

podemos aplicar la función `c` a estos elementos separados por comas.

```
> x=c(1,5,6,2,5,7,8,3,5,2,1,0)
> x
[1] 1 5 6 2 5 7 8 3 5 2 1 0
```

Si queremos crear un vector de palabras con la instrucción `c`, tenemos que entrarlas obligatoriamente entre comillas. R también nos las muestra entre comillas.

```
> nombres=c("Pep","Catalina","Joan","Pau")
> nombres
[1] "Pep" "Catalina" "Joan" "Pau"
> nombres=c(Pep,Catalina,Joan,Pau) #Si nos olvidamos de las
  comillas...
Error: object 'Pep' not found
```

Hemos mencionado que todos los elementos de un vector han de ser del mismo tipo. Por este motivo, si concatenamos datos de diferentes tipos en un vector, R automáticamente los convertirá a un tipo que pueda ser común a todos ellos. El orden de conversión entre los tipos que hemos explicado al principio de la lección es: `character` gana a `complex`, que gana a

`numeric`, que gana a `integer`, que gana a `logical`. Así, cuando alguna entrada de un vector es de tipo palabra, R considera el resto de sus entradas como palabras (y las muestra entre comillas), como se puede ver en el siguiente ejemplo:

```
> c(2,3.5,TRUE,"casa")
[1] "2"      "3.5"    "TRUE"   "casa"
```

Otra posibilidad para crear un vector es usar la función `scan`. Si ejecutamos la instrucción `scan()` (así, con el argumento vacío), R abre en la consola un entorno de diálogo donde podemos ir entrando datos separados por espacios en blanco; cada vez que pulsemos la tecla *Entrar*, R importará los datos que hayamos escrito desde la vez anterior en que la pulsamos y abrirá una nueva línea donde esperará más datos; cuando hayamos acabado, dejamos la última línea en blanco (pulsando por última vez la tecla *Entrar*) y R cerrará el vector.

Por ejemplo, para crear un vector `x_scan` que contenga dos copias de

1 5 6 2 5 7 8 3 5 2 1 0,

podemos hacer lo siguiente:¹

```
> x_scan=scan() #Y pulsamos Entrar
1: 1 5 6 2 5 7 8 3 5 2 1 0
13: 1 5 6 2 5 7 8 3 5 2 1 0
25:
Read 24 items
> x_scan
[1] 1 5 6 2 5 7 8 3 5 2 1 0 1 5 6 2 5 7 8 3 5 2 1 0
```

La función `scan` también se puede usar para copiar en un vector el contenido de un fichero de texto situado en el directorio de trabajo, o del que conozcamos su dirección en Internet. La manera de hacerlo es aplicando `scan` al nombre del fichero o a su *url*, entrados en ambos casos entre comillas. Por ejemplo, para definir un vector llamado `notas` con las notas de un examen que tenemos guardadas en el fichero `http://aprender.uib.es/Rdir/notas.txt`, sólo tenemos que entrar:

```
> notas=scan("http://aprender.uib.es/Rdir/notas.txt")
Read 65 items
> notas
 [1] 4.1 7.8 5.8 6.5 4.8 6.9 1.3 6.4 4.6 6.9 9.4
[12] 3.0 6.8 4.8 5.6 7.7 10.0 4.4 1.7 8.0 6.3 3.0
[23] 7.5 3.8 7.2 5.7 7.3 6.0 5.7 4.7 5.1 1.5 7.0
[34] 7.0 6.0 6.6 7.2 5.0 3.5 3.3 4.7 5.4 7.1 8.2
[45] 6.7 0.1 5.1 6.8 6.9 8.8 4.5 6.6 2.0 3.0 6.7
[56] 7.9 7.7 6.4 3.0 5.3 5.1 5.3 5.1 5.4 3.0
```

Si primero descargamos este fichero, sin cambiarle el nombre, en el directorio de trabajo de R, para definir el vector anterior bastará entrar:

¹ Con el editor de textos hemos copiado la secuencia, y hemos pulsado *Entrar* después de cada pegado. Probadlo vosotros.

```
> notas2=scan("notas.txt")
Read 65 items
> notas2
 [1]  4.1  7.8  5.8  6.5  4.8  6.9  1.3  6.4  4.6  6.9  9.4
[12]  3.0  6.8  4.8  5.6  7.7 10.0  4.4  1.7  8.0  6.3  3.0
[23]  7.5  3.8  7.2  5.7  7.3  6.0  5.7  4.7  5.1  1.5  7.0
[34]  7.0  6.0  6.6  7.2  5.0  3.5  3.3  4.7  5.4  7.1  8.2
[45]  6.7  0.1  5.1  6.8  6.9  8.8  4.5  6.6  2.0  3.0  6.7
[56]  7.9  7.7  6.4  3.0  5.3  5.1  5.3  5.1  5.4  3.0
```

Si usamos el menú «*Import Dataset*» de la pestaña **Environment** para importar un vector contenido en un fichero externo (véase la página 2-6 de la Lección 2), obtendremos en realidad un *data frame* de una sola columna, llamada V1. Para construir un vector con esta columna, podemos usar luego la instrucción

nombre_del_vector=nombre_del_dataframe\$V1.

Véase la Lección 5 para más detalles.

La función **scan** dispone de muchos parámetros, que podéis consultar en su Ayuda. Los más útiles en este momento son los siguientes:

- **sep**: sirve para indicar el signo usado para separar entradas consecutivas si no son espacios en blanco. Para ello se ha de igualar **sep** a este signo, entrecomillado. Por ejemplo, si vamos a entrar las entradas separadas por comas (o si están así en el fichero que vamos a importar), tenemos que especificar **sep=","**.

```
> x_scan2=scan()
1: 1,5,6,2,5,7,8,3,5
1:
Error in scan(file, what, nmax, sep, dec, quote, skip, nlines,
na.strings, : scan() expected 'a real', got '1,5,6,2,5,7,8,3,5'
> x_scan2=scan(sep=",")
1: 1,5,6,2,5,7,8,3,5
13:
Read 12 items
> x_scan2
[1] 1 5 6 2 5 7 8 3 5
```

- **dec**: sirve para indicar el separador decimal cuando no es un punto. Para ello hemos de igualar **dec** al separador decimal entre comillas. Por ejemplo, si queremos crear con **scan** un vector formado por los dos números reales 4,5 y 6,2 escritos exactamente de esta manera, tenemos que especificar **dec=","**.

```
> x_scan3=scan()
1: 4,5 6,2
Error in scan(file, what, nmax, sep, dec, quote, skip, nlines,
na.strings, : scan() expected 'a real', got '4,5'
> x_scan3=scan(dec=",")
1: 4,5 6,2
3:
```

```
Read 2 items
> x_scan3
[1] 4.5 6.2
```

- **what**: sirve para indicar a R de qué tipo tiene que considerar los datos que se le entren. En particular, **what="character"** especifica que los valores que se van a entrar en la consola o el fichero son palabras, aunque no estén entre comillas (si se entran entre comillas, no hace falta especificarlo).

```
> x_scan4=scan(sep=",")
1: Pep, Catalina, Joan, Pau
Error in scan(file, what, nmax, sep, dec, quote, skip, nlines,
na.strings, : scan() expected 'a real', got 'Pep'
> x_scan4=scan(what="character", sep=",")
1: Pep, Catalina, Joan, Pau
5:
Read 4 items
> x_scan4
[1] "Pep" "Catalina" "Joan" "Pau"
```

- **encoding**: sirve para indicar la codificación de alfabeto del fichero externo que se va a importar. Sólo es necesario especificarlo si dicho fichero contiene caracteres que no sean de 7 bits; o sea, letras acentuadas o caracteres especiales. En este caso, si su codificación no es la que espera nuestro ordenador y no la especificamos con este parámetro, estos caracteres se importarán mal. Los dos valores posibles que podemos darle son **"latin1"** y **"UTF-8"**. Por ejemplo, si sois usuarios de Windows, seguramente vuestro ordenador espere que el fichero a importar esté codificado en latin1; entonces, si está codificado en utf8 y contiene letras acentuadas, no las entenderá a no ser que especifiquéis **encoding="UTF-8"**. Los ficheros que usaremos en este curso estarán codificados en utf8, pero no contendrán letras acentuadas ni caracteres especiales, por lo que no será necesario usar este parámetro.

Veamos un ejemplo en sentido contrario: en <http://aprender.uib.es/Rdir/enlatin1.txt> hemos guardado algunos nombres con acentos y hemos codificado el fichero en latin1. En la sesión siguiente se puede ver cómo si lo importamos desde un ordenador Mac sin avisar de la codificación, los acentos se traducen mal.

```
> x_scan5=scan("http://aprender.uib.es/Rdir/enlatin1.txt", what="
character")
Read 3 items
> x_scan5
[1] "Juan" "Mar\xeda" "Jos\xe9"
> x_scan6=scan("http://aprender.uib.es/Rdir/enlatin1.txt", what="
character", encoding="latin1")
Read 3 items
> x_scan6
[1] "Juan" "María" "José"
```

Para definir un vector constante podemos usar la función **rep(a, n)**, que genera un vector que contiene el valor *a* repetido *n* veces.

```
> rep(1, 6)
[1] 1 1 1 1 1 1
> rep("Palma", 5) #Las palabras, siempre entre comillas
[1] "Palma" "Palma" "Palma" "Palma" "Palma"
```

La función `rep` también se puede usar para repetir vectores. Ahora bien, cuando decimos que queremos repetir cinco veces los valores 1, 2, 3, podemos referirnos a una de las dos construcciones siguientes:

1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3 o 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3.

Para especificar el tipo de repetición tenemos que usar el parámetro adecuado en el argumento de `rep`: si añadimos `times=5`, repetiremos el vector en bloque cinco veces (en el primer sentido), y si en cambio añadimos `each=5`, repetiremos cada valor cinco veces (en el segundo sentido).

```
> rep(c(1,2,3), times=5)
[1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
> rep(c(1,2,3), each=5)
[1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
```

Si queremos repetir cada elemento de un vector un número diferente de veces, podemos especificarlo igualando el parámetro `times` al vector de estas multiplicidades.

```
> rep(c(1,2,3,4), times=c(2,3,4,5))
[1] 1 1 2 2 2 3 3 3 3 4 4 4 4 4
```

Las progresiones aritméticas se pueden especificar de manera compacta usando la función `seq`. Una primera manera de hacerlo es mediante la instrucción `seq(a, b, by=p)`, que especifica la progresión aritmética de paso p que empieza en a ,

$$a, a + p, a + 2p, \dots,$$

hasta llegar a b . En concreto, si $a < b$ y $p > 0$, la función `seq(a, b, by=p)` genera un vector con la secuencia creciente $a, a + p, a + 2p, \dots$, hasta llegar al último valor de esta sucesión menor o igual que b .

```
> seq(3, 150, by=4.5)
[1] 3.0 7.5 12.0 16.5 21.0 25.5 30.0 34.5 39.0
[10] 43.5 48.0 52.5 57.0 61.5 66.0 70.5 75.0 79.5
[19] 84.0 88.5 93.0 97.5 102.0 106.5 111.0 115.5 120.0
[28] 124.5 129.0 133.5 138.0 142.5 147.0
```

Si $a > b$ y $p < 0$, entonces `seq(a, b, by=p)` genera un vector con la secuencia decreciente $a, a + p, a + 2p, \dots$, hasta parar en el último valor de esta sucesión *mayor* o igual que b .

```
> seq(80, 4, by=-3.5)
[1] 80.0 76.5 73.0 69.5 66.0 62.5 59.0 55.5 52.0 48.5 45.0
[12] 41.5 38.0 34.5 31.0 27.5 24.0 20.5 17.0 13.5 10.0 6.5
```

Si el signo de p no es el correcto, obtenemos un mensaje de error.

```
> seq(80, 4, by=3.5)
Error in seq.default(80, 4, by = 3.5) : wrong sign in 'by' argument
```

Como vimos en la lección anterior, la instrucción `seq` con paso ± 1 se puede abreviar con el signo «:». La instrucción `a:b` define la secuencia de números consecutivos entre dos números a y b , es decir, la secuencia $a, a+1, a+2, \dots$ hasta llegar a b (si $a < b$), o $a, a-1, a-2, \dots$ hasta llegar a b (si $a > b$).

```
> 1:15
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
> 2.3:12.5
[1] 2.3 3.3 4.3 5.3 6.3 7.3 8.3 9.3 10.3 11.3 12.3
> 34:-5
[1] 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
[20] 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 -1 -2 -3
[39] -4 -5
> -3:5 #Cuidado con los paréntesis
[1] -3 -2 -1 0 1 2 3 4 5
> -(3:5)
[1] -3 -4 -5
```

La función `seq` también se puede usar para definir progresiones aritméticas de otras dos maneras:

- `seq(a, b, length.out=n)` define la progresión aritmética de longitud n que va de a a b ; su paso es, por lo tanto, $p = (b - a)/(n - 1)$ si $n > 1$; si $n = 1$ sólo produce el valor a .
- `seq(a, by=p, length.out=n)` define la progresión aritmética

$$a, a + p, a + 2p, \dots, a + (n - 1)p$$

de longitud n y paso p que empieza en a .

```
> seq(2, 10, length.out=10)
[1] 2.000000 2.888889 3.777778 4.666667 5.555556
[6] 6.444444 7.333333 8.222222 9.111111 10.000000
> seq(2, by=0.5, length.out=10)
[1] 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5
```

A estas alturas habréis observado que cuando el resultado de una instrucción es un vector, R comienza cada línea del resultado con un número entre corchetes `[]`. Este número indica la posición dentro del vector de la primera entrada de la línea correspondiente. De esta manera, en el resultado de `seq(2, 10, length.out=10)`, R nos indica que 2.000000 es el primer elemento de este vector y 6.444444 su sexto elemento.

La función `c` que hemos usado para crear vectores en realidad concatena sus argumentos en un vector (de ahí viene la `c`). Si la aplicamos a vectores, crea un nuevo vector concatenando sus elementos. Podemos mezclar vectores y datos en su argumento.


```

> x=c(rep(1, 10), 2:10)
> x
[1] 1 1 1 1 1 1 1 1 1 1 2 3 4 5 6 7 8 9 10
> x=c(0,x,20,30)
> x
[1] 0 1 1 1 1 1 1 1 1 1 1 2 3 4 5 6 7 8 9
[20] 10 20 30

```

Esta última construcción, `x=c(0,x,20,30)`, muestra que la función `c` se puede usar para añadir valores al principio o al final de un vector sin cambiarle el nombre: en este caso, hemos redefinido `x` añadiéndole un 0 al principio y 20, 30 al final.

Un vector se puede modificar fácilmente usando el editor de datos que incorpora *RStudio*. Para hacerlo, se aplica la función `fix` al vector que queremos editar. *R* abre entonces el vector en una nueva ventana de edición. Mientras esta ventana esté abierta, será la ventana activa de *R* y no podremos volver a nuestra sesión de *R* hasta que la cerremos. Los cambios que hagamos en el vector con el editor de datos se guardarán cuando cerremos esta ventana.

Probadlo. Cread un vector con *R* y abridlo en el editor. Por ejemplo:

```

> x=c(rep(1, 10), 2:10)
> fix(x)

```

Se abrirá entonces una ventana como la que mostramos en la Figura 3.1. Ahora, en esta ventana, podéis añadir, borrar y cambiar los datos que queráis. Por ejemplo, añadid un 0 al principio y 20, 30 al final y guardad el resultado (pulsando «Save» en la ventana del editor). El valor de `x` se habrá modificado, como podréis comprobar entrando `x` en la consola.

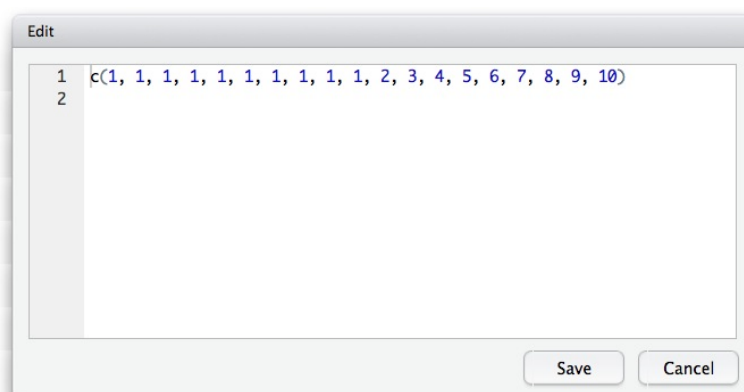


Figura 3.1. Ventana del editor de vectores de *RStudio* para Mac OS X.

3.2. Operaciones con vectores

El manejo de vectores con *R* tiene una propiedad muy útil: podemos aplicar una función a todos los elementos de un vector en un solo paso.

```

> x=seq(2, 30, by=3)
> x

```

```

[1] 2 5 8 11 14 17 20 23 26 29
> x+2.5
[1] 4.5 7.5 10.5 13.5 16.5 19.5 22.5 25.5 28.5 31.5
> 2.5*x
[1] 5.0 12.5 20.0 27.5 35.0 42.5 50.0 57.5 65.0 72.5
> sqrt(x)
[1] 1.414214 2.236068 2.828427 3.316625 3.741657 4.123106 4.472136
[8] 4.795832 5.099020 5.385165
> 2^x
[1] 4 32 256 2048 16384 131072
[7] 1048576 8388608 67108864 536870912
> x^2
[1] 4 25 64 121 196 289 400 529 676 841
> (1:4)^2
[1] 1 4 9 16
> 1:4^2 #Cuidado con los paréntesis
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

```

A veces no es posible aplicar una función concreta a todo un vector entrándolo dentro del argumento de la función, como hemos hecho en los ejemplos anteriores. En estos casos, podemos usar la instrucción

`sapply(vector, FUN=función).`

Por ejemplo, dentro de un rato veremos que la función `mean` calcula la media aritmética de un vector. Supongamos que definimos una función `F` que, aplicada a un número natural x , calcula la media de los números

$$1, 2, \dots, x.$$

```

> F=function(x){mean(1:x)}
> F(20)
[1] 10.5
> F(30)
[1] 15.5

```

Resulta que no podemos aplicar esta función a todas las entradas de un vector x entrando simplemente `F(x)`.

```

> F(20:30)
[1] 10.5
Warning message:
In 1:x : numerical expression has 11 elements: only the first used

```

En casos como este, siempre podemos recurrir a la función `sapply`.

```

> sapply(20:30, FUN=F)
[1] 10.5 11.0 11.5 12.0 12.5 13.0 13.5 14.0 14.5 15.0 15.5

```

También podemos operar término a término las entradas de dos vectores de la misma longitud.

```

> 1:5+1:5 #Suma entrada a entrada
[1] 2 4 6 8 10

```

```
> (1:5)*(1:5) #Producto entrada a entrada
[1] 1 4 9 16 25
> (1:5)^(1:5) #Potencia entrada a entrada
[1] 1 4 27 256 3125
```

Esto nos permite calcular fácilmente vectores de la forma $(x_n)_{n=p,\dots,q}$, formados por los términos x_p, x_{p+1}, \dots, x_q de una sucesión $(x_n)_n$, a partir de la fórmula explícita de x_n como función del índice n : basta aplicar esta fórmula a $p:q$. Por ejemplo, para definir el vector

$$x = (3 \cdot 2^n - 20)_{n=1,\dots,20},$$

podemos entrar lo siguiente:

```
> n=1:20 #Secuencia 1,...,20, y la llamamos n por comodidad
> x=3*2^n-20 #Aplicamos la fórmula a n=1,...,20
> x
[1] -14 -8 4 28 76 172 364
[8] 748 1516 3052 6124 12268 24556 49132
[15] 98284 196588 393196 786412 1572844 3145708
```

De manera similar, para definir el vector

$$y = \left(\frac{n}{n^2 + 1} \right)_{n=0,\dots,20},$$

podemos usar lo siguiente:

```
> n=0:20
> y=n/(n^2+1)
> y
[1] 0.00000000 0.50000000 0.40000000 0.30000000 0.23529412
[6] 0.19230769 0.16216216 0.14000000 0.12307692 0.10975610
[11] 0.09900990 0.09016393 0.08275862 0.07647059 0.07106599
[16] 0.06637168 0.06225681 0.05862069 0.05538462 0.05248619
[21] 0.04987531
```

En ambos casos, y para facilitar la visualización de la construcción, hemos creado el vector n con los índices de los términos de la sucesión, y después hemos obtenido el trozo de sucesión deseado aplicando la función que la define a n . También habríamos podido generar estos vectores escribiendo directamente la sucesión de índices en la fórmula que los define. Por ejemplo:

```
> (0:20)/((0:20)^2+1)
[1] 0.00000000 0.50000000 0.40000000 0.30000000 0.23529412
[6] 0.19230769 0.16216216 0.14000000 0.12307692 0.10975610
[11] 0.09900990 0.09016393 0.08275862 0.07647059 0.07106599
[16] 0.06637168 0.06225681 0.05862069 0.05538462 0.05248619
[21] 0.04987531
```

R dispone de muchas funciones para aplicar a vectores, relacionadas principalmente con la estadística. Veamos algunas que nos pueden ser útiles por el momento, y ya iremos viendo otras a medida que avance el curso:

- `length` calcula la longitud del vector.

- `max` y `min` calculan sus valores máximo y mínimo, respectivamente.
- `sum` calcula la suma de sus entradas.
- `prod` calcula el producto de sus entradas.
- `mean` calcula la media aritmética de sus entradas.
- `diff` calcula el vector formado por las diferencias sucesivas entre entradas del vector original.
- `cumsum` calcula el vector formado por las *sumas acumuladas* de las entradas del vector original: cada entrada de `cumsum(x)` es la suma de las entradas de `x` hasta su posición.
- `sort` ordena los elementos del vector en el orden natural creciente del tipo de datos que lo forman: el orden numérico, el orden alfabético, etc. Si lo queremos ordenar en orden decreciente, podemos incluir en su argumento el parámetro `dec=TRUE`.
- `rev` invierte el orden de los elementos del vector; por lo tanto, `rev(sort(...))` es otra opción para ordenar en orden decreciente.

```
> x=c(1,5,6,2,5,7,8,3,5,2,1,0)
> length(x)
[1] 12
> max(x)
[1] 8
> min(x)
[1] 0
> sum(x)
[1] 45
> prod(x)
[1] 0
> mean(x)
[1] 3.75
> cumsum(x)
[1] 1 6 12 14 19 26 34 37 42 44 45 45
> diff(x)
[1] 4 1 -4 3 2 1 -5 2 -3 -1 -1
> sort(x)
[1] 0 1 1 2 2 3 5 5 5 6 7 8
> sort(x, dec=TRUE)
[1] 8 7 6 5 5 5 3 2 2 1 1 0
> rev(x)
[1] 0 1 2 5 3 8 7 5 2 6 5 1
```

La función `sum` es útil para evaluar sumatorios; por ejemplo, si queremos calcular

$$\sum_{n=0}^{200} \frac{1}{n^2 + 1},$$

sólo tenemos que entrar:

```
> n=0:200
> sum(1/(n^2+1))
[1] 2.071687
```

La función `cumsum` permite definir sucesiones descritas mediante sumatorios; a modo de ejemplo, para definir el vector

$$y = \left(\sum_{i=0}^n 2^{-i} \right)_{n=0,\dots,20},$$

basta aplicar `cumsum` al vector $x = (2^{-i})_{i=0,\dots,20}$ de la manera siguiente:

```
> i=0:20
> x=2^(-i)
> y=cumsum(x)
> y
[1] 1.000000 1.500000 1.750000 1.875000 1.937500 1.968750 1.984375
[8] 1.992188 1.996094 1.998047 1.999023 1.999512 1.999756 1.999878
[15] 1.999939 1.999969 1.999985 1.999992 1.999996 1.999998 1.999999
```

Observamos que esta sucesión tiende a 2, lo que corresponde a la igualdad

$$\sum_{k=0}^{\infty} \frac{1}{2^k} = \frac{1}{1 - \frac{1}{2}} = 2.$$

3.3. Entradas y trozos de vectores

Si queremos extraer el valor de una entrada concreta de un vector, o si queremos referirnos a esta entrada para usarla en un cálculo, podemos emplear la construcción

$$vector[i],$$

que indica la i -ésima entrada del *vector*. En particular, `vector[length(vector)-i]` es la $(i+1)$ -ésima entrada del *vector* empezando por el final: su última entrada es `vector[length(vector)]`, la penúltima es `vector[length(vector)-1]` y así sucesivamente.

Observad que para referirnos a elementos de un vector, empleamos corchetes `[]`, y no los paréntesis redondos usuales.

```
> x=seq(2, 50, by=1.5)
> x
[1] 2.0 3.5 5.0 6.5 8.0 9.5 11.0 12.5 14.0 15.5 17.0
[12] 18.5 20.0 21.5 23.0 24.5 26.0 27.5 29.0 30.5 32.0 33.5
[23] 35.0 36.5 38.0 39.5 41.0 42.5 44.0 45.5 47.0 48.5 50.0
> x(3) #La tercera entrada del vector?
Error: could not find function "x"
> x[3] #La tercera entrada del vector, ahora sí
[1] 5
> x[length(x)] #La última entrada del vector
[1] 50
> x[length(x)-5] #La sexta entrada del vector empezando por el
final
```

```
[1] 42.5
```

También podemos extraer subvectores de un vector. Una primera manera de obtener un subvector es especificando los índices de las entradas que lo han de formar:

- `vector[y]`, donde `y` es un vector (de índices), crea un nuevo vector con las entradas del `vector` original cuyos índices pertenecen a `y`.
- En particular, si `a` y `b` son dos números naturales, `vector[a:b]` crea un nuevo vector con las entradas del `vector` original que van de la `a`-ésima a la `b`-ésima.
- `vector[-y]`, donde `y` es un vector (de índices), es el complementario de `vector[y]`: sus entradas son las del `vector` original cuyos índices *no* pertenecen a `y`.
- En particular, `vector[-i]` borra la entrada `i`-ésima del `vector` original.

Veamos algunos ejemplos:

```
> n=1:10
> x=2*3^n-5*n^3*2^n
> x
[1] -4 -142 -1026 -4958 -19514 -67662
[7] -215146 -642238 -1826874 -5001902
> x[-3] #x sin la tercera entrada
[1] -4 -142 -4958 -19514 -67662 -215146 -642238
[8] -1826874 -5001902
> x[3:7] #Los elementos tercero a séptimo de x
[1] -1026 -4958 -19514 -67662 -215146
> x[7:3] #Los elementos séptimo a tercero de x
[1] -215146 -67662 -19514 -4958 -1026
> x[seq(1, length(x), by=2)] #Los elementos de índice impar de x
[1] -4 -1026 -19514 -215146 -1826874
> x[seq(2, length(x), by=2)] #Los elementos de índice par
[1] -142 -4958 -67662 -642238 -5001902
> x[-seq(1, length(x), by=2)] #Si borramos los elementos de índice
impar, quedan los de índice par
[1] -142 -4958 -67662 -642238 -5001902
> x[(length(x)-5):length(x)] #Los últimos 6 elementos de x
[1] -19514 -67662 -215146 -642238 -1826874 -5001902
> x[length(x)-5:length(x)] #No os dejéis los paréntesis ...
[1] -19514 -4958 -1026 -142 -4
```

Fijaos en las dos últimas instrucciones: si $\ell = \text{length}(x)$, $(\text{length}(x)-5):\text{length}(x)$ es la secuencia de índices

$$\ell - 5, \ell - 4, \ell - 3, \ell - 2, \ell - 1, \ell;$$

en cambio, $\text{length}(x)-5:\text{length}(x)$ es la secuencia

$$\ell - (5, 6, 7, \dots, \ell) = \ell - 5, \ell - 6, \ell - 7, \dots, 1, 0.$$

También podemos extraer las entradas de un vector (o sus índices) que satisfagan alguna condición. Los operadores lógicos que podemos usar para definir estas condiciones son los que damos en la Tabla 3.1. Veamos algunos ejemplos (y observad su sintaxis):

Operador	=	≠	<	>	≤	≥	negación	conjunción	disjunción
Signo	==	!=	<	>	<=	>=	!	&	

Tabla 3.1. Signos de operadores lógicos

```
> x=c(1,5,6,2,5,7,8,3,5,2,1,0)
> x[x>3] #Elementos mayores que 3
[1] 5 6 5 7 8 5
> x[x>2 & x<=5] #Elementos mayores que 2 y menores o iguales que 5
[1] 5 5 3 5
> x[x!=2 & x!=5] #Elementos diferentes de 2 y de 5
[1] 1 6 7 8 3 1 0
> x[x>5 | x<=2] #Elementos mayores que 5 o menores o iguales que 2
[1] 1 6 2 7 8 2 1 0
> x[x>=4] #Elementos mayores o iguales que 4
[1] 5 6 5 7 8 5
> x[!x<4] #Elementos que NO son menores que 4; es equivalente a la
          anterior
[1] 5 6 5 7 8 5
> x[x%%4==0] #Elementos múltiplos de 4
[1] 8 0
```

Analicemos la segunda instrucción, `x[x>3]`. La construcción `x>3` define un vector que, en cada posición, contiene un `TRUE` si el elemento correspondiente del vector `x` es mayor que 3 y un `FALSE` si no lo es.

```
> x>3
[1] FALSE TRUE TRUE FALSE TRUE TRUE TRUE FALSE TRUE
[10] FALSE FALSE FALSE
```

Entonces `x[x>3]` lo que nos da son las entradas del vector `x` correspondientes a los `TRUE` de este vector de valores lógicos.

```
> x[x>3]
[1] 5 6 5 7 8 5
```

Esta construcción también permite extraer las entradas de un vector cuyos índices sean los de las entradas de otro vector que satisfagan una condición lógica. Por ejemplo:

```
> x=c(1,5,6,2,5,7,8,3,5,2,1,0)
> y=c(2,-3,0,1,2,-1,4,-1,-2,3,5,1)
> x[y>0] #Entradas de x correspondientes a entradas positivas de y
[1] 1 2 5 8 2 1 0
```

Para obtener los *índices* de las entradas del vector que satisfacen una condición dada, podemos usar la función `which`. Esta función, aplicada a un vector de valores lógicos, da los índices de las posiciones de los `TRUE`. Así, para saber los índices de las entradas de `x` que son mayores que 3, usamos `which(x>3)`, que nos dará los índices de las entradas `TRUE` del vector `x>3`.

```
> x=c(1,5,6,2,5,7,8,3,5,2,1)
```

```

> x
[1] 1 5 6 2 5 7 8 3 5 2 1
> x[x>3] #Elementos mayores que 3
[1] 5 6 5 7 8 5
> which(x>3) #Índices de los elementos mayores que 3
[1] 2 3 5 6 7 9
> which(x>2 & x<=5) #Índices de los elementos > 2 y <= 5
[1] 2 5 8 9
> which(x!=2 & x!=5) #Índices de los elementos diferentes de 2 y 5
[1] 1 3 6 7 8 11
> which(x>5 | x<=2) #Índices de los elementos > 5 o <= 2
[1] 1 3 4 6 7 10 11
> which(x%%2==0) #Índices de los elementos pares del vector
[1] 3 4 7 10

```

La instrucción `which.min(x)` nos da la primera posición en la que el vector toma su valor mínimo; `which.max(x)` hace lo mismo, pero para el máximo. En cambio, con `which(x==min(x))` obtenemos todas las posiciones en las que el vector toma su valor mínimo y, con `which(x==max(x))`, aquellas en las que toma su valor máximo.

```

> x
[1] 1 5 6 2 5 7 8 3 5 2 1
> which.min(x)
[1] 1
> which(x==min(x))
[1] 1 11

```

Si un vector no contiene ningún término que satisfaga la condición que imponemos, obtenemos como respuesta un vector vacío. R lo indica con `numeric(0)` si es de números, `character(0)` si es de palabras, o `integer(0)` si es de índices de entradas de un vector. Estos vectores vacíos tienen longitud, naturalmente, 0.

```

> x=2^(0:10)
> x
[1] 1 2 4 8 16 32 64 128 256 512 1024
> x[20<x & x<30] #Elementos de x estrictamente entre 20 y 30
numeric(0)
> length(x[20<x & x<30]) #¿Cuántas entradas hay entre 20 y 30?
[1] 0
> which(x>1500) #Índices de elementos mayores que 1500
integer(0)

```

Si R no sabe de qué tipo son los datos que faltan en un vector vacío, lo indica con `NULL`. También podemos usar este valor para definir un vector vacío.

```

> x=c()
> x
NULL
> z=NULL
> z
NULL

```



```
> y=c(x, 2, z)
> y
[1] 2
```

Los operadores lógicos que hemos explicado también se pueden usar para pedir si una condición sobre unos números concretos se satisface o no. Por ejemplo:

```
> exp(pi)>pi^(exp(1)) #¿Es mayor e^pi que pi^e?
[1] TRUE
> 1234567%%9==0 #¿Es 1234567 múltiplo de 9?
[1] FALSE
```

Podemos modificar algunas entradas de un vector simplemente declarando sus nuevos valores. Esto se puede hacer entrada a entrada, o para todo un subvector de golpe.

```
> x=1:10
> x
[1] 1 2 3 4 5 6 7 8 9 10
> x[3]=15 #En la posición 3 escribimos 15
> x[11]=25 #Añadimos en la posición 11 un 25
> x
[1] 1 2 15 4 5 6 7 8 9 10 25
> x[c(2, 3, 4)]=x[c(2, 3, 4)]+10 #Sumamos 10 a las entradas en las
    posiciones 2, 3 y 4
> x
[1] 1 12 25 14 5 6 7 8 9 10 25
> x[(length(x)-2):length(x)]=0 #Igualamos las últimas tres
    entradas a 0
> x
[1] 1 12 25 14 5 6 7 8 0 0 0
> x[length(x)+3]=2
> x
[1] 1 12 25 14 5 6 7 8 0 0 0 NA NA 2
```

Fijaos en la última instrucción. Hemos añadido al vector `x` el valor 2 tres posiciones más allá de su última entrada. Entonces, en las posiciones 12 y 13 ha escrito `NA` antes de añadir en la 14 el 2. Estos `NA`, de *Not Available*, indican que las entradas correspondientes del vector no existen.

Los `NA` serán muy importantes cuando usemos vectores en estadística descriptiva, donde podrán representar valores desconocidos, errores, etc. Serán importantes porque son molestos, puesto que, por norma general, una función aplicada a un vector que contenga algún `NA` da `NA`.

```
> sum(x)
[1] NA
> mean(x)
[1] NA
```

Afortunadamente, muchas de las funciones para vectores admiten un parámetro `na.rm` que, igualado a `TRUE`, hace que la función sólo tenga en cuenta las entradas definidas.

```
> sum(x, na.rm=TRUE)
[1] 80
```

```
> mean(x, na.rm=TRUE)
[1] 6.666667
```

Para extraer las entradas no definidas de un vector x no podemos usar la condición lógica $x==NA$, sino la función `is.na(x)`.

```
> x
[1] 1 12 25 14 5 6 7 8 0 0 0 NA NA 2
> which(x==NA) #¿Índices de entradas NA?
integer(0)
> is.na(x)
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[10] FALSE FALSE TRUE TRUE FALSE
> which(is.na(x)) #Índices de entradas NA
[1] 12 13
> y=x #Creamos una copia de x y la llamamos y
> y[is.na(y)]=mean(y, na.rm=TRUE) #Cambiamos los NA de y por la
  media del resto de entradas
> y
[1] 1.000000 12.000000 25.000000 14.000000 5.000000
[6] 6.000000 7.000000 8.000000 0.000000 0.000000
[11] 0.000000 6.666667 6.666667 2.000000
```

Naturalmente, podemos usar la negación de `is.na(x)` para obtener las entradas definidas de un vector x : formarán el vector `x[!is.na(x)]`.

```
> x
[1] 1 12 25 14 5 6 7 8 0 0 0 NA NA 2
> x[!is.na(x)]
[1] 1 12 25 14 5 6 7 8 0 0 0 2
> sum(x[!is.na(x)])
[1] 80
> cumsum(x)
[1] 1 13 38 52 57 63 70 78 78 78 78 NA NA NA
> cumsum(x, na.rm=TRUE) #cumsum no admite na.rm
Error in cumsum(x, na.rm = TRUE) :
  2 arguments passed to 'cumsum' which requires 1
> cumsum(x[!is.na(x)])
[1] 1 13 38 52 57 63 70 78 78 78 78 80
```

Las entradas no definidas de un vector también se pueden borrar aplicándole la función `na.omit`.

```
> na.omit(x)
[1] 1 12 25 14 5 6 7 8 0 0 0 2
attr(,"na.action")
[1] 12 13
attr(,"class")
[1] "omit"
> sum(na.omit(x))
[1] 80
> cumsum(na.omit(x))
```

```
[1] 1 13 38 52 57 63 70 78 78 78 80
```

Observad el resultado de `na.omit(x)`. Contiene un primer vector formado por las entradas del vector original que no son NA, y luego una serie de información extra llamados *atributos*, e indicados por R con `attr`: los índices de las entradas que ha eliminado y el tipo de acción que ha llevado a cabo. Como podéis ver, estos atributos no interfieren para nada en las operaciones que se realicen con el primer vector, pero si os molestan, se pueden eliminar: la instrucción

```
attr(objeto, atributo)=NULL
```

borra el *atributo* del *objeto*.

```
> x_sinNA=na.omit(x)
> x_sinNA
[1] 1 12 25 14 5 6 7 8 0 0 0 2
attr(,"na.action")
[1] 12 13
attr(,"class")
[1] "omit"
> attr(x_sinNA, "na.action")=NULL
> attr(x_sinNA, "class")=NULL
> x_sinNA
[1] 1 12 25 14 5 6 7 8 0 0 0 2
```

3.4. Factores

Un *factor* es como un vector, pero con una estructura interna más rica que permite usarlo para clasificar observaciones. Para ilustrar la diferencia entre vectores y factores, vamos a crear un vector `Ciudades` con los nombres de algunas ciudades, y a continuación un factor `Ciudades.factor` con el mismo contenido, aplicando a este vector la función `factor`.

```
> Ciudades=c("Madrid","Palma","Madrid","Madrid","Barcelona",
  "Palma","Madrid","Madrid")
> Ciudades
[1] "Madrid"      "Palma"      "Madrid"      "Madrid"      "Barcelona"
[6] "Palma"      "Madrid"      "Madrid"
> Ciudades.factor=factor(Ciudades)
> Ciudades.factor
[1] Madrid      Palma      Madrid      Madrid      Barcelona Palma
[7] Madrid      Madrid
Levels: Barcelona Madrid Palma
```

Observad la diferencia. El factor dispone de un atributo especial llamado *niveles* (*levels*), y cada elemento del factor es igual a un nivel; de esta manera, los niveles «clasifican» las entradas del factor. Podríamos decir, en resumen, que un factor es una lista formada por copias de etiquetas (los niveles), como podrían ser el sexo o la especie de unos individuos.

Cuando tengamos un vector que queramos usar para clasificar datos, conviene definirlo como un factor y así podremos hacer más cosas con él. Para crear un factor, hemos de definir un vector y transformarlo en factor por medio de una de las funciones `factor` o `as.factor`. La diferencia entre estas funciones es que `as.factor` «convierte» el vector en un factor, y toma

como sus niveles los diferentes valores que aparecen en el vector, mientras que `factor` «define» un factor a partir del vector, y dispone de algunos parámetros que permiten modificar el factor que se crea, tales como:

- `levels`, que permite especificar los niveles e incluso añadir niveles que no aparecen en el vector.
- `labels`, que permite cambiar los nombres de los niveles.

De esta manera, con `as.factor` o con `factor` sin especificar `levels`, el factor tendrá como niveles los diferentes valores que toman las entradas del vector, y además aparecerán en su lista de niveles, `Levels`, ordenados en orden alfabético. Si especificamos el parámetro `levels` en la función `factor`, los niveles aparecerán en dicha lista en el orden en el que los entremos en él.

```
> S=c("M","M","F","M","F","F","F","M","M","F")
> Sex=as.factor(S)
> Sex
[1] M M F M F F F M M F
Levels: F M
> Sex2=factor(S) #Esto definirá el mismo factor
> Sex2
[1] M M F M F F F M M F
Levels: F M
> Sex3=factor(S, levels=c("M","F","B")) #Si queremos añadir un
nuevo nivel "B" y cambiar el orden de los niveles
> Sex3
[1] M M F M F F F M M F
Levels: M F B
> Sex4=factor(S, levels=c("M","F","B"),
labels=c("Masc","Fem","Bisex")) #Si queremos cambiar los nombres
> Sex4
[1] Masc Masc Fem Masc Fem Fem Fem Masc Masc Fem
Levels: Masc Fem Bisex
```

Para obtener los niveles de un factor, podemos emplear la función `levels`.

```
> levels(Sex)
[1] "F" "M"
> levels(Sex4)
[1] "Masc" "Fem" "Bisex"
```

Esta función `levels` también permite cambiar los nombres de los niveles de un factor.

```
> Notas=as.factor(c(1,2,2,3,1,3,2,4,2,3,4,2))
> Notas
[1] 1 2 2 3 1 3 2 4 2 3 4 2
Levels: 1 2 3 4
> levels(Notas)=c("Muy.mal","Mal","Bien","Muy.bien")
> Notas
[1] Muy.mal Mal Mal Bien Muy.mal Bien Mal
[8] Muy.bien Mal Bien Muy.bien Mal
```

```
Levels: Muy.mal Mal Bien Muy.bien
```

Observad que los niveles han heredado el orden del factor original.

Con la función `levels` también podemos agrupar varios niveles de un factor en uno solo, simplemente repitiendo nombres al especificarlos; por ejemplo, en el factor de notas anterior, vamos a agrupar los niveles «Muy mal» y «Mal» en uno solo, y lo mismo con los niveles «Muy bien» y «Bien»:

```
> Notas_2niv=Notas
> levels(Notas_2niv)=c("Mal","Mal","Bien","Bien")
> Notas_2niv
[1] Mal Mal Mal Bien Mal Bien Mal Bien Mal Bien Bien Mal
Levels: Mal Bien
```

Nos hemos referido varias veces al orden de los niveles. En realidad, hay dos tipos de factores: simples y ordenados. Hasta ahora sólo hemos considerado los *factores simples*, en los que el orden de los niveles realmente no importa, y si lo modificamos es sólo por razones estéticas o de comprensión de los datos; en este caso, la manera más sencilla de hacerlo es redefiniendo el factor con `factor` y modificando en el parámetro `levels` el orden de los niveles. Pero si el orden de los niveles es relevante para analizar los datos, entonces es conveniente definir el factor como *ordenado*. Esto se lleva a cabo con la función `ordered`, que dispone de los mismos parámetros que `factor`. Así, si queremos que nuestro factor `Notas` sea un factor ordenado, con sus niveles ordenados de «Muy mal» a «Muy bien», hay que entrar lo siguiente:

```
> Notas=ordered(Notas, levels=c("Muy.mal","Mal","Bien","Muy.bien"))
> Notas
[1] Muy.mal Mal Mal Bien Muy.mal Bien Mal
[8] Muy.bien Mal Bien Muy.bien Mal
Levels: Muy.mal < Mal < Bien < Muy.bien
```

Observad que R indica el orden de los niveles de un factor ordenado mediante el signo `<`.

Aunque en la instrucción anterior hemos aplicado la función `ordered` a un factor, también se puede aplicar a un vector, como si usáramos `factor`.

3.5. Listas heterogéneas

Los vectores que hemos estudiado hasta el momento sólo pueden contener datos, y estos datos han de ser de un solo tipo. Por ejemplo, no podemos construir un vector que contenga simultáneamente palabras y números, o cuyas entradas sean a su vez vectores. Este problema se resuelve con las *listas heterogéneas*; para abreviar, las llamaremos por su nombre en R: `list`. Una `list` es una lista formada por objetos que pueden ser de clases diferentes. Así, en una misma `list` podemos combinar números, palabras, vectores, otras `list`, etc. En la Lección 2 ya aparecieron dos objetos de clase `list`: los resultados de `lm(...)` y `summary(lm(...))`.

Supongamos por ejemplo que queremos guardar en una lista un vector, su nombre, su media, y su vector de sumas acumuladas. En este caso, tendríamos que hacerlo en forma de lista heterogénea usando la función `list`.

```
> x=c(1,2,-3,-4,5,6)
> L=list(nombre="x",vector=x,media=mean(x),sumas=cumsum(x))
```

```

> L
$nombre
[1] "x"

$vector
[1] 1 2 -3 -4 5 6

$media
[1] 1.166667

$sumas
[1] 1 3 0 -4 1 7

```

Observad la sintaxis de la función `list`: le hemos entrado como argumento los diferentes objetos que van a formar la lista heterogénea, poniendo a cada uno un nombre adecuado. Este nombre es «interno» de la `list`: por ejemplo, pese a que dentro de la lista hemos definido un objeto llamado `sumas`, en el entorno de trabajo de **R** no tenemos definida ninguna variable con ese nombre (a no ser que la hayamos definido previamente durante la sesión).

```

> sumas
Error: object 'sumas' not found

```

Para obtener una componente concreta de una `list`, tenemos que añadir al nombre de la `list` el sufijo formado por un signo `$` y el nombre de la componente; recordad cómo extraíamos el valor de R^2 de un `summary(lm(...))` o (en el contexto de los *data frames*) cómo extraíamos una columna de un *data frame* en la página 3-3.

```

> L$nombre
[1] "x"
> L$vector
[1] 1 2 -3 -4 5 6
> L$media
[1] 1.166667

```

También podemos indicar el objeto por su posición en la `list` usando un par de dobles corchetes `[[]]`. Si usamos sólo un par de corchetes, como en los vectores, lo que obtenemos es una `list` formada por esa única componente, no el objeto que forma la componente.

```

> L[[1]]
[1] "x"
> L[[4]] #Esto es un vector
[1] 1 3 0 -4 1 7
> 3*L[[4]] #Y podemos operar con él
[1] 3 9 0 -12 3 21
> L[4] #Esto es una list, no un vector
$sumas
[1] 1 3 0 -4 1 7
> 3*L[4] #Y NO podemos operar con él
Error in 3 * L[4] : non-numeric argument to binary operator

```

Para conocer la estructura interna de una `list`, es decir, los nombres de los objetos que la forman y su naturaleza, podemos usar la función `str`. Si sólo queremos saber sus nombres, podemos usar la función `names`. Si la `list` se obtiene con una función de R cuyo resultado sea una estructura de este tipo, como, por ejemplo, `lm`, es recomendable consultar la Ayuda de la función, ya que probablemente explique el significado de los objetos que la forman.

```
> str(L)
List of 4
 $ nombre: chr "x"
 $ vector: num [1:6] 1 2 -3 -4 5 6
 $ media : num 1.17
 $ sumas : num [1:6] 1 3 0 -4 1 7
> names(L)
[1] "nombre" "vector" "media"  "sumas"
```

3.6. Guía rápida de funciones

- `c` sirve para definir un vector concatenando elementos o vectores.
- `scan` crea un vector importando datos que se entren en la consola o contenidos en un fichero. Algunos parámetros importantes:
 - `dec`: indica el separador decimal.
 - `sep`: indica el signo usado para separar las entradas.
 - `what`: indica el tipo de datos que se importan.
 - `encoding`: indica la codificación de alfabeto del fichero externo; sus dos valores posibles son `"latin1"` y `"UTF-8"`.
- `rep` sirve para definir un vector repitiendo un valor o las entradas de otro vector. Algunos parámetros importantes:
 - `each`: cuando aplicamos la función a un vector, sirve para indicar cuántas veces queremos repetir cada entrada del vector.
 - `times`: cuando aplicamos la función a un vector, sirve para indicar cuántas veces queremos repetir todo el vector en bloque.
- `seq` se puede usar para definir progresiones aritméticas. Algunos parámetros importantes:
 - `by`: el paso de la progresión.
 - `length.out`: la longitud de la progresión.

Estos parámetros se pueden usar de las maneras siguientes:

- `seq(a, b, by=p)` define la progresión

$$a, a + p, a + 2p, \dots, b$$

(o parándose en el término inmediatamente anterior a b , si b no pertenece a la progresión).

- `seq(a, b, length.out=n)` define la progresión

$$a, a + p, a + 2p, \dots, b$$

tomando como paso $p = (b - a)/(n - 1)$.

- `seq(a, by=p, length.out=n)` define la progresión

$$a, a + p, a + 2p, \dots, a + (n - 1)p.$$

- `a:b` es sinónimo de `seq(a, b, by=1)` (si $a < b$) o `seq(a, b, by=-1)` (si $a > b$).
- `NULL` indica un vector vacío.
- `fix` abre un vector (o, en general, un objeto de datos: una matriz, un *data frame*...) en el editor de datos.
- Funciones para vectores:

Función	longitud	máximo	mínimo	suma	producto
Signo	<code>length</code>	<code>max</code>	<code>min</code>	<code>sum</code>	<code>prod</code>
Función	media	sumas acumuladas	diferencias	ordenar	invertir el orden
Signo	<code>mean</code>	<code>cumsum</code>	<code>diff</code>	<code>sort</code>	<code>rev</code>

Las funciones `max`, `min`, `sum`, `prod` y `mean` admiten el parámetro siguiente:

- `na.rm`: igualado a `TRUE`, impone que no se tengan en cuenta los valores `NA` del vector al calcularla.
- `sapply(vector, FUN=función)` aplica la *función* a todas las entradas del *vector*.
- `vector[...]` se usa para especificar un elemento o un subvector del *vector*. Las entradas que formarán el subvector pueden especificarse mediante el vector de sus índices o mediante una condición lógica sobre las entradas. Los signos de operadores lógicos que se pueden usar para definir condiciones lógicas son los siguientes:

Operador	<code>=</code>	<code>≠</code>	<code><</code>	<code>></code>	<code>≤</code>	<code>≥</code>	negación	conjunción	disjunción
Signo	<code>==</code>	<code>!=</code>	<code><</code>	<code>></code>	<code><=</code>	<code>>=</code>	<code>!</code>	<code>&</code>	<code> </code>

- `which` sirve para obtener los índices de las entradas de un vector que satisfacen una condición lógica.
- `which.min` y `which.max` dan la primera posición en la que el vector toma su valor mínimo o máximo, respectivamente.
- `is.na` es la alternativa correcta a la condición `==NA`.
- `na.omit` elimina las entradas `NA` de un vector.

- `as.factor` transforma un vector en un factor.
- `factor` crea un factor a partir de un vector. Algunos parámetros importantes:
 - `levels`: sirve para especificar los niveles.
 - `labels`: sirve para cambiar los nombres de los niveles.
- `ordered` crea un factor ordenado a partir de un vector o un factor; sus parámetros son los mismos que los de `factor`.
- `levels` sirve para obtener los niveles de un factor, y también para cambiar sus nombres.
- `list` construye listas heterogéneas.
- `str` sirve para obtener la estructura de una `list`.
- `names` sirve para conocer los nombres de las componentes de una `list`.
- `list$componente` sirve para referirnos al objeto que forma la *componente* de la `list`.
- `list[[i]]` sirve para referirnos al objeto que forma la *i*-ésima componente de la `list`.

3.7. Ejercicio

Tenemos las siguientes notas obtenidas por unos estudiantes en un examen:

7.9, 4.3, 5.5, 7.9, 9.8, 2.7, 4.7, 2.4, 8.3, 7.3, 6.8, 6.3, 4.8, 5.7, 3.8, 6.3, 5.4, 5.4, 80, 4.2, 8.3, 4.7, 6.0, 6.8, 5.7, 6.5, 4.6, 5.4, 3.7, 7.1, 5.5, 6.0, 6.7, 7.0, 7.3, 3.0, 6.6, 6.1, 2.4, 7.1, 9.4, 3.7, 4.5, 5.1, 5.9, 4.7, 5.5, 8.9, 8.1, 8.3, 4.3, 7.1, 9.3, 5.1, 6.1, 3.0, 5.7, 6.8, 3.1, 7.7, 7.3, 7.0, 6.2, 8.8, 5.3, 4.0.

- (a) Cread un vector con estas notas (podéis copiarlas de este documento y pegarlas) y ponedle un nombre adecuado.
- (b) ¿Cuántas notas contiene este vector? ¿Cuál es su valor medio?
- (c) ¡Vaya! El 80 ha sido un error, tenía que ser un 8.0. Cambiad el 80 del vector anterior por un 8.0, sin volver a entrar el resto de notas. Volved a calcular la media de las notas tras haber corregido este error.
- (d) ¿Cuál es la nota mínima obtenida por estos estudiantes? ¿Cuántos estudiantes la han sacado?
- (e) ¿Cuántos estudiantes han logrado un notable (entre 7 y 8.9)? ¿Qué porcentaje del total de estudiantes representan?
- (f) ¿Qué grupo es más numeroso: el de los estudiantes que han sacado entre 4 y 4.9, o el de los que han sacado entre 5 y 5.9?
- (g) Ordenad en orden creciente estas notas y obtened su *mediana*: una vez ordenado el vector, si tiene un número impar de entradas, su mediana es el valor central, y si tiene un número par de entradas, su mediana es la media aritmética de los dos valores centrales.

- (h) La mediana de un vector se puede calcular directamente con la función `median`. Calculad la del vector anterior con esta función. ¿Da lo mismo que el valor obtenido en el punto anterior?
- (i) ¿Cuántos notas diferentes hay en esta muestra? (Podéis emplear astutamente algunas funciones explicadas en esta lección, o podéis consultar `help.search("duplicated")` a ver si encontráis una función que elimine las entradas duplicadas de un vector.)



Universitat
de les Illes Balears

Campus Extens
UIB Virtual

<http://campusextens.uib.cat>



@campusextensUIB



<http://www.scoop.it/t/recursos-i-eines-per-al-professorat>



<http://campusextensrecursos.uib.es/>