Ejercicios Introducción a R

David Criado Ramón

09/11/2019

1. R Interactivo

Crea números del 1 al 30 usando el operador ":"

```
secuencia <- 1:30
secuencia
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26 27 28 29 30
```

Busca en la ayuda lo que hace la función seq(). Crea una secuencia de número aleatorios del 1 al 30 con un incremento de 0.5

seq es una función genérica que permite generar secuencias numéricas. Para delimitar el tamaño de la secuencia, así como la distancia entre dos elementos consecutivos, disponemos de varios parámetros que podemos pasar a la función.

- **from** indica el valor inicial de la secuencia.
- **to** indica el valor final de la secuencia.
- **by** indica la distancia entre dos valores consecutivos. Si no se indica será calculado según **from**, **to** y **length.out**
- **length.out** indica el tamaño deseado de la secuencia.
- **along.with** nos permite pasar un argumento del que se toma la longitud para la secuencia.

Es habitual llamar a la función con **seq(from, to)**, **seq(from, to, by)** o **seq(from, to, length.out)**.

```
secuencia <- seq(1, 30, 0.5)
secuencia
```

```
##  [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5
## [15]  8.0  8.5  9.0  9.5 10.0 10.5 11.0 11.5 12.0 12.5 13.0 13.5 14.0 14.5
## [29] 15.0 15.5 16.0 16.5 17.0 17.5 18.0 18.5 19.0 19.5 20.0 20.5 21.0 21.5
## [43] 22.0 22.5 23.0 23.5 24.0 24.5 25.0 25.5 26.0 26.5 27.0 27.5 28.0 28.5
## [57] 29.0 29.5 30.0
```

Lee los distintos argumentos que puedan utilizar la función seq() para saber qué se puede hacer. Crea una secuencia de 100 números, empezando por el número 20 y con un incremento de 0.5

```
secuencia <- seq(from=20, by=0.5, length.out= 100)
secuencia
```

```
##   [1] 20.0 20.5 21.0 21.5 22.0 22.5 23.0 23.5 24.0 24.5 25.0 25.5 26.0 26.5
##  [15] 27.0 27.5 28.0 28.5 29.0 29.5 30.0 30.5 31.0 31.5 32.0 32.5 33.0 33.5
##  [29] 34.0 34.5 35.0 35.5 36.0 36.5 37.0 37.5 38.0 38.5 39.0 39.5 40.0 40.5
##  [43] 41.0 41.5 42.0 42.5 43.0 43.5 44.0 44.5 45.0 45.5 46.0 46.5 47.0 47.5
##  [57] 48.0 48.5 49.0 49.5 50.0 50.5 51.0 51.5 52.0 52.5 53.0 53.5 54.0 54.5
##  [71] 55.0 55.5 56.0 56.5 57.0 57.5 58.0 58.5 59.0 59.5 60.0 60.5 61.0 61.5
##  [85] 62.0 62.5 63.0 63.5 64.0 64.5 65.0 65.5 66.0 66.5 67.0 67.5 68.0 68.5
##  [99] 69.0 69.5
```

Investiga si existen vectores en R definidos para los siguientes casos:

- Letras minúsculas.

```
minusculas <- letters
minusculas
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

- Letras mayúsculas.

```
mayusculas <- LETTERS
mayusculas
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

- Nombre de los meses del año.

```
meses <- month.name
meses
```

```
##  [1] "January"   "February"  "March"     "April"     "May"
##  [6] "June"      "July"      "August"    "September" "October"
## [11] "November"  "December"
```

- Nombre abreviado de los meses del año

```
meses_abreviados <- month.abb
meses_abreviados
```

```
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov"
## [12] "Dec"
```

Investiga la función rep()

La función **rep** nos permite replicar los elementos que se encuenten en x. Para determinar cómo se replican y cuántas veces disponemos de varios parámetros para la función.

- **times** indica el número de veces a repetir cada elemento si **length.out** no es 1 o a repetir el vector entero si **length.out** es 1.
- **length.out** es el tamaño deseado del vector con las repeticiones.
- **each** indica cuántas veces va a ser repetido un elemento.

Crea una secuencia que contenga los valores del 1 al 8 repetidos cinco veces.

```
uno_a_ocho_cinco_Veces <- rep(seq(1,8), times=5)
uno_a_ocho_cinco_Veces
```

```
##  [1] 1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 8 1 2 3
## [36] 4 5 6 7 8
```

Crea una secuencia que contenga las cuatro primeras letras del abecedario 6 veces.

```
abcd_seis_veces <- rep(letters[1:4], times=6)
abcd_seis_veces
```

```
##  [1] "a" "b" "c" "d" "a" "b" "c" "d" "a" "b" "c" "d" "a" "b" "c" "d" "a"
## [18] "b" "c" "d" "a" "b" "c" "d"
```

2. Vectores

Crea los siguientes vectores utilizando c():

- Un vector del 1 al 20.

```
uno_a_veinte <- c(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20)
uno_a_veinte
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

- Un vector del 20 al 1.

```
veinte_a_uno <- c(20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1)
veinte_a_uno
```

```
##  [1] 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1
```

- Un vector que contenga el siguiente patrón: 1, 2, 3, . . . , 19, 20, 19, 18, . . . , 2, 1.

```
uno_veinte_uno <- c(uno_a_veinte, veinte_a_uno[2:20])
uno_veinte_uno
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 19 18 17
## [24] 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1
```

Genera un vector x que contenga 9 números comprendidos entre 1 y 5.

```
x <- seq(1,5,length.out=9)
x
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

Busca lo que hace la función sequence(). ¿Cuál es la diferencia con la función seq()?

Mientras **seq()** nos permite indicar el inicio, final, tamaño deseado de una secuencia así como la distancias entre dos numéros consecutivos de la secuencia, **sequence()** parte de un vector que contiene uno o más elementos numéricos y genera un nuevo vector de manera que, para cada elemento de vector original **nvec** se genera una secuencia númerica de paso la unidad, que parte del 1 y acaba en el elemento indicado en **nvec**. El proceso se repite hasta llegar al final del vector, concatenando cada una de las secuencias obtenidas en el vector de salida.

Crea el vector numérico x <- c(2.3, 3.3, 4.4) y accede al segundo elemento del vector

```
x <- c(2.3, 3.3, 4.4)
x[2]
```

```
## [1] 3.3
```

Crea un vector numérico "z", que contenga del 1 al 10. Cambia el modo del vector a caracter. ¿Qué pasa si vuelves a poner z en el terminal?

```
z <- 1:10
z <- as.vector(z, mode="character")
z
```

```
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

Ahora tenemos un vector que, en vez de estar formado por números enteros, se encuentra formada por los strings correspondientes a esos números enteros.

Ahora cambia el vector z a numérico de nuevo

```
z <- as.vector(z, mode="numeric")
z
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Crea un vector x con 100 elementos, selecciona de ese vector una muestra al azar de tamaño 5.

```
x <- 1:100
# Muestra tomada sin reemplazo, Usar parámetro reemplaze para controlar
# si queremos reemplazo o no
muestra_x <- sample(x, 5)
muestra_x
```

```
## [1] 66 57 79 75 41
```

Genera un vector de enteros "integer" con 100 números entre el 1 y el 4 de forma random. Para ello mira en la ayuda la función runif.

La función runif nos devuelve valores obtenidos aleatoriamente en el intervalo [min, max) de la distribución uniforme definida por esos parámetros.

```
v <- as.integer(runif(100, 1, 5))
v
```

```
##   [1] 4 4 1 2 2 2 3 2 1 2 4 3 4 4 3 1 1 1 2 2 4 1 4 2 1 2 1 1 2 3 3 2 2 4 1
##  [36] 4 3 4 3 4 1 4 3 2 3 4 4 4 2 2 2 1 1 3 1 4 4 4 4 4 1 4 4 4 3 1 4 1 4 1
##  [71] 3 2 2 4 4 1 1 2 1 4 4 2 3 3 2 1 2 2 4 1 3 2 2 4 3 3 4 4 2 2
```

Ordena el vector por tamaño utilizando la función sort(). ¿Qué devuelve la función sort()?

```
sorted <- sort(v)
sorted
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [36] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4 4 4 4
##  [71] 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
```

La función sort dvuelve el vector ordenado en orden ascendente.También tenemos la opción decreasing=T para realizarlo en orden descendente.

Si quisieras invertir el orden de los elementos que función utilizarías.

```
sorted_desc <- sort(v, decreasing=T)
sorted_desc
```

```
##   [1] 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 3 3
##  [36] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [71] 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Ahora busca en la ayuda la función order(). Aplícala sobre el vector v que hemos generado en los ejercicios anteriores. ¿Cuál es la diferencia con la función sort()?

Order nos devuelve la permutación de los índices que ordenarían el vector. Similarmente a sort, podemos usar un parámetro para ordenar descendentemente en vez de ascendentemente.

```
ordered <- order(v)
ordered
```

```
##   [1]   3   9  16  17  18  22  25  27  28  35  41  52  53  55  61  66  68
##  [18]  70  76  77  79  86  90   4   5   6   8  10  19  20  24  26  29  32
##  [35]  33  44  49  50  51  72  73  78  82  85  87  88  92  93  99 100   7
##  [52]  12  15  30  31  37  39  43  45  54  65  71  83  84  91  95  96   1
##  [69]   2  11  13  14  21  23  34  36  38  40  42  46  47  48  56  57  58
##  [86]  59  60  62  63  64  67  69  74  75  80  81  89  94  97  98
```

Crea un vector x que contenga dos veces los números ordenados del 1 al 10. Investiga la función rep(). Una vez comprobado que funciona elimina las entradas repetidas del vector, para ello consulta la función unique().

```
x <- rep(1:10, 2)
x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10  1  2  3  4  5  6  7  8  9 10
```

```
x <- unique(x)
x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Crea un vector cualquiera. Devuelve de ese vector una muestra de cinco números seleccionados al azar. Usa la función sample(), prueba a determinar los valores que quieres extraer con reemplazo y sin reemplazo

```
x <- 1:100
sample(x, 50, replace=T)
```

```
##  [1] 17  6 61 66  7 24 29 40 60 94 62 27 17 78 48 81 63 77 13 28 39  4 85
## [24] 79 77 42 39 62 63 78 79 12 16 31 51 25 57 26 87 63 53 30 58 96 68 27
## [47] 90 30 11 67
```

```
sample(x, 50, replace=T)
```

```
##  [1]  25  66  32  84  88  47  69  93   8  50  35  43  95  35  21  31  67
## [18]  43  61  35  77  94  41 100  25  52  84  28  37  36  63  25  74   5
## [35]  85  75  81  76  64  26  87  27  89  25  49  79  28  56  33  91
```

```
sample(x, 50, replace=F)
```

```
##  [1] 48 91 56 99  2 10 18 40 88 96 92 77 98 49 54 28 34 95 82 84 73 51 33
## [24] 13 57 42 61  5 45 76 89 55 27  9  1 81 87 63 19 69 60 15 64 66 25 97
## [47] 75 16 43 31
```

```
sample(x, 50, replace=F)
```

```
##  [1] 86 98 71 93 31 55 97 78 43  5 96  7 88 50  8 30 59  4  2 61 60 36 52
## [24] 72 46 34 83 99 94 95 73 92 27  1 54 35 41  6  9 48 25 89 16 33 13 84
## [47] 26 63 76 39
```

En los casos en los hemos utilizado reemplazo podemos observar que en una misma muestra podría repetirse (Primer ejemplo: 77 dos veces). Cuando llamamos sin reemplazo no puede aparecernos dos veces el mismo elemento cuando sacamos la muestra con sample, aunque sí podría salir el mismo elemento si llamamos dos veces (El 5 aparece en las dos muestras sacadas sin reemplazo).

3. Explora el indexado de vectores.

Crea un vector con números del 1 al 100 y extrae los valores del 2 al 23

```
x <- 1:100
x[2:23]
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
```

Del mismo vector, x, extrae ahora todos los valores menos del 2:23

```
x[-(2:23)]
```

```
##  [1]   1  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39
## [18]  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56
## [35]  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73
## [52]  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
## [69]  91  92  93  94  95  96  97  98  99 100
```

Cambia el número en la posición 5 por el valor 99

```
x[5] <- 99
x
```

```
##   [1]   1   2   3   4  99   6   7   8   9  10  11  12  13  14  15  16  17
##  [18]  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34
##  [35]  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51
##  [52]  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
##  [69]  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85
##  [86]  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100
```

Crea un vector lógico del vector letters, (por ejemplo, comprobando si existe c en el vector letters).

```
x <- letters == 'c'
x
```

```
##  [1] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [23] FALSE FALSE FALSE FALSE
```

¿Qué devuelve el siguiente comando?

```
which(rep(letters, 2) == "c")
```

```
## [1]  3 29
```

El comando primero concatena dos veces el vector con las letras minúsculas, es decir, una vez termina el vector original empieza de nuevo el vector. Al igualarlo a c, el vector se convierte a un vector lógico, que indica con valor verdadera las posiciones que contenían una c. Al aplicar which, nos devuelve el índice de las posiciones verdaderas (3 y 29).

¿Qué devuelve el siguiente comando?

```
match(c("c", "g"), letters)
```

```
## [1] 3 7
```

Match nos devuelve los índices de la **primera ocurrencia** de cada elemento del vector pasado como primer parámetro en la estructura pasada como segundo parámetro. Por defecto, si no se encontrase el valor devuelve NA, aunque podemos modificar el valor a devolver con el argumento nomatch.

Crea un vector x de elementos -5, -1, 0, 1, . . . , 6. Escribe un código en R del tipo x['algo'], para extraer:

```
x <- c(-5,-1,0,1:6)
x
```

```
## [1] -5 -1  0  1  2  3  4  5  6
```

- Elementos de x menores que 0

```
x[x < 0]
```

```
## [1] -5 -1
```

- Elementos de x menores o iguales que 0

```
x[x <= 0]
```

```
## [1] -5 -1  0
```

- Elementos de x mayores o iguales que 3

```
x[x >= 3]
```

```
## [1] 3 4 5 6
```

- Elementos de x menores que 0 o menores que 4. Decir ésto es lo mismo que decir sólo menores que 4, pero vamos a especificarlo tal y como se enuncia.

```
x[x < 0 | x < 4]
```

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

- Elementos de x mayores que 0 y menores que 4

```
x[x > 0 & x < 4]
```

```
## [1] 1 2 3
```

- Elementos de x distintos de 0

```
x[x != 0]
```

```
## [1] -5 -1  1  2  3  4  5  6
```

El código is.na() se usa para identificar valores ausentes (NA). Crea el vector x <- c(1, 2, NA) y averigua que pasa cuando escribes is.na(x). Prueba con x[x != NA], ¿obtienes con este comando los "missing values" de x? ¿Cuál es tu explicación?

```
x <- c(1, 2, NA)
is.na(x)
```

```
## [1] FALSE FALSE  TRUE
```

is.na() nos devuelve un vector lógico que nos indica las posiciones en las que hay presentes "missing values".

```
x[x != NA]
```

```
## [1] NA NA NA
```

Si queremos determinar los valores que no contienen "missing values" deberíamos de haber usado x[!is.na(x)]

```
x[!is.na(x)]
```

```
## [1] 1 2
```

Los "missing values", representados en R como NA hacen referencia a datos perdidos. Al aplicar el operador de desigualdad sobre el valor NA, obtenemos un "missing value" de tipo lógico, por lo que al aplicarlo sobre un vector obtenemos un vector de NA lógicos, que al aplicarlos como indexación lógico nos devuelve un vector de missing values al no saber si el valor lógico era verdadero o falso.

Determina qué objetos tienes en tu directorio de trabajo.

Mi directorio de trabajo es la carpeta en la que tengo los materiales de la asignatura, en el directorio de trabajo tengo todos los trabajos relacionados con R y dentro tengo una carpeta llamada Python que contiene los trabajos relacionados con el mismo.

```
list.files()
```

```
##  [1] "cinco.txt"       "dos.txt"         "Ejercicios1.pdf"
##  [4] "Ejercicios1.R"   "Ejercicios1.Rmd" "Ejercicios2.pdf"
##  [7] "Ejercicios2.Rmd" "estructuras.pdf" "estructuras.R"
## [10] "estructuras.Rmd" "fin.txt"         "functions.pdf"
## [13] "functions.R"     "functions.Rmd"   "graficos.pdf"
## [16] "graficos.R"      "graficos.Rmd"    "guion.Rmd"
## [19] "io.pdf"          "io.R"            "io.Rmd"
## [22] "prime.txt"       "Python"          "strings.aux"
## [25] "strings.out"     "strings.pdf"     "strings.R"
## [28] "strings.Rmd"     "tres.txt"
```

Los objetos cargados actualmente en el entorno de R son

```
ls()
```

```
##  [1] "abcd_seis_veces"        "mayusculas"
##  [3] "meses"                  "meses_abreviados"
##  [5] "minusculas"             "muestra_x"
##  [7] "ordered"                "secuencia"
##  [9] "sorted"                 "sorted_desc"
## [11] "uno_a_ocho_cinco_Veces" "uno_a_veinte"
## [13] "uno_veinte_uno"         "v"
## [15] "veinte_a_uno"           "x"
## [17] "z"
```

Crea un vector de valores según la fórmula

$$e^x \cdot cos(x) \;\; para\; x = 3, 3.1, 3.2, ..., 6$$

```
x <- seq(3,6,0.1)
exp(x) * cos(x)
```

```
##  [1] -19.884531 -22.178753 -24.490697 -26.773182 -28.969238 -31.011186
##  [7] -32.819775 -34.303360 -35.357194 -35.862834 -35.687732 -34.685042
## [13] -32.693695 -29.538816 -25.032529 -18.975233 -11.157417  -1.362099
## [19]  10.632038  25.046705  42.099201  61.996630  84.929067 111.061586
## [25] 140.525075 173.405776 209.733494 249.468441 292.486707 338.564378
## [31] 387.360340
```

Calcula la siguiente sumatoria

$$\sum_{i=10}^{100} (i^3 + 4i^2)$$

```
i <- 10:100
i^3 + 4*i^2
```

```
##  [1]    1400    1815    2304    2873    3528    4275    5120    6069
##  [9]    7128    8303    9600   11025   12584   14283   16128   18125
## [17]   20280   22599   25088   27753   30600   33635   36864   40293
## [25]   43928   47775   51840   56129   60648   65403   70400   75645
## [33]   81144   86903   92928   99225  105800  112659  119808  127253
## [41]  135000  143055  151424  160113  169128  178475  188160  198189
## [49]  208568  219303  230400  241865  253704  265923  278528  291525
## [57]  304920  318719  332928  347553  362600  378075  393984  410333
## [65]  427128  444375  462080  480249  498888  518003  537600  557685
## [73]  578264  599343  620928  643025  665640  688779  712448  736653
## [81]  761400  786695  812544  838953  865928  893475  921600  950309
## [89]  979608 1009503 1040000
```

Crea los siguientes vectores. Vas a tener que usar las funciones sort(), order(), mean(), sqrt(), sum() y abs()

```
set.seed(50)
xVec <- sample(0:999, 250, replace=T)
yVec <- sample(0:999, 250, replace=T)
```

- Selecciona las variables de yVec > 600.

```
yVec[yVec > 600]
```

```
##  [1] 702 901 617 726 915 723 941 906 782 681 721 929 827 653 839 800 869
## [18] 692 840 845 769 866 696 685 788 642 902 797 601 656 842 970 680 792
## [35] 662 868 875 795 880 700 665 699 979 796 772 836 974 990 954 846 943
## [52] 658 655 628 623 629 989 738 992 758 870 910 933 641 872 904 647 988
## [69] 753 624 996 621 714 965 920 755 783 856 927 759 700 764 666 667 790
## [86] 654 959 868 963 698 686
```

- R permite encontrar las posiciones en las que se encuentran los elementos que cumplen una determinada condición con which(). Utiliza esta función para obtener las posiciones de los elementos de yVec mayores que 600.

```
which(yVec > 600)
```

```
##  [1]   3   9  10  18  20  22  25  26  27  29  37  41  42  43  45  48  49
## [18]  51  65  67  71  74  79  81  84  85  88  95  98  99 103 106 108 109
## [35] 110 113 114 119 120 129 130 131 138 139 143 147 148 152 154 159 161
## [52] 166 167 168 172 173 174 176 177 183 187 188 189 190 191 194 196 197
## [69] 201 202 204 206 207 211 212 219 223 224 225 227 229 230 233 235 238
## [86] 239 240 243 246 248 249
```

- ¿Qué posiciones de xVec son idénticas (tienen el mismo valor) a las posiciones > 600 de yVec?

Si entendemos que han de tener el mismo valor en la misma posición y ser mayor que 600

```
which(yVec > 600 & xVec == yVec)
```

```
## integer(0)
```

no encontramos ninguna.

Si entendemos que han de tener el mismo valor pero pueden estar en posiciones distintas:

```
indices <- which(xVec %in% yVec[yVec > 600])
indices
```

```
##  [1]  14  24  25  26  50  56  67  73  81  83  92  97 105 122 135 138 140
## [18] 166 191 192 207 219 224 237 239 241 242 247 250
```

Vistos numéricamente:

```
xVec[indices]
```

```
##  [1] 764 795 624 988 996 656 755 915 755 868 662 979 665 988 769 667 692
## [18] 963 665 996 839 601 827 840 680 954 927 800 795
```

- ¿Cuántos valores de yVec están por encima de 200 puntos por debajo del máximo de yVec?

```
sum(yVec < max(yVec) - 200)
```

```
## [1] 207
```

- ¿Cuántos números de xVec son divisibles por 2. Nota: el operador módulo es %% en R. ¿Cuánto vale su suma?

```
divisibles_logico <- xVec %% 2 == 0
c(sum(divisibles_logico), sum(xVec[divisibles_logico]))
```

```
## [1]   117 60056
```

Hay 117 números divisibles por 2 en xVec y la sumatoria de sus valores es 60056.

- Ordena los números de xVec en orden creciente según los valores de yVec. Entiendo que he de tomar los índices de las posiciones que ordenan ascendentemente a yVec y aplicarlo a xVec.

```
xVec[order(yVec)]
```

```
##   [1] 271 725 957 151 374  10 919 996 325 120 216 978 997 409 474 261 607
##  [18] 979 814 271 905 362 692 746 777 793 130  94 257 840 892 435  68 703
##  [35] 862  23 949 853 250 986 813 669 996 441 504 975  49  46  98 239 274
##  [52] 358 598 799 159 885  94 150 114 611 650 339 988 778 881 344 764 189
##  [69] 247 391 180  43 541 487 635 868 180 865 215 830 465 521 253 609  78
##  [86] 440 618 799 259 835 960 921 420 581 927 711 752 257 346 102 966 272
## [103] 665 640 563 104 887 510 276 958 160 855 662 795  40 450 648 656  12
## [120] 234 915 362 765 800 678 786 769 485 251 598 926 805 161 449 310 924
## [137] 369 777  17 765  59 795 367 499 498 778 274 450 651 722 954  55 470
## [154] 526 469 749 477  31 962 811 903 113 201 873 212 324 218 125  78 593
## [171] 680 961 385 963 646 705 317 523 610 568 755 139 935 810 211 319 161
## [188] 983 819 839 926 481 761 770 227 601 232 566   3 335 605 473 814 233
## [205] 402 221 206 286 440 455 852  87  86 398 591 755 576 827 500 261 687
## [222] 773 438 665 640 388 706 332 708 988  25 439 397  79 518 376 624 778
## [239] 777 512 398 757 345 895 667 762 387 561 258 390
```

- Crea un vector con la siguiente fórmula: (Voy a aplicarselo a yVec)

$$(|x_1 - \bar{x}|\frac{1}{2}, |x_2 - \bar{x}|\frac{1}{2}, ..., |x_n - \bar{x}|\frac{1}{2})$$

```
abs(yVec-mean(yVec))/2
```

```
##   [1]  29.68 234.82 112.18 191.32 171.82 120.82 157.82 236.82 211.68  69.68
##  [11] 201.82  20.68 158.32 120.32  32.82 210.82 126.32 124.18 104.32 218.68
##  [21]  72.32 122.68  46.82  26.18 231.68 214.18 152.18 194.82 101.68 238.32
##  [31] 138.82  15.82  94.32 215.32  15.18 202.82 121.68 199.32 197.32 159.32
##  [41] 225.68 174.68  87.68  78.82 180.68 219.32   5.32 161.18 195.68 153.32
##  [51] 107.18 108.32 152.82  82.32 193.32  16.82  57.68  25.18 151.82 213.82
##  [61]  40.32 109.82 238.82 102.32 181.18  30.32 183.68 178.32  32.18  40.68
##  [71] 145.68  35.68  12.32 194.18 133.82 107.32  13.68 148.32 109.18  37.82
##  [81] 103.68  44.18 109.32 155.18  82.18  71.82 153.32 212.18  54.32 173.82
##  [91] 112.32  26.32 130.82 159.82 159.68 153.82 203.32  61.68  89.18  20.68
## [101]  50.18  50.68 182.18  12.82  42.82 246.18  85.82 101.18 157.18  92.18
## [111] 117.32  96.82 195.18 198.68 107.82  38.32  58.18 192.82 158.68 201.18
## [121] 153.32 126.32  52.18  26.82  55.82  37.82  20.82  22.82 111.18  93.68
## [131] 110.68  49.32 130.82 122.82   0.82  36.18 112.32 250.68 159.18 196.32
## [141]  30.32  93.32 147.18  15.68 171.82 168.82 179.18 248.18  10.82 120.82
## [151] 140.82 256.18  37.32 238.18   4.18   2.68   2.32  17.18 184.18 205.82
## [161] 232.68 229.32  39.18  10.82  33.68  90.18  88.68  75.18  18.18  58.18
## [171] 109.82  72.68  75.68 255.68 195.82 130.18 257.18 134.82  18.32 127.32
## [181] 238.82  88.32 140.18  45.82  16.18 140.32 196.18 216.18 227.68  81.68
## [191] 197.18 228.32 117.82 213.18  75.82  84.68 255.18  48.32   9.18 237.82
## [201] 137.68  73.18  88.32 259.18 180.82  71.68 118.18 214.82  61.18  11.68
## [211] 243.68 221.18  99.32 144.32 144.82 148.82  58.82 120.32 138.68  74.82
## [221]  38.68 213.82 152.68 189.18 224.68 131.82 140.68 219.32 111.18 143.18
## [231] 173.32 149.32  94.18 183.32  94.68 117.32 181.82 156.18  88.18 240.68
## [241]  41.18  59.32 195.18 206.82  15.18 242.68   8.82 110.18 104.18  25.82
```

4. Búsqueda de valores idénticos y distintos en Vectores.

Haz la intersección de dos vectores month.name[1:4] y month.name[3:7] utilizando intersect().

```
intersect(month.name[1:4], month.name[3:7])
```

```
## [1] "March" "April"
```

Recupera los valores idénticos entre dos vectores usando %in%. Esta función devuelve un vector lógico de los elementos idénticos. Utiliza posteriormente el vector lógico generado para poder extraer ese subset del vector original.

```
a <- 1:10
b <- 2:5
a[a %in% b]
```

```
## [1] 2 3 4 5
```

Si x = month.name[1:4] e y = month.name[3:7] recupera los valores únicos en el primer vector. Para ello, investiga la función diff(). ¿Es posible usar la función diff() con caracteres? Busca una alternativa. Pista: busca funciones que contengan diff en el nombre. Comprueba si importa el orden en el que se pongan los vectores en la función.

Podemos realizar dicha operación con setdiff(). El orden sí es importante.

```
x <- month.name[1:4]
y <- month.name[3:7]
setdiff(x, y)
```

```
## [1] "January"  "February"
```

```r
setdiff(y, x)
```

```
## [1] "May"  "June" "July"
```

setdiff() realiza la diferencia del primer conjunto con el segundo, es decir, todos los elementos que están en el primero pero no en el segundo.

Une dos vectores sin duplicar las entradas repetidas en uno nuevo llamado x. Investiga la función union().

```r
union(x,y)
```

```
## [1] "January"  "February" "March"    "April"    "May"      "June"
## [7] "July"
```

Recupera las entradas duplicadas de x

```r
x <- c(1:10, 2:5, 2:5)
unique(x[duplicated(x)])
```

```
## [1] 2 3 4 5
```

5. Filtrado de vectores, subset(), which(), ifelse()

R permite extraer elementos de un vector que satisfacen determinadas condiciones utilizando la función subset(), la diferencia entre esta función y el filtrado normal es como funciona con NA, subset() los elimina automáticamente del cálculo. Para el vector x <- c(6, 1:3, NA, 12) calcula los elementos mayores que en 5 en x usando primero el filtrado normal, es decir, con el operador ">" y luego la función subset(). ¿Se obtiene el mismo resultado?

```r
x <- c(6, 1:3, NA, 12)
x[x > 5]
```

```
## [1]  6 NA 12
```

```r
subset(x, x > 5)
```

```
## [1]  6 12
```

R permite extraer encontrar las posiciones en las que se encuentran los elementos que cumplen una determinada posición con which(). Utiliza esta función para encontrar dado el vector z, las posiciones donde el cuadrado de z sea mayor que 8

```r
z <- -5:5
which(z^2 > 8)
```

```
## [1]  1  2  3  9 10 11
```

En R, aparte de encontrarse los típicos bucles if-then-else existe la función ifelse(). Para un vector x devuelve 5 para aquellos números que sean pares (módulo igual a 0) y 23 para los números impares.

```r
x <- 1:11
ifelse(x %% 2 == 0, 5, 23)
```

```
##  [1] 23  5 23  5 23  5 23  5 23  5 23
```

Practica ahora para el vector x <- c(5, 2, 9, 12) y crea un código que devuelva el doble de x si el valor de x es mayor que 6 y el triple si no lo es.

```
x <- c(5, 2, 9, 12)
ifelse(x > 6, 2*x, 3*x)
```

```
## [1] 15  6 18 24
```

6. Matrices

Ejecuta los siguiente comandos y observa qué pasa

```
M <- matrix(data=5, nr=2, nc=2)
M
```

```
##      [,1] [,2]
## [1,]    5    5
## [2,]    5    5
```

El primer caso hemos generado una matriz cuadrada de número filas nr y número de columnas nc rellena con el valor 5.

```
M <- matrix(1:6, 2, 3)
M
```

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

Ahora hemos creado una matriz rectangular con 2 filas (segundo parámetro) y 3 columnas (tercer parámetro), los datos son el vector del 1 al 6 y son introducidos por columnas en la matriz.

```
M <- matrix(1:6, 2, 3, byrow=TRUE)
M
```

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

Similar al caso anterior, pero ahora rellenamos la matriz por filas.

Crea un vector z con los 30 primeros números y crea con el una matriz M con 3 filas y 10 columnas.

```
z <- 1:30
M <- matrix(z, 3, 10)
M
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1    4    7   10   13   16   19   22   25    28
## [2,]    2    5    8   11   14   17   20   23   26    29
## [3,]    3    6    9   12   15   18   21   24   27    30
```

Extrae la tercera columna de M en un vector

```
M[,3]
```

```
## [1] 7 8 9
```

y calcula los efectos de los siguientes comandos

```
M[1,]
```

```
##  [1]  1  4  7 10 13 16 19 22 25 28
```

M[1,] devuelve un vector con la primera fila.

```
M[2,]
```

```
##  [1]  2  5  8 11 14 17 20 23 26 29
```

M[2,] devuelve un vector con la segunda fila.

```
M[,2]
```

```
## [1] 4 5 6
```

M[,2] devuelve un vector con la segunda columna.

```
M[1,2]
```

```
## [1] 4
```

M[1,2] devuelve el elemento de la matriz que se encuentra en la intersección de la primera fila con la segunda columna.

```
M[, 2:3]
```

```
##      [,1] [,2]
## [1,]    4    7
## [2,]    5    8
## [3,]    6    9
```

Devuelve una matriz que se corresponde a la submatriz con todas las filas y las columnas segunda y tercera.

Crea un array de 5x5 y rellénalo con los valores del 1 al 25.

```
array(1:25, dim=c(5,5))
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    6   11   16   21
## [2,]    2    7   12   17   22
## [3,]    3    8   13   18   23
## [4,]    4    9   14   19   24
## [5,]    5   10   15   20   25
```

Dadas las matrices m1 y m2 usa rbind() y cbind() para crear matrices nuevas. ¿En qué se diferencian las matrices creadas?

```
m1 <- matrix(1, nr=2, nc=2)
m2 <- matrix(2, nr=2, nc=2)

mr <- rbind(m1, m2)
mr
```

```
##      [,1] [,2]
## [1,]    1    1
## [2,]    1    1
## [3,]    2    2
```

```
## [4,]    2    2
```

```
mc <- cbind(m1,m2)
mc
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    1    2    2
## [2,]    1    1    2    2
```

La diferencia es que rbind concatena las matrices añadiendo nuevas filas y cbind concatena añadiendo nuevas columnas.

El operador para el producto de dos matrices es "%*%". Por ejemplo, considerando las dos matrices creadas en el ejercicio anterior utilízalo.

```
M <- mr %*% mc
M
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    2    4    4
## [2,]    2    2    4    4
## [3,]    4    4    8    8
## [4,]    4    4    8    8
```

La trasposición de una matriz se realiza con la función t; esta función también funciona con marcos de datos. Prueba con la matriz M del ejercicio anterior.

```
t(M)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    2    4    4
## [2,]    2    2    4    4
## [3,]    4    4    8    8
## [4,]    4    4    8    8
```

En el ejemplo anterior la traspuesta es igual a la matriz original. Probemos con otra matriz.

```
M <- matrix(1:9, 3, 3)
M
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
t(M)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

La función diag() se puede usar para extraer o modificar la diagonal de una matriz o para construir una matriz diagonal. Comprueba lo que ocurre al utilizar el siguiente comando.

```
diag(2.1, nr=3, nc=5)
```

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

```
## [1,]  2.1  0.0  0.0    0    0
## [2,]  0.0  2.1  0.0    0    0
## [3,]  0.0  0.0  2.1    0    0
```

Genera una matriz con todos los valores inicializados a 0 excepto la diagonal principal partiendo del (1, 1) que es inicializada con el valor puesto en el primer parámetro.