Principios y Herramientas de Programación

Dra. Jessica Andrea Carballido

jac@cs.uns.edu.ar

Dpto. de Ciencias e Ingeniería de la Computación

UNIVERSIDAD NACIONAL DEL SUR


```
     5
ames(sort(apply(ejemplo,1,sum)))[1]
1] "d"
> apply(ejemplo,1,sum)
a b c d e f g h
7 6 5 4 5 6 7 8
> sort(apply(ejemplo,1,sum))
d c e b f a g h
4 5 5 6 6 7 7 8
sort(apply(ejemplo,1,sum))[1]


y(ejemplo,1,sum))
```

Vectores

Un vector es una colección ordenada
de datos del mismo tipo.

Usamos la función **c()** o el operador **:** para generarlos.

```
 > aaa <- c (TRUE, FALSE) #   función c()
>   aaa
[1]  TRUE FALSE
> class(aaa)
[1] "logical"


> bbb <- c(2.1,3.8)
>  bbb
[1] 2.1 3.8
> class(bbb)
[1] "numeric"

> ccc <- c("a","b","c")    # vector de caracteres
>  ccc
[1] "a" "b" "c"
> class (ccc)
[1] "character"
```

```
> ddd <- 1:3                    # vector de enteros
>  ddd
[1] 1 2 3
> class(ddd)
[1] "integer"
```

Vectores

Al concatenar valores numéricos con caracteres,
(éstos últimos deben estar entre comillas)
el vector resultante es un vector de caracteres.

```
> nuevovector <- c(1,2,3, Pedro, Juan)
Error: object "Pedro" not found
> nuevovector <- c(1,2,3, "Pedro", "Juan")
> nuevovector
[1] "1"     "2"     "3"     "Pedro" "Juan"
> class(nuevovector)
[1] "character"
```

El intérprete buscó si existía alguna variable llamada Pedro

Vectores

Otras formas de crearlos:

```
> vec=c()
> vec
NULL
> vec=c(vec,1)
> vec
[1] 1
> vec=c(vec,2)
> vec
[1] 1 2
> vec=c(vec,3)
> vec
[1] 1 2 3
>
> vec=c(vec,c(4,3)
+ )
> vec
[1] 1 2 3 4 3
> vec=c(vec,10:20)
> vec
 [1]  1  2  3  4  3 10 11 12 13 14 15 16 17 18 19 20
```

v=2:8

```
> v=seq(2,8)
> v
[1] 2 3 4 5 6 7 8
> w=seq(1, by=3, length=10)
> w
 [1]  1  4  7 10 13 16 19 22 25 28
> z=rep("hola", len=5)
> z
[1] "hola" "hola" "hola" "hola" "hola"
>
```

```
> seq(1, 20, by=3)
[1] 1 4 7 10 13 16 19
```

```
> 1.8:4.6
[1] 1.8 2.8 3.8
```

?seq

Vectores

```
> seq(1, 9, by = pi)
[1] 1.000000 4.141593 7.283185
> seq(1.575, 5.125, by = 0.05)
 [1] 1.575 1.625 1.675 1.725 1.775 1.825 1.875 1.925 1.975 2.025 2.075 2.125 2.175
[14] 2.225 2.275 2.325 2.375 2.425 2.475 2.525 2.575 2.625 2.675 2.725 2.775 2.825
[27] 2.875 2.925 2.975 3.025 3.075 3.125 3.175 3.225 3.275 3.325 3.375 3.425 3.475
[40] 3.525 3.575 3.625 3.675 3.725 3.775 3.825 3.875 3.925 3.975 4.025 4.075 4.125
[53] 4.175 4.225 4.275 4.325 4.375 4.425 4.475 4.525 4.575 4.625 4.675 4.725 4.775
[66] 4.825 4.875 4.925 4.975 5.025 5.075 5.125
> seq(0, 1, length.out = 11)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

seq {base}

Sequence Generation

Description

Generate regular sequences. `seq` is a standard generic with a default method. `seq.int` is a primitive which can be much faster but has a few restrictions. `seq_along` and `seq_len` are very fast primitives for two common cases.

Usage

```
seq(...)

## Default S3 method:
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)
```

Examples

```
seq(0, 1, length.out = 11)
seq(stats::rnorm(20)) # effectively 'along'
seq(1, 9, by = 2)     # matches 'end'
seq(1, 9, by = pi)    # stays below 'end'
seq(1, 6, by = 3)
seq(1.575, 5.125, by = 0.05)
seq(17) # same as 1:17, or even better seq_len(17)
```

The interpretation of the unnamed arguments of seq and seq.int is *not* standard, and it is recommended always to name the arguments when programming.

Vectores

Generación de muestras aleatorias:

- Sample:

población

```
> xy=c("malo", "reg", "bueno")
> sample(xy, 10)
Error en sample(xy, 10) :
  cannot take a sample larger than the population when 'replace = FALSE'
> sample(xy, 3)
[1] "bueno" "malo"  "reg"
> pp=c(0.1,0.1,0.8)
> sample(xy, 10, replace=TRUE, prob=pp)
 [1] "bueno" "bueno" "reg"   "bueno" "reg"   "bueno" "bueno" "bueno" "bueno" "bueno"
```

- Con distribuciones de probabilidad:

r*dist* donde *dist* puede ser uniforme, normal, exponencial, etc.

```
> runif
function (n, min = 0, max = 1)
.External(C_runif, n, min, max)
<bytecode: 0x0000000009cf5928>
<environment: namespace:stats>
> runif(10,20,30)
 [1] 27.81257 23.17090 28.52833 24.94562 23.56907 28.72852 27.39849 22.89908 23.64417 28.72334
```

Vectores

Aritmética Vectorial

Los vectores son manipulados de manera similar a los escalares, con los operadores y funciones que vimos la clase pasada.

Elemento a elemento

```
> x=1:4
> y=5:8
> x
[1] 1 2 3 4
> y
[1] 5 6 7 8
> x*y
[1]  5 12 21 32
> y/x
[1] 5.000000 3.000000 2.333333 2.000000
> y-x
[1] 4 4 4 4
> x^y
[1]     1    64  2187 65536
> cos(x*pi)+cos(y*pi)
[1] -2  2 -2  2
>
```

```
> a=c(2,5,9)
> b=c(3,5,8)
> a
[1] 2 5 9
> b
[1] 3 5 8
> a*2
[1]  4 10 18
> a==b
[1] FALSE  TRUE FALSE
> a>b
[1] FALSE FALSE  TRUE
> sqrt(a+b)
[1] 2.236068 3.162278 4.123106
>
>
```

Vectores

Aritmética Vectorial

Los vectores son manipulados de manera similar a los escalares, con los operadores y funciones que vimos la clase pasada.

```
> x=1:4
> y=5:8
> x
[1] 1 2 3 4
> y
[1] 5 6 7 8
> x*y
```

Elemento a elemento, igual longitud?

```
> a=c(2,5,9)
> b=c(3,5,8)
> a
[1] 2 5 9
> b
```

```
> a=c(1,2,3)
> b=c(2,3)
> a+b
[1] 3 5 5
Warning message:
In a + b : longer object length is not a multiple of shorter object length
>
```

- *Reciclado* - Completa al vector más corto con repeticiones de todos sus elementos

Vectores

```
> v=seq(2,8)
> v
[1] 2 3 4 5 6 7 8
> w=seq(1, by=3, length=10)
> w
 [1]  1  4  7 10 13 16 19 22 25 28
```

Selección de elementos

Acceso a las componentes usando vectores booleanos

```
> v[4]
[1] 5
> w[-2]
[1]  1  7 10 13 16 19 22 25 28
> w[-c(1,3,5)]
[1]  4 10 16 19 22 25 28
> w[-c(1:5)]
[1] 16 19 22 25 28
> w[w%%2==0]
[1]  4 10 16 22 28
```

Elementos de w que son pares

Vectores

Otras funciones interesantes

all(a==b)

Retorna TRUE si todas las comparaciones fueron exitosas

any(a==b)

Retorna TRUE si alguna de las comparaciones fue exitosa

Expresión lógica

all(v1==v2)
  v1 y v2, deben tener igual longitud?
all(v1%%2==0)
any(v2>1)

```
> a=c(1,2,1,2)
> b=c(1,2)
```

```
> all(a==b)
[1] TRUE
> a==b
[1] TRUE TRUE TRUE TRUE
```

Vectores

Búsqueda de elementos

Ver si un elemento está en un arreglo

```
> v=round(runif(10,0,5))
> v
 [1] 1 4 5 5 3 2 4 5 4 1
> which(v==4)
[1] 2 7 9
> min(which(v==4))
[1] 2
> match(4,v)
[1] 2
> which(v==8)
integer(0)
> match(8,v)
[1] NA
```

any(v==4)
Devolverá
TRUE o FALSE

```
if (any(v == 4)) print(match(4, v))
else print("NO ESTA")
```

Vectores

Otras funciones interesantes

Nombre	Operación
`length()`	devuelve la cantidad de componentes del vector
`sum()`	calcula la suma de las componentes del vector
`prod()`	calcula el producto de las componentes del vector
`cumsum(), cumprod()`	calcula sumas y productos acumulados
`sort()`	ordena el vector
`diff()`	calcula diferencias de vectores adecuadamente corridos (por defecto 1)

Además…
min(), max(), mean(), median(), var(), sd()

```
>
> sort(c(4,1,6))
[1] 1 4 6
> sort(c(4,1,6), decreasing=TRUE)
[1] 6 4 1
>
```

```
>
> x=c(3,1,9,7,5)
> length(x)
[1] 5
> sum(x)          #3+1+9+7+5
[1] 25
> prod(x)         #3*1*9*7*5
[1] 945
> cumsum(x)
[1]  3  4 13 20 25
```

Vectores

```
Console ~/
> x=c(1,3,5,7,9)
> y=c(2,3,5,7,11,13)
> x+1; y*2
[1]  2  4  6  8 10
[1]  4  6 10 14 22 26
> length(x); length(y)
[1] 5
[1] 6
> x+y
[1]  3  6 10 14 20 14
Warning message:
In x + y : longer object length is not a multiple of shorter object length
> sum(x>5)
[1] 2
> sum(x[x>5])
[1] 16
> sum(x>5 | x<3)
[1] 3
> y[3]; y[-3]
[1] 5
[1]  2  3  7 11 13
> y[x]
[1]  2  5 11 NA NA
> y[y>=7]
[1]  7 11 13
> which(y==7)
[1] 4
> which.min(x)
[1] 1
> length(x[x%%3==0])
[1] 2
```

Suma 1 por cada TRUE

Acceso a las componentes usando vectores booleanos

Retornan posiciones

No se modificó ni x ni y.

Vectores

Modificación de algunas componentes:

```
> a=1:10
> a
 [1]  1  2  3  4  5  6  7  8  9 10
> a[c(3,4)]=0
> a
 [1]  1  2  0  0  5  6  7  8  9 10
> a[-c(3,4,5)]=100
> a
 [1] 100 100   0   0   5 100 100 100 100 100
> vl=c(TRUE,FALSE,FALSE)
> a[vl]=-20
> a
 [1] -20 100   0 -20   5 100 -20 100 100 -20
```

Acceso a las componentes usando vectores booleanos

Cuando el vector se accede con un vector lógico en **el lado izquierdo** de la asignación, se asignan solamente las posiciones con TRUE

```
>
> logico=c(TRUE, FALSE, FALSE, TRUE, TRUE)
> vector=1:10
> vector[logico]
[1]  1  4  5  6  9 10
>
```

Obtengo las componentes de las posiciones donde hay TRUE

```
> vector[logico]=0
> vector
 [1] 0 2 3 0 0 0 7 8 0 0
```


```
> vector[logico]=c(22,33)
> vector
 [1] 22  2  3 33 22 33  7  8 22 33
```

FUNCION **tapply**

Aplicar alguna función a las partes de un vector definidas por otro vector (un factor)

tapply

```
> edades=round(runif(10, 18,25))
> edades
[1] 24 22 20 23 24 19 24 20 25 24
> sexo=sample(c("F","M"), 10, replace=TRUE)
> sexo
[1] "F" "F" "M" "F" "M" "F" "F" "F" "F" "M"
> tapply(edades, sexo, mean)
F          M
22.42857 22.66667
```

Matrices

Una matriz es una colección de datos del mismo tipo ordenados en filas y columnas.

Usamos la función **matrix()** para generarlas.

Esta función toma un vector con los datos y los transforma en objetos con estructura de matriz.

Tiene 4 argumentos que especifican el **vector de entrada**, el tamaño del objeto matriz creado (**cantidad de filas y columnas**) y el argumento ***byrow*** que toma los valores TRUE o T y FALSE o F para especificar cómo son llenados los valores en la matriz.

Matrices

Usamos la función **matrix()** para generarlas.

```
>
> a <- c(1,2,3,4,5,6,7,8)
> A <- matrix(a,nrow=2,ncol=4, byrow=FALSE) # a is diferente de A
> A
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
>
```

Podríamos haber eliminado **byrow=FALSE**
ya que es el **valor por defecto**.

También la cantidad de columnas ya que se infiere
de la cantidad de elementos del vector y la cantidad de filas.

Matrices

Es más, como los argumentos tienen un orden específico y "byrow" por defecto es falso podríamos haber escrito

> A = matrix(a,2,4)

para obtener el mismo resultado.

O incluso:

> A = matrix(a,2)

```
> a=seq(1:8)
> a
[1] 1 2 3 4 5 6 7 8
> A=matrix(a,2)
> A
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
> 
```

Matrices

```
> a <- c(1,2,3,4,5,6,7,8,9,10)
> A <- matrix(a, nrow = 5, ncol = 2) # entra los valores por
columna
> A
[,1] [,2]
[1,] 1 6
[2,] 2 7
[3,] 3 8
[4,] 4 9
[5,] 5 10

> B <- matrix(a, nrow = 5, ncol = 2, byrow = TRUE) # por fila
> B
[,1] [,2]
[1,] 1 2
[2,] 3 4
[3,] 5 6
[4,] 7 8
[5,] 9 10
> C <- matrix(a, nrow = 2, ncol = 5, byrow = TRUE)
> C
[,1] [,2] [,3] [,4] [,5]
[1,] 1 2 3 4 5
[2,] 6 7 8 9 10
>
```

Matrices

Otra forma de generar matrices
es mediante las funciones **cbind**() y **rbind**().

```
> a=1:3
> b=c(4,1,8)
> m1=cbind(a,b)
> m2=rbind(a,b)
> a
[1] 1 2 3
> b
[1] 4 1 8
> m1
     a b
[1,] 1 4
[2,] 2 1
[3,] 3 8
> m2
  [,1] [,2] [,3]
a    1    2    3
b    4    1    8
```

Podemos hacer:
> M = cbind(M, nuevaCol)
(M siendo matriz, nuevaCol siendo vector)

> rbind(m1, c(1,6))

?

Matrices

Nombre	Operación
`dim()`	dimensión de la matriz (cantidad de filas y columnas)
`as.matrix()`	fuerza un argumento en un objeto tipo matriz
`%*%`	multiplicación matricial
`t()`	traspuesta
`det()`	determinante de una matriz cuadrada
`solve()`	inversa de una matriz. También resuelve un sistema de ecuaciones lineales
`eigen()`	calcula autovalores y autovectores

Matrices

```
> m1
     a b
[1,] 1 4
[2,] 2 1
[3,] 3 8
> m2
  [,1] [,2] [,3]
a    1    2    3
b    4    1    8
> dim(m1)
[1] 3 2
> dim(m2)
[1] 2 3
> m1%*%m2
     [,1] [,2] [,3]
[1,]   17    6   35
[2,]    6    5   14
[3,]   35   14   73
> t(m1)
  [,1] [,2] [,3]
a    1    2    3
b    4    1    8
>
```

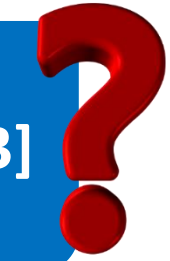
Multiplicación y traspuesta

```
> a=1:9
> m=matrix(a,nrow=3)
> m
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> m%*%m
     [,1] [,2] [,3]
[1,]   30   66  102
[2,]   36   81  126
[3,]   42   96  150
> m*m
     [,1] [,2] [,3]
[1,]    1   16   49
[2,]    4   25   64
[3,]    9   36   81
```

Matrices

```
> M
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> M2 <- cbind(A = c(1,2,3), B = c(4,5,6), C = c(8,9,10))
> rownames(M2) <- c("primero", "segundo", "tercero")
> M2
        A B  C
primero 1 4  8
segundo 2 5  9
tercero 3 6 10
> M2[,1] # da la primera columna
primero segundo tercero
      1       2       3
> M2 [1:2,] # da las filas primera a segunda
        A B C
primero 1 4 8
segundo 2 5 9
> M2 [c(1,3),] # da las filas primera y tercera
        A B  C
primero 1 4  8
tercero 3 6 10
> M2[,"A"] # da la columna de colnames "A"
primero segundo tercero
      1       2       3
```

> M2[c(1,3),3]



¡Lo mismo que hicimos con vectores!

```
> datos=sample(1:100,12)
> datos
 [1] 73 58 76 18 99  2 88 78 28 20 42 22
> m=matrix(datos, nrow=3)
> m
     [,1] [,2] [,3] [,4]
[1,]   73   18   88   20
[2,]   58   99   78   42
[3,]   76    2   28   22
> f=c(T,F,T,F)
> f
[1]  TRUE FALSE  TRUE FALSE
> m[,f]
     [,1] [,2]
[1,]   73   88
[2,]   58   78
[3,]   76   28
> m[3,f]
[1] 76 28
> m[1,f]=0
> m
     [,1] [,2] [,3] [,4]
[1,]    0   18    0   20
[2,]   58   99   78   42
[3,]   76    2   28   22
```

```
> m=matrix(1:9,3)
> m
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> sum(m[m%%2==0])
[1] 20
> m%%2==0
      [,1]  [,2]  [,3]
[1,] FALSE  TRUE FALSE
[2,]  TRUE FALSE  TRUE
[3,] FALSE  TRUE FALSE
> m[m%%2==0]
[1] 2 4 6 8
```

Cuando accedemos a una matriz
con un solo índice se convierte a vector (por columnas) para devolver el resultado.

```
> m=matrix(runif(9,1,9),3)
> m
         [,1]     [,2]     [,3]
[1,] 1.860973 2.691847 7.881910
[2,] 5.823677 4.057190 5.165604
[3,] 4.995363 2.591400 7.553434
> m[5]
[1] 4.05719
> m[6]
[1] 2.5914
```

FUNCION APPLY

Usar ***apply*** es mucho **mas eficiente que hacer un bucle**.
Principalmente para aplicar a una matriz (1er argumento),
por filas (2do argumento=1) o por columnas (2),
una operación (3er argumento).

apply

```
> matriz=matrix(round(runif(9, 0,100)),3)
> matriz
     [,1] [,2] [,3]
[1,]   31   40   63
[2,]   87   13   91
[3,]   32    1   72
> apply(matriz,1,min)
[1] 31 13  1
> apply(matriz,2,max)
[1] 87 40 91
> apply(matriz,1,sum)
[1] 134 191 105
> rowSums(matriz)
[1] 134 191 105
> sum(matriz)
[1] 430
```

```
> v=round(runif(9,0,100))
> matriz=matrix(v, nrow=3)
> matriz
     [,1] [,2] [,3]
[1,]   64  100   11
[2,]   95   97   32
[3,]   78   79   90
> apply(matriz, 1, min)
[1] 11 32 78
> min(matriz)
[1] 11
```

Consejo: Cada vez que vayamos a usar un “for” intentemos sustituirlo por alguna función predefinida de R (apply, rowSums,…)

Matrices

```
>
> datos=matrix(c(20,65,174,22,70,180,19,68,170),nrow=3,byrow=T)
> datos
     [,1] [,2] [,3]
[1,]   20   65  174
[2,]   22   70  180
[3,]   19   68  170
> colnames(datos)=c("edad","peso","altura")
> datos
     edad peso altura
[1,]   20   65    174
[2,]   22   70    180
[3,]   19   68    170
> rownames(datos)=c("luis","juan","pepe")
> datos
     edad peso altura
luis   20   65    174
juan   22   70    180
pepe   19   68    170
> apply(datos, 2, mean)
     edad      peso    altura
 20.33333  67.66667 174.66667
> |
```

Aplica la función (*mean*) a cada fila (2do argumento = 1) o columna (2do argumento = 2) de la matriz (1er argumento).

Alternativamente a usar *colnames* y *rownames*:
dimnames(datos)=list(c("luis","juan","pepe"), c("edad","peso","altura"))

have a nice day!

Vectores

Retornar el elemento más grande de un vector de enteros positivos:

```
mayElem = function (x)
 {
        may = 0
        for (i in 1:length(x))
                if (x[i] > may)
                        may = x[i]
        return(may)
}
```

Retornar el elemento más grande ubicado en una posición par de un vector:

```
mayElemP = function (x) {
  may=0
  for (i in seq(2,length(x),2))
        if (x[i]>may)  may=x[i]
  return(may)
}
```

max(x)

```
> v=c(10,2,4,1,5,8,2,5)
> pos=seq(2,by=2,length(v))
> max(v[pos])
[1] 8
```

Dra. Jessica Andrea Carballido
CONICET - DCIC (UNS)

Retornar el elemento más grande ubicado en una posición par de un vector:

```
mayElemP = function (x) {
  may=0
  for (i in seq(2,length(x),2))
        if (x[i]>may)  may=x[i]
  return(may)
}
```

```
mayElemP=function(v){
   max(v[seq(2, length(v), by=2)])
}
```