Capítol 1

Vectors

Comencem per recordar que un **vector** v és un objecte **estructurat** composat per un nombre finit d'objetes N anomenats components o elements. Tots els components d'un vector són del mateix tipus T. El tipus T és el **tipus base del vector**. És a dir, els vectors són objectes homogenis. De manera genèrica, el vector v del que estem parlant és de la forma:

t_1	t_2	t_3	t_4	t_5	t_6	$\ldots$	t_N

on $\forall i \in \{1, \ldots, N\}$, $t_i \in T$. És a dir, t_i es un valor del tipus base T. Un vector d'enters, de dimensió $N = 8$ podria ser aquest de sota:

1	2	3	4	-4	-3	-2	-1

Mentre que un vector de booleans, de dimensió $N = 4$ pot ser per exemple:

$TRUE$	$TRUE$	$FALSE$	$TRUE$

Pel que fa a requeriment de memòria de les variables de tipus vector, els seus components s'emmagatzemen de manera contigua a un bloc de memòria de mida igual a la dimensió del vector. Per a ser precisos hem d'aclarir que, en realitat, per a una variable vector es reserva espai a memòria equivalent a la dimensió del vector multiplicat per l'espai requerit per emmagatzemar un objecte del tipus base. El tipus base no sempre és un dels tipus base del lleguatge. Això vol dir que no sempre tenim vectors d'enters, booleans, reals i caracters. Podem tenim vectors de tipus estructurats també. En particular, els vector amb tipus base vectors (vectors de vectors) són coneguts com matrius i d'això en parlarem al Capítol 2.

1.1 Creació de vectors al llenguatge R

Al llenguatge R tenim varies funcions que són constructors de vectors.

1. La funció `c()`

   Aquest constructor ens permet crear vectors per enumeració dels seus components. El vector resultant tindrà dimensió igual al nombre d'elements que s'han fet servir a la crida de la funció `c()`.

```
> v <- c(0.4, 0.1)            # N = 2
> v
[1] 0.4 0.1
>
> v <- c(1, 2, 3, 4, 5, 6, 7)       # N = 7
> v
[1] 1 2 3 4 5 6 7
>
> v <- c(TRUE, FALSE)         # N = 2
> v
[1]  TRUE FALSE
>
> v <- c("a", "b", "c","d","e")     # N = 5
> v
[1] "a" "b" "c" "d" "e"
>
```

   S'ha de notar que poden crear vectors buits:

```
> v <- c()
> v
NULL
>
```

2. La funció `vector(Tipus_Base, Dimensió)`

   Aquest altre constructor ens permet crear vectors d'un tipus base i d'una dimensió donada. Si el Tipus_Base no es dona, per defecte, crea un vector de booleans inicialitzat en FALSE.

```
> v<-vector(length=6)
> v
[1] FALSE FALSE FALSE FALSE FALSE FALSE
>
```

3. L'operador `:`

   Aquest operador permet crear vectors d'un rang de números.

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> v <- 1:10
> v[2]
```

```
[1] 2
> v[2] <- 4
> v
 [1]  1  4  3  4  5  6  7  8  9 10
>
> v <- 4:9
> v
[1] 4 5 6 7 8 9
>
> v <- 12:7
> v
[1] 12 11 10  9  8  7
>
```

4. Altres operadors que ens permeten construïr vectors: `seq()` and `rep()`,

```
> v <- seq(from=16,to=24, by=2)
> v
[1] 16 18 20 22 24
>
> v <- seq(from=16,to=4, by=-3)
> v
[1] 16 13 10  7  4
>
> v<- seq(8)
> v
[1] 1 2 3 4 5 6 7 8
>
> v <- rep(1,5)
> v
[1] 1 1 1 1 1
>
> v <- rep(15,3)
> v
[1] 15 15 15
>
```

I també podem combinar els operadors:

```
> v <- c(10,rep(2,5),9)
> v
[1] 10  2  2  2  2  2  9
>
> v <- c(seq(5))
> v
[1] 1 2 3 4 5
> v <- c(13:20)
> v
[1] 13 14 15 16 17 18 19 20
>
> v <- c(seq(from=15, to=25, by=3))
```

```
> v
[1] 15 18 21 24
>
```

Un cop s'ha creat un vector v, no es pot canviar la seva dimensió. Malgrat això, podem "redimensionar" un vector con segueix:

```
> v <- 1:4
> v
[1] 1 2 3 4
> v <- c(v,10)
> v
[1]  1  2  3  4 10
> v <- c(v,10)
> v <- c(10,v)
> v
[1] 10  1  2  3  4 10 10
> v <- c(10:20,v)
> v
 [1] 10 11 12 13 14 15 16 17 18 19 20 10  1  2  3  4
    10 10
```

Amb les instruccions de dalt sembla que estem redimensionant el vector v. En realitat, aquestes instruccions són força costoses perquè cada vegada que s'afegeix un o més components, es crea un nou vector a un altre lloc de memoria i el vector original s'ha de relocalitzar totalment. Llavors, s'ha de fer servir amb molta cura aquest tipus d'instruccions.

1.2 Accés als components d'un vector

Mitjaçant un **operador d'accés directe**, que té com a paràmetre un enter que correspòn a un índex que indica la posició que es desitja visitar, es pot accedir a qualsevol component del vector de manera directa. Matemàticament l'especificació d'aquesta funció és la següent:

$$[\ \ ] : vector \times enter \to T$$

Llavors, donat un vector v i una expressió de tipus enter exp, fem servir la crida

$$v[exp]$$

amb

$$1 \leq avaluacio(exp) \leq N \qquad (1)$$

per referir-nos al component del vector que es troba a la posició que resulti d'avaluar exp.

Aleshores, no cal visitar els components anteriors a un component donat per arribar-hi.

```
> v <- c(0.4, 0.1)
> v[1]
[1] 0.4
```

```
>
> v <- c(1, 2, 3, 4, 5, 6, 7)
> v[5]
[1] 5
>
> v <- c(TRUE, FALSE)
> v[2]
[1] FALSE
> v <- c("a", "b", "c","d","e")
> i <- 2
> v[i]
[1] "b"
> v[i+1]
[1] "c"
> v[i*2]
[1] "d"
> j <- 3
> v[i+j]
[1] "e"
> v[2*i+1]
[1] "e"
>
```

Si assignem un valor numèric a qualsevol component d'un vector creat amb l'operador `vector()`, el tipus base canvia.

```
> v<-vector(length=6)
> v
[1] FALSE FALSE FALSE FALSE FALSE FALSE
>
> v[4] <- 55
> v
[1]  0  0  0 55  0  0
>
```

És molt important vigilar que es satisfaci la restricció (1). En cas que no la respectem tindrem un problema en intentar accedir a una posiciò il.legal dins del vector. En aquests casos, l'R ens respòn amb `NA` (Not Available):

```
> v <- c("a", "b", "c","d","e")
> v[6]
[1] NA
>
```

En particular, quan el vector és buit, tindrem:

```
> v <- c()
> v
NULL
> v[1]
NULL
>
```

S'ha de tenir en compte que, segons la definició de vectors, $v[exp] = t_{avaluacio(exp)} \in T$ i que per tant, aquest component es comporta com una variable del tipus base i podem fer servir qualsevol dels seus operadors.

```
> w <- rep(8,7)
> w
[1] 8 8 8 8 8 8 8
> v <- 5:10
> v
[1]  5  6  7  8  9 10
> cat(v[5]+w[6],"\n")
17
> y <- 5*v[3]
> y
[1] 35
>
```

Ara que hem vist la necessitat que els accessos als components d'un vector siguin posicions permeses, surt la necessitat de conèixer la dimensió d'un vector. Per fer-ho disposem a l'R d'una funció especificada com segueix:

$$length : vector \to enter$$

Així doncs, si la dimensió d'un vector v és N, la crida

$$length(v)$$

ens tornarà N.

```
> v<-c()
> length(v)
[1] 0
>
> v <- c("a", "b", "c","d","e")
> length(v)
[1] 5
>
> v <- c(TRUE, FALSE)
> length(v)
[1] 2
>
```

1.3 Recorregut

Hi ha molts problemes on s'han de processar tots i cadascun dels components d'una seqüència que, aleshores, pot estar representada per un vector. Aquesta classe de problemas se'ls anomena "Recorreguts". Per exemple, per calcular la mitjana d'una seqüència de notes d'estudiants, per avaluar un polinomi donat els seus coeficients, hem de fer un recorregut. Hi ha molts problemes que requereixin que tots els elemnts del'entrada es processin. Llavors, per fer recorreguts d'un vector, tindrem un esquema genèric com aquest:

```
A
for (i in 1:length(v)){
   processar(v[i])
}
B
```

On `A` correspon a un bloc d'instruccions de preprocessament, per exemple, la inicialització d'algunes variables, i `B` correspòn a un bloc d'instruccions de postprocessament, per exemple, imprimir el resultat.

Un primer exemple molt simple és l'escriptura dels components d'un vector:

```
escriure_vector <- function(v){
   for (i in 1:length(v)){
      cat(v[i],"\n")
   }
}
```

I es pot utilitzar d'aquesta forma:

```
> v<- 2:15
> escriure_vector(v)
2
3
4
5
6
7
8
9
10
11
12
13
14
15
>
```

Suposem que necessitem una funció per llegir un vector. Aquesta funció pot ser con segueix

```
llegir_vector <- function(v){
   for (i in 1:length(v)){
      v[i] <- scan(n=1,quiet=TRUE)
   }
   return (v)
}
```

I la podem fer servir de la següent manera:

```
> v <- vector(length = 8)
> v <- llegir_vector(v)
1: 2
1: 4
1: 6
```

```
1: 8
1: 10
1: 12
1: 14
1: 16
> v
[1]  2  4  6  8 10 12 14 16
>
```

Amdós exemples previs, no requereixen cap preprocessament o postprocesament. Considerem ara el problema de trobar el valor màxim dins d'un vector no buit (aquesta és la precondició del problema):

```
maxim <- function(v){
   max<- v[1]
   for(i in 2:length(v)){
      if (v[i] > max) max <- v[i]
   }
   return (max)
}
```

```
> v<-c(12,5,6,7,10,100,4,5, 56,2,20)
> maxim(v)
[1] 100
> v<-c(12,5,6,7,10,100,4,5, 56,2,200)
> maxim(v)
[1] 200
> v<-c(1200,5,6,7,10,100,4,5, 56,2,20)
> maxim(v)
[1] 1200
>
```

Un altre exemple és la funció que calcula el producte escalar de dos vectors de la mateixa dimensió (precondició):

```
producte_escalar <- function(v,w){
   p <- 0
   for (i in 1:length(v)){
      p <- p + v[i]*w[i]
   }
   return (p)
}
```

```
> v <- rep(2,3)
> v
[1] 2 2 2
> w <- 1:3
> w
[1] 1 2 3
> producte_escalar(v,w)
[1] 12
>
```

1.4 Cerca

Hi ha una classe de problemes on no cal visitar o processar tots els elements, és a dir, fer un recorregut, perquè ens demanen confirmar un predicat, és a dir, una propietat, `P(s)`, sobre la seqüència `s` representada amb un vector `v`, com per exemple:

- `P(s)` = "la seqüència `s` té múltiples de 3".
  Aquesta propietat vol dir el seguent: $\exists i : \ 1 \leq i \leq n : \ v[i] \, mod \, 3 = 0$ i això s'ha de comprovar. Anirem processant cadascun dels components del vector i si trobem un que sigui múltiple de 3, hem d'aturar la comprovació i donar una resposta afirmativa. En cas que no hi hagi cap múltiple de 3 al vector, acabarem fent un recorregut i la resposta seria negativa. Un possible script a l'R és el seguent:

```
multiploe3 <- function(v){
   i <- 1
   trobat <- FALSE
   while (i <= length(v) && !trobat){
      trobat <- v[i] %% 3 == 0
      i <- i + 1
   }
   return (trobat)
}
```

- `P(s)` = "la seqüència `s` té un cim".
  Per a aquest problema és necesari conèixer la definició de cim. Atès que la seqüència `s` es representa com un vector `s`, un cim és un component $v[i]$, $2 \leq i \leq n-1$, de v tal que $v[i-1] < v[i] < v[i+1]$. Si, mentre es fa el processament del vector es troba un component que satisfaci aquesta propietat s'ha d'aturar el recorregut perquè ja podem donar una resposta afirmativa. En cas contrari, si haguéssim de fer el processament de tots els components del vector, és a dir, el recorregut del vector, obtindríem una resposta negativa. Obviament, per comprobar si un vector té un cim, es requereix que tingui com a mínim tres elements. Llavors, aquesta és la precondició del problema: la dimensió de v ha de ser més gran o igual a 3. Un possible script a l'R és el seguent:

```
#Precondicio: length(v) >= 3
cim <- function(v){
   ant <- 1
   act <- 2
   suc <- 3
   trobat <- FALSE
   while (suc <= length(v) && !trobat){
      trobat <- v[ant] < v[act] && v[act] > v[suc]
      ant <- act
      act <- suc
      suc <- suc + 1
   }
   return (trobat)
}
```

- `P(s)` = "la seqüència `s` és creixent"
  Si analitzem aquesta propietat, ens adonarem que en realitat vol dir el seguent: $\forall i: 1 \leq i \leq n-1: v[i] \leq v[i+1]$ i això és el que realment hem de comprovar. En aquest cas, la cerca consisteix en trobar un contraexemple d'aquesta propietat. És a dir, si trobem un parell de components consecutius del vector tal que $1 \leq i \leq n-1: v[i] > v[i+1]$ hem d'aturar la comprovació de la propietat i donar una resposta negativa. En aquest cas, si féssim el recorregut, la resposta seria afirmativa. S'ha de notar que, segons la definició, tota seqüència buida o que tingui un sol element és creixent. Un possible script en R és el seguent:

```
1 creixent <- function(v){
2    i <- 1
3    continuar <- TRUE
4    while (i < length(v) && continuar){
5       continuar <- v[i] < v[i+1]
6       i <- i + 1
7    }
8    return (continuar)
9 }
```

- `P(s)` = "la seqüència `s` no té cap element parell".
  Si analitzem aquesta propietat, ens adonarem que en realitat vol dir el seguent: $\forall i: 1 \leq i \leq n: v[i] \; mod \; 2 \neq 0$ no és parell i això és el que realment hem de comprovar. Un altre cop, el problema consisteix en trobar un contraexemple de la propietat. És a dir, si trobem un component parell del vector que representa la seqüència, hem d'aturar la comprovació i donar una resposta negativa. Si féssim el recorregut, la resposta seria afirmativa. Un possible script a l'R és el seguent:

```
1 noparells <- function(v){
2    i <- 1
3    continuar <- TRUE
4    while (i <= length(v) && continuar){
5       continuar <- v[i] %% 2 != 0
6       i <- i + 1
7    }
8    return (continuar)
9 }
```

1.5 Sorting

Una família molt important d'algorismes és la família d'algorismes d'ordenament d'un vector. El problema pot ser vist de la següent manera: donat un vector, s'han de reorganitzar els seus components de forma tal que el vector arribi a representar una seqüència creixent (també pot ser decreixent). Això es pot formalitzar com segueix:

$$sort(v, v') = permutacio(v, v') \text{ i } \forall i: 1 \leq i \leq n-1: v'[i] \leq v'[i+1]$$

On el predicat $permutacio(v, v')$ diu que v' és una permutació de v. Els algorismes bàsics d'ordenament són el `SelectSort` (ordenament per selecció), l'`InsertSort` (ordenament per inserció) i el `BubbleSort` (ordenament per intercanvi o mètode de la bombolla). La dificultat que ténen aquests algorismes és que el vector s'ha d'ordenat sense fer servir cap emmagazatment addicional. És a dir, només s'ha de fer servir l'espai a memòria del vector v. Podeu trobar més detalls d'aquests algorismes a [1]. A sota presentem una implementació a l'R del `SelectSort`.

```
1  buscarmin <- function(v,inicio,final){
2     pos_min <- inicio
3     i <- inicio+1
4     while (i <= final){
5        if (v[i] < v[pos_min]) pos_min <- i
6        i <- i+1
7     }
8     return (pos_min)
9  }
10
11 SelectSort <- function(v){
12    n <- length(v)
13    i <- 1
14    #Durant la resta de l'execucio es mante que
15    #el segment v[1..i-1] esta ordenat
16    while (i <= n-1){
17       #Busca el component minim a v[i..n]
18       pos_min <- buscarmin(v,i,n)
19       #Intercanvia v[i] amb v[pos_min]
20       aux <- v[i]
21       v[i] <- v[pos_min]
22       v[pos_min] <- aux
23       i <- i+1
24    }
25    return (v)
26 }
```

La funció sort a l'R El llenguatge R disposa d'una funció que ordena un vector de manera creixent (opció per defecte) o decreixent. La discusió en relució a l'algorisme d'ordenament que implementa aquesta funció està fora de l'abast d'aquest document.

```
1 > v <- c(9,3,5,1,7,9,5)
2 > sort(v)
3 [1] 1 3 5 5 7 9 9
4 > sort(v,decreasing=TRUE)
5 [1] 9 9 7 5 5 3 1
6 >
```