

# Les séquences : vecteur

## Algo & Prog avec R

---

A. Malapert, B. Martin, M. Pelleau, et J.-P. Roy

21 avril 2019

Université Côte d'Azur, CNRS, I3S, France

`firstname.lastname@univ-cotedazur.fr`

# Qu'est-ce qu'une séquence ?

## Séquence

Une séquence est une suite finie de valeurs numérotées, distinctes ou pas, de n'importe quel type.

### Vecteur

Un vecteur est une séquence d'éléments **du même type**.

```
> 1:6
[1] 1 2 3 4 5 6
> c('foo', 'bar')
[1] "foo" "bar"
```

### Liste

Une liste est une séquence d'éléments **de type quelconque**.

```
> list(1:6, c('foo', 'bar'))
[[1]]
[1] 1 2 3 4 5 6

[[2]]
[1] "foo" "bar"
```

### Mutabilité

Les listes et vecteurs sont mutables, i.e. on peut les modifier.

Occupons nous d'abord des vecteurs.

## De l'utilité des vecteurs ...

Programmons la fonction `Diviseurs` qui retourne la liste des diviseurs non triviaux (distincts de 1 et  $n$ ) d'un entier  $n \geq 2$ .

Ne connaissant pas à l'avance le nombre d'éléments du résultat, on utilise un vecteur, initialement vide, qu'on agrandit au fur et à mesure.

```
Diviseurs <- function(n) {  
  res <- numeric(0);  
  for(d in 2:(n-1)) {  
    if( n %% d == 0) res <- append(res, d);  
  }  
  return(res)  
}
```

`numeric` crée un vecteur numérique.

`append` ajoute un élément en queue d'un vecteur.

```
> Diviseurs(1000)  
[1] 2 4 5 8 10 20 25 40 50 100 125 200 250 500
```

# Accès par rang aux éléments d'une séquence

L'accès par rang est commun aux vecteurs, listes, matrices, data frames.

- ▶ La notation indexée par crochets. Les indices commencent à 1.
- ▶ R supprime les éléments quand leurs indices sont négatifs.
- ▶ R vérifie la validité des arguments

```
> vec <- 1:5
> vec[1] # Les indices commencent à 1.
[1] 1
> vec[2]
[1] 2
> vec[c(1,5)] # accéder simultanément à plusieurs éléments.
[1] 1 5
> vec[-3] # suppression d'éléments avec un index négatif
[1] 1 2 4 5
> vec[0] # Les indices commencent à 1 !
integer(0)
> vec[6] # valeur spéciale pour les éléments manquants.
[1] NA
```

# Accès par contenu

Dans de nombreux langages, la fonction `index` permet de connaître le rang d'un élément dans une séquence donnée.

Elle n'existe pas en R et nous allons donc la programmer !

```
index <- function(x, vec) {  
  for(i in seq_along(vec)) {  
    if( vec[i] == x) return(i)  
  }  
  return(NA)  
}
```

```
> index(2, 1:5)  
[1] 2  
> index(6, 1:5)  
[1] NA  
> index("foo", c("foo", "bar"))  
[1] 1
```

La fonction `seq_along` renvoie le vecteur des indices du vecteur `vec` !

## Défi

Programmer une fonction `positions(x,vec)` retournant le vecteur des indices `i` tels que `vec[i] == x`.

## Autres fonctions d'accès par contenu

R ne dispose pas d'une fonction `index`, mais des fonctions plus puissantes et complexes : `match` et `which`.

# Arithmétique des vecteurs

## Opérations membre à membre

L'opération est appliquée sur les paires d'éléments à la même position.

```
> 1:3 + 3:1  
[1] 4 4 4  
> 1:3 * 3:1  
[1] 3 4 3
```

```
> 1:3 / 1:3  
[1] 1 1 1  
> (1:3) ** (3:1)  
[1] 1 4 3
```

## Recyclage des éléments

Si deux vecteurs ont des longueurs différentes, les éléments du plus court sont recyclés.

```
> 1:4 + 1:2  
[1] 2 4 4 6  
> 1:5 + 1:2  
[1] 2 4 4 6 6  
Warning message:  
In 1:5 + 1:2 :  
  la taille d'un objet plus long n'est pas multiple de la  
  taille d'un objet plus court
```

# Construction d'une *nouvelle* séquence

## Construction en extension

On utilise surtout la fonction `c`.

```
> c(2, 3, 5)
[1] 2 3 5
> c(TRUE, FALSE, TRUE, FALSE, FALSE)
[1] TRUE FALSE TRUE FALSE FALSE
```

## Longueur d'un vecteur

La longueur d'un vecteur est retournée par la fonction `length`.

```
> length(c("aa", "bb", "cc", "dd", "ee"))
[1] 5
```

## Concaténation de deux séquences

On peut coller côte à côte (concaténer) deux séquences pour construire une nouvelle séquence. Le vecteur numérique a été transformé (*coerced*) en chaînes de caractères pour la cohésion de type.

```
> c(1:3, c("foo", "bar"))
[1] "1" "2" "3" "foo" "bar"
```

# Extraction d'une tranche de séquence

La notation des tranches (**slices**) est valide pour toute séquence. Elle permet d'extraire une sous-séquence à partir d'un **vecteur numérique d'indices**.

- ▶ Les indices peuvent être dupliqués.
- ▶ Les indices peuvent être dans n'importe quel ordre.

```
> vec <- 1:5  
> vec[c(1,3,5)]  
[1] 1 3 5  
> vec[c(5,1,5)]  
[1] 5 1 5
```



# Extraction d'une tranche de séquence ...

La notation des tranches permet d'extraire une sous-séquence à partir d'un **vecteur logique d'indices**. Le booléen au rang  $k$  indique si le  $k$ -ème élément appartient à la sous-séquence.

- ▶ Les indices ne peuvent pas être dupliqués.
- ▶ Les indices ne peuvent pas être dans n'importe quel ordre.

```
> vec <- c("a", "b", "c", "d", "e")
> ind <- c(TRUE, TRUE, FALSE, FALSE, FALSE)
> vec[ind]
[1] "a" "b"
```

## Recyclage du vecteur logique

```
> vec[c(TRUE, FALSE)]
[1] "a" "c" "e"
```

## Extraction d'entêtes et de queues : head et tail

```
> head(vec, 3)
[1] "a" "b" "c"
```

```
> tail(vec, 3)
[1] "c" "d" "e"
```

```
> tail(vec, -3)
[1] "d" "e"
```

# Pré-allocation de mémoire

Prévoir l'espace mémoire nécessaire avant l'exécution du programme, en spécifiant la quantité nécessaire dans le code source. Au chargement du programme en mémoire, juste avant l'exécution, l'espace réservé devient alors accessible.

## Réécrivons 1:n avec ou sans préallocation de mémoire

```
Seq1 <- function(n) {  
  vec <- c() # sans  
  for(i in seq(n)) {  
    vec <- append(vec, i)  
  }  
  return(vec)  
}
```

```
Seq2 <- function(n) {  
  vec <- numeric(n) # avec  
  for(i in seq(n)) {  
    vec[i] <- i  
  }  
  return(vec)  
}
```

```
> system.time(replicate(5, Seq1(20000)))  
utilisateur      système      écoulé  
      1.373         0.017         1.393  
  
> system.time(replicate(5, Seq2(20000)))  
utilisateur      système      écoulé  
      0.013         0.000         0.012
```

# Modification d'un vecteur

Les vecteurs et les listes sont mutables, avec l'opérateur crochet, la méthode `append` (en queue ou à un rang donné) ou les indices négatifs (supprime un ou plusieurs éléments).

## Ajout d'éléments

```
> v <- 1:5  
> v[1] <- 5  
> v  
[1] 5 2 3 4 5
```

```
> append(v,6)  
[1] 5 2 3 4 5 6  
> append(v,c(6,7))  
[1] 5 2 3 4 5 6 7
```

```
> append(v,6,2)  
[1] 5 2 6 3 4 5  
> append(v,c(6,7),2)  
[1] 5 2 6 7 3 4 5
```

## Suppression d'éléments

```
> v[-3]  
[1] 1 2 4 5  
> v[c(-2, -3)]  
[1] 1 4 5
```

## Rappel sur les chaînes

- ▶ Les chaînes sont immutables.
- ▶ Ce ne sont pas des séquences.
- ▶ Ce sont des OBJETS !

```
> str <- "foo"  
> str[1]  
[1] "foo"
```

# Conditions et tests vectorisés !

## Opérations de comparaison vectorisées

Tous les opérateurs de comparaison sont vectorisés.

```
> vec <- 1:5
> vec == 2
[1] FALSE TRUE FALSE FALSE FALSE
> vec > 3
[1] FALSE FALSE FALSE TRUE TRUE
```

## Attention aux Opérations logiques vectorisées

Attention, les opérateurs & et && sont très différents pour un vecteur !

```
> vec %% 2 == 0 & vec > 2
[1] FALSE FALSE FALSE TRUE FALSE
> vec %% 2 == 0 && vec > 2
[1] FALSE
```

les opérateurs | et || sont tout aussi différents

# Recherche des bornes d'un vecteur

## Recherche du rang du plus petit/grand élément

```
> which.min(c(2, 3, 1, 5, 4))  
[1] 3
```

```
> which.max(c(2, 3, 1, 5, 4))  
[1] 4
```

## Recherche du plus petit/grand élément

```
> min(c(2, 3, 1, 5, 4))  
[1] 1
```

```
> max(c(2, 3, 1, 5, 4))  
[1] 5
```

## Recherche du plus petit et du plus grand élément

```
> range(c(2, 3, 1, 5, 4))  
[1] 1 5
```

## Le cas du NA : cela ne marche plus, supprimons-les

```
> range(c(2, 3, 1, 5, 4, NA))  
[1] NA NA  
> range(c(2, 3, 1, 5, 4, NA), na.rm = TRUE)  
[1] 1 5
```

## Recherche dans un vecteur : which

La fonction `which` renvoie un vecteur numérique d'indices dont les éléments respectent la condition.

```
> vec <- 1:5
> vec == 2
[1] FALSE TRUE FALSE FALSE FALSE
> vec %% 2 == 0 & vec > 2
[1] FALSE FALSE FALSE TRUE FALSE
> which(vec == 2)
[1] 2
> which(vec %% 2 == 0)
[1] 2 4
> which(vec %% 2 == 0 & vec > 2)
[1] 4
> which(vec %% 2 == 0 && vec > 2)
integer(0)
```

## Recherche dans un vecteur : match

La fonction `match` renvoie un vecteur numérique d'indices du premier élément de son premier argument contenu dans le second.

```
> match(1:5, 2)
[1] NA 1 NA NA NA
> 1:5 %in% 2
[1] FALSE TRUE FALSE FALSE FALSE
> which(1:5 %in% 2)
[1] 2
> match( 1:5, c(2,6,3))
[1] NA 1 3 NA NA
> match( 1:5, c(2,6,3), nomatch=0)
[1] 0 1 3 0 0
```

# Fonctions all et any sur un vecteur logique

**all** renvoie TRUE si tous les éléments sont TRUE.

```
> all(c(TRUE, TRUE))  
[1] TRUE  
> all(c(FALSE, TRUE))  
[1] FALSE  
> all(c(TRUE, TRUE, NA))  
[1] NA  
> all(c(TRUE, TRUE, NA), na.rm=TRUE)  
[1] TRUE  
> all(1:10 > 1)  
[1] FALSE
```

**any** renvoie TRUE si au moins un élément est TRUE.

```
> any(c(FALSE, FALSE))  
[1] FALSE  
> any(c(FALSE, TRUE))  
[1] TRUE  
> any(1:10 < 2)  
[1] TRUE
```



# Tri d'un vecteur : sort

## Dans un ordre donné

```
> x <- sample(10)
> sort(x)
[1] 1 2 3 4 5 6 7 8 9 10
> sort(x, decreasing=TRUE)
[1] 10 9 8 7 6 5 4 3 2 1
```

## En retournant éventuellement les indices

```
> x
[1] 9 3 10 1 4 8 5 7 6 2
> sort(x, index.return = TRUE)
$x
[1] 1 2 3 4 5 6 7 8 9 10
$ix
[1] 4 10 2 5 7 9 8 6 1 3
```

## De n'importe quel type

```
> sort(head(letters, 10), decreasing=TRUE)
[1] "j" "i" "h" "g" "f" "e" "d" "c" "b" "a"
```

Consultez notre superbe [reference card](#) !

# Extraction des nombres pairs d'un vecteur

## Méthode classique

Facilement transportable dans d'autres langages de programmation.

```
ExtractionNombresPairs <- function(x) {  
  res <- numeric(0);  
  for(v in x) {  
    if( v %% 2 == 0) res <- append(res, v);  
  }  
  return(res)  
}
```

```
> x <- 1:10  
> ExtractionNombresPairs(x)  
[1] 2 4 6 8 10
```

## Méthodes plus courtes, mais spécifiques à R

```
> x[ x %% 2 == 0]  
> x[ which(x %% 2 == 0)]  
> Filter(function(n) n %% 2 == 0, x)
```

Questions?

Retrouvez ce cours sur le site web

[www.i3s.unice.fr/~malapert/R](http://www.i3s.unice.fr/~malapert/R)