Formation **R** Perfectionnement

Martin Chevalier (Insee)

Cette page comporte l'ensemble des cas pratiques de la formation **R** perfectionnement de décembre 2016 accompagnés de leur correction.

Le support de présentation est téléchargable ici et les données nécessaires aux cas pratiques là

Cette page a été réalisée avec Rstudio sous Rmarkdown et compilée le 2016-12-09.

Quelques rappels : vecteurs, matrices, listes et `data.frame`

Ces trois premiers cas pratiques reprennent certains éléments fondamentaux du langage de **R**. Pour une présentation plus progressive et approfondie, se référer au second module de la formation **R** initiation.

Cas pratique 1 Savoir manipuler des vecteurs

Les vecteurs sont les éléments fondamentaux de **R** : ils sont utilisés dans la plupart des opérations sur les objets plus complexes (listes, `data.frame`).

a. Créez le vecteur `a1` en utilisant trois méthodes différentes :

```
a1
##  [1]  6  7  8  9 10 11 12 13 14 15
```

Quelles sont ses caractéristiques (type, longueur, etc.) ?
Afficher/masquer la solution

```
# Méthode 1 : définition explicite avec c()
a1 <- c(6, 7, 8, 9, 10, 11, 12, 13, 14, 15)

# Méthode 2 : définition de la séquence avec `:`
a1 <- 6:15

# Méthode 3 : définition de la séquence avec seq()
a1 <- seq(6, 15, by = 1)

# Caractéristiques : fonctions str(), length(), class()
# et typeof()
str(a1)
##  num [1:10] 6 7 8 9 10 11 12 13 14 15
```

```
length(a1)
## [1] 10
class(a1)
## [1] "numeric"
typeof(a1)
## [1] "double"
# Note : la méthode 1 produit un vecteur de numériques
# (typeof(a1) vaut "double") alors que les méthodes 2 et 3
# produisent un vecteur d'entiers (typeof(a1) vaut "integer")
```

b. Sélectionnez l'élément en deuxième position par deux méthodes différentes. Utilisez alors la méthode la plus appropriée pour sélectionner : les éléments en position 8 et 5 (dans cet ordre) ; tous les éléments sauf le neuvième ; tous les éléments dont la valeur est strictement supérieure à 10 ; tous les éléments dont la valeur est paire.
Afficher/masquer la solution

```
# L'opérateur `[` permet d'extraire les éléments d'un vecteur
# selon trois méthodes

# Méthode 1 : position de l'élément dans le vecteur
a1[2]
## [1] 7

# Méthode 2 : vecteur logique de longueur length(a1)
a1[c(FALSE, TRUE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE)]
## [1] 7

# Méthode 3 : noms des éléments du vecteur
# Aucun nom n'est associé ici au vecteur a1, cf. la question
# suivante.

# La méthode 1 est la plus adaptée aux deux premières sous-questions
a1[c(8, 5)]
## [1] 13 10
a1[-9]
## [1]  6  7  8  9 10 11 12 13 15

# La méthode 2 est la plus adaptée aux deux dernières sous-questions
a1[a1 > 10]
## [1] 11 12 13 14 15
a1[a1 %% 2 == 0]
## [1]  6  8 10 12 14
# Note : x %% y renvoie le reste de la division euclidienne de
# x par y.
```

c. Que renvoie `names(a1)` ? À quoi cela correspond-il ? Utilisez le vecteur `letters` pour nommer les éléments de `a1` : a, b, c, d, e, f, g, h, i, j. Quelle est la valeur de l'élément dont le nom est `"b"` ?
Afficher/masquer la solution

```
# La fonction names(x) renvoie la valeur du vecteur de noms
# associé à l'objet x
names(a1)
## NULL
# Ici names(a1) vaut NULL car les éléments de a1 ne sont pas nommés :
# aucun vecteur de nom ne correspond à a1.

# letters est un vecteur de type caractère et de longueur 26
# stockant les 26 lettres de l'alphabet (en minuscules)
letters
##  [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"

# Utilisé avec <-, la fonction names() permet d'affecter
# un vecteur de noms à un objet
names(a1) <- letters[1:10]
a1
##  a  b  c  d  e  f  g  h  i  j
##  6  7  8  9 10 11 12 13 14 15

# Dès lors qu'un objet dispose d'un vecteur de noms,
# il est possible de les utiliser pour sélectionner ses éléments.
a1["b"]
## b
## 7
```

d. On définit le vecteur `a2 <- 1:10`. Que renvoie `c(a1, a2)` ? Utilisez les fonctions `union()`, `intersect()` et `setdiff()` pour déterminer les valeurs présentes dans `a1` ou `a2`, dans `a1` et `a2`, dans `a1` mais pas dans `a2`, dans `a2` mais pas dans `a1`.
Afficher/masquer la solution

```
# On définit a2 avec `:`
a2 <- 1:10

# Appliquée à deux vecteurs, la fonction c() en renvoie
# la concaténation (dans l'ordre)
c(a1, a2)
```

```
##  a  b  c  d  e  f  g  h  i  j
##  6  7  8  9 10 11 12 13 14 15  1  2  3  4  5  6  7  8  9 10

# Les fonctions union(), intersect() et setdiff()
# permettent d'effectuer des opérations ensemblistes
# sur les vecteurs
union(a1, a2) # Eléments dans a1 ou dans a2
##  [1]  6  7  8  9 10 11 12 13 14 15  1  2  3  4  5
intersect(a1, a2) # Eléments dans a1 et dans a2
## [1]  6  7  8  9 10
setdiff(a1, a2) # Eléments dans a1 mais pas dans a2
## [1] 11 12 13 14 15
setdiff(a2, a1) # Eléments dans a2 mais pas dans a1
## [1] 1 2 3 4 5

# Note : setdiff() est particulièrement utile pour comparer
# les valeurs prises par un identifiant dans deux tables
# différentes.

# Par exemple,  dans une enquête présentant une table
# de niveau ménage et une table de niveau individu,
# setdiff(ind$idmen, men$idmen)
# permet de vérifier que tous les individus appartiennent
# bien dans un ménage référencé dans la table de niveau
# ménage.
```

e. Générez un vecteur `a3` de longueur 100 et tiré dans une loi uniforme avec la fonction `runif()` :
   — quel est son maximum ?
   — combien de valeur de `a3` sont strictement supérieures à 0,80 ?
   — remplacer toutes les valeurs de `a3` inférieures à 0,50 par leur opposé.

   Afficher/masquer la solution

```
# La fonction runif(n) permet de générer des vecteurs de longueur
# n dont les observations sont tirées dans une loi uniforme sur [0;1]
# (par défaut).
a3 <- runif(100)

# Les trois sous-questions font appel à des fonctions ou
# structures particulièrement utiles en pratique
max(a3)
## [1] 0.9881736
sum(a3  > 0.80)
```

```
## [1] 31
a3[a3 < 0.50] <- - a3[a3 < 0.50]

# Note : la génération de a3 reposant sur des séquences
# de nombres pseudo-aléatoires, les résultats
# que vous obtenez de ceux de la correction.
# La fonction set.seed() permet de spécifier
# un point d'initialisation pour la séquence de nombres
# pseudo-aléatoire et donc de reproduire parfaitement
# une séquence générée aléatoirement. Son utilisation
# est illustrée dans le cas pratique 5.
```

Cas pratique 2 Savoir manipuler des matrices

Les matrices s'apparentent aux vecteurs en cela qu'elles ne peuvent contenir que des données d'un seul type (numérique, caractère, logique). Elles ont néanmoins deux dimensions, ce qui permet de leur appliquer des fonctions spécifiques.

a. Utlisez les fonctions `matrix()` pour générer une matrice `b1` de 4 lignes et de 5 colonnes dont les valeurs sont celles du vecteur `1:20` (en colonnes). Quelles sont ses caractéristiques (type, dimensions) ?
Afficher/masquer la solution

```
# La syntaxe de la fonction matrix() est la suivante :
# matrix(data, nrow, ncol)
b1 <- matrix(1:20, nrow = 4)
b1
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20

# Caractéristiques
str(b1)
##  int [1:4, 1:5] 1 2 3 4 5 6 7 8 9 10 ...
class(b1)
## [1] "matrix"
typeof(b1)
## [1] "integer"
dim(b1)
```

```
## [1] 4 5
nrow(b1)
## [1] 4
ncol(b1)
## [1] 5
```

b. Sélectionnez l'élément en deuxième ligne, quatrième colonne de `b1` par deux méthodes différentes. Utilisez alors la méthode la plus appropriée pour sélectionner : sa troisième ligne ; les colonnes impaires.
Afficher/masquer la solution

```
# Comme pour les vecteurs, l'opérateur `[` permet d'extraire
# les éléments d'un vecteur selon trois méthodes. La matrice
# étant à deux dimensions, `[` doit comporter deux positions
# [ligne, colonne].

# Méthode 1 : position de l'élément dans la matrice
b1[2, 4]
## [1] 14

# Méthode 2 : vecteurs logiques de longueur
# nrow(b1) et ncol(b1)
b1[c(FALSE, TRUE, FALSE, FALSE), c(FALSE, FALSE, FALSE, TRUE, FALSE)]
## [1] 14

# Méthode 3 : noms des lignes ou des colonnes
# de la matrice
# Aucun nom n'est associé ici à la matrice b1.

# Pour extraire des lignes entières, la seconde
# position de `[` doit rester vide
b1[3, ]
## [1]  3  7 11 15 19

# Pour extraire des colonnes entières, la première
# position de `[` doit rester vide
b1[, 1:ncol(b1) %% 2 == 1]
##      [,1] [,2] [,3]
## [1,]    1    9   17
## [2,]    2   10   18
## [3,]    3   11   19
## [4,]    4   12   20
```

c. Utilisez les fonctions `colSums()` et `rowSums()` pour déterminer :
   — la somme des éléments de chaque ligne de `b1` ;
   — pour chaque colonne de `b1`, le nombre d'éléments pairs.
   Afficher/masquer la solution

```
# Les fonctions colSums() et rowSums() calculent
# des sommes respectivement selon les colonnes et
# les lignes d'une matrice.
rowSums(b1)
## [1] 45 50 55 60

# Appliqué à une matrice, une expression logique
# produit une matrice de mêmes dimensions composée
# de valeur TRUE ou FALSE
b1 %% 2 == 0
##       [,1]  [,2]  [,3]  [,4]  [,5]
## [1,] FALSE FALSE FALSE FALSE FALSE
## [2,]  TRUE  TRUE  TRUE  TRUE  TRUE
## [3,] FALSE FALSE FALSE FALSE FALSE
## [4,]  TRUE  TRUE  TRUE  TRUE  TRUE
# Il n'y a alors plus qu'à utiliser la fonction
# colSums() pour sommer ces valeurs en colonne
# (TRUE est converti en 1 et FALSE en 0).
colSums(b1 %% 2 == 0)
## [1] 2 2 2 2 2
```

Cas pratique 3 Savoir manipuler des listes et des data.frame

Les listes constituent un type d'objet particulièrement souple dans **R** : contrairement aux vecteurs ou aux matrices, elles peuvent contenir des objets de types ou de dimensions différents. Les `data.frame` sont des cas particuliers de listes dans lesquels tous les éléments ont la même longueur (et sont en général de dimension 1). La plupart des données statistiques sont stockées sous la forme de `data.frame`.

a. Utilisez la fonction `list()` pour créer la liste `c1` contenant le vecteur `a1` du cas pratique 1 et la matrice `b1` du cas pratique 2. Assignez à chaque élément de la liste le nom de son objet d'origine.
   Afficher/masquer la solution

```
# La fonction list() permet de créer une liste
# à partir d'autres objets.
```

```r
c1 <- list(a1, b1)
c1
## [[1]]
##  a  b  c  d  e  f  g  h  i  j
##  6  7  8  9 10 11 12 13 14 15
##
## [[2]]
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
names(c1)
## NULL

# Pour créer c1 en associant à chaque objet un nom,
# il suffit de l'indiquer dans la fonction list()
c1 <- list(a1 = a1, b1 = b1)
c1
## $a1
##  a  b  c  d  e  f  g  h  i  j
##  6  7  8  9 10 11 12 13 14 15
##
## $b1
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
names(c1)
## [1] "a1" "b1"
```

b. Comparez les caractéristiques de `c1[1]` et `c1[[1]]`. Quelle valeur renvoie `c1[[1]][2]` ? Quelle syntaxe alternative pourriez-vous utilisez ?
Afficher/masquer la solution

```r
# Appliqué à une liste, l'opérateur d'extraction `[` retourne
# une sous-liste de la liste originale
c1[1]
## $a1
##  a  b  c  d  e  f  g  h  i  j
##  6  7  8  9 10 11 12 13 14 15
is.list(c1[1])
## [1] TRUE
```

```r
# On peut utiliser un vecteur logique ou un nom avec `[`
c1[c(TRUE, FALSE)]
## $a1
##  a  b  c  d  e  f  g  h  i  j
##  6  7  8  9 10 11 12 13 14 15
c1["a1"]
## $a1
##  a  b  c  d  e  f  g  h  i  j
##  6  7  8  9 10 11 12 13 14 15

# En revanche, `[[` renvoie non pas une sous-liste mais l'élément
# de la liste lui-même
c1[[1]]
##  a  b  c  d  e  f  g  h  i  j
##  6  7  8  9 10 11 12 13 14 15
is.list(c1[[1]])
## [1] FALSE
str(c1[[1]])
##  Named num [1:10] 6 7 8 9 10 11 12 13 14 15
##  - attr(*, "names")= chr [1:10] "a" "b" "c" "d" ...
# En l'occurrence ici, c1[[1]] renvoie la valeur du vecteur a1
# du premier cas pratique (au moment où c1 a été créée).
# On ne peut pas utiliser de vecteur logique avec `[[`
c1[[c(TRUE, FALSE)]]
## Error in c1[[c(TRUE, FALSE)]]: tentative de sélectionner moins d'un élément
c1[["a1"]]
##  a  b  c  d  e  f  g  h  i  j
##  6  7  8  9 10 11 12 13 14 15


# De ce fait, c1[[1]][2] renvoie la valeur de l'élément en
# deuxième position dans l'élément en première position de c1
c1[[1]][2]
## b
## 7

# On peut aussi utiliser les noms
c1[["a1"]]["b"]
## b
## 7

# Et même le signe $ pour extraire des valeurs par noms de la liste
c1$a1["b"]
## b
## 7
```

c. On définit le `data.frame` `c2` :

```
c2 <- data.frame(
  var1 = 1:20
  , var2 = letters[20:1]
  , var3 = rep(c(TRUE, FALSE), times = 10)
)
```

Quelles sont les caractéristiques de `c2` ? Vérifiez qu'il s'agit bien d'une liste.
Afficher/masquer la solution

```
# Caractéristiques
str(c2)
## 'data.frame':    20 obs. of  3 variables:
##  $ var1: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ var2: Factor w/ 20 levels "a","b","c","d",..: 20 19 18 17 16 15 14 13 12 11
##  $ var3: logi  TRUE FALSE TRUE FALSE TRUE FALSE ...
dim(c2)
## [1] 20  3
length(c2)
## [1] 3
# Note : la fonction length() renvoie le nombre de colonnes
# de c2. Cela traduit le fait que c2 est en fait une liste
# dont chaque colonne est un élément.
is.list(c2)
## [1] TRUE
as.list(c2)
## $var1
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
##
## $var2
##  [1] t s r q p o n m l k j i h g f e d c b a
## Levels: a b c d e f g h i j k l m n o p q r s t
##
## $var3
##  [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
## [12] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

d. Quel est le type de la variable `var2` ? Comment l'expliquez-vous et comment feriez-vous pour qu'il n'en soit pas ainsi ?
Afficher/masquer la solution

```r
# La variable var2 est de type factor
class(c2$var2)
## [1] "factor"
# Par défaut, la fonction data.frame() convertit
# les variables caractères en type factor. Pour
# empêcher la conversion, utiliser l'option
# stringsAsFactors = FALSE dans la fonction data.frame().
c3 <- data.frame(
  var1 = 1:20
  , var2 = letters[20:1]
  , var3 = rep(c(TRUE, FALSE), times = 10)
  , stringsAsFactors = FALSE
)
class(c3$var2)
## [1] "character"
```

e. Sélectionnez la variable `var3` en utilisant son nom de quatre manières différentes. Sélectionner les variables `var2` et `var1` (dans cet ordre). Sélectionnez toutes les variables sauf la variable `var2`.
Afficher/masquer la solution

```r
# Le data.frame est un liste susceptible d'être représentée
# par un tableau à deux dimensions. De ce fait, le fonctionnement
# des opérateurs d'extraction est proche de celui des listes
# mais aussi de celui des matrices

# Méthode 1 : `[` comme une liste
c2["var3"]
##     var3
## 1   TRUE
## 2  FALSE
## 3   TRUE
## 4  FALSE
## 5   TRUE
## 6  FALSE
## 7   TRUE
## 8  FALSE
## 9   TRUE
## 10 FALSE
## 11  TRUE
## 12 FALSE
## 13  TRUE
## 14 FALSE
## 15  TRUE
```

```
## 16 FALSE
## 17  TRUE
## 18 FALSE
## 19  TRUE
## 20 FALSE
```

```
# Méthode 2 : `[` comme une matrice
c2[, "var3"]
##  [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
## [12] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

```
# Méthode 3 : `[[`
c2[["var3"]]
##  [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
## [12] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

```
# Méthode 4 : `$`
c2$var3
##  [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
## [12] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

```
# Sélection des variables var2 et var1
c2[, c("var2", "var1")]
##    var2 var1
## 1     t    1
## 2     s    2
## 3     r    3
## 4     q    4
## 5     p    5
## 6     o    6
## 7     n    7
## 8     m    8
## 9     l    9
## 10    k   10
## 11    j   11
## 12    i   12
## 13    h   13
## 14    g   14
## 15    f   15
## 16    e   16
## 17    d   17
## 18    c   18
## 19    b   19
## 20    a   20
```

```r
# Sélection de toutes les variables sauf var2

# Note : l'idée est de se ramener au cas précédent
# en déterminant le vecteur des variables à conserver.
# Cette liste n'est rien d'autre que le vecteur des
# noms de variables de c2 sans la variable var2.
# La fonction setdiff() permet d'effectuer cette opération :
setdiff(names(c2), "var2")
## [1] "var1" "var3"

# Il ne reste plus qu'à utiliser cette structure dans `[`
c2[, setdiff(names(c2), "var2")]
##    var1  var3
## 1     1  TRUE
## 2     2 FALSE
## 3     3  TRUE
## 4     4 FALSE
## 5     5  TRUE
## 6     6 FALSE
## 7     7  TRUE
## 8     8 FALSE
## 9     9  TRUE
## 10   10 FALSE
## 11   11  TRUE
## 12   12 FALSE
## 13   13  TRUE
## 14   14 FALSE
## 15   15  TRUE
## 16   16 FALSE
## 17   17  TRUE
## 18   18 FALSE
## 19   19  TRUE
## 20   20 FALSE
```

f. Sélectionnez les observations pour lesquelles `var3` est fausse ; les observations pour lesquelles `var1` est impaire ; les observations pour lesquelles `var2` vaut d, e ou f.
Afficher/masquer la solution

```r
# La sélection des observations dans un data.frame est analogue
# à celle qui s'opère dans une matrice. En particulier, on peut
# avoir recours à un vecteur logique.

# Ainsi, la première condition demandée (var3 fausse) est évaluée
# par
```

```
c2$var3 == FALSE
##  [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
## [12]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
# Le vecteur logique correspondant peut donc être utilisé dans
# l'opérateur `[`
c2[c2$var3 == FALSE, ]
##    var1 var2  var3
## 2     2    s FALSE
## 4     4    q FALSE
## 6     6    o FALSE
## 8     8    m FALSE
## 10   10    k FALSE
## 12   12    i FALSE
## 14   14    g FALSE
## 16   16    e FALSE
## 18   18    c FALSE
## 20   20    a FALSE

# Les autres cas ne sont que des variations sur cette structure
c2[c2$var1 %% 2 == 1, ]
##    var1 var2 var3
## 1     1    t TRUE
## 3     3    r TRUE
## 5     5    p TRUE
## 7     7    n TRUE
## 9     9    l TRUE
## 11   11    j TRUE
## 13   13    h TRUE
## 15   15    f TRUE
## 17   17    d TRUE
## 19   19    b TRUE
c2[c2$var2 %in% c("d", "e", "f"), ]
##    var1 var2  var3
## 15   15    f  TRUE
## 16   16    e FALSE
## 17   17    d  TRUE
```

Savoir utiliser les fonctions `*apply`, `do.call()` et `Reduce()`

Le *package* `microbenchmark` est souvent utilisé dans cette partie et les suivantes pour mesurer la performance des des solutions testées :

```r
install.packages("microbenchmark")
library(microbenchmark)
```

Cas pratique 4 Fonctions et environnements

Tout ce qui se passe dans **R** correspond à un appel de fonction. Comprendre le fonctionnement des fonctions et savoir en créer soi-même est donc crucial.

a. Utilisez les guillemets simples inversés (AltGr + 7 sur le clavier azerty) pour afficher le code associé au signe `+`. Utilisez-le comme une fonction classique avec la syntaxe `nomFonction(parametre1, parametre2)`.
Afficher/masquer la solution

```r
# Pour afficher le code d'une fonction, il suffit
# de saisir son nom. Dans le cas des signes
# arithmétiques, il convient d'entourer le nom
# de guillemets inversés
`+`
## function (e1, e2)  .Primitive("+")

# En utilisant les guillemets inversés, on peut
# utiliser la fonction `+`() comme n'importe
# quelle fonction (en appelant ses arguments dans
# la parenthèse).
`+`(2, 2)
## [1] 4
```

b. Définissez la fonction `monCalcul(x, puissance)` qui pour un vecteur numérique `x` quelconque :

1. calcule sa somme ;
2. met la somme à la puissance `puissance`.

Dans un second temps, donnez à `puissance` la valeur `2` par défaut et faites en sorte que la fonction prenne en charge les vecteurs `x` présentant des valeurs manquantes `NA`.
Afficher/masquer la solution

```r
# Associée à `<-`, la fonction function()
# permet de définir une nouvelle fonction
monCalcul <- function(x, puissance){
  resultat <- sum(x)^puissance
  return(resultat)
```

```
}
monCalcul(1:3, puissance = 2)
## [1] 36

# La syntaxe suivante est équivalente :
monCalcul <- function(x, puissance) sum(x)^puissance
monCalcul(1:3, puissance = 2)
## [1] 36

# Pour ajouter un argument par défaut, il suffit
# de le spécifier dans la parenthèse de function()
monCalcul <- function(x, puissance = 2) sum(x)^puissance
monCalcul(1:3)
## [1] 36

# L'argument na.rm = TRUE de la fonction sum()
# permet d'exclure automatiquement les valeurs manquantes.
monCalcul <- function(x, puissance = 2) sum(x, na.rm = TRUE)^puissance
monCalcul(c(1:3, NA))
## [1] 36
```

c. Modifier la fonction pour ajouter une étape de vérification du type de de `x` : prévoyez un message d'erreur si `x` est de type `character` ou `factor`.
Afficher/masquer la solution

```
# Les fonctions is.character() et is.factor() permettent
# de tester le type de x.
monCalcul <- function(x, puissance = 2){
  if(is.character(x) || is.factor(x)) stop("x est de type caractère ou factor.")
  sum(x, na.rm = TRUE)^puissance
}
monCalcul(c("a", "c"))
## Error in monCalcul(c("a", "c")): x est de type caractère ou factor.
```

d. Quelle est la valeur de l'objet `T` ? Comment expliquez-vous que cet objet soit défini alors que vous ne l'avez pas vous-même créé (vous pouvez utilisez la fonction `getAnywhere()` pour répondre) ?
Afficher/masquer la solution

```
# L'objet T a par défaut la valeur TRUE
T
## [1] TRUE
```

```
# T est défini dans le package base de R comme un alias de TRUE
getAnywhere(T)
## A single object matching 'T' was found
## It was found in the following places
##   package:base
##   namespace:base
## with value
##
## [1] TRUE
base::T
## [1] TRUE
```

e. Soumettez le code `T <- 3`. Que vaut désormais l'objet `T` ? Pourquoi **R** n'accède-t-il plus à la valeur stockée par défaut (vous pouvez utiliser la fonction `search()` pour répondre) ? Comment accéder désormais à la valeur par défaut ? Que retenez-vous quant à l'utilisation de `T` et de `F` en lieu et place de `TRUE` et `FALSE` dans un code ?
Afficher/masquer la solution

```
T <- 3
T
## [1] 3
# R accède en premier lieu à l'environnement global, puis
# aux packages dans l'ordre de search()
search()
##  [1] ".GlobalEnv"             "package:microbenchmark"
##  [3] "package:stats"          "package:graphics"
##  [5] "package:grDevices"      "package:utils"
##  [7] "package:datasets"       "package:methods"
##  [9] "Autoloads"              "package:base"
# Le package base est situé en dernière position.

# Pour accéder explicitement à l'élément du package base,
# on utilise ::
base::T
## [1] TRUE

# Le fait que T et F ne soit que des alias pour TRUE et FALSE
# et donc que leur valeur soit modifiable invite à la plus
# grande prudence dans leur utilisation. Elle est en fait
# à proscrire dans la réalisation de projets importants
# (par exemple l'écriture d'un package).
```

Cas pratique 5 `apply()` : Appliquer une fonction selon les dimensions d'une matrice

a. Créez une matrice `e1` de 3 lignes et de 5 colonnes en utilisant la fonction `runif()`. Ses valeurs sont-elles identiques si vous la générez une seconde fois ? Comment faire pour que cela soit le cas ?
Afficher/masquer la solution

```
# Par défaut,  l'utilisation sucessive de la fonction
# runif() ne conduit pas aux mêmes résultats
e1 <- matrix(runif(15), ncol = 5)
e1
##           [,1]      [,2]       [,3]      [,4]      [,5]
## [1,] 0.1178220 0.1817914 0.98906768 0.2446630 0.5374668
## [2,] 0.7017961 0.5424616 0.05518202 0.5026829 0.6360702
## [3,] 0.5554569 0.7330166 0.07799456 0.5497169 0.3654879
e1b <- matrix(runif(15), ncol = 5)
e1b
##           [,1]       [,2]       [,3]      [,4]      [,5]
## [1,] 0.4671328 0.37910439 0.02767167 0.7825179 0.6929564
## [2,] 0.3456576 0.01836591 0.06224703 0.7743206 0.6739887
## [3,] 0.8499960 0.42726688 0.59305615 0.4324565 0.7560812
identical(e1, e1b)
## [1] FALSE

# La fonction set.seed() permet en revanche d'initialiser
# le générateur de nombres pseudo-aléatoires de R.
# Utilisé après la fonction set.seed(), deux fonctions
# runif() donneront toujours le même résultat.
set.seed(2016)
e1 <- matrix(runif(15), ncol = 5)
e1
##           [,1]      [,2]        [,3]       [,4]      [,5]
## [1,] 0.1801636 0.1335746 0.616631283 0.05345881 0.2225800
## [2,] 0.1429437 0.4775025 0.890547575 0.38868792 0.8765744
## [3,] 0.8416465 0.1212584 0.002623455 0.27295359 0.2466688
set.seed(2016)
e1b <- matrix(runif(15), ncol = 5)
e1b
##           [,1]      [,2]        [,3]       [,4]      [,5]
## [1,] 0.1801636 0.1335746 0.616631283 0.05345881 0.2225800
## [2,] 0.1429437 0.4775025 0.890547575 0.38868792 0.8765744
```

```
## [3,] 0.8416465 0.1212584 0.002623455 0.27295359 0.2466688
identical(e1, e1b)
## [1] TRUE
```

b. Utilisez la fonction `apply()` pour calculer la somme des termes de chaque ligne de `e1`. Comment auriez-vous pu faire autrement ? Utilisez la fonction `microbenchmark()` pour comparer ces deux solutions.

Afficher/masquer la solution

```
# La fonction apply() permet d'appliquer
# une même fonction aux lignes ou aux colonnes
# d'une matrice.
apply(e1, 1, sum)
## [1] 1.206408 2.776256 1.485151
# Le second argument indique la dimension selon
# laquelle appliquer la fonction (1 pour les lignes,
# 2 pour les colonnes).

# Cette opération est en fait nativement implémentée
# via la fonction rowSums()
rowSums(e1)
## [1] 1.206408 2.776256 1.485151

# La fonction microbenchmark() permet de comparer
# systématiquement ces deux stratégies (apply() ou
# rowSums())
microbenchmark(times = 1e4
  , apply = apply(e1, 1, sum)
  , rowSums = rowSums(e1)
)
## Unit: microseconds
##     expr    min     lq      mean median      uq      max neval
##    apply 14.878 16.482 19.302062 17.125 18.4445 1169.158 10000
##  rowSums  2.896  3.579  4.402847  3.976  4.3640 1820.644 10000
```

c. Utilisez la fonction `apply()` pour centrer-réduire toutes les colonnes de `e1` (*i.e.* leur soustraire leur moyenne puis les diviser par leur écart-type).

Afficher/masquer la solution

```
# Etape 1 : fonction pour centrer-réduire une variable
x <- runif(10)
mean(x) # Moyenne de x
```

```
## [1] 0.3157225
sd(x) # Ecart-type de x
## [1] 0.2328214
centrer_reduire <- function(x) ( x - mean(x) ) / sd(x)
z <- centrer_reduire(x)
mean(z)
## [1] 5.551115e-17
sd(z)
## [1] 1

# Etape 2 : utilisation dans un apply()
e2 <- apply(e1, 2, centrer_reduire)
colMeans(e2)
## [1]  0.000000e+00  3.700743e-17  1.110223e-16 -1.850372e-17  7.401487e-17
apply(e2, 2, sd)
## [1] 1 1 1 1 1

# Tout en une seule étape :
e3 <- apply(e1, 2, function(x) ( x - mean(x) ) / sd(x))
identical(e2, e3)
## [1] TRUE
```

Cas pratique 6 `lapply()` et `sapply()` : Appliquer une fonction aux éléments d'un vecteur, d'une liste ou aux colonnes d'un data.frame

a. On définit le vecteur `f1` par `f1 <- 5:15`. Utilisez la fonction `sapply()` pour calculer la somme cumulée des éléments de `f1`. Quelle(s) alternative(s) envisageriez-vous ? Utilisez la fonction `microbenchmark()` pour comparer ces solutions.
Afficher/masquer la solution

```
# INDICATION : commencer par définir la fonction
# sumfirst(x, i) qui calcule la somme des i premiers
# éléments de x, puis utilisez-la dans un sapply().

f1 <- 5:15

# Méthode 1 : sapply()
# L'objectif est d'obtenir un vecteur de longueur length(f1)
# qui pour chaque élément i renvoie la somme des éléments 1 à i
# de f1
```

```r
# On commence par définir la fonction sumfirst(x, i) :
sumfirst <- function(x, i) sum(x[1:i])
sumfirst(f1, 1)
## [1] 5
sumfirst(f1, 2)
## [1] 11

# Il ne reste plus qu'à appliquer sumfirst() pour chaque
# indice de f1.
sapply(1:length(f1), function(i) sumfirst(i, x = f1))
##  [1]   5  11  18  26  35  45  56  68  81  95 110
# ou encore
sapply(seq_along(f1), sumfirst, x = f1)
##  [1]   5  11  18  26  35  45  56  68  81  95 110
# ou en une seule étape
sapply(seq_along(f1), function(i) sum(f1[1:i]))
##  [1]   5  11  18  26  35  45  56  68  81  95 110

# Méthode 2 : cumsum()
# La fonction cumsum() effectue nativement cette opération
cumsum(f1)
##  [1]   5  11  18  26  35  45  56  68  81  95 110

# Comparaison avec microbenchmark()
microbenchmark(times = 1e4
  , sapply = sapply(seq_along(f1), function(i) sum(f1[1:i]))
  , cumsum = cumsum(f1)
)
## Unit: nanoseconds
##    expr   min      lq       mean median      uq     max neval
##  sapply 21533 23425.5 26412.3013  24821 25995.5 1426711 10000
##  cumsum   163   204.0   316.6458    290   371.0   21418 10000
```

b. On définit la liste `f2` par

```r
f2 <- list(
  sample.int(26, 10, replace = TRUE)
  , sample.int(26, 100, replace = TRUE)
  , sample.int(26, 1000, replace = TRUE)
)
```

Utilisez les fonctions `lapply()` ou `sapply()` pour :

— retrouver la longueur de chacun des éléments de `f2` ;

— extraire les 5 premiers éléments de chacun des éléments de `f2` ;
— remplacer chaque élément de `f2` par le vecteur de lettres (en minuscules) dont il représente les positions dans l'alphabet.
Afficher/masquer la solution

```
# Les fonctions sapply() et lapply() permettent
# d'appliquer systématiquement une fonction aux éléments
# d'une liste.

# Pour connaître la longueur de chaque élément de f2,
# il suffit ainsi d'appliquer à chacun la fonction length()
lapply(f2, length)
## [[1]]
## [1] 10
##
## [[2]]
## [1] 100
##
## [[3]]
## [1] 1000

# sapply() effectue exactement le même traitement que
# lapply() mais essaie en plus de simplifier le résultat
# si c'est possible
sapply(f2, length)
## [1]   10  100 1000
# Ici c'est le cas : on passe d'une liste avec lapply()
# à un vecteur avec sapply()

# Pour extraire les éléments 1 à 10 de chacun des éléments
# de f2, on utilise l'opérateur [
lapply(f2, function(x) x[1:10])
## [[1]]
##  [1]  2 17 13 12 25 25 17 16  4 20
##
## [[2]]
##  [1]  4  3  4  9 11  8  7  9 10 17
##
## [[3]]
##  [1] 18  5 15  8 14  6  7 13 26  2
# On aurait aussi pu soumettre de façon équivalente
lapply(f2, `[`, 1:10)
## [[1]]
##  [1]  2 17 13 12 25 25 17 16  4 20
##
```

```
## [[2]]
##  [1]  4  3  4  9 11  8  7  9 10 17
##
## [[3]]
##  [1] 18  5 15  8 14  6  7 13 26  2
# car x[1:10] est équivalent à `[`(x, 1:10)

# Remplacer les éléments d'un des éléments de f2
# par les lettres dont il représente la position
# est relativement simple :
letters[f2[[1]]]
##  [1] "b" "q" "m" "l" "y" "y" "q" "p" "d" "t"
# On reprend donc le principe de la sous-question
# précédente dans un lapply()
# lapply(f2, function(x) letters[x])
# (commenté ici pour ne pas saturer la sortie).
```

c. On définit le `data.frame` `f3` par

```
set.seed(1)
f3 <- data.frame(
  id = letters[1:20]
  , by = rep(letters[1:5], times = 4)
  , matrix(runif(100), ncol = 5)
  , stringsAsFactors = FALSE
)
```

Utilisez les fonctions `lapply()` ou `sapply()` pour :

— déterminer le type de chacune des variables de `f3` ;
— calculer la moyenne de toutes les variables numériques de `f3` ;
— convertir toutes les variables de type `character` en facteurs.
Afficher/masquer la solution

```
# Un data.frame étant un cas particulier de liste,
# on peut lui appliquer les fonctions lapply() et
# sapply() exactement comme on le ferait pour une liste.

# Pour connaître le type de chacun de ses éléments,
# on leur applique donc systématiquement la fonction
# typeof()
sapply(f3, typeof)
##          id          by          X1          X2          X3          X4
## "character" "character"    "double"    "double"    "double"    "double"
```

```
##          X5
##    "double"

# Pour appliquer une même fonction à toutes ses variables
# de type numériques, on commence par les identifier
# avec un is.numeric() dans un sapply()
num <- sapply(f3, is.numeric)
num
##    id    by    X1    X2    X3    X4    X5
## FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
# Puis on leur applique la fonction désirée
sapply(f3[num], mean)
##        X1        X2        X3        X4        X5
## 0.5551671 0.4738822 0.5072134 0.5665520 0.4864206

# De même pour les variables de type caractère
char <- sapply(f3, is.character)
char
##    id    by    X1    X2    X3    X4    X5
##  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
f3[char] <- lapply(f3[char], as.factor)
str(f3)
## 'data.frame':    20 obs. of  7 variables:
##  $ id: Factor w/ 20 levels "a","b","c","d",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ by: Factor w/ 5 levels "a","b","c","d",..: 1 2 3 4 5 1 2 3 4 5 ...
##  $ X1: num  0.266 0.372 0.573 0.908 0.202 ...
##  $ X2: num  0.935 0.212 0.652 0.126 0.267 ...
##  $ X3: num  0.821 0.647 0.783 0.553 0.53 ...
##  $ X4: num  0.913 0.294 0.459 0.332 0.651 ...
##  $ X5: num  0.435 0.713 0.4 0.325 0.757 ...
```

Cas pratique 7 `tapply()` : Appliquer une fonction selon les modalités d'un vecteur

a. On définit le vecteur `g1 <- sample(20)` et le vecteur `g2 <- rep(c("H", "F"), times = 10)`. Utilisez `tapply()` pour calculer la moyenne de `g1` selon les groupes définis par `g2`.
Afficher/masquer la solution

```
set.seed(2)
g1 <- sample(20)
```

```r
g2 <- rep(c("H", "F"), times = 10)

# La fonction tapply() permet d'appliquer une fonction
# à un vecteur selon les modalités d'un autre vecteur
tapply(g1, g2, mean)
##    F    H
## 10.7 10.3
```

b. On repart du `data.frame` `f3` du cas pratique précédent. Utilisez `tapply()` pour calculer le total de la variable `X1` selon les modalités de la variable `by`.
Afficher/masquer la solution

```r
# Il suffit d'utiliser tapply() sur les variables de f3
tapply(f3$X1, f3$by, sum)
##        a        b        c        d        e
## 1.867572 2.210974 2.912580 2.301461 1.810755
```

c. Combinez la fonction `split()` avec `sapply()` pour obtenir le même résultat. Comment calculeriez-vous le total de toutes les variables numériques de `f3` selon les modalités de la variable `by` ?
Afficher/masquer la solution

```r
# La fonction split() éclate en une liste un vecteur
# ou un data.frame selon les modalités d'une ou plusieurs
# variables.
f3[,c("by", "X1")]
##    by         X1
## 1   a 0.26550866
## 2   b 0.37212390
## 3   c 0.57285336
## 4   d 0.90820779
## 5   e 0.20168193
## 6   a 0.89838968
## 7   b 0.94467527
## 8   c 0.66079779
## 9   d 0.62911404
## 10  e 0.06178627
## 11  a 0.20597457
## 12  b 0.17655675
## 13  c 0.68702285
## 14  d 0.38410372
## 15  e 0.76984142
```

```
## 16  a 0.49769924
## 17  b 0.71761851
## 18  c 0.99190609
## 19  d 0.38003518
## 20  e 0.77744522
split(f3$X1, f3$by)
## $a
## [1] 0.2655087 0.8983897 0.2059746 0.4976992
##
## $b
## [1] 0.3721239 0.9446753 0.1765568 0.7176185
##
## $c
## [1] 0.5728534 0.6607978 0.6870228 0.9919061
##
## $d
## [1] 0.9082078 0.6291140 0.3841037 0.3800352
##
## $e
## [1] 0.20168193 0.06178627 0.76984142 0.77744522

# Ce faisant, on peut avec une fonction lapply()
# ou sapply() reproduire le comportent de la fonction
# tapply()
sapply(split(f3$X1, f3$by), sum)
##        a        b        c        d        e
## 1.867572 2.210974 2.912580 2.301461 1.810755

# L'avantage de cette technique est qu'elle
# permet d'appliquer le même type de traitement
# non à une seule variable mais à plusieurs
# d'un seul coup
sapply(split(f3[num], f3$by), function(x){
  sapply(x, sum)
})
##           a        b        c        d        e
## X1 1.867572 2.210974 2.912580 2.301461 1.810755
## X2 2.471366 1.619339 1.635547 1.905174 1.846217
## X3 2.187388 1.847873 2.216894 2.192152 1.699960
## X4 2.402164 2.475928 1.962049 1.527737 2.963161
## X5 1.674290 1.937845 1.574059 2.257980 2.284239
# ... ou de façon équivalente (mais un peu difficile
# à relire !)
sapply(split(f3[num], f3$by), sapply, sum)
##           a        b        c        d        e
```

```
## X1 1.867572 2.210974 2.912580 2.301461 1.810755
## X2 2.471366 1.619339 1.635547 1.905174 1.846217
## X3 2.187388 1.847873 2.216894 2.192152 1.699960
## X4 2.402164 2.475928 1.962049 1.527737 2.963161
## X5 1.674290 1.937845 1.574059 2.257980 2.284239
# ... ou plus efficacement avec colSums()
sapply(split(f3[num], f3$by), colSums)
##           a        b        c        d        e
## X1 1.867572 2.210974 2.912580 2.301461 1.810755
## X2 2.471366 1.619339 1.635547 1.905174 1.846217
## X3 2.187388 1.847873 2.216894 2.192152 1.699960
## X4 2.402164 2.475928 1.962049 1.527737 2.963161
## X5 1.674290 1.937845 1.574059 2.257980 2.284239
```

Cas pratique 8 `do.call()` : Appliquer une fonction simultanément à l'ensemble des éléments d'une liste

De nombreuses fonctions peuvent porter sur un nombre indéterminé d'éléments : `c()`, `sum()`, `rbind()`, etc. Pour les appliquer à l'ensemble des éléments d'une liste sans avoir à tous les écrire un à un, il suffit d'utiliser `do.call()`.

a. On définit la liste `h1 <- list(1:5, 6:10, 11:15)`. Que se passe-t-il si vous soumettez `sum(h1)` ? Utilisez `do.call()` pour sommer l'ensemble des éléments de `h1`. Comparez avec le résultat de `sapply(h1, sum)`.
Afficher/masquer la solution

```
h1 <- list(1:5, 6:10, 11:15)

# La fonction sum() ne peut pas porter sur une liste
sum(h1)
## Error in sum(h1): 'type' (list) de l'argument incorrect

# En revanche, il est possible d'appliquer la fonction
# sum() à l'ensemble des éléments de h1
sum(h1[[1]], h1[[2]], h1[[3]])
## [1] 120

# Pour automatiser cette expression, on peut utiliser
# do.call()
do.call(sum, h1)
## [1] 120
```

```
# A l'inverse, sapply(h1, sum) applique la fonction
# sum() à chaque élément de h1 :
sapply(h1, sum)
## [1] 15 40 65
```

b. Réunissez les éléments de `h1` en un seul vecteur avec la fonction `base::c()`. Comparez avec le résultat de `lapply(h1, base::c)`
Afficher/masquer la solution

```
# La fonction base::c() permet de concaténer
# les vecteurs qui constituent h1
c(h1[[1]], h1[[2]], h1[[3]])
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15

# Pour automatiser cette expression, on peut utiliser
# do.call()
do.call(base::c, h1)
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
# Note : on utilise ici base::c pour bien indiquer que
# c'est la fonction c() du package base que l'on
# souhaite utiliser. En effet, si l'on avait défini
# on objet c dans l'environnement global il y aurait
# eu un conflit de noms.

# lapply(h1, base::c) applique la fonction base::c()
# à chaque élément de h1, ce qui ne change rien
lapply(h1, base::c)
## [[1]]
## [1] 1 2 3 4 5
##
## [[2]]
## [1]  6  7  8  9 10
##
## [[3]]
## [1] 11 12 13 14 15
```

c. On définit la liste de matrices `h2`

```
h2 <- list(
  matrix(1:6, nrow = 2)
  , matrix(7:12, nrow = 2)
```

```
  , matrix(13:18, nrow = 2)
)
```

Utilisez `do.call()` avec les fonctions `rbind()` et `cbind()` pour concaténer l'ensemble des éléments de `h2` en ligne ou en colonne respectivement.
Afficher/masquer la solution

```
# Pour concaténer les éléments de h2 en ligne,
# on utilise rbind()
rbind(h2[[1]], h2[[2]], h2[[3]])
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
## [3,]    7    9   11
## [4,]    8   10   12
## [5,]   13   15   17
## [6,]   14   16   18

# Pour automatiser cette expression, on peut utiliser
# do.call()
do.call(rbind, h2)
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
## [3,]    7    9   11
## [4,]    8   10   12
## [5,]   13   15   17
## [6,]   14   16   18

# De même en colonne avec cbind()
do.call(cbind, h2)
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]    1    3    5    7    9   11   13   15   17
## [2,]    2    4    6    8   10   12   14   16   18
```

Cas pratique 9 `Reduce()` : Appliquer une fonction successivement à l'ensemble des éléments d'une liste

De nombreuses fonctions portent sur deux éléments précisément : opérations arithmétiques, `merge()`, etc. Pour les appliquer successivement à l'ensemble des éléments d'une liste, il suffit d'utiliser `Reduce()`.

a. On repart de la liste `h1` du cas pratique précédent. Observez le résultat de `Reduce('+', h1)` : comment le comprenez-vous ? Comparez avec `sapply(h1, sum)`.

Afficher/masquer la solution

```
# Reduce(f, x) applique la fonction f() aux éléments de x
# 2 à 2 et successivement.
Reduce(`+`, h1)
## [1] 18 21 24 27 30
# Ici, cette expression est équivalente à
( h1[[1]] + h1[[2]] ) + h1[[3]]
## [1] 18 21 24 27 30
# Le vecteur obtenu est la somme terme à terme de
# tous les éléments qui composent h1

# A l'inverse, sapply(h1, sum) applique la fonction
# sum() à chaque élément de h1 :
sapply(h1, sum)
## [1] 15 40 65
```

b. On définit la liste de `data.frame` `i1`

```
i1 <- list(
  data.frame(id = letters[1:4], var1 = 1:4, stringsAsFactors = FALSE)
  , data.frame(id = letters[2:5], var2 = 5:8, stringsAsFactors = FALSE)
  , data.frame(id = letters[3:6], var3 = 9:12, stringsAsFactors = FALSE)
)
```

Utilisez `Reduce()` pour fusionner l'ensemble des éléments de `i1` selon la variable `id`. Comment ajusteriez-vous ce code pour pouvoir utiliser l'option `all = TRUE` de la fonction `merge()` ?

Afficher/masquer la solution

```
# La fonction Reduce() est particulièrement utile
# pour fusionner de nombreux data.frame selon la
# même variable. Par défaut, la fonction merge()
# fusionne les data.frame sur les variables qu'ils
# ont en commun.
Reduce(merge, i1)
##   id var1 var2 var3
## 1  c    3    6    9
## 2  d    4    7   10
# Ce code fournit le résultat désiré.
```

```r
# Pour spécifier explicitement la variable de fusion,
# il faut redéfinir à la volée la fonction à appliquer
# dans le Reduce() (comme dans un *apply) :
Reduce(function(x, y) merge(x, y, by = "id"), i1)
##   id var1 var2 var3
## 1  c    3    6    9
## 2  d    4    7   10

# De même pour ajouter d'autres options à la fonction
# merge()
Reduce(function(x, y) merge(x, y, by = "id", all = TRUE), i1)
##   id var1 var2 var3
## 1  a    1   NA   NA
## 2  b    2    5   NA
## 3  c    3    6    9
## 4  d    4    7   10
## 5  e   NA    8   11
## 6  f   NA   NA   12
```

Travailler efficacement sur des données avec base R

L'ensemble des cas pratiques qui suivent portent sur les données de l'enquête Emploi en continu stockées dans le fichier `eect4.rds`. La fonction `readRDS()` permet de le charger en mémoire :

```r
setwd("U:/R_perfectionnement")
eec <- readRDS("eect4.rds")
str(eec)
```

```
## 'data.frame':    34913 obs. of  19 variables:
##  $ IDENT    : chr  "G0A56JP6" "G0A56JP6" "G0A56JR6" "G0A56JS6" ...
##  $ TRIM     : chr  "4" "4" "4" "4" ...
##  $ NOI      : chr  "01" "02" "01" "01" ...
##  $ REG      : chr  "11" "11" "11" "11" ...
##  $ AGE      : chr  "66" "29" "27" "29" ...
##  $ SEXE     : chr  "2" "1" "2" "2" ...
##  $ CSE      : chr  "56" "81" "38" "37" ...
##  $ DIP11    : chr  "71" "42" "10" "11" ...
##  $ ACTEU    : chr  "1" "2" "1" "1" ...
##  $ SALRED   : int  596 NA 2700 2666 11967 NA 2000 2800 2333 3500 ...
##  $ STC      : chr  "2" NA "2" "2" ...
##  $ TAM1D    : chr  NA NA NA NA ...
```

```
##  $ AIDREF   : chr  NA "5" NA NA ...
##  $ TPP      : chr  "1" NA "1" "1" ...
##  $ NBAGENF  : chr  "0" "0" "0" "0" ...
##  $ DUHAB    : chr  "7" NA "7" "7" ...
##  $ PUB3FP   : chr  "4" NA "4" "4" ...
##  $ NAIA     : chr  "1946" "1983" "1985" "1983" ...
##  $ EXTRI1613: num  1777 1777 2045 1898 1754 ...
```

Cas pratique 10 Sélection d'observations

a. Utilisez l'opérateur `[` pour sélectionner l'individu appartenant au ménage dont l'`IDENT` est `"GFO5NVUE"` et dont le numéro d'ordre `NOI` est `"02"`.
Afficher/masquer la solution

```
# Il suffit de créer le vecteur logique correspondant
# et de l'utiliser das `[`
eec[eec$IDENT == "GFO5NVUE" & eec$NOI == "02", ]
##           IDENT TRIM NOI REG AGE SEXE CSE DIP11 ACTEU SALRED STC TAM1D
## 360720 GFO5NVUE    4  02  42  56    1  63    50     1   2004   2  <NA>
##        AIDREF TPP NBAGENF DUHAB PUB3FP NAIA EXTRI1613
## 360720   <NA>   1       0     6      4 1956  990.2715
```

b. Cherchez de la documentation sur la fonction `subset()`. Comparez ses performances avec celles de l'opérateur `[` à l'aide de la fonction `microbenchmark()`.
Afficher/masquer la solution

```
# subset() permet d'évaluer une clause logique
# sans avoir à répéter le nom du data.frame
# d'origine
subset(eec, IDENT == "GFO5NVUE" & NOI == "02")
##           IDENT TRIM NOI REG AGE SEXE CSE DIP11 ACTEU SALRED STC TAM1D
## 360720 GFO5NVUE    4  02  42  56    1  63    50     1   2004   2  <NA>
##        AIDREF TPP NBAGENF DUHAB PUB3FP NAIA EXTRI1613
## 360720   <NA>   1       0     6      4 1956  990.2715

# Comparaison des performances
microbenchmark(times = 1e2
  , "[" = eec[eec$IDENT == "GFO5NVUE" & eec$NOI == "02", ]
  , subset = subset(eec, IDENT == "GFO5NVUE" & NOI == "02")
)
## Unit: milliseconds
```

```
##    expr      min       lq     mean   median       uq      max neval
##       [ 4.080503 4.172534 4.958683 4.249307 5.987093 6.476791   100
##  subset 4.975475 5.047462 6.104903 6.789003 6.888070 8.092015   100
```

c. Concaténez les valeurs des variables `IDENT` et `NOI` et utilisez le résultat comme noms de ligne. Retrouvez alors l'individu de la question a à l'aide de son nom de ligne. Comparez les performances de cette méthode avec celles de l'opérateur `[`.
Afficher/masquer la solution

```
# La foncton paste0() permet de concaténer des chaînes
# de caractères sans utiliser de délimiteur
row.names(eec) <- paste0(eec$IDENT, eec$NOI)

# Sélection de l'individu par son nom de ligne
eec["GFO5NVUE02", ]
##               IDENT TRIM NOI REG AGE SEXE CSE DIP11 ACTEU SALRED STC TAM1D
## GFO5NVUE02 GFO5NVUE    4  02  42  56    1  63    50     1   2004   2  <NA>
##            AIDREF TPP NBAGENF DUHAB PUB3FP NAIA EXTRI1613
## GFO5NVUE02   <NA>   1       0     6      4 1956  990.2715

# Comparaison des performances
microbenchmark(times = 1e2
  , "[" = eec[eec$IDENT == "GFO5NVUE" & eec$NOI == "02", ]
  , names = eec["GFO5NVUE02", ]
)
## Unit: microseconds
##   expr      min       lq      mean   median       uq       max neval
##      [ 4090.764 4154.567 5887.5253 4303.185 6969.056 50965.131   100
##  names  578.838  630.122  909.5403  853.793 1015.394  3351.487   100
```

Cas pratique 11 Création de variables

On souhaite recoder la variable `AGE` en trois modalités : 15-30 ans, 31-60 ans et plus de 60 ans. On part du code suivant :

```
for(i in 1:nrow(eec)) eec$trage1[i] <- if(as.numeric(eec$AGE[i]) < 31) "15-30" else if(a
```

a. Mesurez le temps d'exécution du code proposé. Que pensez-vous de cette syntaxe ?
Afficher/masquer la solution

```
# Mesure du temps d'exécution de cette première version
microbenchmark(times = 1
  , v1 = {
for(i in 1:nrow(eec)) eec$trage1[i] <- if(as.numeric(eec$AGE[i]) < 31) "15-30" else
  }
)
## Unit: seconds
##  expr      min       lq     mean   median       uq      max neval
##    v1 9.370603 9.370603 9.370603 9.370603 9.370603 9.370603     1
# Note : on ne fait qu'une seule itération car le
# temps d'exécution est très long (plusieurs secondes !).

# Cette syntaxe présente un énorme problème : elle utilise
# une boucle là où des opérations vectorisées beaucoup
# plus rapides sont disponibles.
```

b. On propose une seconde version du code :

```
eec$trage2 <- ifelse(as.numeric(eec$AGE) < 31, "15-30", ifelse(
  as.numeric(eec$AGE) < 61, "31-60", "61 et +"
))
```

Vérifiez que le résultat est identique à la proposition initiale (par exemple avec `identical()`ou `all.equal()`) et mesurez le temps temps d'exécution de cette deuxième version. Êtes-vous surpris du gain de performances ?
Afficher/masquer la solution

```
eec$trage2 <- ifelse(as.numeric(eec$AGE) < 31, "15-30", ifelse(
  as.numeric(eec$AGE) < 61, "31-60", "61 et +"
))

# Vérification de la correspondance
# avec la première version
all.equal(eec$trage1, eec$trage2)
## [1] TRUE

# Comparaison des performances
microbenchmark(times = 1e1
  , v2 = {
eec$trage2 <- ifelse(as.numeric(eec$AGE) < 31, "15-30", ifelse(
  as.numeric(eec$AGE) < 61, "31-60", "61 et +"
))
  }
)
```

```
## Unit: milliseconds
##  expr      min       lq     mean  median       uq      max neval
##    v2 31.74871 32.25048 38.20883 32.5345 34.82258 81.17398    10

# On n'est pas surpris du gain de performance : en R,
# les boucles sont beaucoup moins efficaces que les
# opérations vectorisées (qui reposent sur des
# boucles dans des langages de plus bas niveau).
```

c. Comment recoderiez-vous la proposition précédente pour ne plus faire appel à la fonction `ifelse()` ? Cela est-il susceptible d'améliorer les performances ? Mettez en oeuvre cette solution et mesurez-en les performances.
Afficher/masquer la solution

```
# Il est possible de se passer de ifelse() en utilisant
# l'opérateur [ pour effectuer des remplacements successifs.
eec$trage3 <- "15-30"
eec$trage3[as.numeric(eec$AGE) > 30 & as.numeric(eec$AGE) < 61] <- "31-60"
eec$trage3[as.numeric(eec$AGE) > 60] <- "61 et +"

# Vérification de la correspondance
# avec la première version
all.equal(eec$trage1, eec$trage3)
## [1] TRUE

# Comparaison des performances
microbenchmark(times = 1e1
  , v2 = {
eec$trage2 <- ifelse(as.numeric(eec$AGE) < 31, "15-30", ifelse(
  as.numeric(eec$AGE) < 61, "31-60", "61 et +"
))
  }
  , v3 = {
eec$trage3 <- "15-30"
eec$trage3[as.numeric(eec$AGE) > 30 & as.numeric(eec$AGE) < 61] <- "31-60"
eec$trage3[as.numeric(eec$AGE) > 60] <- "61 et +"
  }
)
## Unit: milliseconds
##  expr      min       lq     mean   median       uq      max neval
##    v2 29.46915 32.35999 32.30286 32.56482 32.73211 33.04035    10
##    v3 11.56582 11.60436 13.14490 11.89369 15.25443 15.46838    10
```

d. Combien de fois appelez-vous la fonction `as.numeric()` dans le code qui précède ? Proposez une dernière version qui minimise les opérations effectuées par **R** et synthétisez le gain en termes de performances.
Afficher/masquer la solution

```
# La fonction as.numeric() est appelée trois fois dans
# le code qui précède. Pour gagner du temps, on
# crée la variable t temporaire que l'on utilise
# en lieu et place de as.numeric(eec$AGE)
t <- as.numeric(eec$AGE)
eec$trage4 <- "15-30"
eec$trage4[t > 30 & t < 61] <- "31-60"
eec$trage4[t > 60] <- "61 et +"

# Vérification de la correspondance
# avec la première version
all.equal(eec$trage1, eec$trage4)
## [1] TRUE


# Comparaison des performances
microbenchmark(times = 10
  , v2 = {
eec$trage2 <- ifelse(as.numeric(eec$AGE) < 31, "15-30", ifelse(
  as.numeric(eec$AGE) < 61, "31-60", "61 et +"
))
  }
  , v3 = {
eec$trage3 <- "15-30"
eec$trage3[as.numeric(eec$AGE) > 30 & as.numeric(eec$AGE) < 61] <- "31-60"
eec$trage3[as.numeric(eec$AGE) > 60] <- "61 et +"
  }
  , v4 = {
t <- as.numeric(eec$AGE)
eec$trage4 <- "15-30"
eec$trage4[t > 30 & t < 61] <- "31-60"
eec$trage4[t > 60] <- "61 et +"
  }
)
## Unit: milliseconds
##  expr       min        lq      mean    median       uq       max neval
##    v2 32.329528 32.507772 38.123102 32.791472 35.89258 78.551367    10
##    v3 11.599261 11.731208 12.850544 11.855953 15.26732 15.536842    10
##    v4  5.736214  5.790105  6.564499  5.842058  6.00878  9.551809    10
```

```r
# Le passage de la version 1 à la version 4 du code s'est
# donc traduit par un gain en termes de performances
# de l'ordre d'un facteur 1 000.
```

Cas pratique 12 Agrégation par groupes : salaire moyen par région

On cherche à calculer le plus efficacement possible le salaire moyen (variable `SALRED`) par région (variable `REG`), d'abord sans pondérer puis en pondérant par le poids de sondage `EXTRI1613`.

a. Comparez les performances des fonctions `aggregate()`, `by()`, `sapply()` (avec `split()`) et `tapply()` pour le calcul du salaire moyen **non-pondéré** par région.
Afficher/masquer la solution

```r
# INDICATION : consultez l'aide de chacune de ces fonctions
# pour vous approprier leur syntaxe.

# On adapte à chaque fonction la syntaxe à mettre en oeuvre
# et on compare leurs performances
microbenchmark(times = 10
  , aggregate = aggregate(eec$SALRED, list(eec$REG), mean, na.rm = TRUE)
  , by = by(eec$SALRED, eec$REG, mean, na.rm = TRUE)
  , sapply = sapply(split(eec$SALRED, eec$REG), mean, na.rm = TRUE)
  , tapply = tapply(eec$SALRED, eec$REG, mean, na.rm = TRUE)
)
## Unit: milliseconds
##       expr       min        lq      mean    median        uq       max
##  aggregate 39.292281 39.364568 40.153831 39.517682 39.930893 43.819960
##         by  4.592239  4.625098  5.829585  5.014660  8.133503  8.227018
##     sapply  2.639614  2.683146  3.789832  2.829214  6.117389  6.339385
##     tapply  3.820977  3.857704  5.656975  5.746443  7.380845  7.472033
##  neval
##     10
##     10
##     10
##     10
# En règle générale sapply() et tapply() conduisent aux meilleures
# performances et sont proches l'une de l'autre.
```

b. Quelles pistes envisageriez-vous pour améliorer encore les performances dans ce type de situation ?
Afficher/masquer la solution

```
# Plusieurs pistes sont envisageables :
# - utiliser des manipulations purement vectorielles ;
# - utiliser des manipulations matricielles sur des matrices lacunaires avec le pac
# - paralléliser l'exécution avec le package `parallel`;
# - coder la fonction en C++.
```

c. Utilisez la fonction `sapply()` pour calculer le salaire moyen **pondéré** (par le poids de sondage `EXTRI1613`) par région. Y parvenez-vous également avec `tapply()` ?
Afficher/masquer la solution

```
# INDICATION 1 : appliquez la fonction split() à un objet
# contenant le salaire et le poids de sondage pour
# pouvoir utiliser les deux variables dans le *apply().
# INDICATION 2 : vous pouvez calculer la moyenne pondérée
# "à la main" ou utiliser la fonction weighted.mean()

# Par rapport à la question a., la principale différence
# vient du fait que l'on a besoin de plusieurs éléments
# dans chaque bloc éclaté par la fonction split() :
# le salaire d'une part, le poids de sondage d'autre part.

# Avec la fonction weighted.mean()
sapply(
  split(eec[c("SALRED", "EXTRI1613")], eec$REG)
  , function(x) weighted.mean(x$SALRED, x$EXTRI1613, na.rm = TRUE)
)

# Manuellement en pensant bien à exclure les poids de sondage
# des individus pour lesquels SALRED est NA
sapply(
  split(eec[c("SALRED", "EXTRI1613")], eec$REG)
  , function(x) sum(x$SALRED * x$EXTRI1613, na.rm = TRUE) / sum((!is.na(x$SALRED))
)
```

Cas pratique 13 Fusion de tables : nombre d'individus au chômage par ménage

L'objectif de ce cas pratique est de créer, dans la table `eec`, une variable indiquant pour chaque individu le nombre d'individus au chômage dans son ménage. La position sur le marché du travail est codée par la variable `ACTEU` (`ACTEU == "2"` correspond au chômage) et l'identifiant du ménage est la variable `IDENT` (les individus d'un même ménage ont la même valeur pour la variable `IDENT`).

a. Utilisez la fonction `tapply()` pour déterminer le nombre de personnes au chômage dans chaque ménage et stockez cette information dans un objet appelé `nbcho`. Quelles sont ses caractéristiques ?

Afficher/masquer la solution

```
# Il suffit d'appliquer la fonction sum() par ménage
# à l'indicatrice de chômage eec$ACTEU == 2
nbcho <- tapply(eec$ACTEU == "2", eec$IDENT, sum, na.rm = TRUE)

# Caractéristiques de nbcho
str(nbcho)
##  int [1:18903(1d)] 1 0 0 0 0 0 0 0 0 0 ...
##  - attr(*, "dimnames")=List of 1
##   ..$ : chr [1:18903] "G0A56JP6" "G0A56JR6" "G0A56JS6" "G0A56JT6" ...
# nbcho est un vecteur nommé dont les noms sont
# les identifiants de ménage.
```

b. Utilisez la fonction `merge()` pour refusionner le résultat de la question précédente avec la table `eec` et créer la variable `nbcho`.

Afficher/masquer la solution

```
# Etape 1 : constituer un data.frame à partir de nbcho
# avec IDENT comme identifiant
nbcho_df <- data.frame(IDENT = names(nbcho), nbcho1 = nbcho)

# Etape 2 : fusionner eec et nbcho_df par IDENT
eec <- merge(eec, nbcho_df, by = "IDENT")
```

c. Utilisez habilement les noms de vecteur et l'opérateur `[` pour reproduire plus efficacement le résultat de la question b (toujours en repartant de `nbcho`). Vérifiez que la variable créée est bien identique.

Afficher/masquer la solution

```
# INDICATION : essayer de sélectionner par leur nom
# les valeurs de nbcho correspondant aux 10 premières observations
```

```
# de eec

# Le point essentiel est de remarquer que l'on peut
# utiliser les noms pour réarranger les valeurs
# de nbcho de sorte à ce qu'elles correspondent
# à l'ordre de eec

# Identifiant des 10 premières observations de eec
eec$IDENT[1:10]
##  [1] "G0A56JP6" "G0A56JP6" "G0A56JR6" "G0A56JS6" "G0A56JT6" "G0A56JU6"
##  [7] "G0A56JV6" "G0A56JV6" "G0A56JW6" "G0A56JX6"

# Pour extraire les valeurs de nbcho correspondantes,
# il suffit d'exploiter le fait que nbcho soit
# nommé à l'aide des identifiants de ménage.

# Par exemple :
# - Valeur de nbcho pour le premier ménage de eec
nbcho["G0A56JP6"]
## G0A56JP6
##        1
# - Valeur de nbcho pour les quatre premiers ménages
# de eec
nbcho[c("G0A56JP6", "G0A56JP6", "G0A56JR6", "G0A56JS6")]
## G0A56JP6 G0A56JP6 G0A56JR6 G0A56JS6
##        1        1        0        0
# - Valeur de nbcho pour les 10 premiers ménages
# de eec
nbcho[eec$IDENT[1:10]]
## G0A56JP6 G0A56JP6 G0A56JR6 G0A56JS6 G0A56JT6 G0A56JU6 G0A56JV6 G0A56JV6
##        1        1        0        0        0        0        0        0
## G0A56JW6 G0A56JX6
##        0        0

# On peut donc par ce biais reconstituer l'intégralité
# du vecteur correspondant aux observations de eec.
eec$nbcho2 <- nbcho[eec$IDENT]

# Ce vecteur est bien identique à celui obtenu par fusion
# à la question précédente.
all.equal(eec$nbcho1, eec$nbcho2)
## [1] TRUE
```

d. Comparez la syntaxe et les performances des méthodes mises en oeuvre aux question

b. et c.
Afficher/masquer la solution

```
# La syntaxe de la deuxième option est plus concise
# mais aussi plus complexe pour un relecteur moins
# averti des fonctionnalités de R en matière d'utilisation
# des noms de vecteur.

# Comparaison des performances
microbenchmark(times = 10
  , merge1 = merge(eec, data.frame(IDENT = names(nbcho), nbcho1 = nbcho), by = "IDE
  , merge2 = merge(eec, nbcho_df, by = "IDENT")
  , names = nbcho[eec$IDENT]
)
## Unit: milliseconds
##    expr        min         lq       mean     median         uq        max
##  merge1 139.076006 187.509802 180.748140 189.500565 192.781051 194.312278
##  merge2 119.919936 165.570853 162.560386 173.070987 174.026367 180.678040
##   names   2.162281   2.278111   3.882023   2.333215   6.105578   6.656698
##  neval
##     10
##     10
##     10
# C'est sans commune mesure : que l'on tienne
# compte (merge1) ou pas (merge2) de l'étape
# de constitution du data.frame, la méthode
# reposant sur les noms de vecteur est beaucoup
# plus rapide.
```

Travailler efficacement sur des données avec `dplyr`

Les cas pratiques de cette partie reposent sur le *package* `dplyr` :

```
install.packages("dplyr")
library(dplyr)
```

Plusieurs vignettes sont disponibles sur la page de documentation du *package*. Rstudio a également conçu un aide-mémoire (*cheatsheet*) traduit en français. **N'hésitez pas à vous référer à ces documents pour répondre aux cas pratiques de cette partie**.

Cas pratique 14 Sélection d'observations, de variables et tris

a. Utilisez le verbe `filter()` pour afficher les observations des femmes (`SEXE == "2"`) actives occupées (`ACTEU == "1"`) en Île-de-France (`REG == "11"`). Utilisez la syntaxe classique puis celle faisant appel à l'opérateur *pipe* `%>%`.
Afficher/masquer la solution

```
# La fonction filter() permet de ne pas avoir à répéter
# le nom de la table et de pouvoir séparer les
# différentes clauses par des ,
filter(eec, SEXE == "2", ACTEU == "1", REG == "11")
eec %>% filter(SEXE == "2", ACTEU == "1", REG == "11")
```

b. Utilisez le verbe `select()` pour supprimer toutes les variables créées à la sous-partie précédente (`trage1`, `trage2`, `trage3`, `trage4`, `nbcho1`, `nbcho2`). Pensez à consulter les exemples de l'aide de `select()` pour l'utiliser au mieux.
Afficher/masquer la solution

```
# select() dispose de nombreuses fonctions outils
# pour simplement sélectionner (ou exclure avec -)
# des variables selon les caractères qu'elles contiennent.
eec %>%
  select(-starts_with("trage"), -contains("nbcho")) ->
  eec
# Note : utilisé à l'issue de plusieurs instruction,
# l'opérateur `->` permet d'assigner des valeurs
# à un objet situé à sa droite (et non à sa gauche comme `<-`)
```

c. Utilisez le verbe `arrange()` pour trier la table par région et identifiant de ménage croissants puis par numéro d'ordre dans le ménage décroissants.
Afficher/masquer la solution

```
# Il suffit d'indiquer la liste des variables sur
# lesquelles trier, éventuellement avec la fonction
# desc() quand l'ordre est décroissant.
eec %>%
  arrange(REG, IDENT, desc(NOI)) ->
  eec
```

Cas pratique 15 Création de variables

a. Utilisez le verbe `mutate()` pour effectuer le recodage en classes d'âge présenté lors de la partie précédente.
Afficher/masquer la solution

```
# mutate() est analogue à la fonction de base R
# transform() mais permet l'utilisation directe
# des variables créées dans le même appel de fonction.
eec %>%
  mutate(
age_num = as.numeric(AGE)
, trage_dplyr = ifelse(age_num < 31, "15-30", ifelse(age_num < 61, "31-60", "61 et
  ) ->
  eec
```

b. Comparez l'ergonomie et les performances de `mutate()` avec la méthode la plus efficace de base **R**.
Afficher/masquer la solution

```
# mutate() est relativement ergonomique dans la mesure
# où elle permet de ne pas répéter le nom de la table,
# de créer simultanément plusieurs variables et de
# réutiliser immédiatement les variables créées.

# Comparaison des performances
microbenchmark(times = 100
  , base = {
t <- as.numeric(eec$AGE)
eec$trage4 <- "15-30"
eec$trage4[t > 30 & t < 61] <- "31-60"
eec$trage4[t > 60] <- "61 et +"
  }
  , dplyr = {
eec %>%
  mutate(
    age_num = as.numeric(AGE)
    , trage_dplyr = ifelse(age_num < 31, "15-30", ifelse(age_num < 61, "31-60", "6
  ) ->
  eec
  }
)
## Unit: milliseconds
##   expr       min        lq      mean    median        uq      max neval
```

```
##   base  5.765481  5.817881  6.832731  5.875419  6.160417 55.60787   100
##  dplyr 27.304466 27.691600 30.033281 29.944651 30.307037 77.46204   100

# Le gain en termes d'ergonomie de mutate() a donc
# un coût non-négligeable en termes de performances.
```

Cas pratique 16 Agrégation par groupes : salaire moyen par région

Comme dans le cas pratique correspondant de la partie précédente, l'objectif est d'estimer le salaire moyen par région, d'abord sans pondération puis pondéré par le poids de sondage `EXTRI1613`.

a. Utilisez la fonction `summarise()` pour calculer le salaire moyen non-pondéré et pondéré pour l'ensemble de la France métropolitaine. Intercalez ensuite la fonction `group_by()` pour ventiler ces calculs par région.
Afficher/masquer la solution

```
# On commence par calculer les moyennes
# sur l'ensemble de la France métropolitaine
# grâce à la fonction summarise().
eec %>%
  summarise(
nonpond = mean(SALRED, na.rm = TRUE)
, pond = weighted.mean(SALRED, EXTRI1613, na.rm = TRUE)
  )

# Pour ventiler les calculs par région, il suffit
# d'intercaler la fonction group_by()
eec %>% group_by(REG) %>%
  summarise(
nonpond = mean(SALRED, na.rm = TRUE)
, pond = weighted.mean(SALRED, EXTRI1613, na.rm = TRUE)
  )
```

b. Comparez l'ergonomie et les performances des fonctions de `dplyr` avec la méthode la plus efficace de base **R**.
Afficher/masquer la solution

```
# La syntaxe est beaucoup plus ergonomique avec
# les fonctions de dplyr, notamment du fait que
```

```
# la ventilation par une ou plusieurs variables
# ne nécessite qu'une adaptation minimale du code.

# Comparaison des performances
microbenchmark(times = 100
  , base = tapply(eec$SALRED, eec$REG, mean, na.rm = TRUE)
  , dplyr = eec %>% group_by(REG) %>%
  summarise(SALRED = mean(SALRED, na.rm = TRUE))
)
## Unit: milliseconds
##   expr      min       lq     mean   median       uq      max neval
##   base 3.808255 3.885876 4.581301 3.935264 4.560939 7.389360   100
##  dplyr 3.071125 3.206148 3.635747 3.320200 3.464294 6.989891   100
# Ici dplyr est aussi rapide sinon plus que base R.
# On gagne donc sur tous les tableaux.
```

Cas pratique 17 Fusion de tables : recodage de la PCS

La variable `CSE` de la table `eec` code la Profession et catégorie socio-professionnelle (PCS) au niveau 3 de la nomenclature Insee (*cf.* le site de l'Insee). On souhaite passer du niveau 3 au niveau 2. Le fichier `pcs2003_c_n4_n1.dbf` est la table de passage entre l'ensemble des niveaux de la nomenclature (de 1 à 4).

a. Utilisez le *package* `foreign` et la fonction `read.dbf()` pour lire le fichier `pcs2003_c_n4_n1.dbf`.
Afficher/masquer la solution

```
# La fonction read.dbf() du package foreign
# permet de lire les fichier .dbf
library(foreign)
pcs <- read.dbf("pcs2003_c_n4_n1.dbf")
```

b. Utilisez le verbe `distinct()` pour restreindre la table aux observations distinctes pour les niveaux `N2` et `N3`. Créez également la variable `CSE`, version caractère de `N3`.
Afficher/masquer la solution

```
# Ces deux opérations peuvent être effectuées
# en une seule instruction avec des %>%
pcs %>%
  distinct(N2, N3) %>%
```

```
  mutate(CSE = as.character(N3)) ->
  pcs
```

c. Utilisez le verbe `left_join()` pour fusionner la table `eec` avec la table de passage des PCS de niveau 3 à niveau 2.
Afficher/masquer la solution

```
# La syntaxe est rendue particulièrement claire
# (et proche de SQL) par l'utilisation des %>%
eec %>%
  left_join(pcs, by = "CSE") ->
  eec
```

d. Une solution purement vectorielle en base **R** n'aurait-elle pas également été possible ? Comparez les performances de `dplyr` avec cette solution alternative.
Afficher/masquer la solution

```
# On aurait aussi pu réutiliser le mécanisme
# vu au cas pratique 13 en passant par un vecteur nommé
pcs2 <- setNames(pcs$N2, pcs$N3)
eec$N2_base <- pcs2[eec$CSE]
all.equal(eec$N2, eec$N2_base)
## [1] TRUE

# Comparaison des performances
microbenchmark(times = 10
  , base = pcs2[eec$CSE]
  , dplyr = eec %>% left_join(pcs, by = "CSE")
)
## Unit: milliseconds
##   expr      min        lq      mean    median        uq       max neval
##   base 721.4964 721.72964 722.63653 722.10409 723.54421 724.83541    10
##  dplyr  11.2205  11.30665  12.58783  11.68048  14.11914  15.01254    10
# Il est possible que les perforamnces de la version
# en base R soit affectées par le grand nombre de NA.
```

Travailler efficacement sur des données avec `data.table`

Les cas pratiques de cette partie reposent sur le *package* `data.table` :

```
install.packages("data.table")
library(data.table)
```

On crée le `data.table` correspondant au `data.frame` `eec` :

```
eec_dt <- data.table(eec)
```

Plusieurs vignettes sont disponibles sur la page de documentation du *package*. **N'hésitez pas à vous référer à ces documents pour répondre aux cas pratiques de cette partie**.

Cas pratique 18 Sélection d'observations et tris

a. Utilisez l'argument `i` de `[` pour afficher les observations des femmes (`SEXE == "2"`) actives occupées (`ACTEU == "1"`) en Île-de-France (`REG == "11"`). Quelle différence constatez-vous avec une sélection dans un `data.frame` ?
Afficher/masquer la solution

```
# Dans un data.frame et avec le [ de base R,
# il est nécessaire de répéter le nom de la table
eec[eec$SEXE == "2" & eec$ACTEU == "1" & eec$REG == "11", ]

# Ce n'est pas le cas dans un data.table
eec_dt[SEXE == "2" & ACTEU == "1" & REG == "11", ]
```

b. Utilisez la fonction `setkey()` pour faire de `SEXE`, `ACTEU` et `REG` des clés pour `eec_dt` et utilisez-les pour reproduire la sélection de la question précédente. Comparez alors l'ergonomie et les performances d'une sélection d'observations :

1. avec une clause logique dans un `data.frame` ;
2. avec une clause logique en utilisant le verbe `filter()` de `dplyr` ;
3. avec une clause logique dans un `data.table` ;
4. avec un jeu de clés dans un `data.table`.

Afficher/masquer la solution

```
# La fonction setkey() permet de facilement créer
# des clés pour un data.table donné.
setkey(eec_dt, SEXE, ACTEU, REG)

# La sélection sur la base de clés s'effetue
# avec un argument sous la forme d'une liste
# DANS L'ORDRE DES CLES
eec_dt[list("2", "1", "11")]
```

```
# Comparaison des performances
microbenchmark(times = 100
  , base = eec[eec$SEXE == "2" & eec$ACTEU == "1" & eec$REG == "11", ]
  , dplyr = eec %>% filter(SEXE == "2", ACTEU == "1", REG == "11")
  , data.table1 = eec_dt[SEXE == "2" & ACTEU == "1" & REG == "11"]
  , data.table2 = eec_dt[list("2", "1", "11")]
)
## Unit: milliseconds
##         expr      min       lq     mean   median        uq       max neval
##         base 7.904094 8.013236 9.599683 8.210234 10.498250 62.221990   100
##        dplyr 6.506363 6.626642 6.880191 6.711130  6.824236  9.341599   100
##  data.table1 5.141950 5.273032 5.690626 5.396326  5.550179  9.782614   100
##  data.table2 1.612380 1.686116 1.936770 1.821257  1.872345  5.227913   100

# Les différentes version de data.table sont plus efficaces
# que base R et dplyr, en particulier quand il est fait
# usage des clés.
```

c. Utilisez la fonction `order()` (comme dans un `data.frame`) pour trier la table `eec_dt` par région et identifiant de ménage croissants puis par numéro d'ordre dans le ménage décroissants. Comparez les performances de base **R**, `arrange()` de `dplyr` et `data.table`.

Afficher/masquer la solution

```
# La principale différence avec la fonction order()
# appliquée à un data.frame est qu'il n'est pas
# nécessaire de répéter le nom de la table et
# qu'il est possible d'utiliser le signe - devant
# des variables caractère
eec_dt <- eec_dt[order(REG, IDENT, -NOI)]

# Comparaison des performances
microbenchmark(times = 10
  , base = eec[order(eec$REG, eec$IDENT, -as.numeric(eec$NOI)), ]
  , dplyr = eec %>% arrange(REG, IDENT, desc(NOI))
  , data.table = eec_dt[order(REG, IDENT, -NOI)]
)
## Unit: milliseconds
##        expr      min       lq      mean   median        uq       max neval
##        base 91.43793 92.89436 106.16409 94.79749 110.56226 152.40287    10
##       dplyr 61.96490 64.88916  71.79786 65.80203  67.83572 118.08390    10
##  data.table 13.25075 13.72903  20.75144 16.17314  16.66785  68.46394    10
# Le gain en termes de performances est sensible à nouveau.
```

Cas pratique 19 Agrégation par groupes : salaire moyen par région

Comme dans le cas pratiques correspondants des parties précédentes, on cherche à calculer le salaire moyen par région non-pondéré puis pondéré par le poids de sondage `EXTRI1613`.

a. Utilisez les arguments `j` et `by` de `[` pour calculer le salaire non-pondéré et pondéré d'abord sur l'ensemble de la France métropolitaine, puis par région. Comparez l'utilisation de `by` et `keyby` : pourquoi les résultats sont-ils ici identiques à votre avis ?
Afficher/masquer la solution

```
# L'argument `j` de [ permet de créer de nouvelles
# variables dans un data.table
eec_dt[, j = list(
  nonpond = mean(SALRED, na.rm = TRUE)
  , pond = weighted.mean(SALRED, EXTRI1613, na.rm = TRUE)
)]
##     nonpond     pond
## 1: 1819.209 1833.879
# Pour ventiler par région, il suffit d'ajouter un argument by
eec_dt[, j = list(
  nonpond = mean(SALRED, na.rm = TRUE)
  , pond = weighted.mean(SALRED, EXTRI1613, na.rm = TRUE)
), by = REG]
##     REG  nonpond     pond
##  1:  11 2153.045 2172.578
##  2:  21 1692.069 1686.691
##  3:  22 1585.062 1594.508
##  4:  23 1722.671 1727.632
##  5:  24 1702.413 1725.168
##  6:  25 1633.108 1625.475
##  7:  26 1640.106 1617.565
##  8:  31 1730.126 1714.465
##  9:  41 1729.073 1693.117
## 10:  42 1974.987 2012.017
## 11:  43 1747.486 1750.870
## 12:  52 1609.202 1616.999
## 13:  53 1784.060 1814.576
## 14:  54 1587.502 1547.045
## 15:  72 1703.824 1667.706
## 16:  73 1756.656 1829.931
## 17:  74 1633.896 1657.696
```

```
## 18:  82 1948.086 1954.849
## 19:  83 1633.669 1621.692
## 20:  91 1557.426 1538.386
## 21:  93 1804.827 1806.120
## 22:  94 1828.872 1796.574
##     REG  nonpond     pond
# Les résultats sont identiques ici selon que l'on utilise
# by ou keyby car les données sont triées par région.
# Si cela n'était pas le cas, by conserverait l'ordre du
# fichier (en affichant les groupes par ordre de rencontre)
# alors que keyby trierait les résultats par ordre
# alphabétique de région.
```

b. Comparez l'ergonomie et les performances de la solution en base **R**, avec `dplyr` et `data.table`.
Afficher/masquer la solution

```
# La solution en data.table est plus ergonomique que
# celle en base R, en particulier en raison de la
# facilité à ventiler les traitements par région.
# Néanmoins, elle ne dispose pas de la capacité
# à séquencer les traitements en petites opérations
# simples, ce que permet l'opérateur %>% qu'utilise
# intensément dplyr.

# Comparaison des performances
microbenchmark(times = 100
  , base = tapply(eec$SALRED, eec$REG, mean, na.rm = TRUE)
  , dplyr = eec %>% group_by(REG) %>% summarise(SALRED = mean(SALRED, na.rm = TRUE)
  , data.table = eec_dt[, j = list(mean(SALRED, na.rm = TRUE)), by = REG]
)
## Unit: microseconds
##        expr      min       lq     mean   median       uq       max neval
##        base 3725.662 3866.179 5094.095 3923.690 6258.657 56279.629   100
##       dplyr 3249.171 3407.872 3678.204 3493.867 3561.886  6170.888   100
##  data.table  959.950 1058.709 1240.075 1184.424 1230.423  3667.298   100
# Là encore data.table est beaucoup plus rapide.
```

Cas pratique 20 Fusion de tables : nombre d'individus au chômage par ménage

Comme dans le cas pratique 13, on cherche ici à associer à chaque individu le nombre d'individus au chômage (`ACTEU == "2"`) dans son ménage (individus avec la même valeur pour la variable `IDENT`).

a. Utilisez les arguments `j` et `by` de `[` pour calculer le nombre d'individus au chômage par ménage. Utilisez la structure `j :=` pour automatiquement refusionner ce résultat avec la table de départ.
Afficher/masquer la solution

```
# On reprend la syntaxe de la question précédente
# pour calculer le nombre d'individu au chômage par ménage
eec_dt[, sum(ACTEU == "2", na.rm = TRUE), by = "IDENT"]
##            IDENT V1
##     1: G0A56JP6  1
##     2: G0A56JR6  0
##     3: G0A56JS6  0
##     4: G0A56JT6  0
##     5: G0A56JU6  0
##    ---
## 18899: GXZ5OX1F  0
## 18900: GY05O5DF  0
## 18901: GY05O85F  0
## 18902: GY05OAXF  1
## 18903: GY05ODPF  0

# Pour refusionner ces résultats avec la table d'origine,
# il suffit d'utiliser l'opérateur := au niveau de l'argument j
eec_dt <- eec_dt[, nbcho := sum(ACTEU == "2", na.rm = TRUE), by = "IDENT"]
```

b. Comment mèneriez vous ce traitement dans la logique de `dplyr` ? Comparez l'ergonomie et les performances de la solution en base **R**, `dplyr` et `data.table`.
Afficher/masquer la solution

```
# Avec dplyr, on serait tenté de fusionner la table eec
# avec une table de statistique (comme en base R)
eec %>%
  left_join(
eec %>% group_by(IDENT) %>% summarize(nbcho = n())
, by = "IDENT"
  ) ->
  eec

# Comparaison des performances
microbenchmark(times = 10
```

```
  , base = tapply(eec$ACTEU == "2", eec$IDENT, sum, na.rm = TRUE)[eec$IDENT]
  , dplyr = {
eec %>%
  left_join(
    eec %>% group_by(IDENT) %>% summarize(nbcho = n())
    , by = "IDENT"
  )
  }
  , data.table = eec_dt[, nbcho := sum(ACTEU == "2", na.rm = TRUE), by = "IDENT"]
)
## Unit: milliseconds
##        expr       min        lq      mean    median        uq       max
##        base  51.17290  54.50705  60.82165  55.62895  56.72165 108.09253
##       dplyr 122.45642 126.21747 137.41056 128.25152 131.41873 180.62253
##  data.table  15.75786  15.84539  17.48081  16.35169  19.74838  21.08564
##  neval
##     10
##     10
##     10
# Comme toujours data.table est le plus rapide.
# Néanmoins ici, il n'est pas à exclure qu'il existe
# dans dplyr une méthode plus efficace (une possibilité
# d'autofusion après agrégation par exemple).
```

Réaliser des graphiques avec R

Les cas pratiques de cette partie reposent sur l'utilisation du *package* `ggplot2` :

```
install.packages("ggplot2")
library(ggplot2)
```

Cas pratique 21 Graphiques à partir de la table `mpg`

L'objectif de ce cas pratique est de reproduire les graphiques du support ainsi que ceux présentés par H. Wickham dans le chapitre 2 de son ouvrage **ggplot2 : Elegant Graphics for Data Analysis** (dont le `.pdf` recompilé est dans le dossier de la formation).

Dans les deux cas, la table utilisée est `mpg` (table d'exemple du *package* `ggplot2`) : après avoir chargé `ggplot2`, utilisez la fonction `data()` pour "rapatrier" la table `mpg` dans l'environnement global et tapez `? mpg` pour obtenir une description détaillée de ses variables.

Cherchez à reproduire les graphiques du support ou de ceux du chapitre 2 de **ggplot2 : Elegant Graphics for Data Analysis** en expérimentant avec les fonctionnalités de `ggplot2`. En particulier :

a. Utilisez les mots-clés `colour`, `shape` et `size` pour faire varier la représentation des points avec `geom_point()`. Pour chacun des mots-clés, comparez ce qu'il se passe quand vous utilisez une variable de type numérique ou une variable de type caractère ou facteur.
b. Comparez l'utilisation du mot-clé `colour` dans la fonction `aes()` et en dehors de la fonction `aes()`.
c. Testez les différents types de représentation possibles en les adaptant à la nature des données à représenter. Pour chaque fonction `geom_*()`, recherchez dans l'aide les paramètres qui lui sont spécifiques et testez des valeurs différentes de leurs valeurs par défaut.
d. Expérimentez les différentes possibilités de *facetting*.
e. Sauvegarder un graphique dans un objet **R**. Utilisez `ggsave()` pour exporter un graphique en `.png` et `.pdf`.
f. Affichez le code d'une fonction `geom_*()` et utilisez ces informations pour reconstituer manuellement l'instruction `layer()` correspondante.
g. Tentez de reproduire un graphique standard de `ggplot2` avec les fonctions du *package* `graphics`.

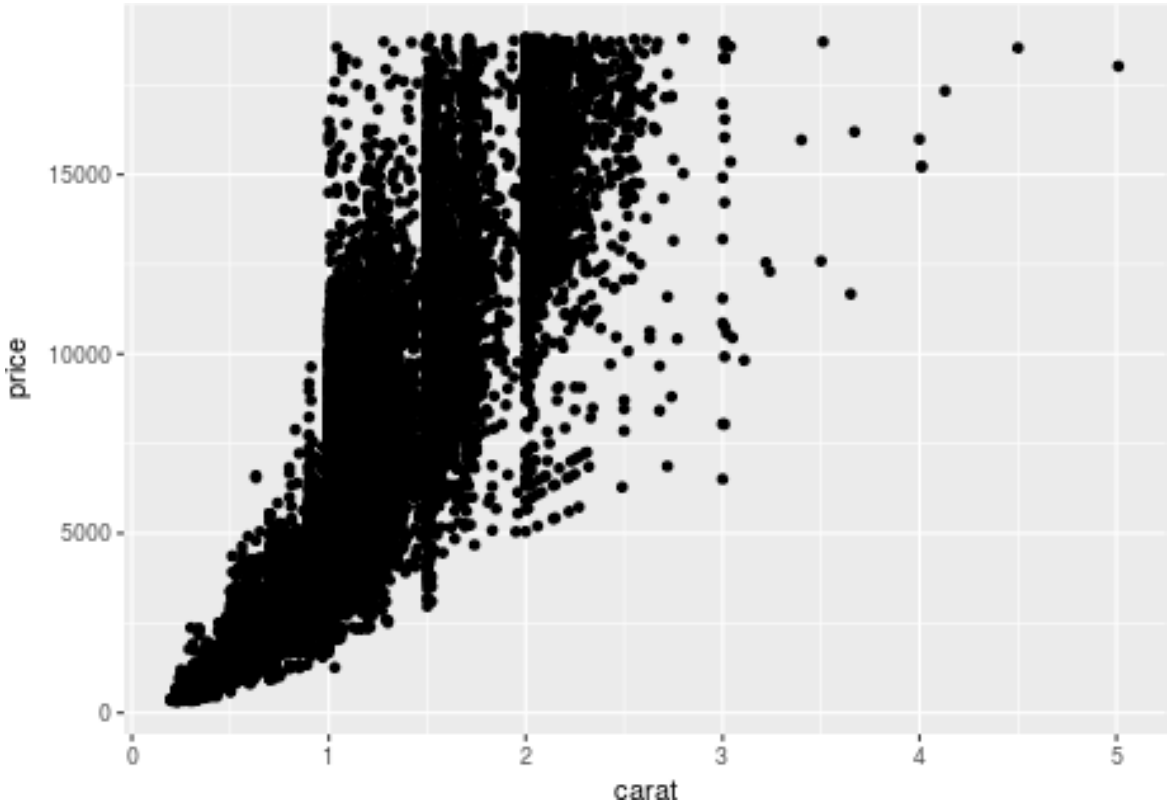
Cas pratique 22 Graphiques à partir de la table `diamonds`

La table `diamonds` est le second fichier de démonstration classique de `ggplot2` : utilisez `data()` pour le "rapatrier" dans l'environnement global et tapez `? diamonds` pour obtenir une description détaillée de ses variables.

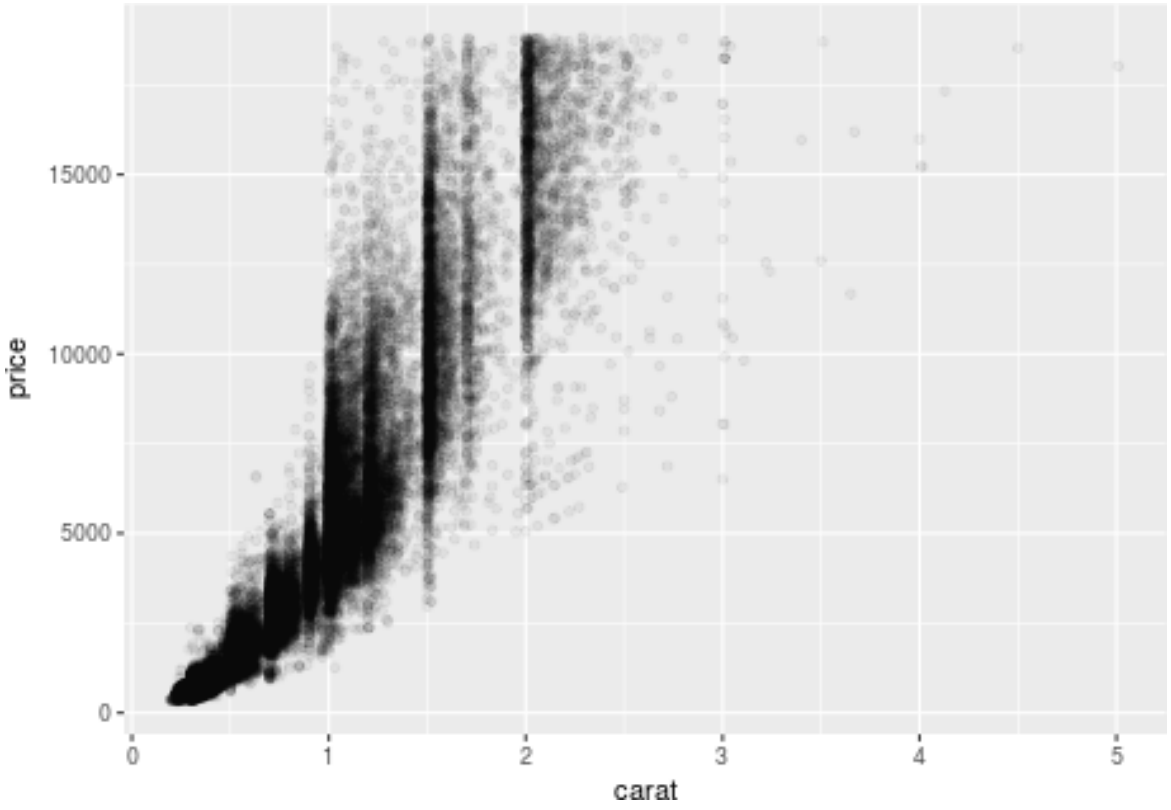
a. Représentez la relation entre poids du diamant (`carat`) et prix (`price`). Utilisez le paramètre `alpha` pour limiter la saturation du graphique par le très grand nombre de points. Ajoutez une droite de régression linéaire au graphique.

   Afficher/masquer la solution

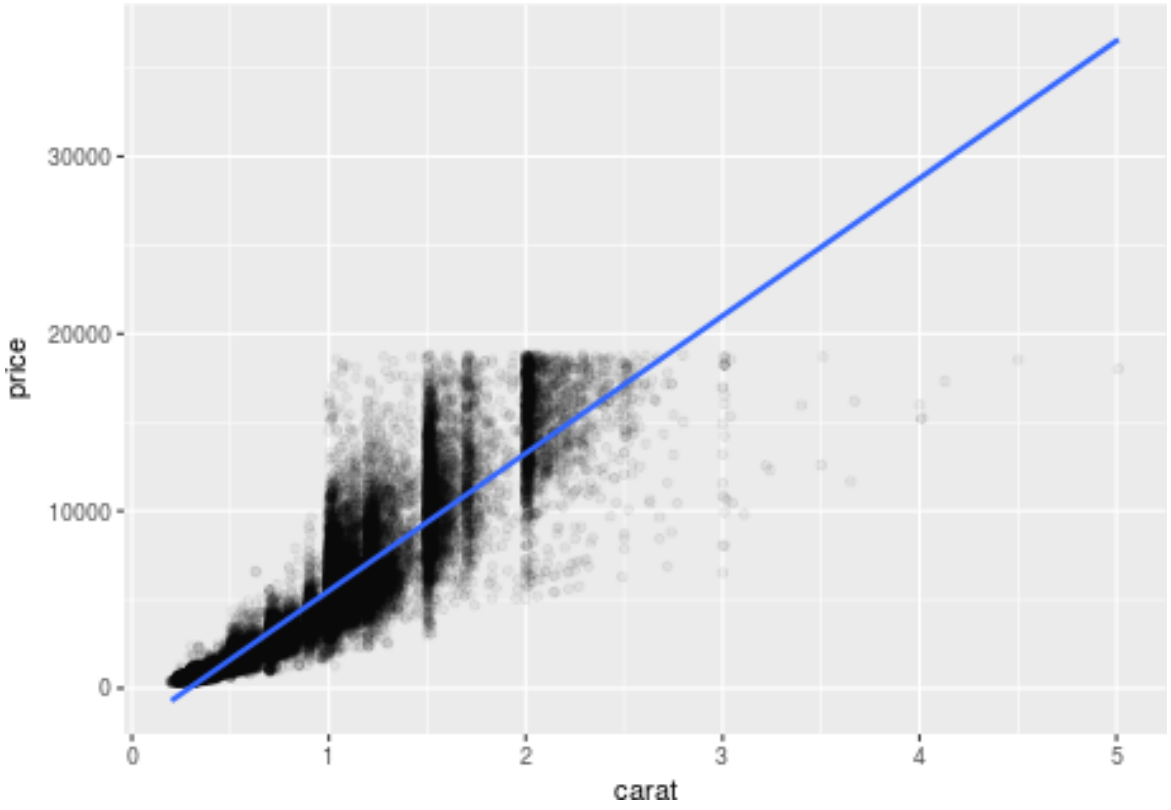
```
# Par défaut, les points sont totalement opaques :
# on ne peut pas visualiser les points qui se superposent
# les uns aux autres (on parle d'over-plotting)
ggplot(diamonds, aes(carat, price)) + geom_point()
```




```
# Le paramètres alpha indique de rendre les points
# en partie transparent. Avec alpha = 0.05, il faut
# 20 points pour obtenir une zone totalement opaque
ggplot(diamonds, aes(carat, price)) + geom_point(alpha = 0.05)
```

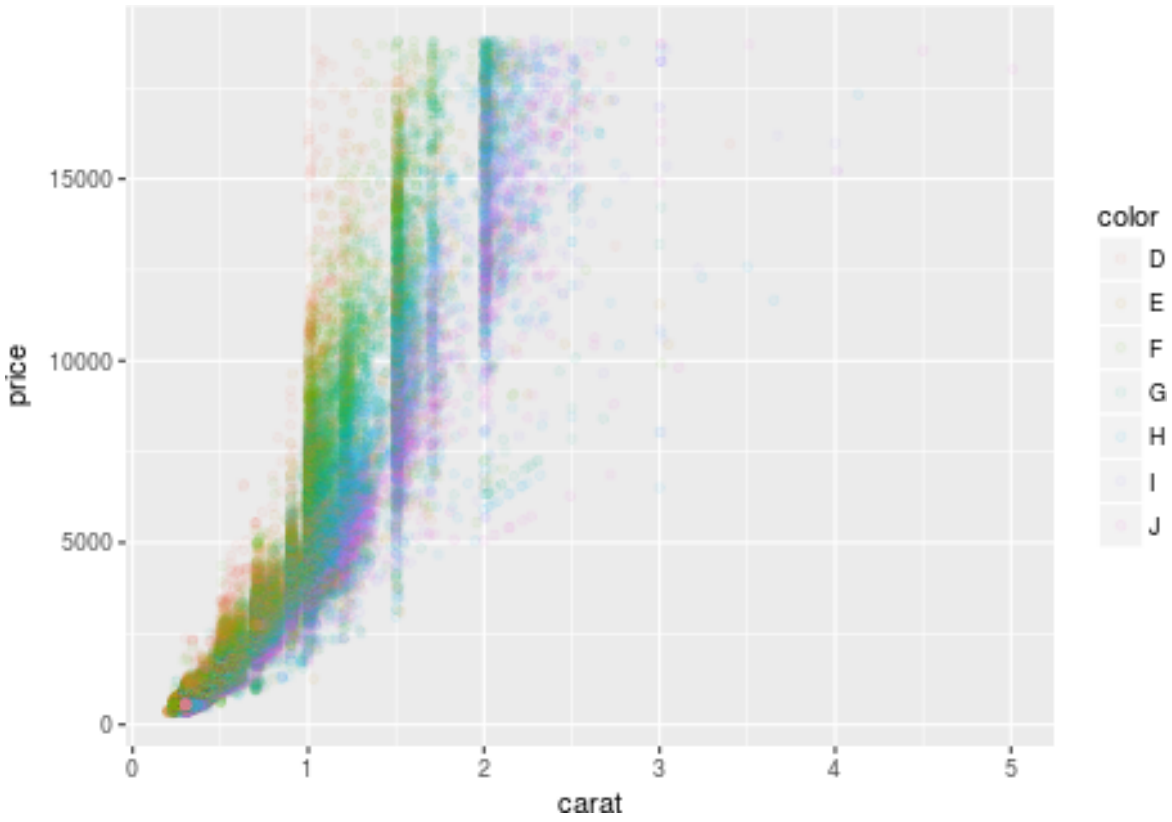



```
# Pour ajouter une droite de régression, il suffit
# d'utiliser la fonction geom_smooth() avec l'option
# method = "lm"
ggplot(diamonds, aes(carat, price)) + geom_point(alpha = 0.05) +
  geom_smooth(method = "lm")
```

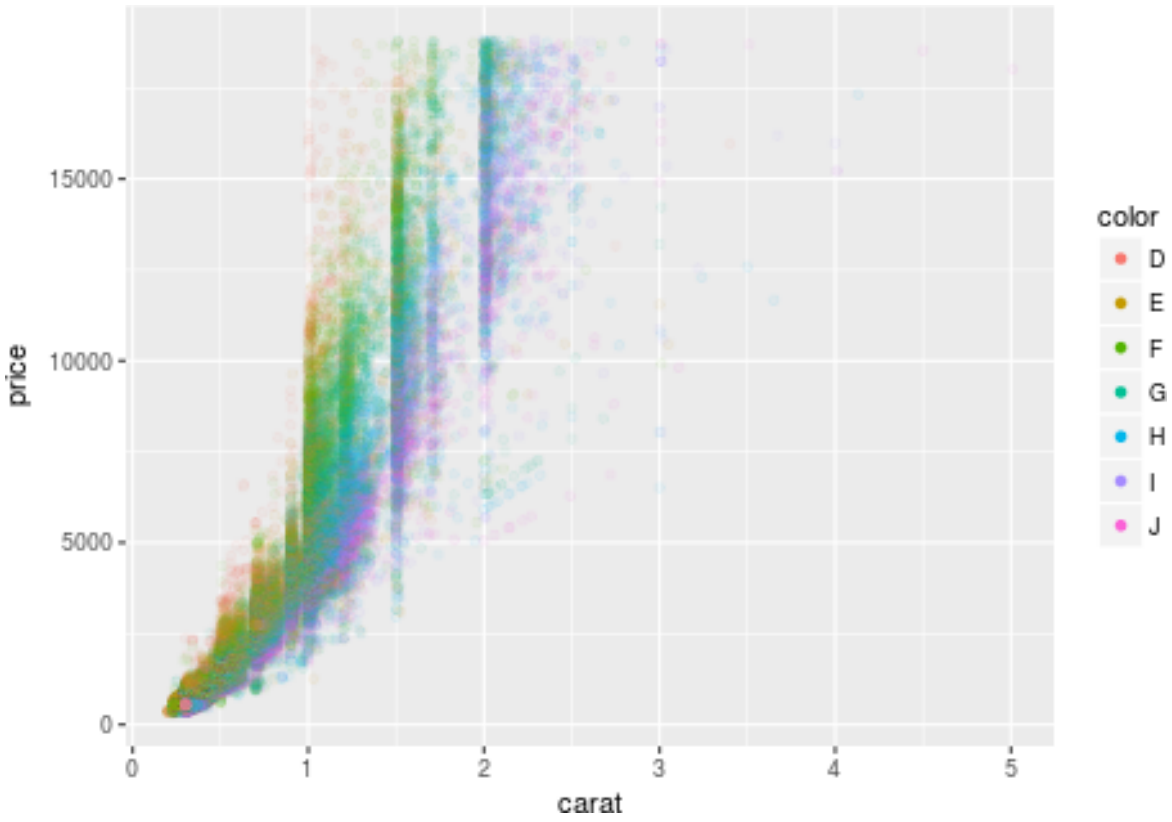


b. Représentez l'influence de la couleur (`color`) sur le prix de plusieurs manières.
Afficher/masquer la solution

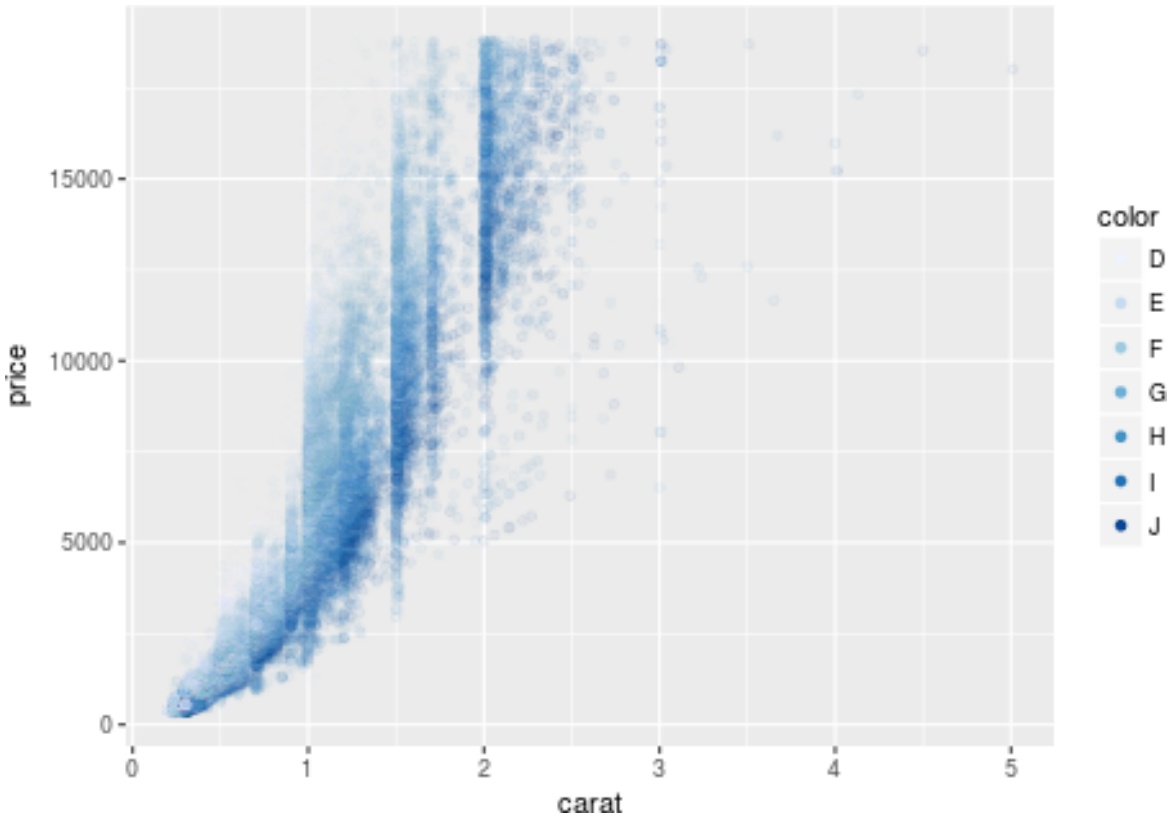
```
# Idée 1 : faire varier la couleur des points
# sur le graph précédent
g1 <- ggplot(diamonds, aes(carat, price, colour = color)) + geom_point(alpha = 0.05
g1
```



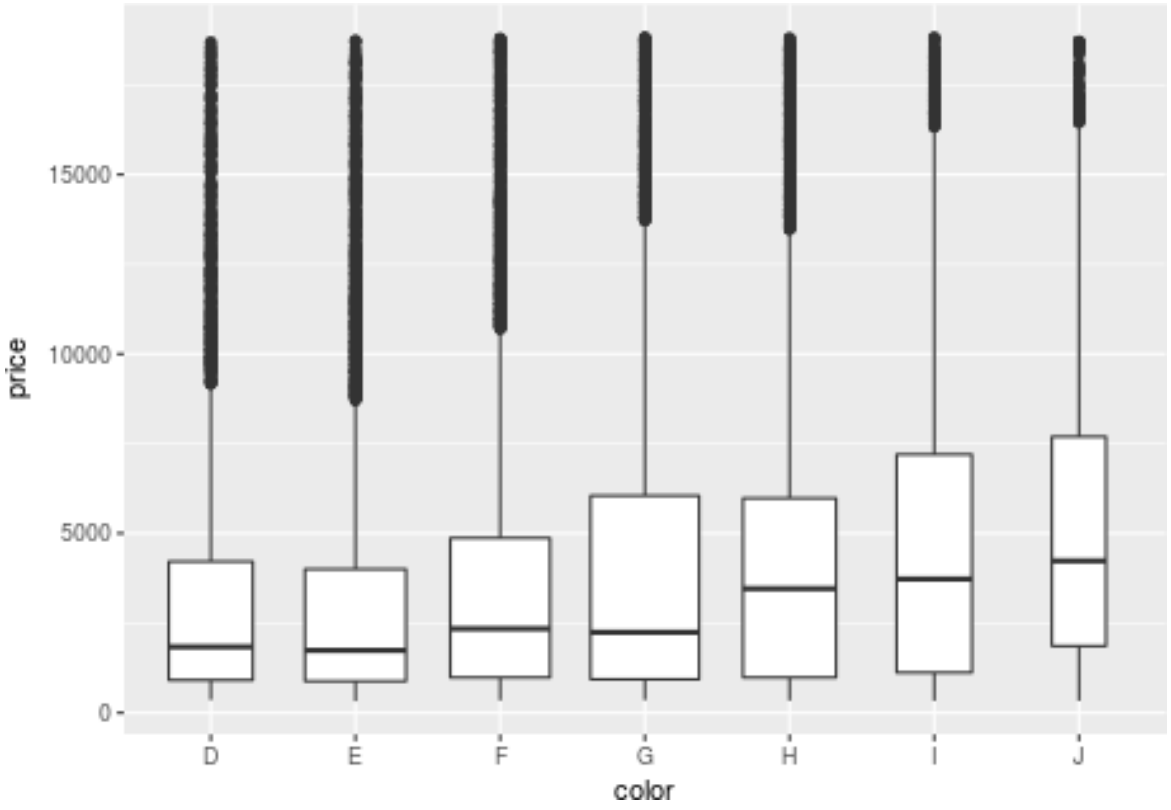

```
# Il est possible d'améliorer le graphique précédent
# 1) en ajustant la manière dont les couleurs sont représentées
# dans la légende
g1 <- g1 + guides(colour = guide_legend(override.aes = list(alpha = 1)))
g1
```




```
# 2) en adoptant une palette de couleurs formant un gradient
# (car la variable color est ordonnée de la pire (D) à la
# meilleure (J))
g1 <- g1 + scale_colour_brewer(palette = 1)
g1
```

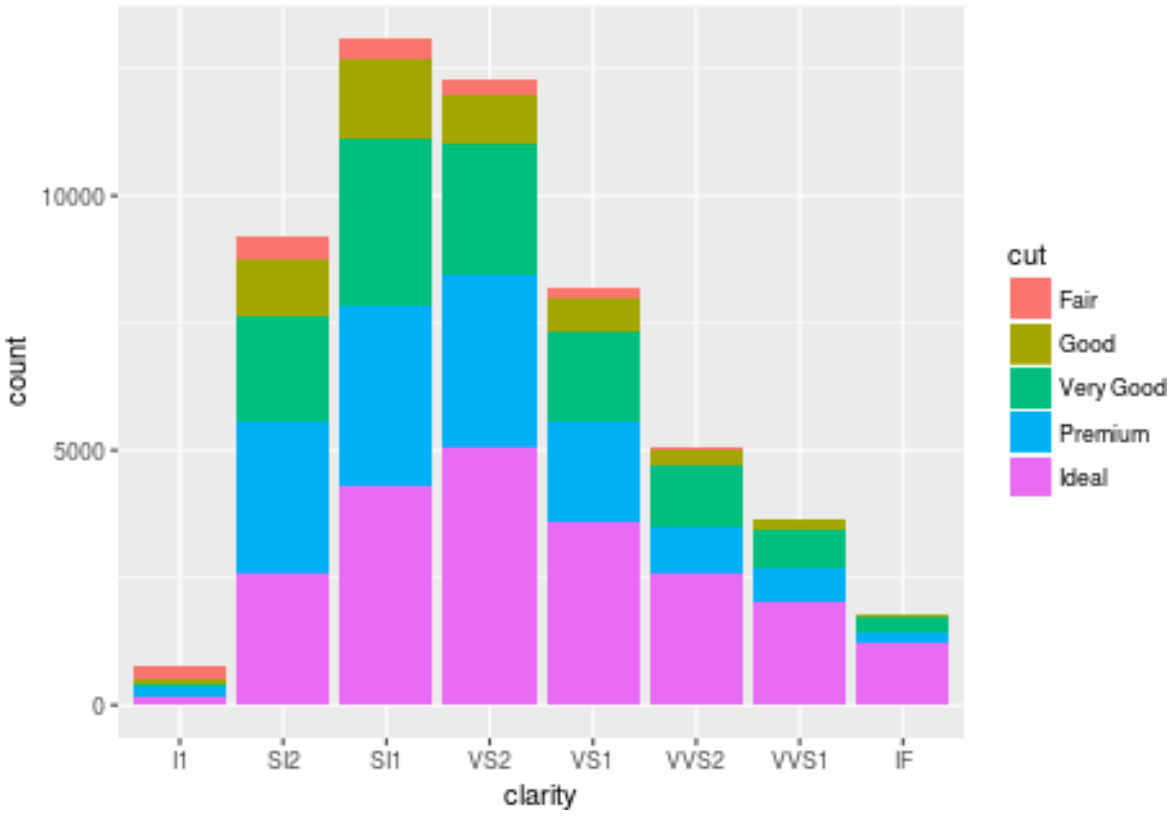



```
# Idée 2 : dessiner des boîtes à moustaches
g2 <- ggplot(diamonds, aes(color, price)) + geom_boxplot(varwidth = TRUE)
g2
```

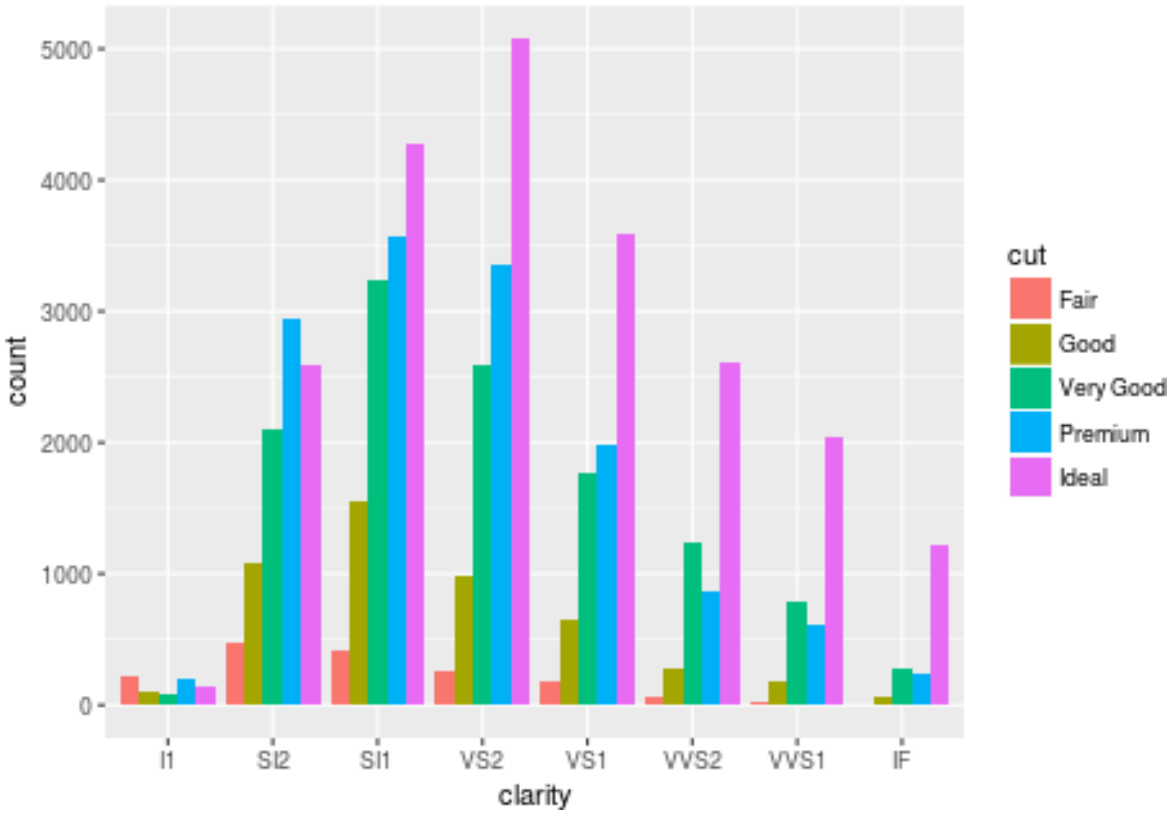


c. Représentez la ventilation des diamants selon la qualité de leur taille (`cut`) et leur clarté (`clarity`).
Afficher/masquer la solution

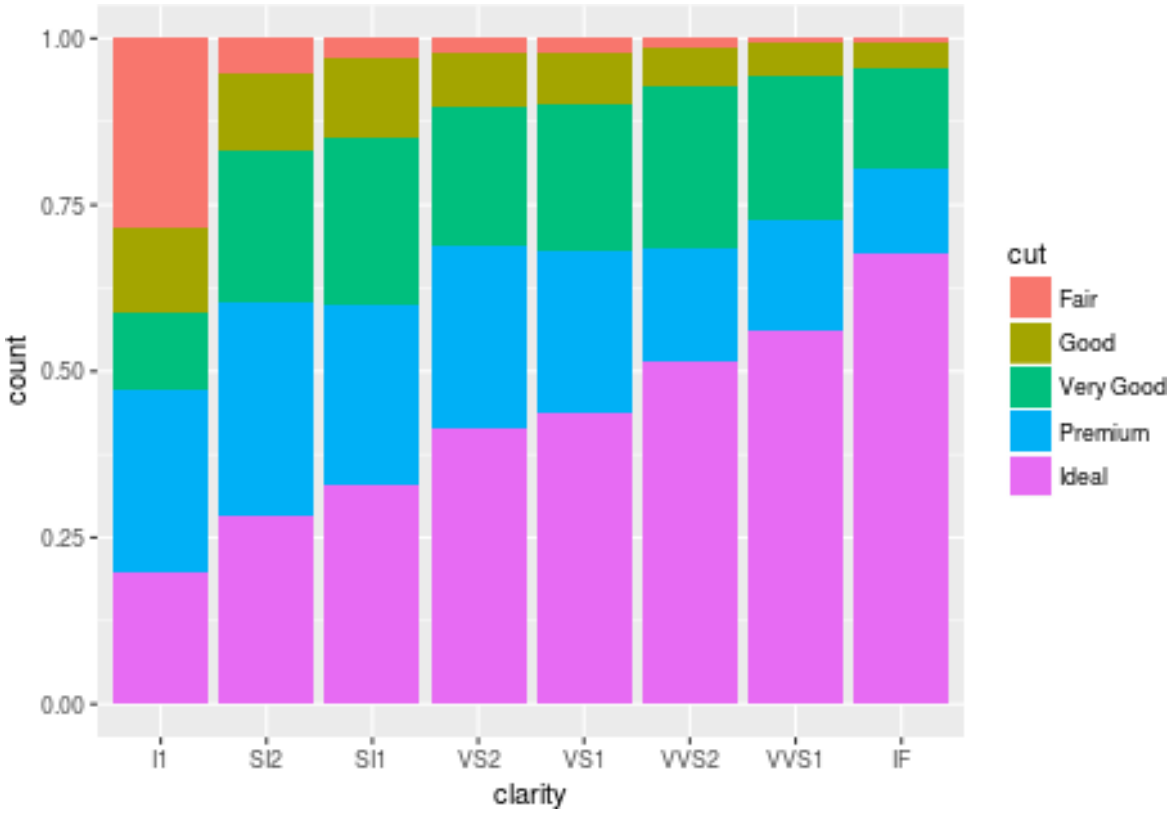
```
# Le plus simple est de représenter l'histogramme bivarié
ggplot(diamonds, aes(clarity, fill = cut)) + geom_bar()
```




```
# Quelques variations sur le positionnement des blocs
ggplot(diamonds, aes(clarity, fill = cut)) + geom_bar(position = "dodge")
```

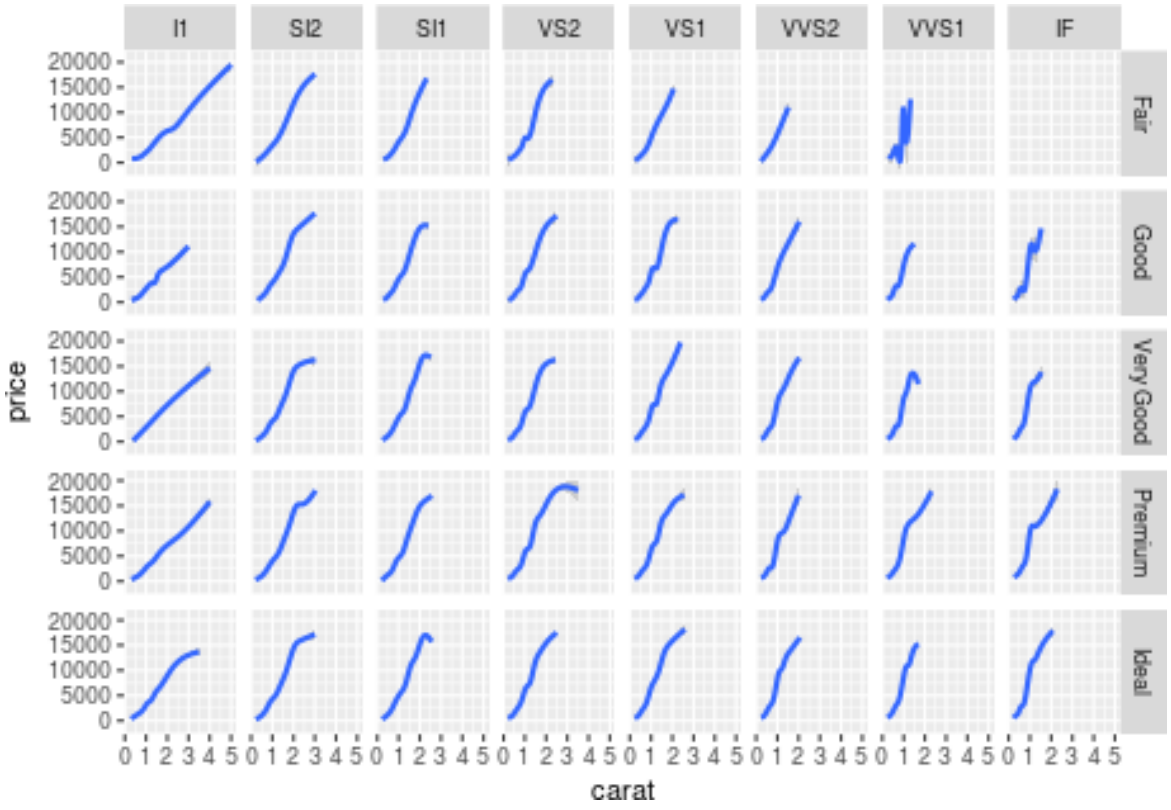



```
ggplot(diamonds, aes(clarity, fill = cut)) + geom_bar(position = "fill")
```



d. Vérifiez graphiquement si la relation entre poids et prix ne varie pas en fonction de la qualité de la taille (`cut`) et la clarté du diamant (`clarity`).
Afficher/masquer la solution

```
# L'idée ici est d'utiliser le facetting pour ventiler
# le premier graphique par qualité de la taille
g3 <- ggplot(diamonds, aes(carat, price)) + geom_smooth() +
  facet_grid(cut ~ clarity)
g3
## `geom_smooth()` using method = 'gam'
## Warning: Computation failed in `stat_smooth()`:
## x has insufficient unique values to support 10 knots: reduce k.
```



Cas pratique 23 Graphiques à partir de la table `raisin`

Le fichier `raisin.rds` comporte des informations sur la maturation du raisin dans des exploitations viticoles de Saône-et-Loire sur la période sur la période 2000-2012.

a. Chargez ce fichier en mémoire avec la fonction `readRDS()` et analysez les caractéristiques des variables de ce fichier (modalités, distributions, etc.).

Afficher/masquer la solution

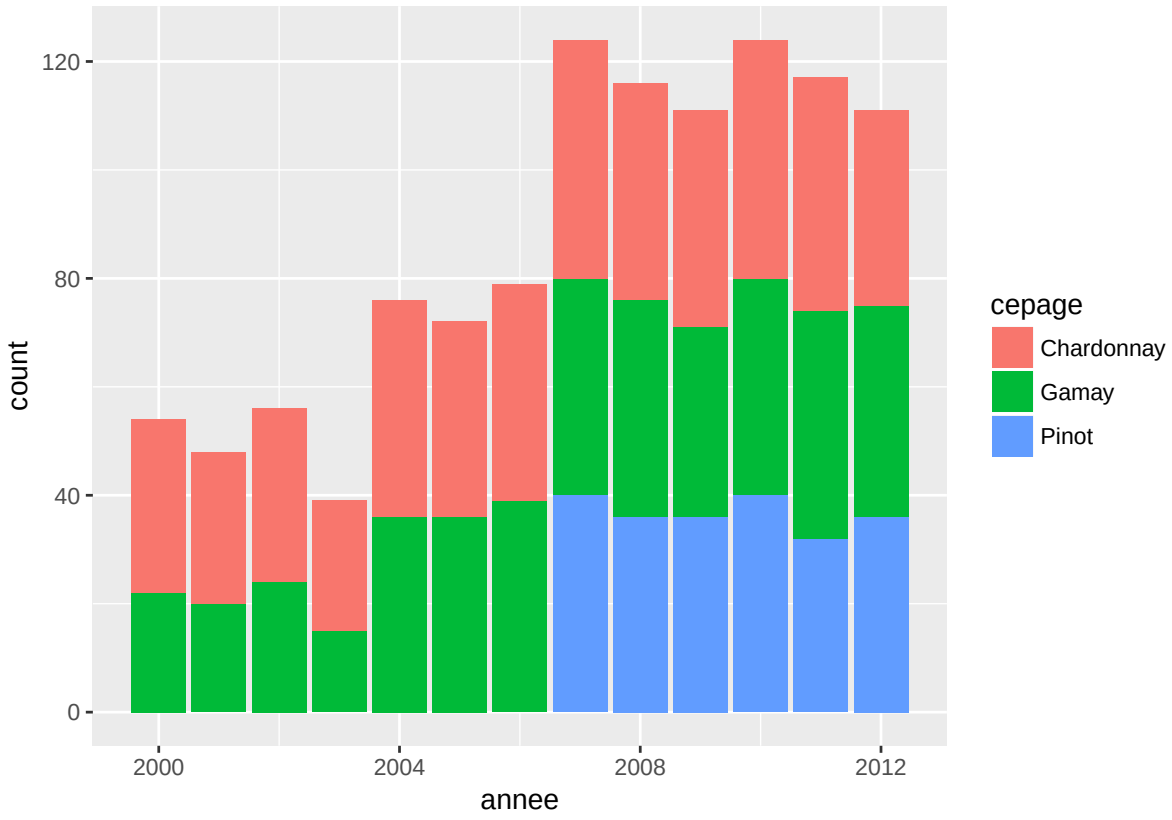
```
# Chargement en mémoire du fichier raisin.rds
# situé dans le répertoire de travail
raisin <- readRDS("raisin.rds")

# Caractéristiques des variables de raisin
table(raisin$annee)
##
## 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012
##   54   48   56   39   76   72   79  124  116  111  124  117  111
table(raisin$secteur)
##
##  Couchois Maconnais  Maranges
##       110       907       110
table(raisin$cepage)
##
## Chardonnay      Gamay      Pinot
##        479        428        220
table(raisin$commune)
##
##   Cheilly-les-Maranges                 Fuissé               St.Amour
##                    110                    479                    428
## St.Maurice-les-Couches     St.Sernin-du-Plain
##                     55                     55
summary(raisin)
##    secteur             cepage            commune              sucres
##  Length:1127        Length:1127        Length:1127        Min.   :  48.0
##  Class :character   Class :character   Class :character   1st Qu.: 136.0
##  Mode  :character   Mode  :character   Mode  :character   Median : 167.0
##                                                           Mean   : 348.4
##                                                           3rd Qu.: 197.0
##                                                           Max.   :2057.0
##
##  acidite_totale         ph         acide_tartrique    potasse
##  Min.   : 3.000   Min.   : 1.00   Min.   : 1.00   Min.   : 1.00
##  1st Qu.: 7.000   1st Qu.: 4.00   1st Qu.: 9.00   1st Qu.:12.00
##  Median : 8.000   Median : 7.00   Median :74.00   Median :19.00
##  Mean   : 8.987   Mean   :21.56   Mean   :55.91   Mean   :21.22
##  3rd Qu.:11.000   3rd Qu.:32.00   3rd Qu.:87.00   3rd Qu.:26.00
##  Max.   :21.000   Max.   :99.00   Max.   :99.00   Max.   :99.00
##                                                   NA's   :1
##    azote_amm         annee
```

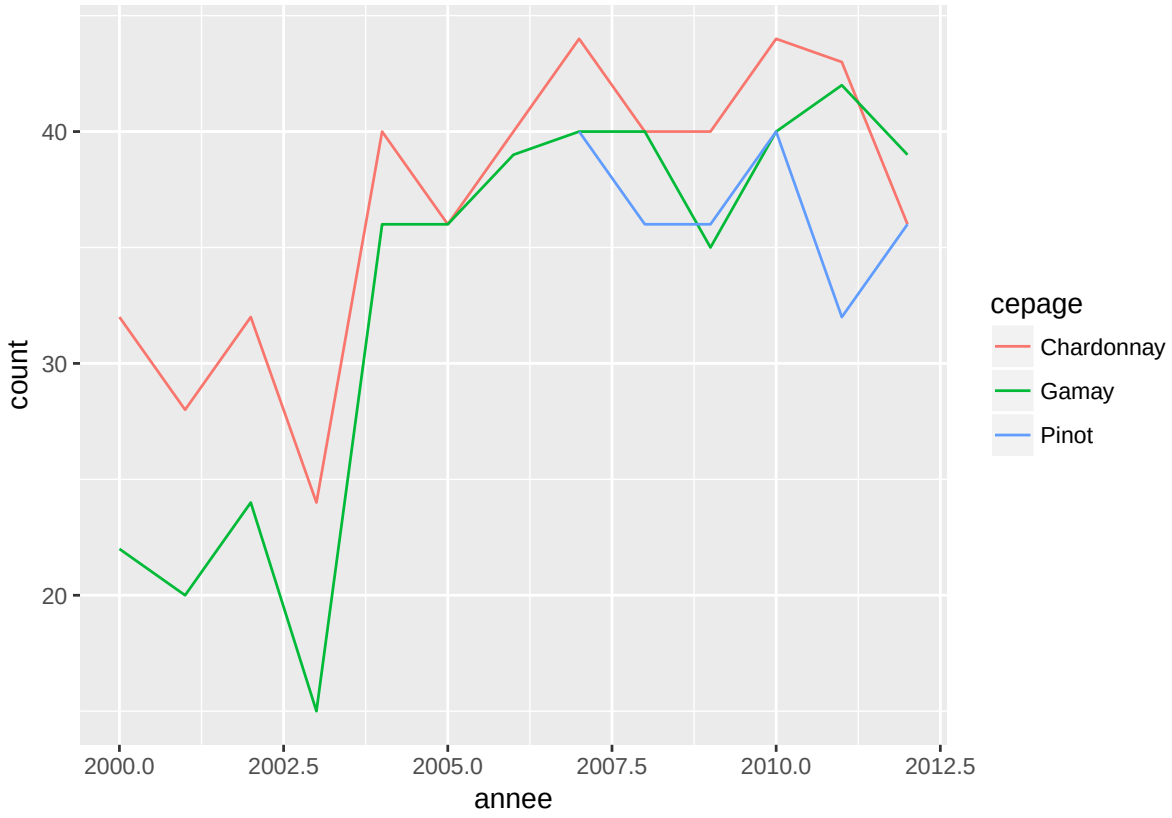
```
##  Min.   : 1.00   Min.   :2000
##  1st Qu.:12.00   1st Qu.:2005
##  Median :43.00   Median :2008
##  Mean   :43.76   Mean   :2007
##  3rd Qu.:71.00   3rd Qu.:2010
##  Max.   :99.00   Max.   :2012
##  NA's   :28
```

b. Analysez la fréquence des différents cépages en fonction du temps.
Afficher/masquer la solution

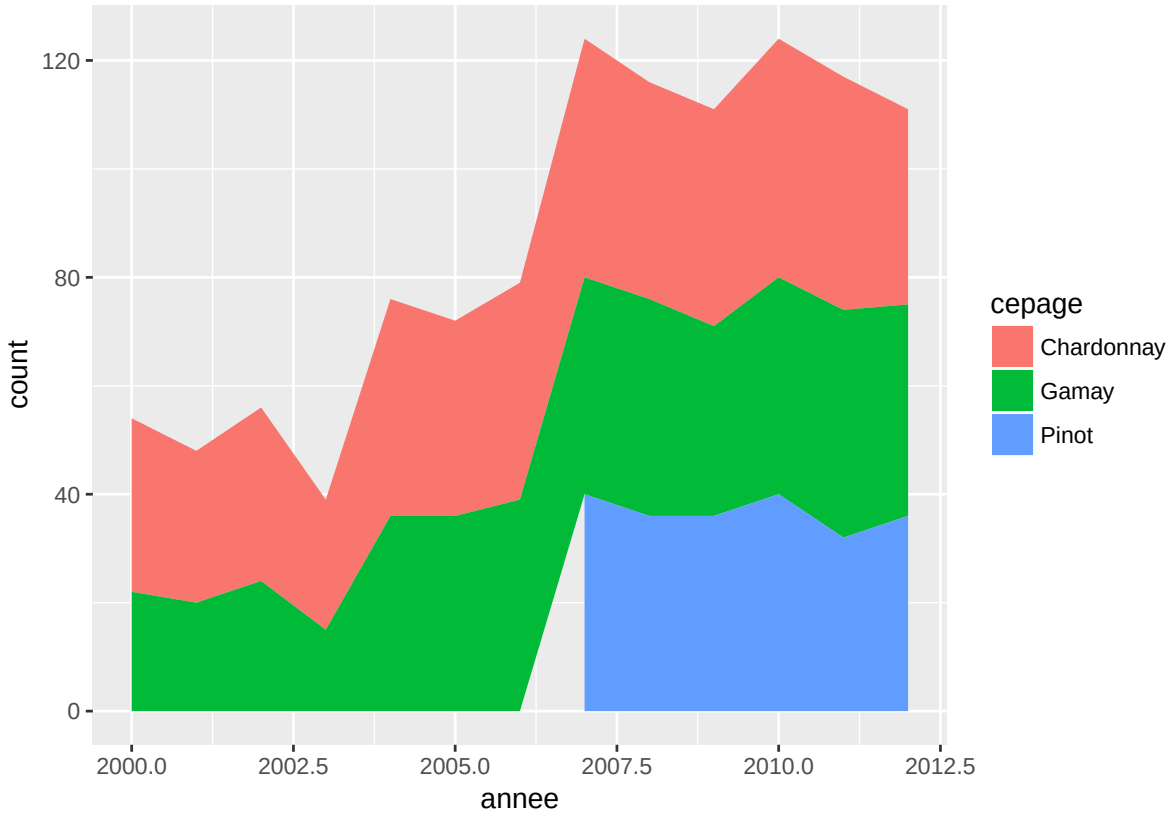
```
# Le plus simple est ici de faire un diagramme en bâton
# en fonction du temps
ggplot(raisin, aes(annee, fill = cepage)) + geom_bar()
```




```
# On peut utiliser explicitement la statistique "count"
# associée par défaut à geom_bar() (taper geom_bar) pour
# le vérifier à d'autres fonctions pour modifier cette
# représentation
ggplot(raisin, aes(annee, colour = cepage)) + geom_line(stat = "count")
```

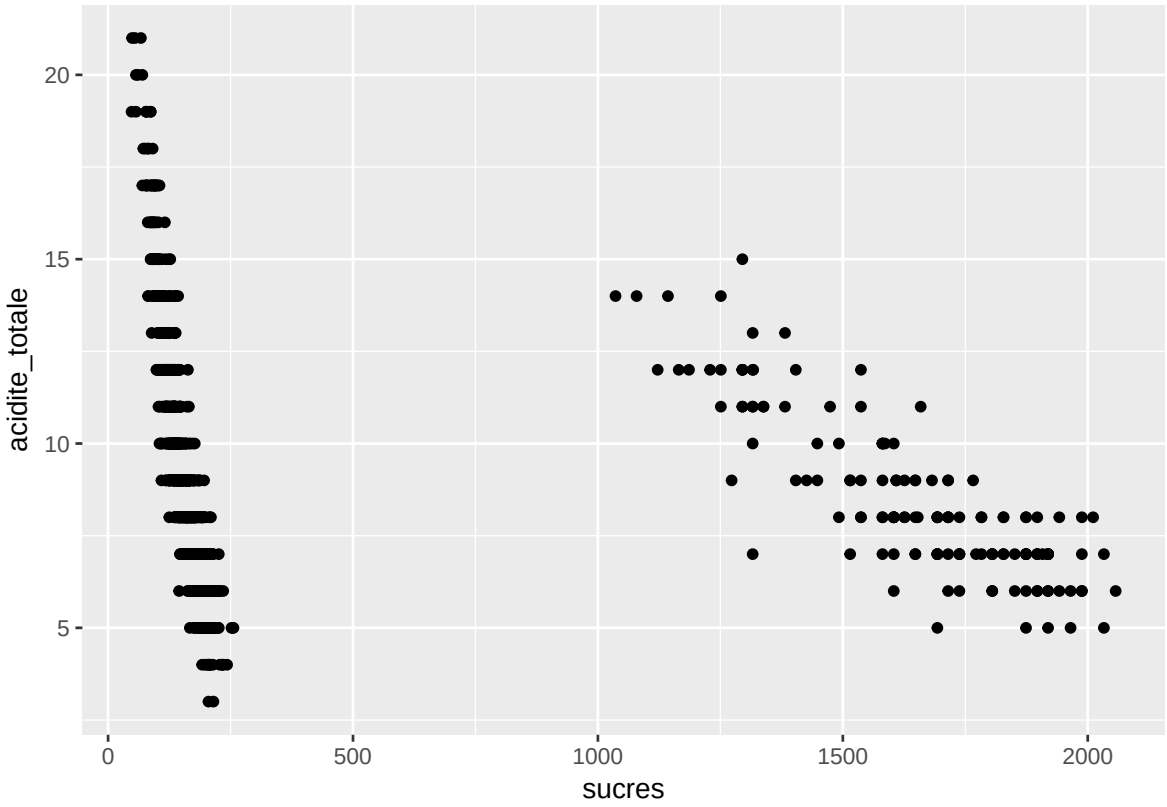



```
ggplot(raisin, aes(annee, fill = cepage)) + geom_area(stat = "count")
```

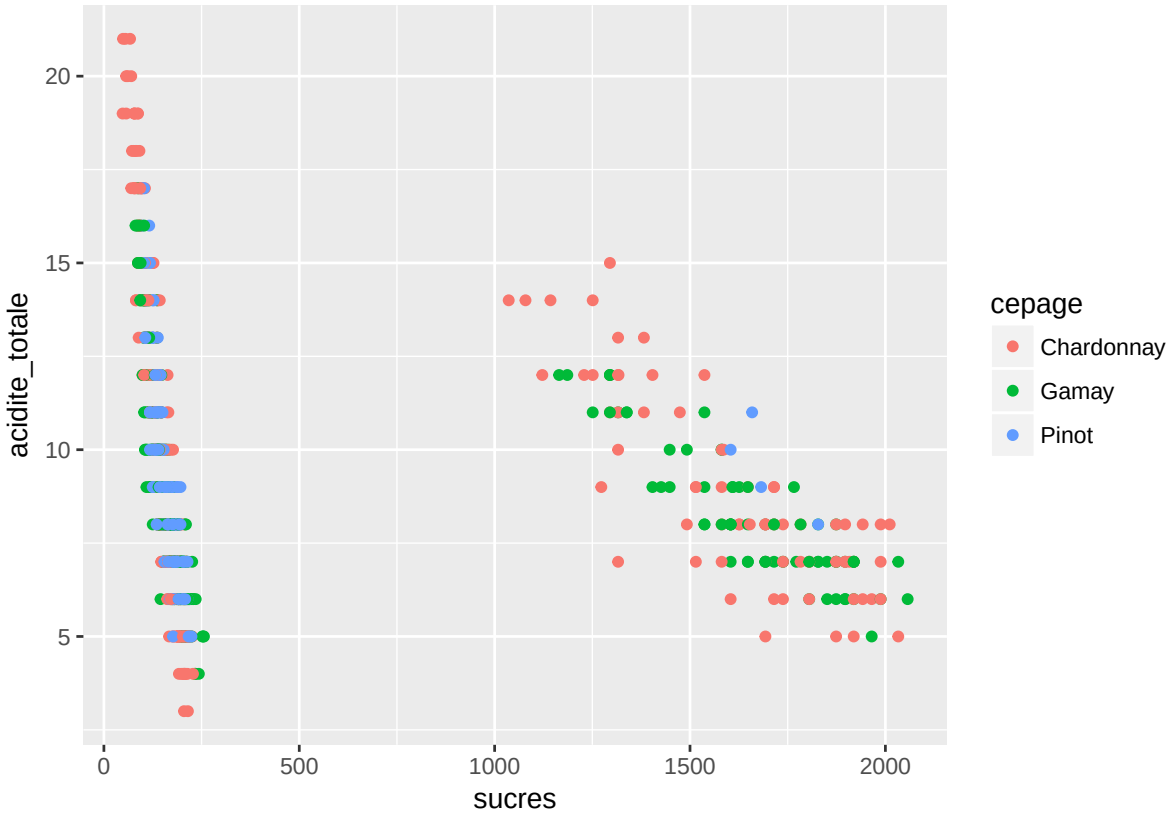


c. Analysez graphiquement la relation entre `sucres` et `acidite_totale`. Utilisez d'autres variables pour tenter de rendre compte de cette distribution.
Afficher/masquer la solution

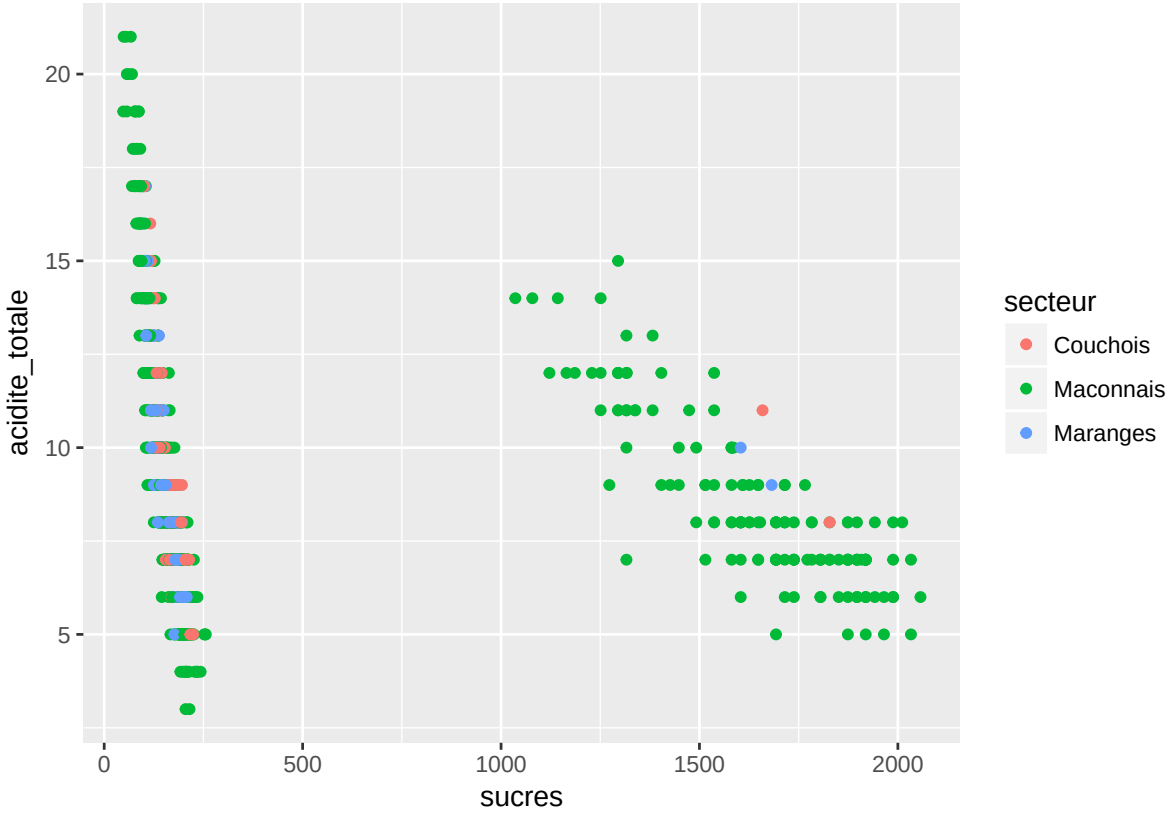
```
# On utilise tout simplement un nuage de points pour représenter
# ces deux variables quantitatives
ggplot(raisin, aes(sucres, acidite_totale)) + geom_point()
```



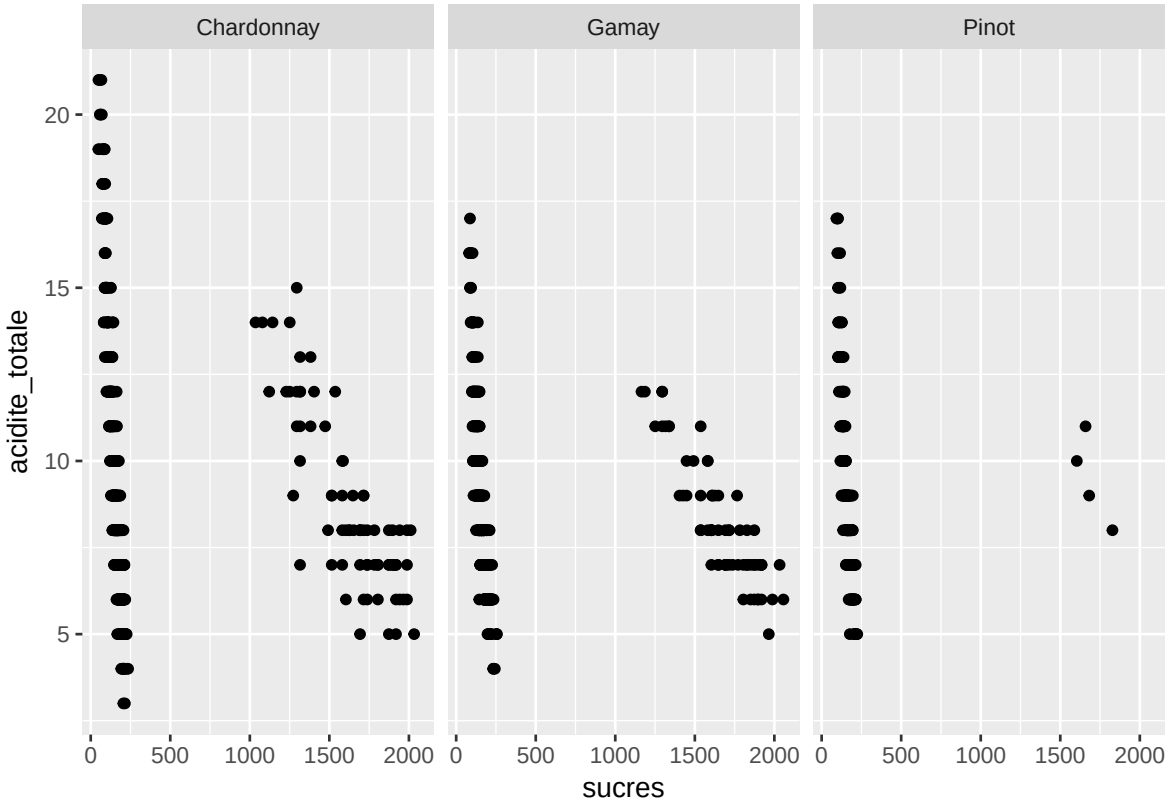

```
# On perçoit très nettement deux groupes : sont-ils expliqués
# par le secteur ou le cépage ?
ggplot(raisin, aes(sucres, acidite_totale, colour = cepage)) + geom_point()
```



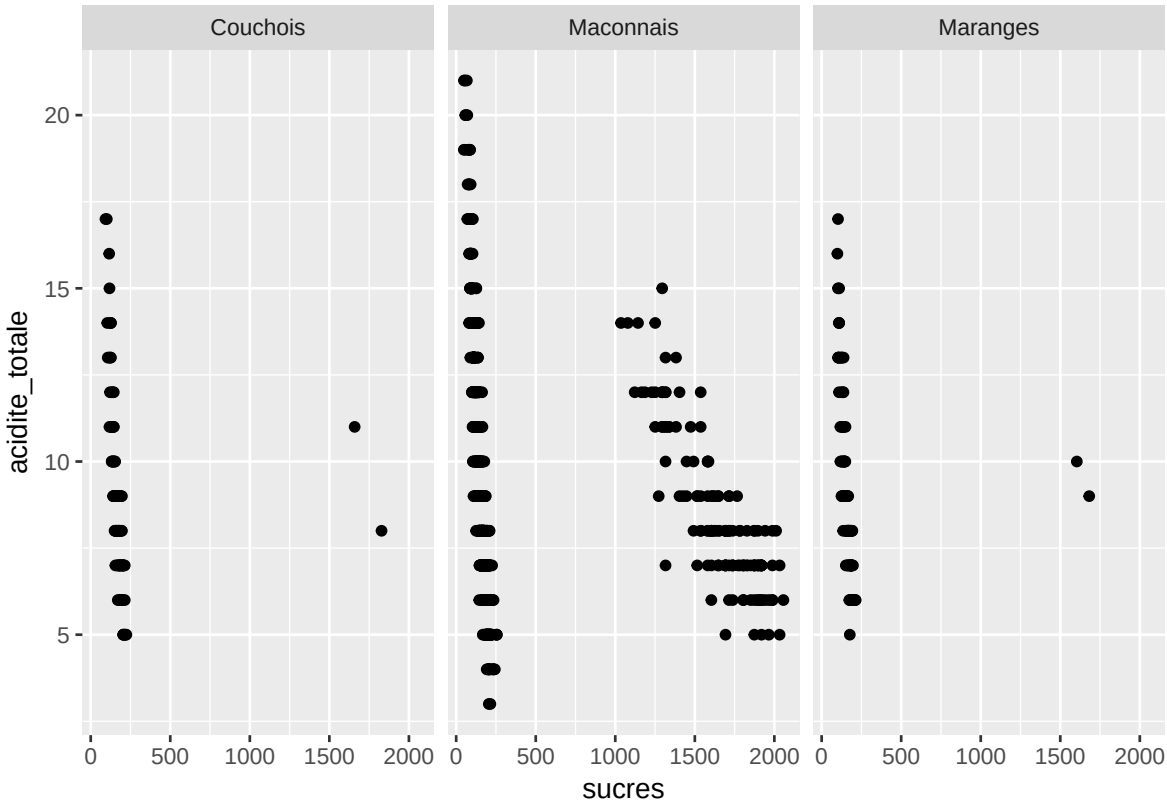

```
ggplot(raisin, aes(sucres, acidite_totale, colour = secteur)) + geom_point()
```



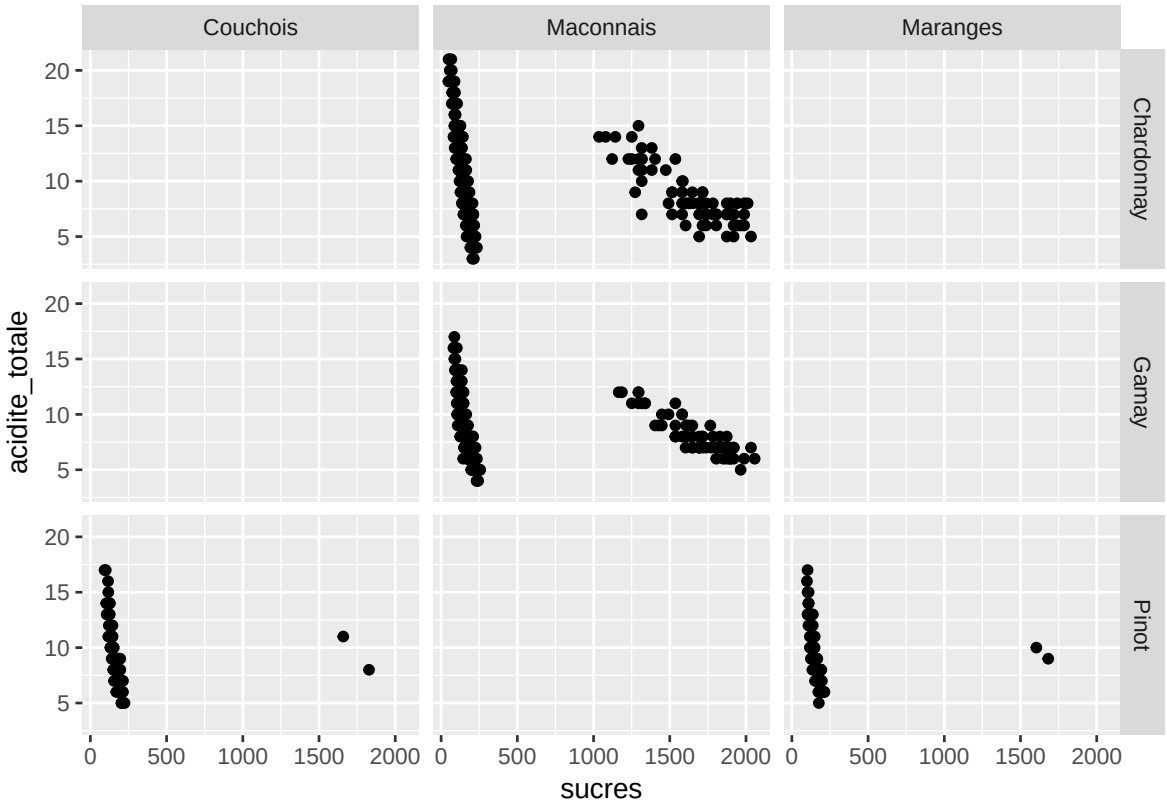
```
# Pour confirmer et ne pas se laisser abuser
# par la superposition de certains points, on
# utilise le facetting
ggplot(raisin, aes(sucres, acidite_totale)) + geom_point() +
  facet_wrap(~cepage)
```




```
ggplot(raisin, aes(sucres, acidite_totale)) + geom_point() +
  facet_wrap(~secteur)
```

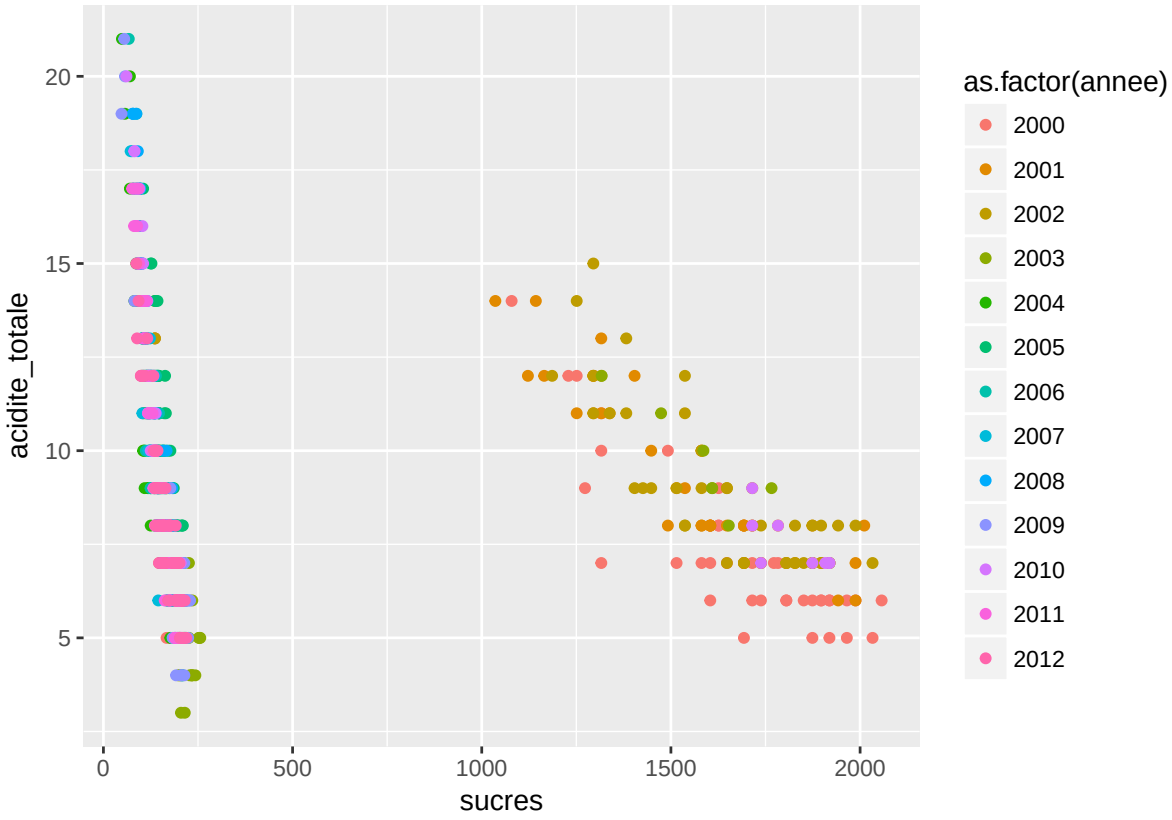



```
ggplot(raisin, aes(sucres, acidite_totale)) + geom_point() +
  facet_grid(cepage~secteur)
```




```
# Il apparaît clairement que c'est la localisation dans le
# maconnais ou les cépages chardonnay et gamay qui lui
# sont spécifiques (au sein de la Saône-et-Loire) qui
# semble expliquer l'oscillation entre deux types de relations
# entre sucres et acidité totale.

# Peut-être cette oscillation dépend-elle des années ?
ggplot(raisin[raisin$secteur == "Maconnais", ], aes(sucres, acidite_totale, colour
```




```
# Ce n'est pas évident, à nouveau on utilise le facetting
ggplot(raisin[raisin$secteur == "Maconnais", ], aes(sucres, acidite_totale, colour
```


```
# Le raisin du secteur du maconnais, quel que soit sa commune,
# semble présenter un profil sucres-acidite_totale particulier
# de 2000 à 2003 (plus sucré).
```

Cas pratique 24 Représentations à partir du fichier de l'EEC

On cherche à représenter les liens entre : position sur le marché du travail (variable `ACTEU`) et caractéristiques socio-démographiques d'une part ; salaire redressé (varible `SALRED`) et caractéristiques socio-démographiques d'autre part. Proposez des représentations graphiques pertinentes à l'aide de `ggplot2`.