Aide-mémoire pour IFT-1902

Gabriel Crépeault-Cauchon

20 décembre 2017

Table des matières

1 Introduction

Section plus qualitative à compléter plus tard

2 Terminal

2.1 Commandes de base

Voici quelques commandes essentielles à connaître pour naviguer terminal Bash :

Important : certaines fonctions du *Shell* demandent un espace, d'autre des tirets.

- `cd` pour *change directory* (nous permet de changer de dossier dans le terminal) ;
- `pwd` pour savoir le chemin d'accès dans lequel on se trouve en ce moment ;
- `ls` pour faire apparaître la liste de tous les fichiers dans le dossier actuel ;
- `ls -a` fait apparaître **tous les dossiers**, même ceux qui commence par un point (exemple, *.Renviron*) ;
- `touch` Créer un fichier de texte brut ;
- `rm`permet
- `mkdir` Créer un dossier dans le répertoire où l'on se trouve ;
- `rmdir` Supprimer un dossier dans le répertoire où l'on se trouve ;
- `mv <nomfichier> <destination>` : dépalacer un fichier
- `nano <nomfichier>` : éditeur de texte intégré à Bash pour modifier un fichier de texte brut (pas très convial, mais permet de faire rapidement des petites modifications).
  - Pour sortir de l'éditeur *nano*, on fait `Ctrl+X`

2.2 Fonctions utiles du terminal et les Regex

2.2.1 Expressions régulières (*Regex*)

Voici un aide-mémoire sur les expressions régulières

Expression	définition	Exemples
`*`	Cherche 1 ou plusieurs occurences du caractère précédent	`ga*` va trouver ga, g et gaaaa
`?`	Cherche 1 ou 0 occurences du caractère précédent	`ga` va trouver ga et g, mais pas gaaaa
`+`	Cherche 1 ou plusieurs du caractère précédent	`ga+` va trouver
strut	Quand on cherche un caractère spécial (utilisé dans les *Regex*)	`Hungry\?` va trouver Hungry ?
`.`	Cherche n'importe quel caractères	`ga.` va trouver gab, garage, gabon, etc.
`( )`	cherche une chaîne de caractères	
`[ ]`	cherche parmi une liste de caractères	`[gb]ateaux` va trouver gateaux et bateaux
`|`	Va chercher la chaîne de caractères avant ou après le symbole	`(lun)|(mar)di` va trouver lundi et mardi

Expression	définition	Exemples
`{ }`	Spécifie le nombre (consécutif) d'occurence	
`^`	le caractère doit se trouver au début de la ligne	`^http` nous permet de trouver des URL
`$`	le caractère doit terminer la ligne	`(.com)$` nous permet de trouver les adresses internet se terminant par .com

2.2.2 `grep`

2.2.3 `awk`

```
awk -F "," '{print $1 " " $5}' IAG.TO.csv
```

```
## Date Close
## 2017-11-17 59.990002
## 2017-11-20 59.900002
## 2017-11-21 60.330002
## 2017-11-22 60.189999
## 2017-11-23 59.700001
## 2017-11-24 59.669998
## 2017-11-27 59.380001
## 2017-11-28 59.049999
## 2017-11-29 59.139999
## 2017-11-30 60.169998
## 2017-12-01 59.889999
## 2017-12-04 60.270000
## 2017-12-05 60.279999
## 2017-12-06 59.860001
## 2017-12-07 59.459999
## 2017-12-08 60.000000
## 2017-12-11 59.570000
## 2017-12-12 59.650002
## 2017-12-13 59.779999
## 2017-12-14 58.820000
## 2017-12-15 58.980000
```

2.3 Utiliser Git sur le terminal

2.3.1 Configuration

À la première utilisation de git via le terminal *Bash* il faut configurer quelques informations :

- `git config --global user.name "<Nom>"` Configurer son nom tel qu'on désire qu'il apparaisse sur Git.
- `git config --global user.email "<courriel>"` Configurer l'adresse courriel associée.
- `git config --global core.editor open` Si vous oubliez de préciser une description lors d'un *commit* (sera vu à la prochaine section), il va simplement ouvrir un fichier texte brut.

— `git config --list` Juste pour valider que les informations entrées ci-dessus sont enregistrées adéquatement.

2.3.2 Collaborer sur un projet

Pour faire le suivi des versions d'un projet informatique, il est utile d'utiliser Git. Voici un résumé des fonctions (du terminal) à savoir utiliser :

— `git init` : Créer un répertoire (*repository*) *Git* Dans le dossier actuel. *Astuce* : il est plus simple de créer son *repository* directement sur Github puis le cloner dans le dossier désiré ;
— `git clone <https://...>` *Cloner* (ou si on préfère, télécharger) un répertoire Git dans le dossier actuel. **Attention**, on va cloner une seule fois un répertoire, car par la suite on va *pull* les modifications du dépôt ;
— `git pull` commandes qu'on utilise seulement si on a déjà cloné le répertoire. Nous permet d'avoir les mises à jour ;
— `git status` permet de voir si il y a des fichiers dans notre dossier qui n'apparaissent pas (ou que les modifications n'apparaissent pas) sur le répertoire Git. Si c'est le cas, elles seront affichés en **rouge**.
— `git add` pour ajouter les modifications dans le dépôt. Après cette étape, on doit confirmer nos modifications par la commande *commit*
  — Si on utilise `git add -A`, tous les fichiers seront ajoutés au prochain *commit*
— `git commit -m "<description de la modif.>"` On confirme notre modification au travail et on décrit *très brièvement* ce qu'on a fait comme modification
— `git push` *pousser* au serveur les modifications qu'on a fait. Après cette étape, si on va sur Github, nos modifications apparaîtront.

3 Programmation en R

3.1 Commandes et expressions de R

3.1.1 Commandes de base

— `save image()` Si on veut sauvegarder l'espace de travail et son environnement. **Rarement utilisée**, sauf si on veut sauvegarder la valeur d'une variable (qui est longue à obtenir)
— `getwd()` obtenir le répertoire de travail dans lequel on se trouve actuellement
— `setwd(<chemin d'accès>)` Changer le répertoire de travail actuel
— `help()` obtenir de l'aide sur une fonction ou une commande en particulier. On peut aussi accéder au manuel d'instruction de R avec `help.start`
— `ls()` : voir tous les objets de l'environnement global
— `rm()` : pour supprimer un objets
— `rm(list = ls())` : supprimer tous les objets dans l'environnement global

3.1.2 Opérateurs de R

Opérateur	Fonction
`$`	extraction d'une liste
`[]`	indiçage
`^`	puissance
`-`	changement de signe
`:`	génération d'une suite

Opérateur	Fonction
%*% %% %/%	produit matriciel, modulo, division entière
* /	multiplication, division
+ -	addition, soustraction
< <= == >= > !=	plus petit, plus petit ou égal, égal, plus grand ou égal, plus grand, différent de
!	négation logique
&	ET logique
|	OU logique
<-	affectation (méthode la plus utilisée)

3.1.3 Création d'une suite ou séquence

Séquence de chiffres

```
seq(from = 10, to = 20, by = 2)
```

```
## [1] 10 12 14 16 18 20
```

```
x <- c(1,2,3,8)
seq(x)
```

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

```
seq_len(5)
```

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

```
y <- c(10,14,3,2)
seq_along(y)
```

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

Échantillon de données aléatoire

```
x <- sample(1:5, size = 8, replace = TRUE, prob = c(0.1,0.2,0.2,0.25,0.25)) ; x
```

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

Séquence de lettres

```
x <- c(1,2,3)
letters[x]
```

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

```
x <- c(24,25,26)
LETTERS[x]
```

```
## [1] "X" "Y" "Z"
```

3.2 Objets de R

3.2.1 Information sur un objet de R

— `mode()` : Mode d'un objet
— `length()` : Longueur d'un objet
— `nchar()` : nombre de caractères
— `class()` : classe d'un objet
— `summary()` : beaucoup d'information sur l'objet

3.2.2 Mode d'un objet

Mode	Contenu de l'objet
`numeric`	nombres réels
`complex`	nombres complexes
`logical`	valeurs booléennes
`character`	chaînes de carctères
`function`	fonction
`list`	liste
`expression`	expressions non évaluées

```
char <- c("a","b","c")
mode(char)
```

```
## [1] "character"
```

```
num <- c(1:5)
mode(num)
```

```
## [1] "numeric"
```

Si on créé un vecteur qui contient des données de plus d'un mode, il va convertir les autres données dans le mode le plus «puissant», en respectant l'ordre suivant :

1. `list`
2. `character`
3. `numeric`
4. `logical`

```
a <- c(TRUE, "test",1:2,list(1)) ; a
```

```
## [[1]]
## [1] TRUE
##
## [[2]]
## [1] "test"
##
## [[3]]
## [1] 1
##
## [[4]]
## [1] 2
##
```

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

```
mode(a)
```

```
## [1] "list"
```

```
b <- c(FALSE,0:2,"test") ; b
```

```
## [1] "FALSE" "0"     "1"     "2"     "test"
```

```
mode(b)
```

```
## [1] "character"
```

```
c <- c(FALSE,1:2) ; c
```

```
## [1] 0 1 2
```

```
mode(c)
```

```
## [1] "numeric"
```

```
x <- c(TRUE,1,4,"GAB")
class(x)
```

```
## [1] "character"
```

```
mode(x)
```

```
## [1] "character"
```

3.2.3 NA, NaN, NULL et Inf

```
0/0
```

```
## [1] NaN
```

```
Inf/Inf
```

```
## [1] NaN
```

```
Inf-Inf
```

```
## [1] NaN
```

```
1/0
```

```
## [1] Inf
```

```
Inf
```

```
## [1] Inf
```

```
Inf^Inf
```

```
## [1] Inf
```

```
5 + NULL
```

```
## numeric(0)
```

3.3 Vecteurs

3.3.1 Création d'un Vecteur

```
(x <- c(1,2,3))
```

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

```
(y <- vector(mode = "numeric", length = 5))
```

```
## [1] 0 0 0 0 0
```

3.3.2 Ajout d'étiquettes au vecteur

il existe 2 façon :

```
x <- c(1,2)
names(x) <- c("a","b") ; x
```

```
## a b
## 1 2
```

ou bien

```
x <- c(a=1, b=2) ; x
```

```
## a b
## 1 2
```

3.3.3 Indiçage du vecteur

```
x <- c(A = 1, B = 2, C = 3, D = 4, E = TRUE, G = NA) ; x
```

```
##  A  B  C  D  E  G
##  1  2  3  4  1 NA
```

```
x[3]            # 3e élément
```

```
## C
## 3
```

```
x[c(1,5)]       # 1er et 5e élément
```

```
## A E
## 1 1
```

```
x[-3]           # tout sauf le 3e élément
```

```
##  A  B  D  E  G
##  1  2  4  1 NA
```

```
x[-c(3,4)]      # tous sauf le 3e et 4e élément
```

```
##  A  B  E  G
##  1  2  1 NA
```

```
x[!is.na(x)]    # tout ce qui n'est pas NA
```

```
## A B C D E
## 1 2 3 4 1
```

```
x[x<4]          # tous les éléments qui sont <4
```

```
##    A    B    C    E <NA>
##    1    2    3    1   NA
```

```
x[c("G","B")]   # seulement les éléments taggé "G" et "B"
```

```
##  G  B
## NA  2
```

```
x[]             # tous les éléments
```

```
##  A  B  C  D  E  G
##  1  2  3  4  1 NA
```

3.3.4 Opérations sur les vecteurs

3.4 Matrices

3.4.1 Création d'une matrice

On peut créer une matrice avec la fonction `matrix` :

```
x <- (matrix(1:10, nrow = 2, ncol = 5, byrow = T))
```

L'option *byrow* est pour que la matrice se remplisse par ligne.

3.4.2 informations supplémentaires sur notre matrice

```
attributes(x)
```

```
## $dim
## [1] 2 5
```

```
nrow(x)         # nombre de lignes
```

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

```
ncol(x)         # nombre de colonnes
```

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

3.4.3 Tableau

```
(tableau <- array(1:30, dim =c(2,5,3)))
```

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

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

```
?array
```

3.4.4 Indiçage d'une matrice

Comme un vecteur, on peut indicer une matrice.

```
x[1]            # 1er élément
```

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

```
x[1,2]          # 1ère ligne, 2e colonne
```

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

```
x[2,]           # toute la 2e ligne
```

```
## [1]  6  7  8  9 10
```

```
x[,1]           # toute la 1ère colonne
```

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

```
x[c(1,2),2]     # les 2 premiers éléments de la première colonne
```

```
## [1] 2 7
```

3.4.5 Opérations sur des matrices

```
rbind(x,x[1,]*x[2,])    # ajout de ligne à la matrice
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]    6   14   24   36   50
```

```
cbind(x,5)              # ajout d'une colonne à la matrice
```

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

3.5 Listes

3.5.1 Construction d'une liste

```
(sondage <- list(nom = c("Justin","William","Charlie"),
          age = c(35,23,14),
          job = c("Politicien","Menuisier","Etudiant"),
          citoyen = c(T,T,F)))
```

```
## $nom
## [1] "Justin"  "William" "Charlie"
##
## $age
## [1] 35 23 14
##
## $job
## [1] "Politicien" "Menuisier"  "Etudiant"
##
## $citoyen
## [1]  TRUE  TRUE FALSE
```

3.5.2 Indiçage d'une liste

```
sondage[2]          # 2e élément de la liste
```

```
## $age
## [1] 35 23 14
```

```
sondage$age         # idem
```

```
## [1] 35 23 14
```

```
# pour pouvoir travailler avec les données, il faut faire un double indiçage :
mean(sondage[2])
```

```
## Warning in mean.default(sondage[2]): argument is not numeric or logical:
## returning NA
```

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

```
mean(sondage[[2]])
```

```
## [1] 24
```

```
sondage[[c(1,2)]]   # 2e élément du 1er élément de la liste
```

```
## [1] "William"
```

3.5.3 Défaire une liste

On peut *défaire* une liste avec `unlist`. La conversion se fait vers le mode le plus puissant.

3.6 Data frame

3.6.1 construire un Data frame

Avec la fonction `dataframe`, on peut faire quelque

```
(form <- data.frame(prenom = c("Suzie","Mario","Jean"),
           nom = c("Tremblay","Gagnon","Cote"),
           age = c(12,13,14),
           fumeur = c(T,T,F),
           salaire = c(40000,45000,135000)))
```

```
##   prenom      nom age fumeur salaire
## 1  Suzie Tremblay  12   TRUE   40000
## 2  Mario   Gagnon  13   TRUE   45000
## 3   Jean     Cote  14  FALSE  135000
```

3.6.2 fonction subset

La fonction `subset` permet d'extraire de l'information dans le data frame en appliquant un filtre.

```
subset(form, salaire >42000, select = age)
```

```
##   age
## 2  13
## 3  14
```

3.7 Importation et exportation de données

3.7.1 Exportation

```
# Pour la fonction cat, on peut ajouter plusieurs objets de R dans la concaténation
cat(num,"Ceci est un commentaire", file = "sondage.data")
# La fonction write n'accepte qu'un seul objet (ou matrice)
write(tableau, file = "sondage.data", ncolumns = 5)
# spécifiquement pour exporter en .csv (virgule)
write.csv(tableau,file = "sondage.csv")
# encore en .csv (mais séparé par des points-virgules)
write.csv2(char, file = "caracteres.csv2")
```

3.7.2 Importation

```
scan("sondage.data")    # pour importer des données
```

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

```
read.table(file = "IAG.TO.csv",header = T, sep = ",")
```

```
##          Date  Open  High   Low Close Adj.Close Volume
## 1  2017-11-17 60.46 60.56 59.86 59.99  59.61127 154000
## 2  2017-11-20 59.88 60.17 59.82 59.90  59.52183 168400
## 3  2017-11-21 60.00 60.50 59.85 60.33  59.94912 153700
```

```
## 4  2017-11-22 60.39 60.64 59.85 60.19  59.81000 152200
## 5  2017-11-23 60.01 60.01 59.47 59.70  59.70000  57800
## 6  2017-11-24 59.87 60.12 59.61 59.67  59.67000  70100
## 7  2017-11-27 59.70 59.97 59.36 59.38  59.38000 103800
## 8  2017-11-28 59.37 59.62 58.67 59.05  59.05000 192000
## 9  2017-11-29 59.20 59.76 59.05 59.14  59.14000 159900
## 10 2017-11-30 59.38 60.69 58.90 60.17  60.17000 367900
## 11 2017-12-01 60.19 60.31 59.24 59.89  59.89000 156000
## 12 2017-12-04 60.00 60.52 59.83 60.27  60.27000 189200
## 13 2017-12-05 60.35 61.02 60.08 60.28  60.28000 170400
## 14 2017-12-06 60.14 60.19 59.44 59.86  59.86000 148100
## 15 2017-12-07 59.85 59.91 59.31 59.46  59.46000 101000
## 16 2017-12-08 59.58 60.18 59.45 60.00  60.00000  90200
## 17 2017-12-11 59.99 60.07 59.42 59.57  59.57000  87000
## 18 2017-12-12 59.52 59.94 59.31 59.65  59.65000  91900
## 19 2017-12-13 59.61 60.16 59.60 59.78  59.78000 113500
## 20 2017-12-14 59.78 59.85 58.78 58.82  58.82000  90900
## 21 2017-12-15 59.07 59.53 58.92 58.98  58.98000 236645
```

```
read.csv(file = "IAG.TO.csv")   # Comprends par défaut qu'il y a un titre
```

```
##          Date  Open  High   Low Close Adj.Close Volume
## 1  2017-11-17 60.46 60.56 59.86 59.99  59.61127 154000
## 2  2017-11-20 59.88 60.17 59.82 59.90  59.52183 168400
## 3  2017-11-21 60.00 60.50 59.85 60.33  59.94912 153700
## 4  2017-11-22 60.39 60.64 59.85 60.19  59.81000 152200
## 5  2017-11-23 60.01 60.01 59.47 59.70  59.70000  57800
## 6  2017-11-24 59.87 60.12 59.61 59.67  59.67000  70100
## 7  2017-11-27 59.70 59.97 59.36 59.38  59.38000 103800
## 8  2017-11-28 59.37 59.62 58.67 59.05  59.05000 192000
## 9  2017-11-29 59.20 59.76 59.05 59.14  59.14000 159900
## 10 2017-11-30 59.38 60.69 58.90 60.17  60.17000 367900
## 11 2017-12-01 60.19 60.31 59.24 59.89  59.89000 156000
## 12 2017-12-04 60.00 60.52 59.83 60.27  60.27000 189200
## 13 2017-12-05 60.35 61.02 60.08 60.28  60.28000 170400
## 14 2017-12-06 60.14 60.19 59.44 59.86  59.86000 148100
## 15 2017-12-07 59.85 59.91 59.31 59.46  59.46000 101000
## 16 2017-12-08 59.58 60.18 59.45 60.00  60.00000  90200
## 17 2017-12-11 59.99 60.07 59.42 59.57  59.57000  87000
## 18 2017-12-12 59.52 59.94 59.31 59.65  59.65000  91900
## 19 2017-12-13 59.61 60.16 59.60 59.78  59.78000 113500
## 20 2017-12-14 59.78 59.85 58.78 58.82  58.82000  90900
## 21 2017-12-15 59.07 59.53 58.92 58.98  58.98000 236645
```

On aurait pu aussi utiliser `read.csv2()` si on a un jeu de données où les champs sont séparés par des points virgules.

3.8 Fonctions

3.8.1 Quelques fonctions déjà programmées dans R

3.8.1.1 Statistiques

Il existe plusieurs fonctions déjà installées pour des calculs statistiques simples :

```r
x <- sample(1:20, 10, replace=T)
y <- sample(1:30, 10, replace = T)
sum(x)      # Somme d'un vecteur
```

```
## [1] 100
```

```r
mean(x)     # Moyenne
```

```
## [1] 10
```

```r
var(x)      # Variance
```

```
## [1] 47.11111
```

```r
cov(x,y)    # Covariance (nécessite de mettre 2 vecteurs en argument)
```

```
## [1] 19.77778
```

Lorsqu'on a un vecteur ou une matrice, on peut utiliser des fonctions qui font certains calculs par ligne ou par colonne :

```r
tableau <- matrix(1:6, nrow = 3, ncol = 2, byrow = T)
rowSums(tableau)
```

```
## [1]  3  7 11
```

```r
rowMeans(tableau)
```

```
## [1] 1.5 3.5 5.5
```

```r
colSums(tableau)
```

```
## [1]  9 12
```

```r
colMeans(tableau)
```

```
## [1] 3 4
```

3.8.1.2 les fonctions "is.blablabla"

Il y a plusieurs fonctions dans R qui permettent d'obtenir une réponse booléenne **vrai** ou **faux** en validant une information.

```r
is.matrix(y)        # une liste n'est pas une matrice
```

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

```r
is.array(y)         # comme on voit, une matrice est aussi un tableau
```

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

```r
is.function(y)      # Est-ce que l'argument est une fonction?
```

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

```r
is.character(char)  # de mode "character"?
```

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

```r
is.numeric(num)     # de mode "numeric"?
```

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

```r
is.vector(y)        # est-ce que c'est un vecteur?
```

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

3.8.1.3 Manipulation de données

Triage de données : malgré le fait qu'on a bâti en cours des algorithmes (voir section plus loin), il est beaucoup plus efficace d'utiliser la fonction `sort()` déjà conçue de R.

3.8.2 Fonctions d'applications

Lorsqu'on travaille avec des *data frame* ou bien des tableaux dans R, on est mieux d'utiliser des fonctions d'applications (plutôt que des boucles qui ralentissent le temps d'éxécution d'une fonction).

- le premier argument est la vecteur ou le tableau sur lequel on veut *appliquer* la fonction
- le 2e argument est la dimension (1 = ligne, 2= colonne, etc.) sur laquelle on veut appliquer itérativement la fonction
- La fonction désirée sur chaque ligne ou colonne est donnée en 3e argument
- si la fonction a besoin de plusieurs arguments de spécifiés, on peut les spécifier par la suite en 4e, 5e argument etc...

Exemples

```r
apply(tableau, 1, sum)    # fait la somme des lignes
```

```
## [1]  3  7 11
```

```r
apply(tableau, 2, mean)    # idem pour colonne
```

```
## [1] 3 4
```

3.8.3 Algorithmes

3.8.3.1 Algorithme de tri

3.8.3.1.1 Insertionsort

$$runtime = O(N^2)$$

```r
insertionsort <- function(x)
{
  xlen <- length(x)
  for(i in seq_len(xlen))
  {
      for (j in seq_len(i-1))
      {
        if (x[j] > x[i])
        x <- c(x[seq_len(j-1)],
                  x[i],x[j],
                  x[j + seq_len(i-j-1)],
                  x[i + seq_len(xlen-i)])
      }
```

```
  }
  x
}
```

3.8.3.1.2 Selectionsort

$$runtime = O(N^2)$$

```
selectionsort <- function(x)
{
  xlen <- length(x)      # sert souvent
  for (i in seq_len(xlen))
  {
    ## recherche de la position de la plus petite valeur
    ## parmi x[i], ..., x[xlen]
    i.min <- i
    for (j in i:xlen)
    {
      if (x[j] <x[i.min])
        i.min <- j
      ## à mesure que i augmente, j ne considère pas
      ## les première position, puisqu'elles sont déjà triées
    }
    ## échange de x[i] et x[i.min]
    x[c(i, i.min)] <- x[c(i.min, i)]
  }
  x
}
```

3.8.3.1.3 Bubblesort

$$runtime = O(N^2)$$

```
bubblesort <- function(x)
{
    ind <- 2:length(x)     # suite sert souvent

    not_sorted <- TRUE     # entrer dans la boucle
    while (not_sorted)
    {
        not_sorted <- FALSE
        for (i in ind)
        {
            j <- i - 1
            if (x[i] < x[j])
            {
                x[c(i, j)] <- x[c(j, i)]
                not_sorted <- TRUE
                next
            }
        }
```

```
    }
    x
}
```

3.8.3.1.4 Countingsort

$$runtime = O(N + M)$$

```
countingsort <- function(x, min, max)
{
    min1m <- min - 1               # sert souvent
    counts <- numeric(max - min1m) # initialisation

    for (i in seq_along(x))
    {
        j <- x[i] - min1m
        counts[j] <- counts[j] + 1
    }

    ## suite des nombres de 'min' à 'max' répétés chacun le
    ## bon nombre de fois
    rep(min:max, counts)
}
```

3.8.3.1.5 Bucketsort

On place nos données dans des paniers (*buckets*), puis on les place en ordre. Ensuite on remets nos *buckets* dans le bon ordre.

$$runtime = O(N + M)$$

```
bucketsort <- function(data, nbuckets)
{
  # Création des buckets
  max = max(data)
  bucket  <- vector(mode = "list", length = nbuckets)
  etendue <- max/nbuckets

  # distribution des données dans les buckets
  for (i in seq_along(data))  # on veut la longueur de data
  {
    j <- ceiling(data[i]/etendue) # pour sélectionner le bon
    bucket[[j]] <- c(bucket[[j]],data[i])
  }

  # tri des différents buckets
  for (i in seq_len(nbuckets))  # on veut le chiffre de la variable
  {
    if (!is.null(bucket[[i]]))  # is pas NULL, on le trie!
    bucket[[i]] <- sort(bucket[[i]])
  }

  # On défait" la liste pour avoir nos données triées
```

```
  unlist(bucket)
}
```

3.8.3.2 Algorithme de recherche

3.8.3.2.1 Linear Search

Cet algorithme va seulement fonctionner si les données sont triées.

$$runtime = O(N)$$

```
linearsearch <- function(x,target)
{
  for (i in seq_along(x))
  {
    if  (x[i] == target)
        return(i)     # retourne la position de l'élément recherché
    if (x[i] > target)
        return("la valeur n'est pas dans le vecteur")
  }
  NA
}
```

3.8.3.2.2 Binary Search

$$runtime = O(\log(N))$$

Principe de l'algorithme : on fixe une valeur au millieu. si la valeur recherchée est plus grande, on vient d'éliminer la moitié des données. La recherche va se faire dans la partie supérieure.

```
binary_search <- function(x, target)
{
  min = 1
  max = length(x)

  while (min <= max)
  {
    mid = floor((max + min)/2)    # choisir floor/ceiling arbitraire
    if (target < x[mid])
      max <- mid - 1
    else if (target > x[mid])
      min <- mid + 1
    else return(mid)
  }
  NA      # la valeur qu'on cherche n'est pas dans le vecteur!
}
```

3.8.3.2.3 Interpolation search

va faire un *guess* sur l'endroit approximatif de la valeur, considérant que les données sont triées. Ensuite, il va interpoler dans le bon sens pour se rapprocher (de façon itérative) de la bonne valeur.

$$runtime = O(\log(\log(N)))$$

```
interpolation_search <- function(x, target)
{
  min <-  1
  max <-  length(x)
  while (min <= max)
  {
    mid <- min + floor((max - min)*
      (target - x[min]) / (x[max] - x[min]))
    if (target < x[mid])
      max <- mid - 1
    else if (target > x[mid])
      min <- mid  + 1
    else
      return(mid)
  }
  "La valeur cherchée n'est pas dans le vecteur"
}
```