Séance 4

Algorithmique I

BIO109 - Dominique Gravel

Laboratoire d'écologie intégrative | Integrative Ecology Lab [IE]

Retour sur les fonctions

Trucs pour essayer de programmer une fonction

Commencer en travaillant localement

```
quadrats ← read.csv2(file = "donnees/quadrats.csv", header = TRUE,
    stringsAsFactors = FALSE, row.names = 1)
acsa_abondance ← quadrats$acsa
# Abondance moyenne
moy ← mean(acsa_abondance)
# Abondance totale
tot ← sum(acsa_abondance)
# Coefficient de variation de l'abondance
CV ← sd(acsa_abondance)/mean(acsa_abondance)
# Densité maximale dans un quadrat
max_ab ← max(acsa_abondance)
# Densité minimale dans un quadrat
min_ab ← min(acsa_abondance)
```

Trucs pour essayer de programmer une fonction

Tout envelopper dans une fonction

```
ma_fonction ← function(abondance) {

    # Abondance moyenne
    moy ← mean(abondance)
    # Abondance totale
    tot ← sum(abondance)
    # Coefficient de variation de l'abondance
    CV ← sd(abondance)/mean(abondance)
    # Densité maximale dans un quadrat
    max_ab ← max(abondance)
    # Densité minimale dans un quadrat
    min_ab ← min(abondance)
    # Retour extérieur
    return(c(moy, tot, CV, max_ab, min_ab))
 }
```

Trucs pour essayer de programmer une fonction

On fait un essai sur un premier objet que l'on connait

```
ma_fonction(acsa_abondance)
```

```
## [1]    6.665331 3326.000000    1.228033   49.000000    0.000000
```

Ensuite on fait un test sur un second objet

```
ma_fonction(quadrats$abba)
```

```
## [1]    5.202405 2596.000000    1.366285   36.000000    0.000000
```

Trucs pour essayer de programmer une fonction

Finalement on peut l'appliquer sur l'ensemble des données

```
apply(quadrats, 2, ma_fonction)
```

Les algorithmes

Définition

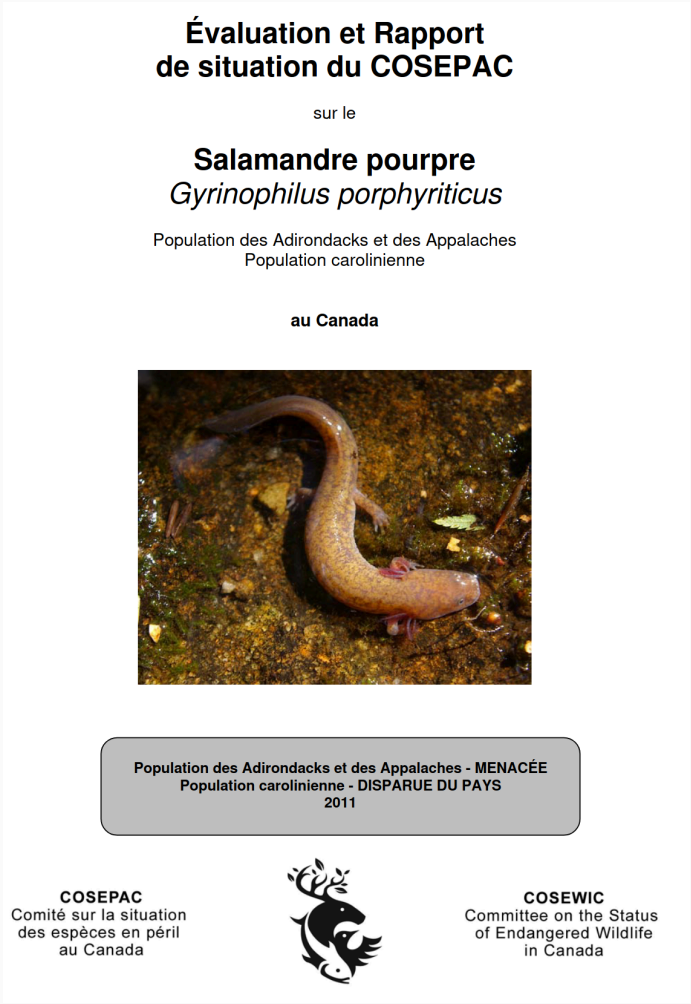
"Ensemble de règles opératoires dont l'application permet de résoudre un problème énoncé au moyen d'un nombre fini d'opérations."
- Larousse -

Une recette de cuisine est une sorte d'algorithme

Les boucles

Mise en situation

Vous étudiez la démographie de la population de salamandres pourpres dans le ruisseau du massif des monts Sutton. Vos données sont très simples, vous avez une mesure d'abondance à deux points d'échantillonnage au long du ruisseau. Vous devez vérifier si la population est en croissance, stable ou en déclin sur une séquence de 5 ans.



Évaluation et Rapport de situation du COSEPAC

sur le

Salamandre pourpre
Gyrinophilus porphyriticus

Population des Adirondacks et des Appalaches
Population carolinienne

au Canada

Population des Adirondacks et des Appalaches - MENACÉE
Population carolinienne - DISPARUE DU PAYS
2011

COSEPAC
Comité sur la situation des espèces en péril au Canada

COSEWIC
Committee on the Status of Endangered Wildlife in Canada

Mise en situation

Le taux de croissance est donné par l'équation suivante :

$$\frac{N_{t+1} - N_t}{N_t}$$

où

$N_t =$ nombre d'individus à l'année t et

$N_{t+1} =$ nombre d'individus à l'année suivante.

Mise en situation

Les données ressemblent à ce qui suit:

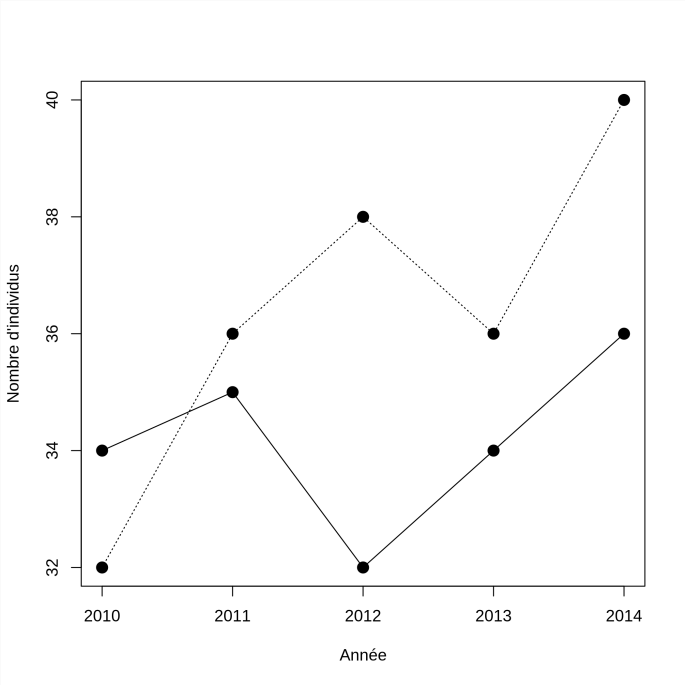
```
donnees ← matrix(nr = 5, nc = 2)
donnees[,1] ← c(34,35,32,34,36)
donnees[,2] ← c(32,36,38,36,40)
an ← c(2010:2014)
rownames(donnees) ← an
colnames(donnees) ← c("Site1", "Site2")
head(donnees)
```

```
##      Site1 Site2
## 2010    34    32
## 2011    35    36
## 2012    32    38
## 2013    34    36
## 2014    36    40
```

Mise en situation

Les données ressemblent à ce qui suit:

```
plot(an,donnees[,1],type = "l", xlab = "Année",ylab = "Nombre d'individus", ylim=c(3
lines(an,donnees[,2], lty = 3)
points(an,donnees[,1], pch = 19,cex=1.5)
points(an,donnees[,2], pch = 19,cex=1.5)
```



Mise en situation

Maintenant, on doit calculer le taux de croissance annuel pour chaque population. Commençons pour l'intervalle entre l'an 1 et l'an 2, sur le site 1 :

```
# 0. Sauvons les données dans une matrice "lambda"
lambda ← matrix(nr = 4, nc = 2)

# 1. Calculons le taux de croissance
lambda[1,1] ← (donnees[2,1] - donnees[1,1]) / donnees[1,1]
```

Ensuite, on fait l'an 2 :

```
lambda[2,1] ← (donnees[3,1] - donnees[2,1]) / donnees[2,1]
```

Mise en situation

Très rapidement, on réalise que c'est assez fastidieux de refaire cet exercice à la main, mais c'est faisable.

Imaginez cependant que vous découvrez un jour qu'un passionné des salamandres a déjà fait une étude similaire sur ce site, et par miracle vous obtenez des séries temporelles de 50 ans réparties sur 25 points d'échantillonnage. Il faudra changer de technique...

Mise en situation

La semaine dernière, nous avons vu comment généraliser une séquence d'opérations au moyen d'une fonction. Pratique !

Mais nous devrions appliquer la fonction pour chacun des points d'échantillonnage. 25 appels de la fonction, avec 25 chances de se tromper...

Comment est-ce qu'on peut généraliser des opérations qui sont répétées très souvent ? C'est le principe de la boucle.

Mise en situation

La solution rapide ressemblerait à

```
n_sites ← ncol(donnees)
n_annees ← nrow(donnees)
lambda ← matrix(nr = n_annees-1, nc = n_sites)
for(i in 1:(n_annees-1)) {
    for(j in 1:n_sites) {
        lambda[i,j] ← (donnees[i+1,j] - donnees[i,j]) / donnees[i,j]
    }
}
```

donnees

```
##      Site1 Site2
## 2010    34    32
## 2011    35    36
## 2012    32    38
## 2013    34    36
## 2014    36    40
```

lambda

```
##             [,1]        [,2]
## [1,]  0.02941176  0.12500000
## [2,] -0.08571429  0.05555556
## [3,]  0.06250000 -0.05263158
## [4,]  0.05882353  0.11111111
```

Définition

Une boucle est une commande qui permet de répéter une série d'instructions sous des conditions définies de départ et de fin. C'est une commande de base de l'algorithmique.

Anatomie de la boucle

```
depart ← 1
fin ← 5
for(etape in depart:fin) {
    print(etape)
}
```

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

`etape` est un compteur de la position dans la boucle.

L'exécution une fois d'un groupe de commandes dans un boucle s'appelle une **itération**.

Quelques exemples simples

Par défaut R augmente toujours le compteur de 1 unité

```
for(etape in 1:5) {
    print(etape*2)
}
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 8
## [1] 10
```

`print` est une fonction simple qui permet d'écrire dans le terminal une information dans la console. Ici, le résultat du compteur multiplié par 2.

Un premier exercice simple

Transformez en celcius une séquence de température en fahrenheit qui va de -50 à 100, par bond de 1. Rappelez vous que la conversion est (F - 32) * 5/9.

- Déterminez le point de départ de la séquence
- Déterminez le point de fin de la séquence
- Faites le calcul approprié
- Imprimez le résultat

Un premier exercice simple

```
PROGRAM CONVERSION

FOR F IN -50:100
    C = (F - 32) * 5/9
    PRINT C
END FOR
```

Le pseudo-code est une façon de décrire un algorithme sans utiliser de syntaxe spécifique à un langage de programmation. C'est une description générale.

Un premier exercice simple

```
for(F in -50:100) {
    C = (F - 32) *5/9
    cat("Pour F = ",F," on obtient C = ",C, '\n')
}
```

```
## Pour F =  -50  on obtient C =  -45.55556
## Pour F =  -49  on obtient C =  -45
## Pour F =  -48  on obtient C =  -44.44444
## Pour F =  -47  on obtient C =  -43.88889
## Pour F =  -46  on obtient C =  -43.33333
## Pour F =  -45  on obtient C =  -42.77778
## Pour F =  -44  on obtient C =  -42.22222
## Pour F =  -43  on obtient C =  -41.66667
## Pour F =  -42  on obtient C =  -41.11111
## Pour F =  -41  on obtient C =  -40.55556
## Pour F =  -40  on obtient C =  -40
## Pour F =  -39  on obtient C =  -39.44444
## Pour F =  -38  on obtient C =  -38.88889
## Pour F =  -37  on obtient C =  -38.33333
## Pour F =  -36  on obtient C =  -37.77778
## Pour F =  -35  on obtient C =  -37.22222
## Pour F =  -34  on obtient C =  -36.66667
## Pour F =  -33  on obtient C =  -36.11111
```

Boucles et indexation

Les boucles sont souvent utilisées pour accéder à des positions dans un objet de façon récursive. Le compteur de la boucle peut alors être utilisé directement comme indice pour accéder à de l'information dans l'objet. Par exemple :

```
vect5 ← runif(5, 0,1)
for(etape in 1:5) {
    print(vect5[etape])
}
```

```
## [1] 0.1753
## [1] 0.03872077
## [1] 0.06892257
## [1] 0.7993519
## [1] 0.6382885
```

Boucles et indexation

De même, on peut réaliser des opérations mathématiques sur cette variable

```
vect500 ← runif(500, 0,1)
for(etape in 1:5) {
    print(vect500[etape * 5])
}
```

```
## [1] 0.9177307
## [1] 0.1851929
## [1] 0.5487566
## [1] 0.5153519
## [1] 0.2135038
```

Compteur

La séquence ne commence pas toujours par 1, et donc parfois on doit avoir un compteur indépendant pour l'indexation. Il est donc pratique de calculer à quelle position on se situe dans la boucle.

```
n = 1
for(etape in -2:5) {
    cat("etape = ",etape, " n = ",n, '\n')
    n ← n + 1
}
```

```
## etape =  -2  n =  1
## etape =  -1  n =  2
## etape =  0  n =  3
## etape =  1  n =  4
```

Dans cet exemple, on ne pourrait pas indexer à la position de départ (-2). On crée donc un objet `n` qui a une valeur de 1 au départ et qui augmente de 1 unité à chaque itération. On peut maintenant utiliser ce compteur pour indexer un objet (vecteur, matrice, liste etc...).

Exercice

Enregistrez les résultats de votre conversion dans une matrice avec pour colonne 1 la valeur en Fahrenheit et la colonne 2 la valeur en celcius.

Solution

```
resultat = matrix(nr = 151, nc = 2)
n = 1
for(F in -50:100) {
    resultat[n,1] = F
    resultat[n,2] = (F-32)*5/9
    n = n + 1
}
head(resultat)
```

```
##      [,1]      [,2]
## [1,]  -50 -45.55556
## [2,]  -49 -45.00000
## [3,]  -48 -44.44444
## [4,]  -47 -43.88889
## [5,]  -46 -43.33333
## [6,]  -45 -42.77778
```

On complexifie le problème ....

Il est possible de nicher une boucle dans une boucle. On réalise notamment cette opération pour faire des calculs sur des matrices, des listes...

```
ma_matrice ← matrix(nr = 3, nc = 5)
n ← 1
for(i in 1:3) {
    for(j in 1:5) {
        ma_matrice[i,j] ← n
        n ← n + 1
    }
}
ma_matrice
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15
```

Notons qu'on remplit ici la matrice par rangée. On pourrait inverser les boucles pour y aller par colonne.

La boucle while

La boucle de type *while*, pour 'pendant que', répète une série d'intructions tant qu'une condition n'a pas été atteinte. Par exemple, on peut tirer deux pièces de monnaie jusqu'à ce que l'on obtienne la combinaison pile-pile.

```
piece ← c("pile","face")
combinaison = "face-face"
while(combinaison ≠ "pile-pile") {
    essai1 ← sample(piece, 1)
    essai2 ← sample(piece, 1)
    combinaison ← paste(essai1, "-" , essai2, sep = "")
    print(combinaison)
}
```

```
## [1] "face-face"
## [1] "pile-face"
## [1] "face-face"
## [1] "pile-face"
## [1] "face-pile"
## [1] "face-pile"
## [1] "pile-pile"
```

La boucle va s'arrêter quand le résultat de la condition sera **FALSE**

Exercice

Utilisez la boucle `while` pour calculez combien de fois vous avez à tirer un dé pour obtenir un 6.

- Ça prend un dé de 6 faces
- Ça prend un compteur
- et une série de tirages

Solution

```
PROGRAM BOUCLE WHILE
DEFINE de = 1:6
DEFINE n = 1
DEFINE tirage = 1

WHILE tirage ≠ 6
    tirage = tirage du dé
    n = n + 1
END WHILE
```

Solution

```
de ← c(1:6)
n ← 1
tirage ← 1 # Il faut initier le tirage
while(tirage ≠ 6) {
    tirage ← sample(de,1) # tirage du dé
    n ← n + 1 # mise à jour du compter pour tester le nombre de tirages
}
cat(n, "tirages pour obtenir un 6 ", '\n')
```

```
## 2 tirages pour obtenir un 6
```

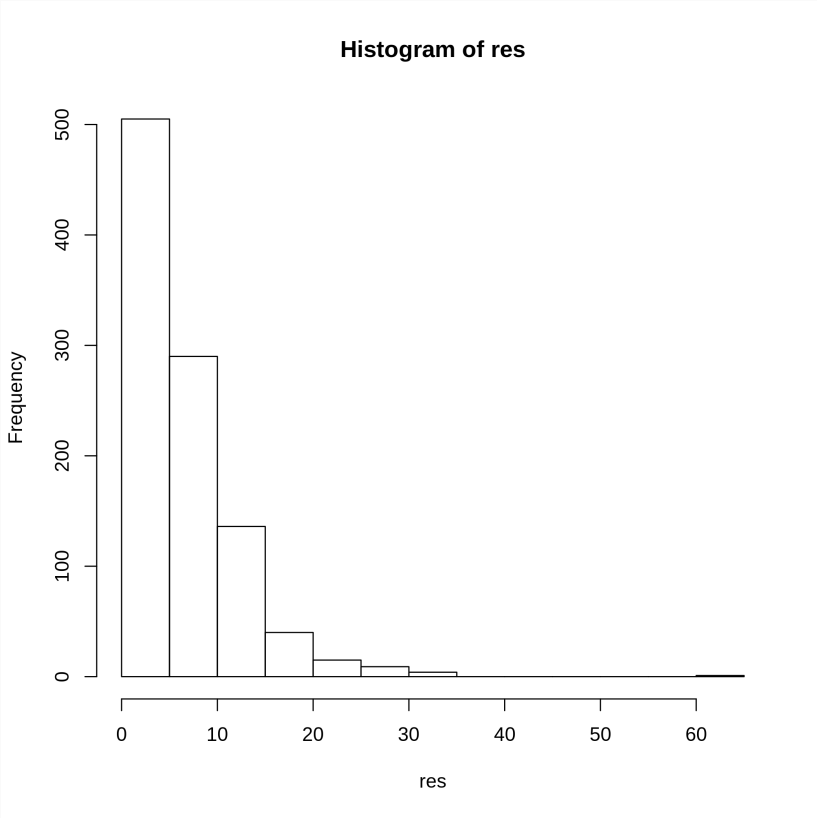
Solution : on répète plusieurs fois

```
de ← c(1:6)
n_test ← 1000
res ← numeric(n_test)
for(i in 1:n_test) {
    n ← 1
    tirage ← 1 # Il faut initier le tirage
    while(tirage ≠ 6) {
        tirage ← sample(de,1) # tirage du dé
        n ← n + 1 # mise à jour du compter pour tester le nombre de tirages
    }
    res[i] ← n # Enregistrement du nombre de tirage
}
mean(res)
```

```
## [1] 7.03
```

Solution

```
hist(res)
```



Exercice intermédiaire

Croissance logistique

La croissance d'une population sujette à la densité-dépendance, en temps discret, se calcul bien au moyen d'une boucle. La densité au temps $t+1$ (l'année suivante) se calcule ainsi :

$$N_{t+1} = N_t + r \times N_t \times (1 - N_t/K)$$

Henri Meunier a importé environ 220 cerfs sur l'île d'Anticosti au début du 20ème siècle et on compte aujourd'hui environ 200 000 bêtes, ce qui correspond à la capacité de support du milieu (K). Si on fixe le taux de croissance (r) à 0.3, combien de temps fut nécessaire à la population pour atteindre 50% de la capacité de support ?

> **Astuce** : On cherche à compter le nombre d'années nécessaires pour que N atteigne la valeur de $K/2$.

Solution 1

```
PROGRAM CROISSANCE CERFS
DEFINE N = 220
DEFINE K = 200000
DEFINE r = 0.3
DEFINE step = 1

WHILE N < K/2
    N = N + r * N * (1 - N/K)
    step = step + 1
END WHILE
```

Solution 1

```
r ← 0.3
N ← 220
K ← 200000
step ← 1
while(N < K/2) {
    N ← N + r * N * (1 - N/K)
    step ← step + 1
}
step
```

```
## [1] 27
```

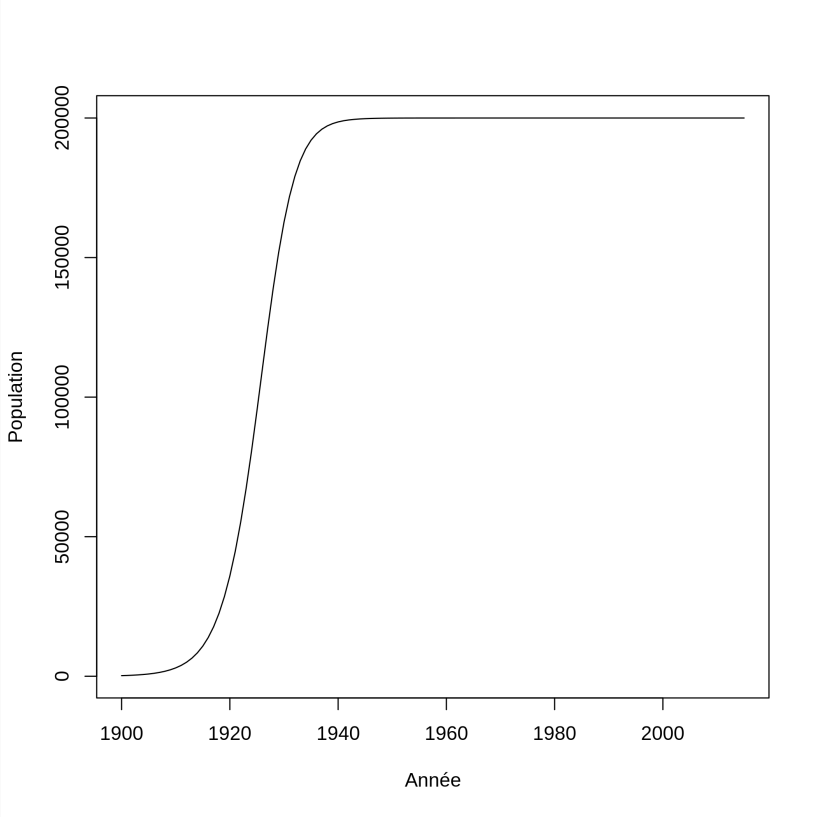
Solution 2

```
r ← 0.3
K ← 200000
n_steps ← 115
N ← numeric(n_steps+1)
N[1] ← 220

for(i in 2:(n_steps+1)) {
    N[i] ← N[i-1] + r * N[i-1] * (1 - N[i-1]/K)
}
```

Solution 2

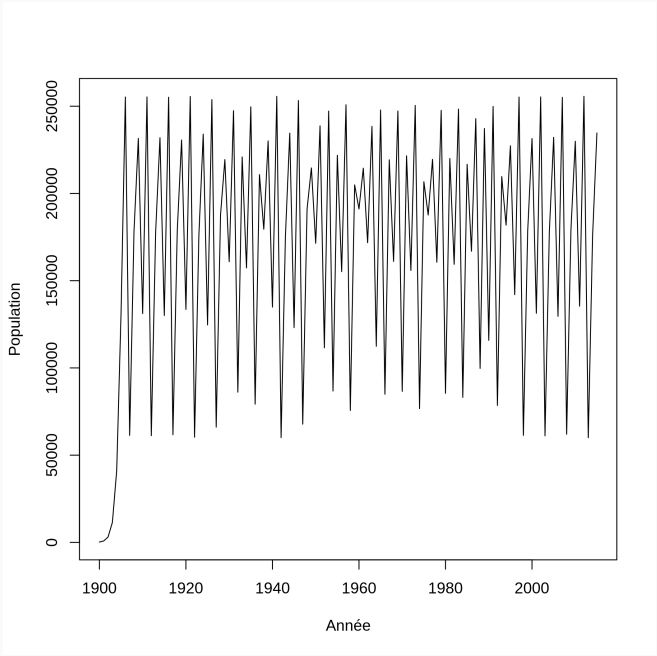
```
plot(c(1900:2015), N, type = "l", xlab = "Année", ylab = "Population")
```



Qu'est-ce qui arrive avec $r = 2.75$?

Qu'est-ce qui arrive avec $r = 2.75$?

```
n_steps ← 115
N ← numeric(n_steps+1)
N[1] ← 220
for(i in 2:(n_steps+1)) {
    N[i] ← N[i-1] + 2.75 * N[i-1] * (1 - N[i-1]/200000)
}
plot(c(1900:2015), N, type = "l", xlab = "Année", ylab = "Population")
```



Expressions conditionnelles

Principe

Très souvent en programmation on a à prendre des décisions du type **si la condition X est remplie, alors faire Y, sinon faire Z**. Nous avons déjà vu certains exemples depuis le début du cours.

La structure de base d'une expression conditionnelle est la suivante:

```
if (condition) {
    # instruction 1
    }
else {
    # instruction 2
}
```

Arbre décisionnel

Un exemple,

```
todo ← function(jour) {
    if(jour == "mardi") {
        print("Je dois aller au cours BIO109")
    }
    else {
        if(jour == "lundi") {
        print("Ai-je fais mon travail pour le cours BIO109 ?")
        }
        else {
            if(jour == "samedi") {
                print("Je peux encore dormir un peu")
            }
            else {
                print("Bof, j'ai encore du temps !")
            }
        }
    }
}
```

Astuce

L'indentation (l'utilisation d'espaces dans le code) est fort utile pour s'y retrouver lorsque les conditions deviennent nombreuses. Reprenons l'exemple précédent.

Arbre décisionnel

```
todo ← function(jour) {
if(jour == "mardi") {
print("Je dois aller au cours BIO109")
}
else {
if(jour == "lundi") {
print("Ai-je fais mon travail pour le cours BIO109 ?")
}
else {
if(jour == "samedi") {
print("Je peux encore dormir un peu")
}
else {
print("Bof, j'ai encore du temps !")
}
}
}
}
```

Opérateurs logiques

Les classiques :

Opérateur	Signification	Exemple
==	égal à	X == Y
!=	différent de	X != Y
>	supérieur à	X > Y
<	inférieur à	X < Y
>=	supérieur ou égal à	X >= Y
<=	inférieur ou égal à	X <= Y

Exercice

1. Tirez au hasard 10 chiffres entre 0 et 1 avec `runif()`.
2. Pour chacun de ces chiffres, déterminez s'il est plus petit ou plus grand que la valeur seuil de 0.3.
3. Inscrivez votre résultat dans un autre objet.

Solution

```
chiffres ← runif(n = 10)
res ← numeric(10)
for(i in 1:10) {
    if(chiffres[i] < 0.3) {
        res[i] ← "plus petit"
    }
    else {
        res[i] ← "plus grand"
    }
}
cbind(chiffres,res)
```

```
##        X                     res
##  [1,] "0.968936904333532"   "plus grand"
##  [2,] "0.0827734367921948"  "plus petit"
##  [3,] "0.140440867748111"   "plus petit"
##  [4,] "0.924323483137414"   "plus grand"
##  [5,] "0.00891387672163546" "plus petit"
##  [6,] "0.678747127763927"   "plus grand"
##  [7,] "0.527182457968593"   "plus grand"
##  [8,] "0.624553735135123"   "plus grand"
##  [9,] "0.885842786170542"   "plus grand"
## [10,] "0.118931479519233"   "plus petit"
```

Opérateurs logiques

On peut aussi combiner plusieurs conditions :

- La condition A et la condition B : `A & B`
- La condition A ou la condition B : `A | B`

Exercice

Reprenez la même séquence de chiffres, mais cette fois-ci déterminez si le chiffre est situé entre les valeurs de 0.2 et 0.6.

Solution

```
res2 ← numeric(10)
for(i in 1:10) {
    if(chiffres[i] > 0.2 & chiffres[i] < 0.6) {
        res2[i] ← "oui"
    }
    else {
        res2[i] ← "non"
    }
}
cbind(chiffres, res2)
```

```
##       chiffres             res2
##  [1,] "0.703685053624213"  "non"
##  [2,] "0.684623213019222"  "non"
##  [3,] "0.987691749818623"  "non"
##  [4,] "0.262822555378079"  "oui"
##  [5,] "0.0844127237796783" "non"
##  [6,] "0.773586681578308"  "non"
##  [7,] "0.0383469699881971" "non"
##  [8,] "0.281646347371861"  "oui"
##  [9,] "0.695975542766973"  "non"
## [10,] "0.930912127019837"  "non"
```

Conditions sur des vecteurs

R a la particularité d'être optimisé pour les objets sous forme de matrice ou de vecteur. Ainsi, vous pouvez évaluer des expressions conditionnelles sur un vecteur sans avoir à passer par une boucle. On peut reprendre l'exemple précédent, d'une façon beaucoup plus rapide :

```
res2 ← numeric(10)
res2[X > 0.2 & X < 0.6] ← "oui"
res2[X < 0.2 | X > 0.6] ← "non"
cbind(X,res2)
```

```
##        X                    res2
##  [1,] "0.715908034238964"  "non"
##  [2,] "0.26107022468932"   "oui"
##  [3,] "0.964512130245566"  "non"
##  [4,] "0.208013951545581"  "oui"
##  [5,] "0.681774899596348"  "non"
##  [6,] "0.712057070806623"  "non"
##  [7,] "0.219971334096044"  "oui"
##  [8,] "0.0483853481709957" "non"
##  [9,] "0.0879302744287997" "non"
## [10,] "0.0375943079125136" "non"
```

Cette syntaxe est moins lisible et intuitive, vous n'avez pas à l'utiliser pour le cours. Il faut néanmoins pouvoir la reconnaitre et surtout savoir qu'elle est de 100 à 1000 plus rapide que la boucle `for`.

Exercice de la semaine

Une situation qui peut arriver tous les jours

1. On jette en face de vous 5 lettres d'un jeu de scrabble
2. Un maniaque vous demande d'écrire un programme qui ordonne les 5 lettres

Prenez le temps de distinguer les étapes que vous réalisez lorsque vous triez les lettres. Vous devez les écrire sous forme de pseudo-code.

Pseudo-code

Ordonner un vecteur `X` composé de 5 lettres de Scrabble

```
DEFINE is.sort = FALSE
READ X

WHILE is.sort = FALSE
  is.sort = TRUE
  FOR pos IN 1:4
    IF X[pos] > X[pos+1]
      INVERSE
      is.sort = FALSE
    END IF
  END FOR
END WHILE
```