Correction TP Programmation sous R

Structures de contrôle

Les conditions

1. Entier pair ou impair

```
n = sample(1:100, size = 1)# Initialisation aléatoire
if (n%%2){ # On utilise le modulo pour déterminer la parité n%%2==1 ssi n est impaire
  cat(n,'est impaire')
} else{
  cat(n,'est paire')
}
```

```
96 est paire
```

La même chose avec `ifelse`:

```
n = sample(1:100, size = 1)# Initialisation aléatoire
ifelse(n%%2, paste(n,'est impaire'),paste(n,'est paire'))
```

```
[1] "28 est paire"
```

La même chose avec `switch`:

```
n = sample(1:100, size = 1)# Initialisation aléatoire
switch(n %% 2+1, # prend comme valeurs 1 et 2  et donne l'instruction à exécuter
       print(paste("n est pair, il est égal à ", n)), # instruction 1
       print(paste("n est impair, il est égal à ", n))) # instruction 2
```

```
[1] "n est impair, il est égal à  79"
```

2.

```
n=100
if(sqrt(n)==floor(sqrt(n))){
  cat(n,'est un carré parfait')
}else{
  cat(n,'n\'est pas un carré parfait')
}
```

```
100 est un carré parfait
```

3. On peut mettre toutes les conditions dans le même `if`.

```
n=0
if(sqrt(n)==floor(sqrt(n)) && n%%2 && n%%3==0){
  cat(n, 'est le caré d\'un entier impaire, multiple de 3')
}else{
  cat(n, 'n\'est pas le caré d\'un entier impaire, multiple de 3')
}
```

```
0 n'est pas le caré d'un entier impaire, multiple de 3
```

On peut également verifier les conditions une par une

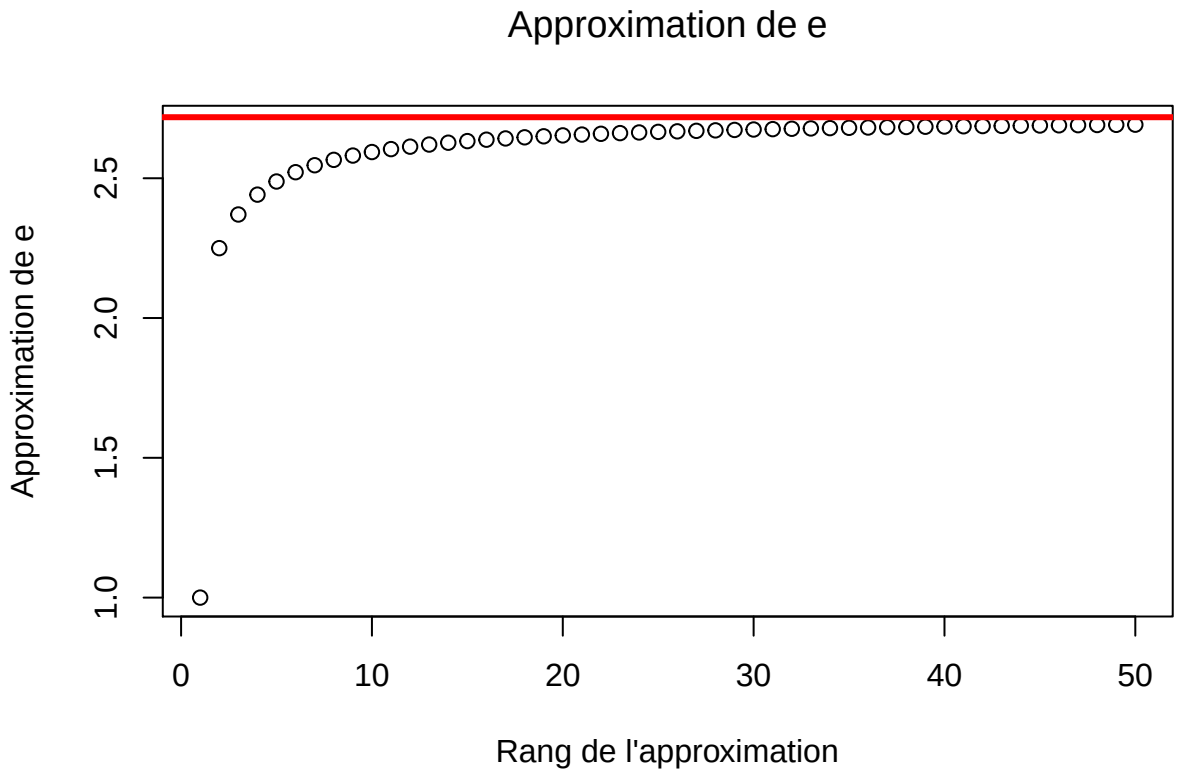
```r
n=36
if(sqrt(n)!=floor(sqrt(n))){
   cat(n,'n\'est pas un carré parfait')
}else{
  if(n%%2==0){
    cat(n,'est le carré d\'un nombre paire')
  }else{
    if(n%%3){
      cat(n,'est le carré d\'un nombre impaire mais pas multiple de 3')
    }
    else{
      cat(n,'est le caré d\'un entier impaire, multiple de 3')
    }
  }
}
```

```
36 est le carré d'un nombre paire
```

Boucles for

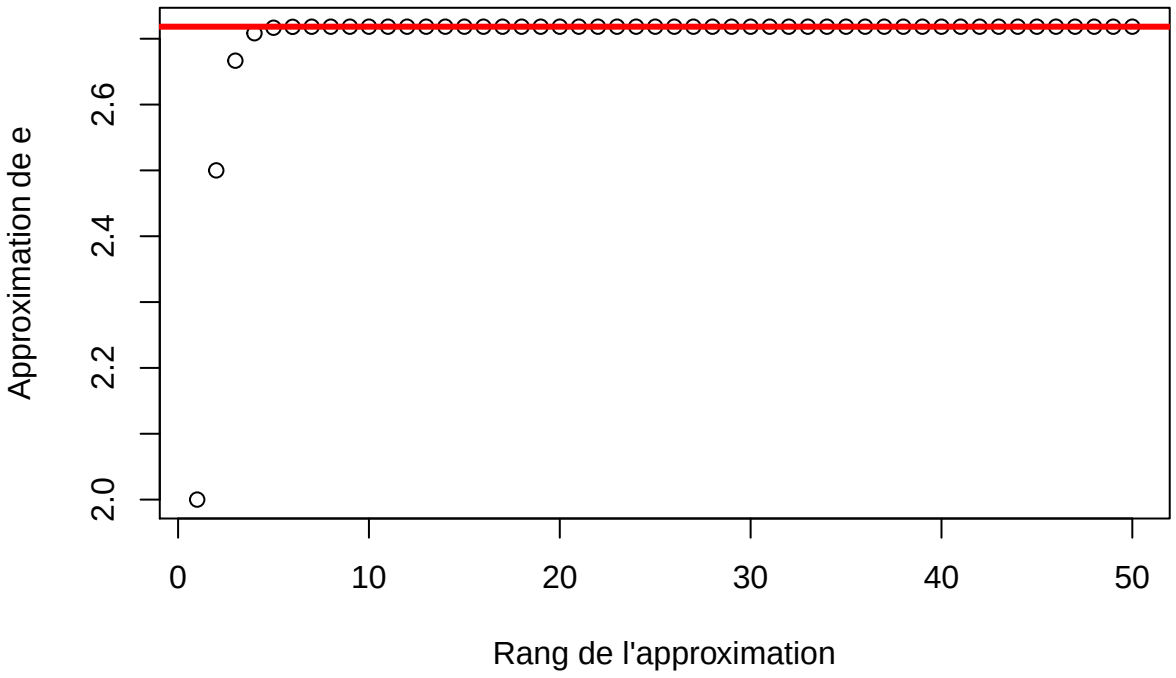
1. **Approximation de e:**

```r
n=50 # Ordre de l'approximation
ae <- 1 # Valeur de l'approximation pour n=1
for (k in 2:n) {
  ae=c(ae,(1+1/k)^k) # Concaténation du vecteur avec le nouveau terme
}
plot(1:n,ae,ylab=expression(Approximation~de~e),
     xlab = "Rang de l'approximation", main=expression(Approximation~de~e))
abline(exp(1),0,col='red',lw=3)
```



2.

```r
n=50 # Ordre  maximal de l'approximation
aproxe <- rep(2, 50) #pour n=0, l'aprox est égale à 1 et on va faire calculer l'
for(k in 2:n){
  aproxe[k] <- aproxe[k-1]+1/factorial(k)
}
plot(1:n,aproxe,ylab=expression(Approximation~de~e),
     xlab = "Rang de l'approximation", main=expression(Approximation~de~e))
abline(exp(1),0,col='red',lw=3)
```



2. **Calcul d'intérêts**.

```
NombreAnnees <- 10
CapitalInitiel <-Capital <- 1000 #Capital initiel pour le Livret A avec le taux de 2019

CapitalLivretA2008Initiel <- CapitalLivretA2008 <- 1000 #Capital init. Livret A avec le tauxx de 2008
TotalInteretLivretA <- 0
Taux=0.0075
TauxLivretA2008=0.04
for(i in 1:NombreAnnees){
  Capital <- Capital + Taux * Capital

  ## Meme chose pour le livret A de 2008
  CapitalLivretA2008 <- CapitalLivretA2008 + TauxLivretA2008* CapitalLivretA2008
}
TotalInteret <- Capital-CapitalInitiel
TotalInteretLivretA2008 <- CapitalLivretA2008-CapitalLivretA2008Initiel
print(paste("Avec un taux annuel de",Taux*100, "%, le montant total d'intérêts sur",
            NombreAnnees,"ans est:",TotalInteret))
```

```
[1] "Avec un taux annuel de 0.75 %, le montant total d'intérêts sur 10 ans est: 77.582545470739"
```

```
print(paste("Avec un taux annuel de",TauxLivretA2008*100, "%, le montant total d'intérêts sur",
            NombreAnnees,"ans est:",TotalInteretLivretA2008))
```

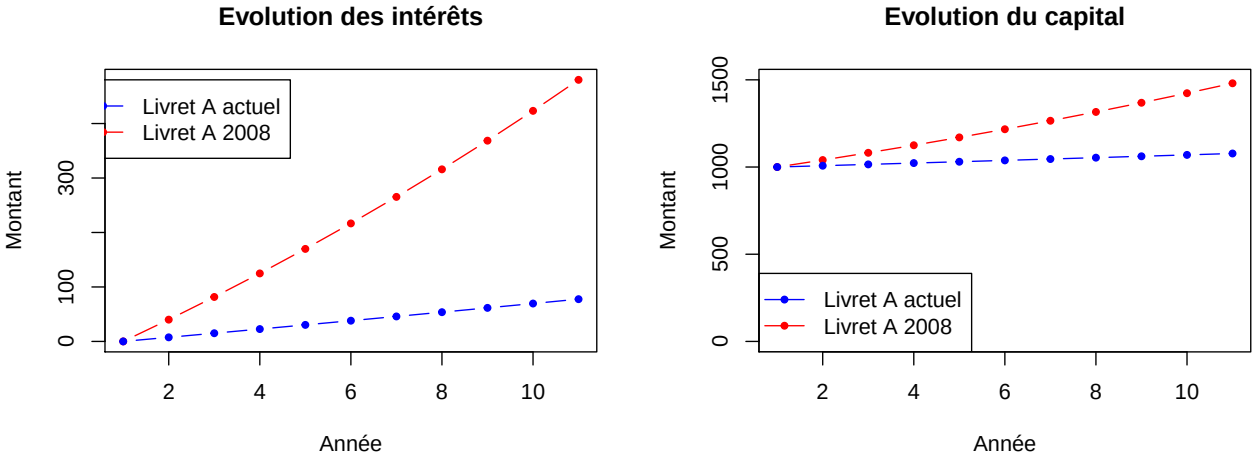
```
[1] "Avec un taux annuel de 4 %, le montant total d'intérêts sur 10 ans est: 480.244284918344"
```

Pour visualiser l'evolution des intérêts (ou du capital) on peux sauvegarder les valeurs intermedieres.

```
NombreAnnees <- 10
# On initialise en créant le vecteur,
# il va donner le capital au début de chaque année pour le Livret A actuel:
```

```r
Capital <- rep(1000, NombreAnnees+1)
TotalInteret <- rep(0, NombreAnnees+1)
#Même chose pour le livret A avec le taux de 2008
CapitalLivretA <- rep(1000, NombreAnnees)
TotalInteretLivretA <- rep(0, NombreAnnees)
Taux=0.0075
TauxLivretA2008=0.04
for(i in 2:(NombreAnnees+1)){
  InteretAnnuel <-Taux * Capital[i - 1]
  TotalInteret[i] <- TotalInteret[i - 1] + InteretAnnuel
  Capital[i] <- Capital[i - 1] + InteretAnnuel

  ## Meme chose pour le livret A
  InteretAnnuelLivretA <- TauxLivretA2008 * CapitalLivretA[i - 1]
  TotalInteretLivretA[i] <- TotalInteretLivretA[i - 1] + InteretAnnuelLivretA
  CapitalLivretA[i] <- CapitalLivretA[i - 1] + InteretAnnuelLivretA
}
```

On peut représenter graphiquement l'évolution des intérêts ou du capital



Remarque: L'évolution du capital C_n à l'année n, peut être décrite par une série géométrique de ratio $q = 1 + \text{Taux}$ et terme initial $C_0 = 1000$. Donc, les intérêts I_n à l'année n s'écrivent $1000 \times ((1 + \text{Taux})^n - 1)$. En utilisant ça et les propriétés de vectorisation de R, on n'a pas besoin de faire de boucle. En effet, la boucle plus haut donne le même résultat que:

```r
(InteretAnnuelLivretA <- 1000*((1+TauxLivretA2008)^(0:10) -1))
```

```
 [1]   0.0000  40.0000  81.6000 124.8640 169.8586 216.6529 265.3190
 [8] 315.9318 368.5691 423.3118 480.2443
```

4. Jeu de données `UScereal`

Calcul sans boucles:

```r
data(UScereal,package="MASS")
moy=aggregate(UScereal$calories, list(UScereal$mfr),mean)
# ou moy=by(UScereal$calories,UScereal$mfr,mean), Voir TP2
et=aggregate(UScereal$calories, list(UScereal$mfr),sd)
colnames(moy)=c('producteur', 'moyenne')
colnames(et)=c('producteur', 'ecart-type')
d=merge(et,moy,by= "producteur")
d
```

```
  producteur ecart-type  moyenne
1          G   44.99660 137.7879
2          K   45.77379 149.6710
3          N   44.91372 160.2593
4          P  122.89988 194.7578
5          Q   55.90069 135.8507
6          R   20.82300 124.8521
```

Calcul avec une boucle:

```
moy=c() # Création d'un vecteur vide
ect=c()
for (i in levels(UScereal$mfr)){
moy <- c(moy, mean(subset(UScereal, mfr == i)$calories))#Voir l'aide de subset ou le TP2
  ect  <- c(ect,#
               sd(subset(UScereal, mfr == i)$calories))
}
(ResCal <- data.frame(Moyenne = moy, EcartType = ect,# Spécification du nom des colonnes
                              row.names = levels(UScereal$mfr)))#Spécification des noms de ligne
```

```
   Moyenne EcartType
G 137.7879  44.99660
K 149.6710  45.77379
N 160.2593  44.91372
P 194.7578 122.89988
Q 135.8507  55.90069
R 124.8521  20.82300
```

Calcul avec deux boucles:

```
moy=c() # Création d'un vecteur vide
ect=c()
ectr=c()
for (p in levels(UScereal$mfr)) {m=0
  for (x in UScereal$calories[UScereal$mfr==p]) {m=m+x
  }
  n=length(UScereal$calories[UScereal$mfr==p])
  m=m/n
  moy=c(moy,m)
  s=0
  for (x in UScereal$calories[UScereal$mfr==p]) {s=s+(x-m)^2
  }
  s=sqrt(s/n)
  ect=c(ect,s)
  ectr=c(ectr,s*sqrt(n/(n-1))) #estimateur utilisé par R
  }

res=data.frame(Moyenne=moy, Ecart_Type= ect, Ecart_Type_R=ectr, row.names = levels(UScereal$mfr))
res
```

```
   Moyenne Ecart_Type Ecart_Type_R
G 137.7879   43.96205     44.99660
K 149.6710   44.67065     45.77379
N 160.2593   36.67190     44.91372
P 194.7578  115.87112    122.89988
Q 135.8507   49.99910     55.90069
R 124.8521   18.62465     20.82300
```

Boucle while

1. La puissance de 2 dans la décomposition en facteurs premiers d'un entier n est le p maximal tel que 2^p est un diviseur de n.

```
n=64
p=0
while(n%%2==0){
  p <- p+1
  n <- n/2
}
print(paste("La puissance de 2 dans la décomposition  de n est",p))
```

```
[1] "La puissance de 2 dans la décomposition  de n est 6"
```

2. Trouver un nombre entier `myst`:

```
myst=sample(1:100,1)
count=1
while(sample(1:100,1)!=myst) {
  count=count+1
}
print(paste("Nombre de tirages necessaires:", count))
```

```
[1] "Nombre de tirages necessaires: 8"
```

Boucle repeat

Q.S. Pour la question 2, on peut également utiliser `repeat` :

```
myst=sample(1:100,1)
count=0
repeat{
  count=count+1
  if (sample(1:100,1)==myst) break
}
print(paste("Nombre de tirages necessaires:", count))
```

```
[1] "Nombre de tirages necessaires: 201"
```

2. **Suite de Syracuse**.

```
#n = sample(1:200, size = 1) #Initialisation aléatoire
n=1
suite=c(n) # initialisation de la suite
repeat{
  if(n %% 2 ==0) {
     n=n/2
  } else{
    n=n*3+1
  }
  suite=c(suite,n)
  if (n==1) break
}
print(paste("En partant de",suite[1], ", il a fallu", length(suite)-1,
            "opérations avant de s'arrêter."))
```

```
[1] "En partant de 1 , il a fallu 3 opérations avant de s'arrêter."
```

```
#print(suite)
```

```
#Remarque, ici si n=1, l'algorithme s'arrete sans engendrer la suite 1,4,2,1
n = sample(1:200, size = 1) #Initialisation aléatoire
suite=c(n) # initialisation de la suite
while (n!=1) {
  if(n %% 2 ==0) {
     n=n/2
  } else{
    n=n*3+1
  }
  suite=c(suite,n)

}
print(paste("En partant de",suite[1], ", il a fallu", length(suite)-1,
            "opérations avant de s'arrêter."))
```

```
[1] "En partant de 130 , il a fallu 28 opérations avant de s'arrêter."
```

Création de fonctions

1. Fonction *calcul d'intérêts*

```
CalculInteret <- function(C, n, tx){
  InteretFinal <- C*((1 + tx)^n -1)# La boucle aurait fonctionné aussi
  return(InteretFinal)
}
#Calcul d'intérêts sur 10 ans pour un livret A au plafond avec le taux de 2019
CalculInteret(C= 22950,n= 10, tx=0.0075)
```

```
[1] 1780.519
```

```
#Calcul d'intérêts sur 10 ans sur la même somme, avec un taux de 4%
CalculInteret(C= 22950,n= 10, tx=0.04)
```

```
[1] 11021.61
```

2. On créé une fonction **NombreTirages**

```
NombreTirages <- function(n){
  #On reprend le même code que précédemment, où myst est remplacé par n
  count=1
  while(sample(1:100,1)!=n) {
   count=count+1
  }

  return(count)
}
```

```
m = 1000;
NombreNecessaire <- rep(NA, m)
myst=sample(1:100,1) #on choisit un nombre de maniere aleatoire
for (i in 1:m) {
  NombreNecessaire[i]=NombreTirages(myst)
```

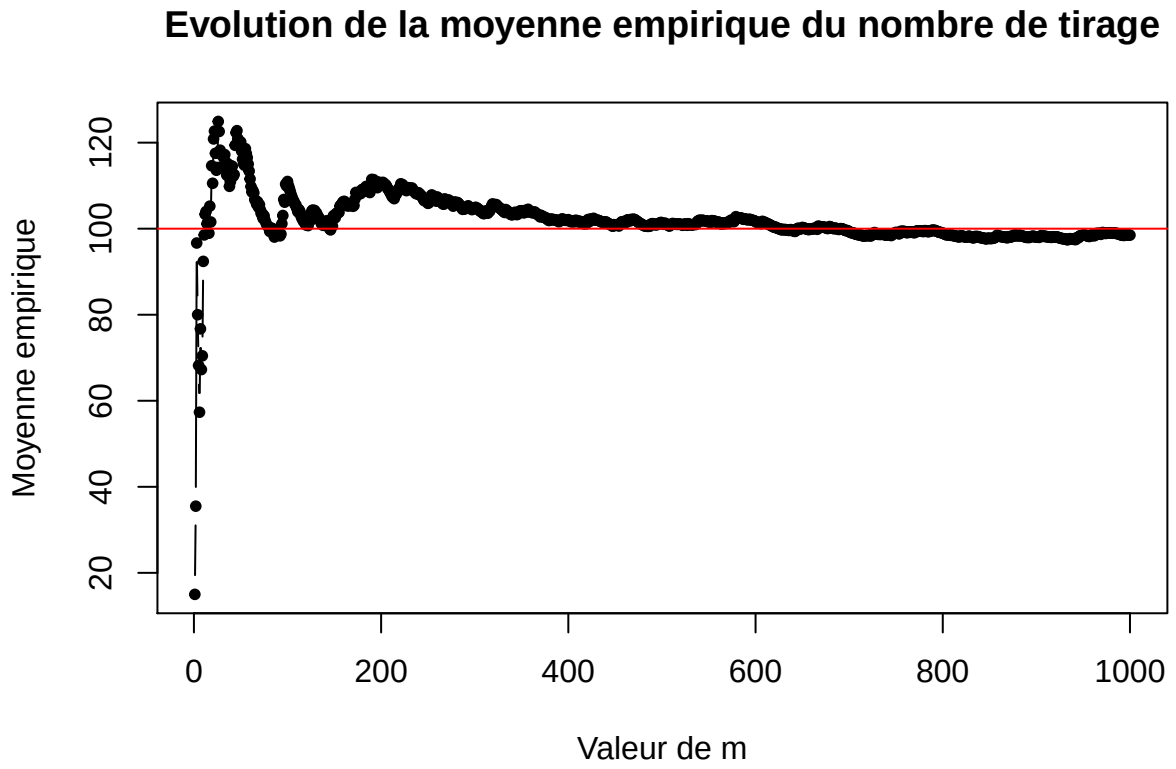
```
}
  print(mean(NombreNecessaire))
```

```
[1] 98.499
```

La valeur moyenne converge vers 100. On a ici une illusatration empirique de la loi des grands nombres. La loi du nombre de tirages est une loi géométrique de paramètre $p = 0.01$, son espérance est donc 100 $(1/p)$. On effectue m réplications indépendantes de cette loi, la moyenne empirique de l'échantillon tend vers 100 quand m grandit.

L'évolution de la moyenne en fonction du nombre de répétions m:

```
MoyEmpiriques <- cumsum(NombreNecessaire)/(1:m)# Vecteurs avec les moyennes aggrégées

plot(1:m, MoyEmpiriques, main = "Evolution de la moyenne empirique du nombre de tirage",
     xlab = "Valeur de m", ylab = "Moyenne empirique", type = "b", pch = 20)
abline(h = 100, col = "red")# Ajout de l'espérance théorique
```



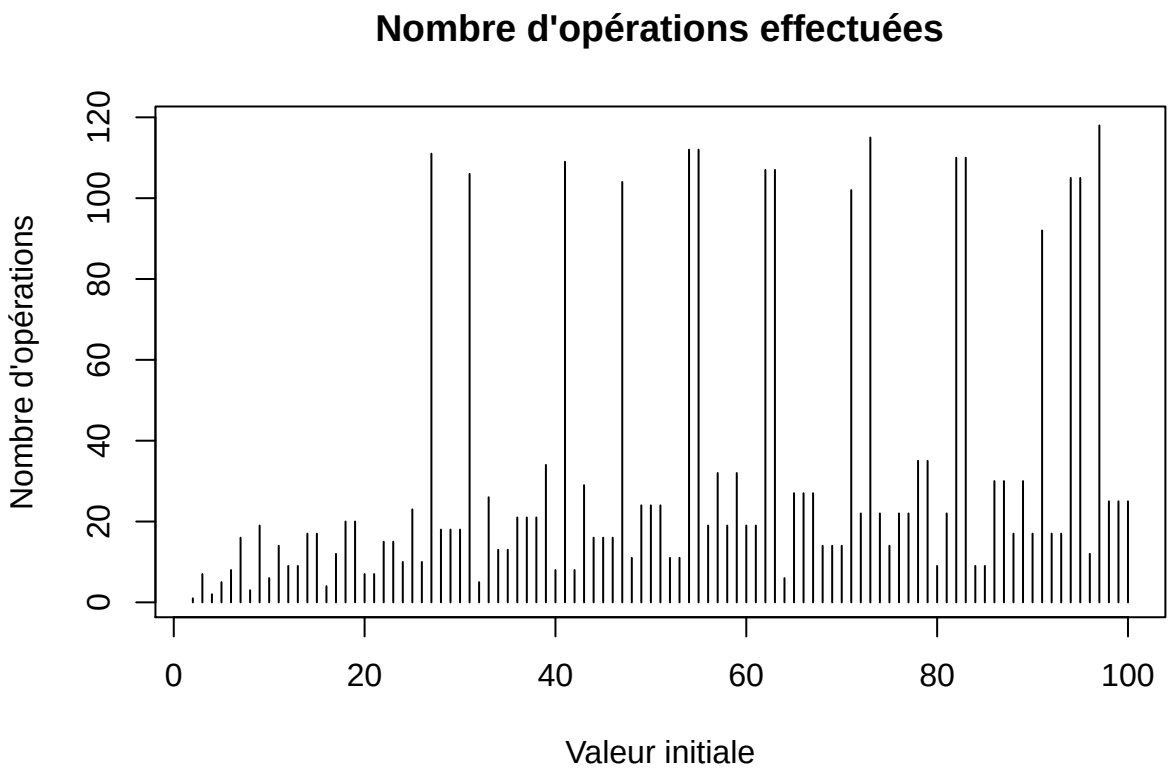
3. Fonction `Syracuse`:

```
Syracuse <- function(n){
suite=c(n) # initialisation de la suite
while (n!=1) {
  if(n %% 2 ==0) {
     n=n/2
  } else{
    n=n*3+1
  }
  suite=c(suite,n)

}
```

```
return(list(Suite=suite, Longuer=length(suite)-1))
}
```

Répresenter les longuers des suites de Syracuse pour n entre 2 et 100:

```
ValeursInitiales <- 2:100
LonguerSuite <- ValeursInitiales # La valeur de l'initialisation n'est pas importante ici
for (i in ValeursInitiales) {
  LonguerSuite[i-1]=Syracuse(i)$Longuer

}
plot(ValeursInitiales,LonguerSuite,type = 'h', main = "Nombre d'opérations effectuées", xlab = "Valeur
    ylab = "Nombre d'opérations")
```



Calculer le nombre maximum atteint:

```
ValeursInitiales <- 2:100
MaxSuite <- ValeursInitiales # La valeur de l'initialisation n'est pas importante ici
for (i in ValeursInitiales) {
  MaxSuite[i-1]=max(Syracuse(i)$Suite)

}
ValMax <- max(MaxSuite)
ValInitialeMax <- ValeursInitiales[which(MaxSuite==ValMax)] # pour avoir toutes les argmax
cat(" Le nombre maximal atteint dans les suites est", ValMax,".\n", #pasage a la ligne
            "Ce nombre est dans les suites ayant comme valeurs initiales:\n", ValInitialeMax)
```

```
 Le nombre maximal atteint dans les suites est 9232 .
 Ce nombre est dans les suites ayant comme valeurs initiales:
 27 31 41 47 54 55 62 63 71 73 82 83 91 94 95 97
```

Comment gérer des cas particuliers

1. Calculer la fonction factorielle:

```
fact=function(n){
  p=1;
  while(1<n)
    {p=p*n
    n=n-1
    }
  return(p)
}
fact(0); fact(5);fact(3)
```

```
[1] 1
```

```
[1] 120
```

```
[1] 6
```

```
#vérifier qu'on obtient les mêmes valeurs
all.equal(c(fact(0), fact(8),fact(153)),c(factorial(0), factorial(8),factorial(153)))
```

```
[1] TRUE
```

```
fact=function(n){
  if(n<0 ){ # On s'arrête si le nombre estu négatif
    stop(" Le nombre n'est pas positif")
  } else{
    if(n!= floor(n)){
      n <- floor(n)
      warning('La factorielle de la partie entiere de n est retourné')
    }else{
  p=1;
  while(1<n)
    {p=p*n
    n=n-1
    }
  return(p)
  }
  }
}
 fact(1.5);
```

```
Warning in fact(1.5): La factorielle de la partie entiere de n est retourné
```

```
 fact(-3)
```

```
Error in fact(-3):  Le nombre n'est pas positif
```

```
fact(4); fact(0);  # Remarque: 0!=1 donc on n'a pas besoin d'une structure de controle en plus
```

```
[1] 24
```

```
[1] 1
```

2. Option **break**

```
NombreTirages <- function(n){
  count=1
  while(sample(1:100,1)!=n) {
```

```
   if(count >= 100){
     count=NULL # Mise du résultat à NULL
     cat("L'algorithme n'a pas convergé. \n ",append = FALSE)
     break # Sortie de la boucle
   }
    count=count+1
  }

  return(count)
}
x=NombreTirages(107)
```

```
L'algorithme n'a pas convergé.
```

```
x
```

```
NULL
```

3. Option **next**.

Le bloc suivant n'est pas necessaire, c'est juste pour pouvoir utliser la fonction sans le message d'erreur. On pourrait egalement reprendre la fonction crée à la question 2. Section 2.1.

```
NombreTirages <- function(n, sw = 0){
   # sw est un parametre pour controler l'aparition d'un message d'errreur quand l'algorithme
   # ne converge pas. Ici sa valeur par défault est 0
  count=1
  while(sample(1:100,1)!=n) {
   if(count >= 100){
     count=NULL # Mise du résultat à NULL
     if(sw) { # si sw == 0 le message d'erreur  n'apparait pas
     cat("L'algorithme n'a pas convergé. \n ")
     }
       break # Sortie de la boucle
   }
    count=count+1
  }

  return(count)
}
NombreTirages(107,0)
```

```
NULL
```

La commande **next** améme le courseur d'exécution du programme au départ de la boucle. L'itération suivante est exécuté (si elle existe).

```
N=1000;
Tirages=c()
myst=sample(1:100,1)
for (i in 1:N) {
    t=NombreTirages(myst,0)
  if (!is.null(t)){
    Tirages=c(Tirages,t)
  } else{
    next
```

```
  }

}
cat(paste("Le nombre moyen de tirages est: ",mean(Tirages),".\n",sep = ""))
```

```
Le nombre moyen de tirages est: 42.9954058192956.
```

```
cat(paste("L'algorithme n'a pas convergé", N-length(Tirages), "fois."))
```

```
L'algorithme n'a pas convergé 347 fois.
```

Opératuers

```
"%+%" <- function(x,y){
  return(paste(x,y))
}
'hello'%+%'world'
```

```
[1] "hello world"
```

Famille des fonctions apply

1. Utiliser les qualités (et connaitre les faiblesses) de R permet bien souvent de gagner du temps. Il faut au mieux utiliser les propriétés vectorielles des fonctions, et essayer d'éviter d'utiliser des boucles lors de schémas non itératifs.

```
# Boucle for sans taille prédéfinie
n <- 10000
T1 <- Sys.time()
Resultat <- c()
for(i in 1:n){
  Resultat <- c(Resultat, cos(i))
}
T2 <- Sys.time()
print(paste("La boucle for sans taille initiale a mis ",
            round(difftime(T2, T1, units = "secs"), 3), "secondes"))
```

```
[1] "La boucle for sans taille initiale a mis  0.121 secondes"
```

```
# Boucle for avec taille prédéfinie
T1 <- Sys.time()
Resultat <- rep(NA, n)
for(i in 1:n){
  Resultat[i] <- cos(i)
}
T2 <- Sys.time()
print(paste("La boucle for avec taille initiale a mis ",
            round(difftime(T2, T1, units = "secs"), 3), "secondes"))
```

```
[1] "La boucle for avec taille initiale a mis  0.005 secondes"
```

```
# Application Vectorielle
T1 <- Sys.time()
```

```
Resultat <- cos(1:n)
T2 <- Sys.time()
print(paste("L'application vectorielle a mis ",
            round(difftime(T2, T1, units = "secs"), 3), "secondes"))
```

```
[1] "L'application vectorielle a mis  0.002 secondes"
```

```
# Avec apply
T1 <- Sys.time()
Resultat <- sapply(1:n, cos)
T2 <- Sys.time()
print(paste("La sapply a mis ", round(difftime(T2, T1, units = "secs"), 3), "secondes"))
```

```
[1] "La sapply a mis  0.005 secondes"
```

```
## Pour verifier le temps avec system.time():
Resultat <- c()
system.time(
 for(i in 1:n){
  Resultat <- c(Resultat, cos(i))
})
```

```
   user  system elapsed
  0.108   0.008   0.116
```

2. On reprend l'importation du TP2

```
DataBladder <- read.table(file = "bladder.txt",  # Adapter le chemin....
                      dec = ".", header = F, na.strings = "-500")
#Ne pas oublier de spécifier que les -500 sont des NAs!!
```

Calcul de la moyenne:

Pour gérer les `NA`, on peut utiliser l'argument `na.rm` de `mean`. On peut spécifier dans les apply les arguments optionnels des fonctions utilisées.

```
MoyLigne <- apply(X = DataBladder, MARGIN = 1, FUN = mean, na.rm = T)
# On applique la fonction mean (FUN) à toutes les lignes de bladder (X) en enlevant les NA (na.rm)
MoyCol <- apply(X = DataBladder, MARGIN = 2, FUN = mean, na.rm = T)
# On applique la fonction mean (FUN) à toutes les colonnes de bladder (X) en enlevant les NA (na.rm)
sum(is.na(MoyLigne))
```

```
[1] 0
```

```
sum(is.na(MoyCol))
```

```
[1] 0
```

On peut faire la même chose en utilisant les raccourcis `colMeans()` et `rowMeans()`:

```
MoyLigne2 <- colMeans(DataBladder,na.rm = TRUE)
MoyCol2 <- rowMeans(DataBladder,na.rm = TRUE)
#vérifier l'égalité:
all.equal(MoyCol,MoyLigne2)
```

```
[1] TRUE
```

```
all.equal(MoyLigne, MoyCol2)
```

```
[1] TRUE
```

Calcul de l'écart type:

```
MySd <- function(x,rm.na = TRUE){ #Par défaut on retire les valeurs manquantes.
  if(rm.na){
  x <- x[!is.na(x)] #on enleve les NA's
  }
  m <- mean(x)
  n <-length(x)
  return(sqrt(sum((x-m)^2)/(n-1))) #on utilise l'estimateur sans bias de l'ecart type
  }
MySdLigne <- apply(X = DataBladder, MARGIN = 1, FUN = MySd)
MySdCol <- apply(X = DataBladder, MARGIN = 2, FUN = MySd)
sum(is.na(MySdLigne))
```

```
[1] 0
```

```
sum(is.na(MySdCol))
```

```
[1] 1
```

Comparaison de résultats avec la fonction `sd()`:

```
SdLigne <- apply(X = DataBladder, MARGIN = 1, FUN = sd, na.rm=TRUE)
SdCol <- apply(X = DataBladder, MARGIN = 2, FUN = sd,na.rm=TRUE)
sum(is.na(SdLigne))
```

```
[1] 0
```

```
sum(is.na(SdCol))
```

```
[1] 1
```

```
all.equal(SdLigne,MySdLigne)
```

```
[1] TRUE
```

```
all.equal(SdCol,MySdCol)
```

```
[1] TRUE
```

La valeur `NA` de `SdCol` ou `MySdCol` correspond à une colonne ayant un seul élément différent de `NA` (et donc d'après notre définition $\hat{\sigma} = 0/0$). On peut corriger ça avec un `if`:

```
MySd <- function(x,rm.na = TRUE){ #Par défaut on retire les valeurs manquantes.
  if(rm.na){
  x <- x[!is.na(x)] #on enleve les NA's
  }
  m <- mean(x)
  n <-length(x)
  if (n==1) {
    return(0)
  }else{
  return(sqrt(sum((x-m)^2)/(n-1))) #on utilise l'estimateur sans bias de l'ecart type
    }
  }
MySdLigne <- apply(X = DataBladder, MARGIN = 1, FUN = MySd)
MySdCol <- apply(X = DataBladder, MARGIN = 2, FUN = MySd)
sum(is.na(MySdLigne))
```

```
[1] 0
```

```r
sum(is.na(MySdCol))
```

```
[1] 0
```

Question supplementaire:

```r
DC <- sweep(DataBladder,MARGIN = 2,STATS = MoyCol ) #le tableau de données centrées
sum(abs(colMeans(DC,na.rm = TRUE))>0.000001) #verification
```

```
[1] 0
```

```r
DCR <- sweep(DataBladder,MARGIN = 2,STATS = MySdCol,FUN = "/") #normaliser les données
```

Pour obtenir des données centrées réduites, on peut également utiliser directement la fonction `scale`.

```r
DataBladderCR <- scale(DataBladder)
```

3. De manière générale, si une `for` boucle n'est pas itérative (i.e. l'étape i ne dépend pas de l'étape $i-1$) alors, elle peut être effectuée par une fonction de type apply.

Exemples:

- appliquer la fonction crée en Section 2. m fois pour ensuite calculer la moyenne sur les resultats:

```r
NombreTirages <- function(n){
  count=1
  while(sample(1:100,1)!=n) {
   count=count+1
  }

  return(count)
}
m <- 100
myst <- sample(1:100,1)
mean(sapply(rep(myst,m),FUN = NombreTirages))
```

```
[1] 112.36
```

- calculer l'approché de e:

```r
n <- 50
sapply(1:n,FUN = function(x){(1+1/x)^x})
```

```
 [1] 2.000000 2.250000 2.370370 2.441406 2.488320 2.521626 2.546500
 [8] 2.565785 2.581175 2.593742 2.604199 2.613035 2.620601 2.627152
[15] 2.632879 2.637928 2.642414 2.646426 2.650034 2.653298 2.656263
[22] 2.658970 2.661450 2.663731 2.665836 2.667785 2.669594 2.671278
[29] 2.672849 2.674319 2.675696 2.676990 2.678208 2.679355 2.680439
[36] 2.681464 2.682435 2.683357 2.684232 2.685064 2.685856 2.686612
[43] 2.687333 2.688022 2.688681 2.689312 2.689917 2.690497 2.691053
[50] 2.691588
```

- calculer la moyenne de calories par producteur:

```r
library(MASS)
tapply(UScereal$calories,UScereal$mfr,mean)
```

```
       G        K        N        P        Q        R
137.7879 149.6710 160.2593 194.7578 135.8507 124.8521
```

Parfois, on peut utiliser des fonctions de type `apply` pour effectuer des boucles `for` itératives. On peut prendre comme exemple l'approximation de e effectuée à la deuxième question.

```
n <- 50
approximation <- function(k){ #approximation de e à l'ordre k
  return(sum(sapply(0:k, FUN = function(x){1/factorial(x)})))
}

VectAprox <- sapply(1:n,approximation)
```

4. Avec `for`:

```
x = seq(0,100,by=10)
y = sample(1:100,15)
PlusProche = function(x,y){
  n=length(y)
  Result=rep(x[1],n ) # on suppose que le premier element est le plus proche
  for(i in 1:n){
    for (xj in x) {
      if (abs(y[i]-xj)< abs(y[i]-Result[i])){
        Result[i] <- xj
      }
    }
  }
  return(Result)
}
y
```

```
 [1] 81 71 45 32 89 72 51 65 62  3 40 87 76  9 95
```

```
PlusProche(x,y)
```

```
 [1] 80 70 40 30 90 70 50 60 60  0 40 90 80 10 90
```

Avec `sapply`:

```
PlusProcheA=function(x,y){
  return(sapply(y,FUN=function(yj){ x[which.min(abs(x-yj))]}))
}
PlusProcheA(x,y)
```

```
 [1] 80 70 40 30 90 70 50 60 60  0 40 90 80 10 90
```

Si on veut s'intéresser seulement a x et y, définis dans la consigne du TP, en utilisant leur structure particulière, on peut obtenir le même résultat plus facilement :

```
floor((y/10))*10
```

```
 [1] 80 70 40 30 80 70 50 60 60  0 40 80 70  0 90
```

5. Produit matriciel:

```
p <- 3
n <- 2
v <- 1:p
A <- matrix(1:(n*p), byrow = TRUE, nrow = n, ncol = p)
A%*%v # produit usuel
```

```
     [,1]
[1,]   14
```

```
[2,]   32
```

```
apply(A,MARGIN = 1, FUN = function(x){sum(x*v)})
```

```
[1] 14 32
```

```
# avec apply le resultat est un vecteur et non une matrice
```

6. Avec `tapply` on peut calculer la consommation moyenne (`mpg`) en fonction deux facteurs: la transmission (`am`) et le nombre de cylindres (`cyl`) :

```
data("mtcars")
tapply(mtcars$mpg, INDEX = list(Transmission = mtcars$am, NbCyl = mtcars$cyl), FUN = mean)
```

```
            NbCyl
Transmission      4        6     8
           0 22.900 19.12500 15.05
           1 28.075 20.56667 15.40
```