TP de Probabilités

NASSREDDINE, HUA, CROWLEY

12/05/2020

Ce TP de probabilités a été réalisé en deux séances qui ont porté chacune sur une problématique différente. La première s'est intéressée à la génération de distributions pseudo-aléatoires et à différentes approches pour tester la qualité (i.e caractère aléatoire) des différents générateurs mis en place. La seconde, quant à elle, a été une application concrète d'un modèle Markovien de simulation de files d'attente. En bonus, on s'est appuyé sur des principes de simulations pour générer des lois de probabilités quelconques à partir d'une loi uniforme. Sa supervision a été assurée par Mme. Irène Gannaz, Mme. Léa Laporte et M. Stefan Duffner. Le rapport suivant a été rédigé par Mme CROWLEY Sophie, M. NASSREDDINE Zakaria et M. HUA Zihao.

Partie 1: Tests de générateurs pseudo-aléatoires

Au niveau de cette partie, on crée deux générateurs pseudo-aléatoires simples qui se rajoutent donc aux deux autres déjà fournis, et on leur fait passer des tests statistiques dans le but de comparer la qualité des séquences qu'ils produisent.

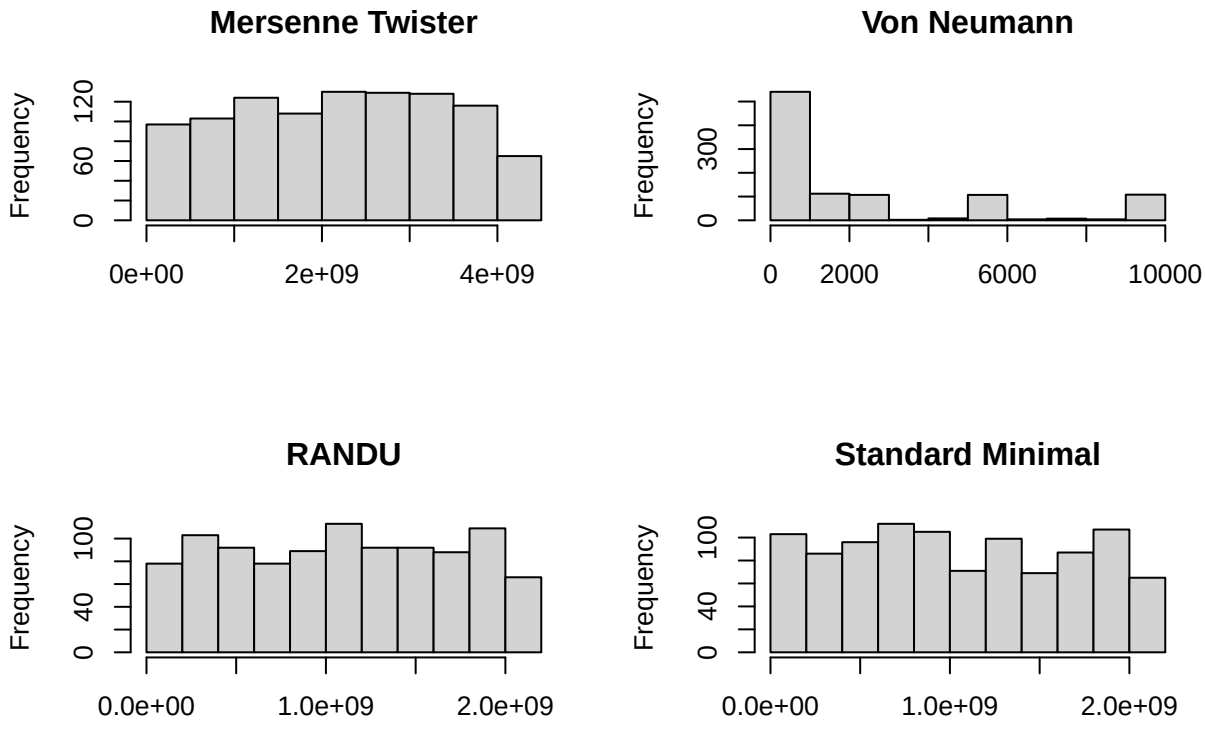
Question 1

On a commencé par mettre en place deux générateurs à congruence linéaire: Standard Minimal et RANDU. Veuillez vous référer au fichier 'generateurs.R' pour l'implémentation.

Question 2.1

On commence par tester visuellement ces générateurs en traçant leurs histogrammes de sortie pour une suite de k = 1000 valeurs.

```
source('generateurs.R');

sVN <- 9721 ; sMT <- 2504 ; Nsimu <- 1000 ; Nrepet <- 1 ;grain <- 999

vn <- VonNeumann(Nsimu,Nrepet,sVN)
mt <- MersenneTwister(Nsimu,Nrepet,sMT)
ru <- RANDU(Nsimu, grain)
sm <- STANDARD_MINI(Nsimu, grain)
par(mfrow=c(2,2))
hist(mt[,1],xlab='',main='Mersenne Twister')
hist(vn[,1],xlab='',main='Von Neumann')
hist(ru[,1],xlab='',main='RANDU')
hist(sm[,1],xlab='',main='Standard Minimal')
```

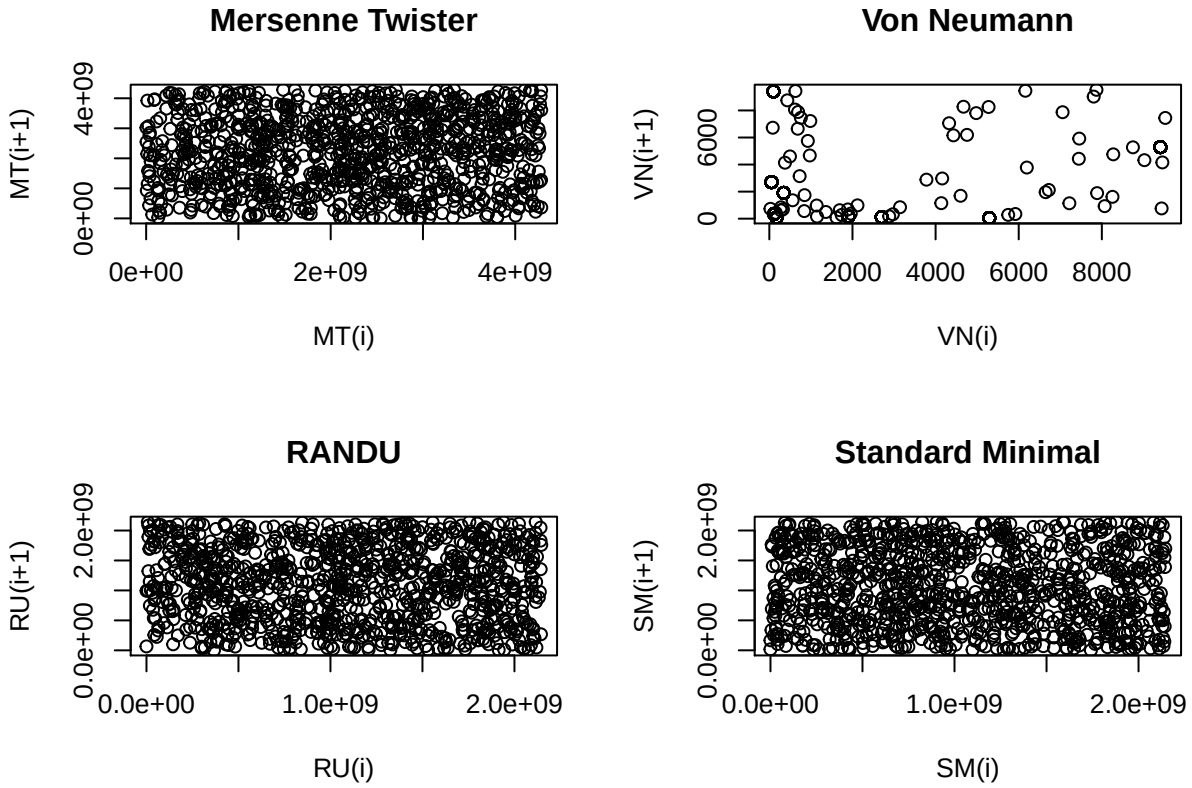


On observe que le générateur de Von Neumann est le moins performant. Ses valeurs générées ne sont pas distribuées de façon aussi uniforme que les autres.

Question 2.2

On suit ici l'évolution de chaque valeur générée en fonction de celle qui la précède. Nous obtenons:

```
par(mfrow=c(2,2))
plot(mt[1:(Nsimu-1),1],mt[2:Nsimu,1],xlab='MT(i)', ylab='MT(i+1)', main='Mersenne Twister')
plot(vn[1:(Nsimu-1),1],vn[2:Nsimu,1],xlab='VN(i)', ylab='VN(i+1)', main='Von Neumann')
plot(ru[1:(Nsimu-1),1],ru[2:Nsimu,1],xlab='RU(i)', ylab='RU(i+1)', main='RANDU')
plot(sm[1:(Nsimu-1),1],sm[2:Nsimu,1],xlab='SM(i)', ylab='SM(i+1)', main='Standard Minimal')
```



Conformément aux observations précédentes au niveau des histogrammes, le générateur de Von Neumann génère des valeurs très serrées et mal réparties ce qui nous permet donc de prédire, à vue d'oeil, ne réussit pas à produire des valeurs suivant une loi uniforme.

Question 3

On a d'abord implémenté la fonction réalisant le test de fréquence monobit comme indiqué sur le bout de code suivant:

```
Frequency <- function(x,nb)
{
  s<-0
  for(i in 1:length(x))
  {
    bin=binary(x[i])
    for(j in 0:(nb-1))
    {
      s <- s + (2*bin[32-j]-1)
    }
  }
  Sobs <- abs(s)/sqrt(nb*length(x))
  Pvaleur <- 2*(1-pnorm(Sobs))
  return(Pvaleur)
}
```

Il faut noter que cette fonction prend en paramètre, en plus de la séquence à évaluer, un entier indiquant

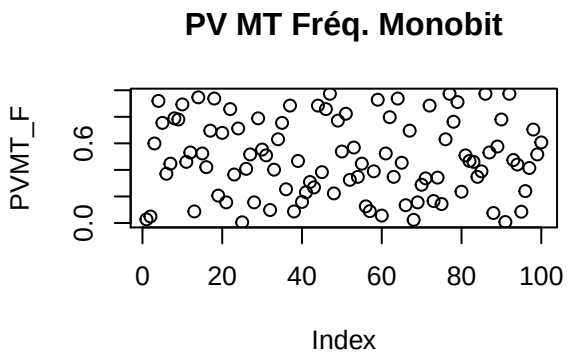
le nombre de bits à prendre en considération. En effet, chacun des générateurs étudiés génère des valeurs en "intervenant" sur un nombre défini de bits traduisant ces nombres en binaire. Il serait donc injuste de prendre en compte toute la séquence de bits lors des tests alors qu'un générateur donné n'aurait pu travailler que sur un nombre plus petit de bits, ce qui fausserait nos statistiques. Ci-dessous le nombre de bits à prendre en compte pour chacun:

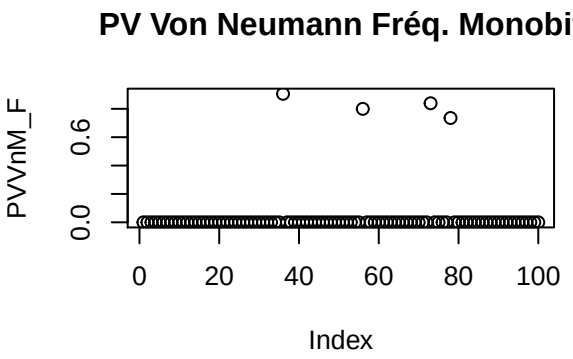
Générateur	Nombre de bits à considérer
Mersenne Twister	32
RANDU	31
Standard Minimal	31
Von Neumann	14

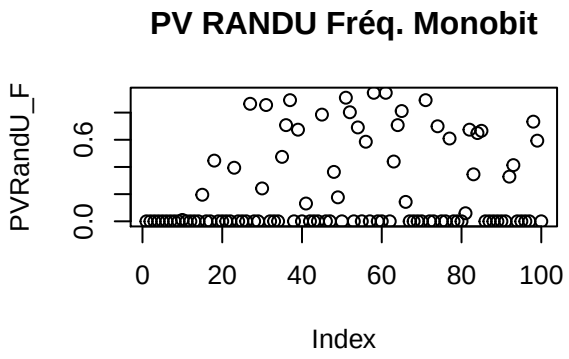
Tableau 1: Nombre de bits à considérer par générateur

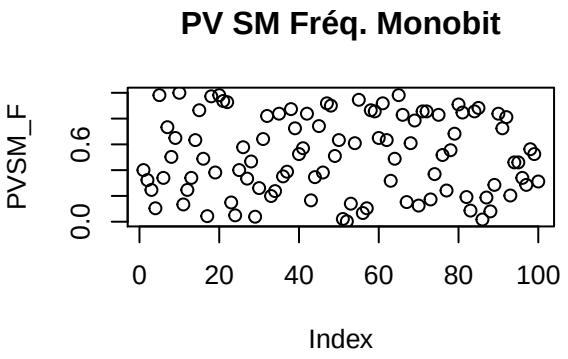
On réalise par la suite 100 mesures pour avoir un échantillon statistiquement significatif. On trace l'évolution de la p valeur pour chacun des générateurs et observe les comportement suivants:

```
samples = sample.int(100000,100)

PVMT_F = matrix(nrow=length(samples), ncol=1)
PVRandU_F = matrix(nrow=length(samples), ncol=1)
PVSM_F = matrix(nrow=length(samples), ncol=1)
PVVnM_F = matrix(nrow=length(samples), ncol=1)

for(i in 1:length(samples))
{
  vn <- VonNeumann(Nsimu,Nrepet,samples[i])
  mt <- MersenneTwister(Nsimu,Nrepet,samples[i])
  randu = RANDU(Nsimu, samples[i])
  sm = STANDARD_MINI(Nsimu, samples[i])

  PVMT_F [i] = Frequency(mt, 32)
  PVRandU_F [i] = Frequency(randu, 31)
  PVSM_F [i] = Frequency(sm,31)
  PVVnM_F [i] = Frequency(vn, 14)
}


par(mfrow=c(2,2))
plot(PVMT_F, main ='PV MT Fréq. Monobit' )
plot(PVVnM_F,main ='PV Von Neumann Fréq. Monobit')
plot(PVRandU_F,main ='PV RANDU Fréq. Monobit' )
plot(PVSM_F,main ='PV SM Fréq. Monobit')
```









Visuellement, et sans surprise, Von Neumann ne passe pas non plus ce test. RANDU n'a pas l'air de très bien se débrouiller non plus avec beaucoup de p-valeurs autour de 0. Pour mieux illustrer, on a calculer la moyenne, l'écart type ainsi que le nombre de valeurs supérieures à 1%:

```
vnP_F = mean(PVVnM_F)
mtP_F = mean(PVMT_F)
randuP_F = mean(PVRandU_F)
smP_F = mean(PVSM_F)

vnSD_F = sd(PVVnM_F)
mtSD_F = sd(PVMT_F)
rdSD_F = sd (PVRandU_F)
smSD_F = sd(PVSM_F)

good_PVMT_F <- 0
for (i in 1:length(samples) )
{
  if (PVMT_F [i] > 0.01)
  {
    good_PVMT_F  <- good_PVMT_F  +1
  }
}
```

Générateur	Moyenne	Écart Type	Nombre de p valeurs > 1%
Mersenne Twister	0.487407	0.2858662	98
Standard Minimal	0.5116205	0.3015067	99
RANDU	0.2085862	0.3189946	36
Von Neumann	0.03280238	0.1619859	4

Tableau 2: Distribution des p-valeurs pour le test de fréquence monobit

Ces indices nous montre donc que les générateurs qui passent le mieux ce test sont Mersenne Twister et Standard Minimal. L'étude sur RANDU donne beaucoup de p valeurs inférieures à 1% ce qui permettrait de l'éliminer. Von Neumann est complètement à côté de la plaque.

Question 4

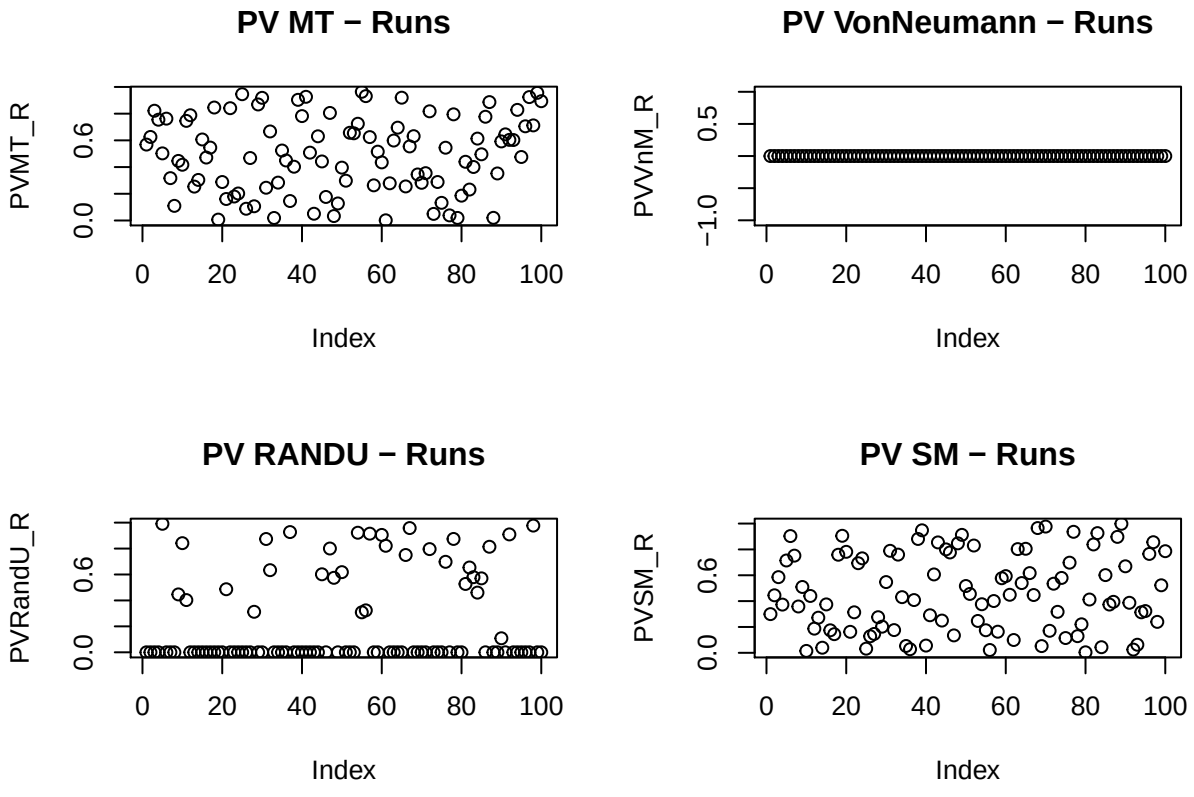
Nous avons implémenté le test des Runs de la manière suivante:

```
Runs <- function(x,nb)
{
  #Obtention de la séquence concaténée
  V = binary(x[1])
  V = V[(32-nb+1):32]

  for(i in 2:length(x)) {
    bin = binary(x[i])
    V = c(V,bin[(32-nb+1):32])
  }
  n <- length(V)
  #pre-test
  pi <- sum(V)/n
  tau <- 2/sqrt(n)
  Pvaleur <- 0
  if(abs(pi-0.5)<tau)
  {
    Vnobs <- 1
    for(j in 1:(n-1))
    {
      if(V[j]!=V[j+1]){
        Vnobs <- Vnobs + 1
      }
    }
    Pvaleur <- 2*(1-pnorm(abs(Vnobs-2*n*pi*(1-pi))/(2*sqrt(n)*pi*(1-pi))))
  }
  return(Pvaleur)
}
```

De la même façon, nous obtenons les indices suivants:

```
# Test des Runs

samples = sample.int(100000,100)
```

```
PVMT_R = matrix(nrow=length(samples), ncol=1)
PVRandU_R = matrix(nrow=length(samples), ncol=1)
PVSM_R = matrix(nrow=length(samples), ncol=1)
PVVnM_R = matrix(nrow=length(samples), ncol=1)

for(i in 1:length(samples))
{
  vn <- VonNeumann(Nsimu,Nrepet,samples[i])
  mt <- MersenneTwister(Nsimu,Nrepet,samples[i])
  randu = RANDU(Nsimu, samples[i])
  sm = STANDARD_MINI(Nsimu, samples[i])

  PVMT_R [i] = Runs(mt, 32)
  PVRandU_R [i] = Runs(randu, 31)
  PVSM_R [i] = Runs(sm,31)
  PVVnM_R [i] = Runs(vn, 14)
}

par(mfrow=c(2,2))
plot(PVMT_R,main ="PV MT - Runs")
plot(PVVnM_R,main ="PV VonNeumann - Runs")
plot(PVRandU_R,main ="PV RANDU - Runs")
plot(PVSM_R,main ="PV SM - Runs")
```



Générateur	Moyenne	Écart Type	Nombre de p valeurs > 1%
Mersenne Twister	0.5608341	0.2776735	99
Standard Minimal	0.4934389	0.3028145	98
RANDU	0.2360877	0.3425761	38
Von Neumann	0	0	0

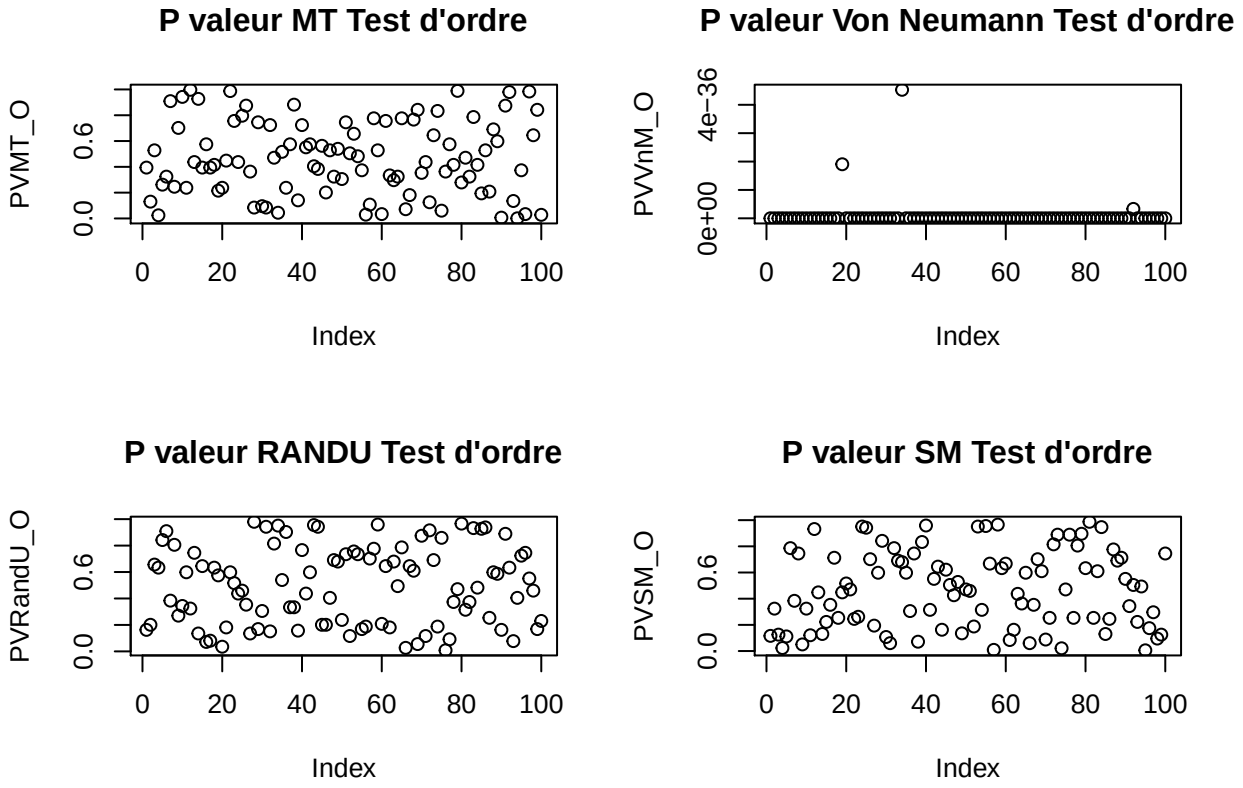
Tableau 3: Distribution des p-valeurs pour le test des Runs

Ce test étant une version plus poussée et mieux raffinée de celui de la fréquence monobit, il met plus à l'épreuve nos différents générateurs. Von Neumann s'en retrouve complètement écrasé et Mersenne Twister et Standard Minimal réussisent toujours aussi bien.

Question 5

Contrairement aux deux tests derniers qui étudiaient les séquences de bits générés, ce dernier test, dit d'ordre, s'intérsse directement aux suite de nombres obtenus.

```
samples = sample.int(100000,100)

PVMT_O = matrix(nrow=length(samples), ncol=1)
PVRandU_O = matrix(nrow=length(samples), ncol=1)
PVSM_O = matrix(nrow=length(samples), ncol=1)
PVVnM_O = matrix(nrow=length(samples), ncol=1)

for(i in 1:length(samples))
{
  vn <- VonNeumann(Nsimu,Nrepet,samples[i])
  mt <- MersenneTwister(Nsimu,Nrepet,samples[i])
  randu = RANDU(Nsimu, samples[i])
  sm = STANDARD_MINI(Nsimu, samples[i])

  PVMT_O [i] = randtoolbox::order.test(mt[,1], d=4, echo=FALSE)$p.value
  PVRandU_O [i] = randtoolbox::order.test(randu[,1], d=4, echo=FALSE)$p.value
  PVSM_O [i] = randtoolbox::order.test(sm[,1], d=4, echo=FALSE)$p.value
  PVVnM_O [i] = randtoolbox::order.test(vn[,1], d=4, echo=FALSE)$p.value
}
par(mfrow=c(2,2))
plot(PVMT_O,main ="P valeur MT Test d'ordre")
plot(PVVnM_O,main ="P valeur Von Neumann Test d'ordre")
plot(PVRandU_O,main ="P valeur RANDU Test d'ordre")
plot(PVSM_O,main ="P valeur SM Test d'ordre")
```



Les résultats sont récapitulés dans le tableau suivant:

Générateur	Moyenne	Écart Type	Nombre de p valeurs > 1%
Mersenne Twister	0.53	0.26	100
Standard Minimal	0.48	0.26	98
RANDU	0.52	0.29	100
Von Neumann	1.8e-37	1.7e-36	0

Tableau 4: Distribution des p-valeurs pour le test d'ordre

On pourrait remarquer que ce test est moins discriminant avec assez souvent un total de 100 p valeurs supérieures à 1% pour les deux meilleurs générateurs, et on observe aussi qu'il laisse davantage passer RANDU qui serait plus sévèrement éliminé par un test plus poussé comme celui des Runs.

Partie 2: Simulation de lois de probabilités quelconques

Au niveau de cette partie, on part d'un loi de probabilités uniforme pour en générer d'autre quelconques. Pour les lois, discrètes, on s'appuie sur le principe de simulation d'une loie discrète. Pour celles qui sont continues, on fait appel à deux algorithmes: La simulation par inversion et la simulation par rejet.

Question 1

On implémente ici une fonction qui génère une réalisation d'une loi binomiale de paramètres n et p donnés à partir d'une loi uniforme. Cette partie étant traitée en bonus, nous n'avons malheureusement pas eu la chance de faire valider nos résultats par un.e des enseignant.e.s. Nous avons commencé par l'implémentation qui nous a parue la plus intuitive de cette fonction:

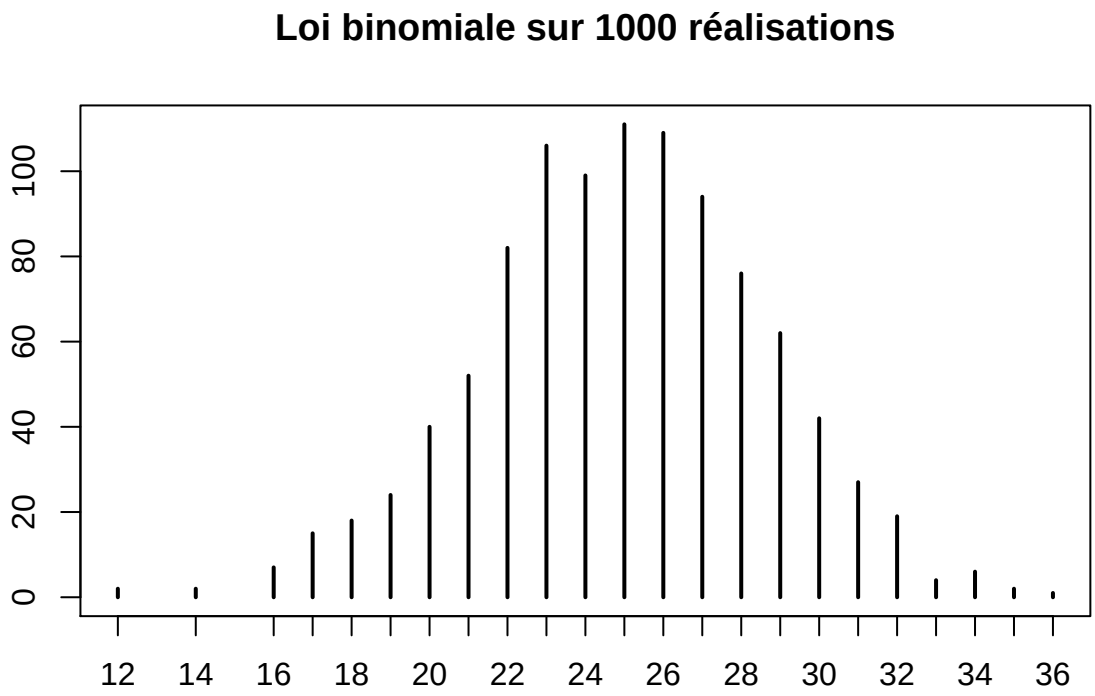
```
source('distributions.R')
LoiBinomiale <- function (n,p,k)
{
  B =  c()
  for (i in 1:k)
  {
    U = runif(n,min = 0,max = 1)
    B <- c(B, sum(U < p))
  }

  return (B)
}
```

Avec plus de recul et en comprenant mieux le bout d'algorithme de l'énoncé, nous avons implémenté la deuxième version:

```
BinomialeSj <- function (n,p,j) # Algo du sujet
{
  B = c()
  for (i in 1:j)
  {
    k = -1
    somme = 0
    U = runif(1,min = 0,max = 1)
    while (U > somme)
    {
      k <- k+1
      somme = somme + choose(n,k) * (p^k) * ((1-p)^(n-k))
    }
    B <- c(B, k)
  }
  return (B)
}
```

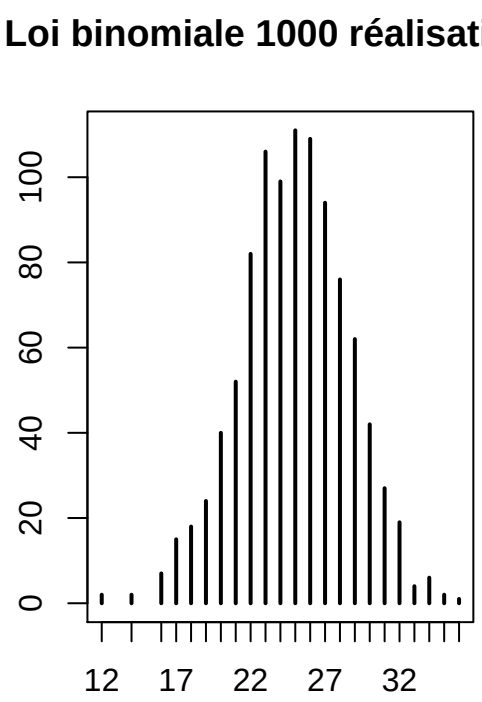
On représente ainsi le diagramme en bâtons obtenu sur 1000 réalisations:

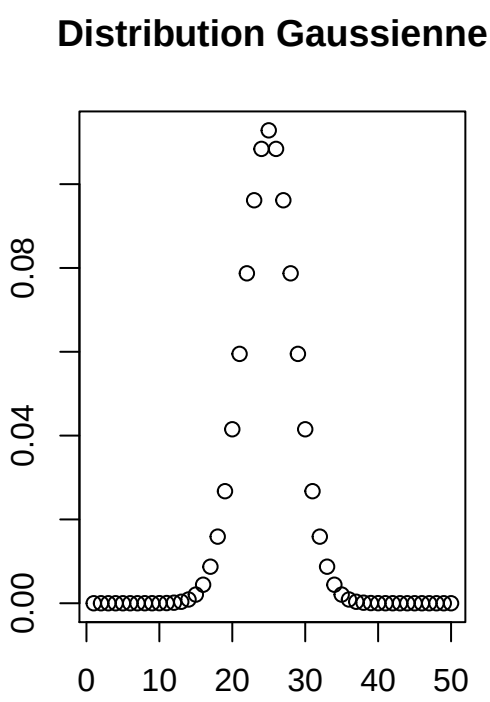
```
n = 50
p = 0.5
B = LoiBinomiale(n,p,1000)
plot(table(B),xlab='', ylab='',main ="Loi binomiale sur 1000 réalisations")
```



Comparons ensuite ce résultat à la densité d'une loi Gaussienne N(np, np(1-p)).

```
X = seq(from = 1, to = n, by =  1)
par(mfrow=c(1,2))
plot(table(B),xlab='', ylab='',main ="Loi binomiale 1000 réalisations")
plot(dnorm(X,n*p, sqrt(n*p*(1-p))), xlab='', ylab='',main ="Distribution Gaussienne")
```





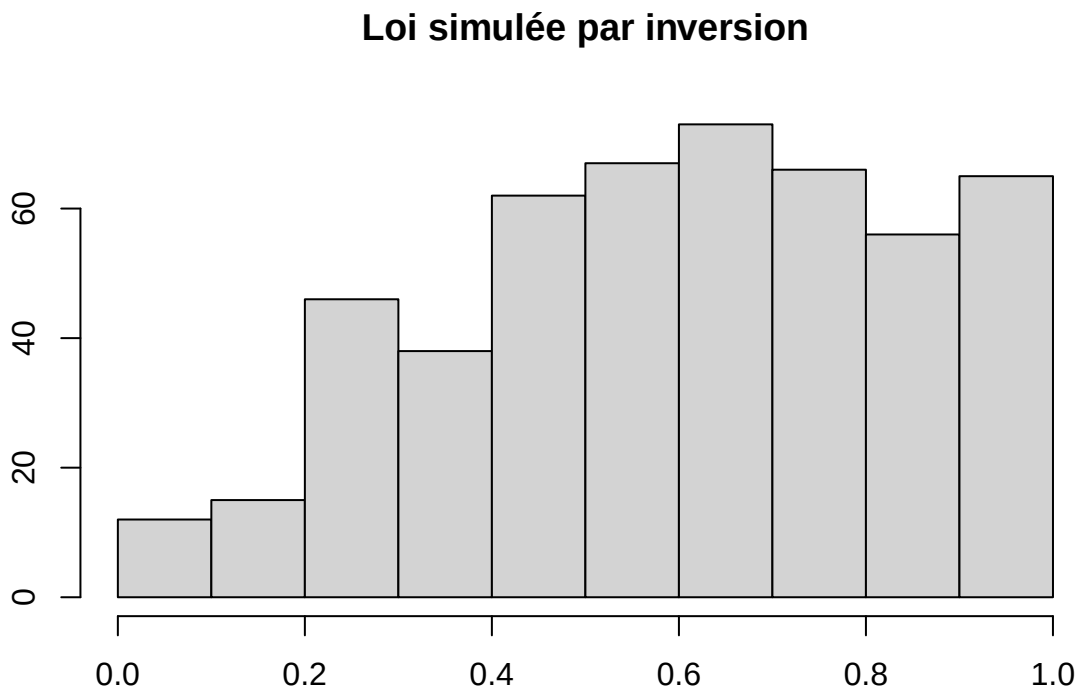
Question 2

On implémente d'abord l'algorithme de simulation par rejet qui est bien applicable à cette loi de probabilité.

```
Inversion <- function(n)
{
  X = c()
  for (i in 1:n)
  {
    U = runif(1,min = 0,max = 1)
    inv = exp(sqrt(U)*log(2))-1
    X <- c(X, inv)
  }
  return (X)
}
```

Observons le résultat obtenu:

```
n2 = 500
X2 = Inversion(n2)
hist(X2,xlab='', ylab='',main ="Loi simulée par inversion")
```

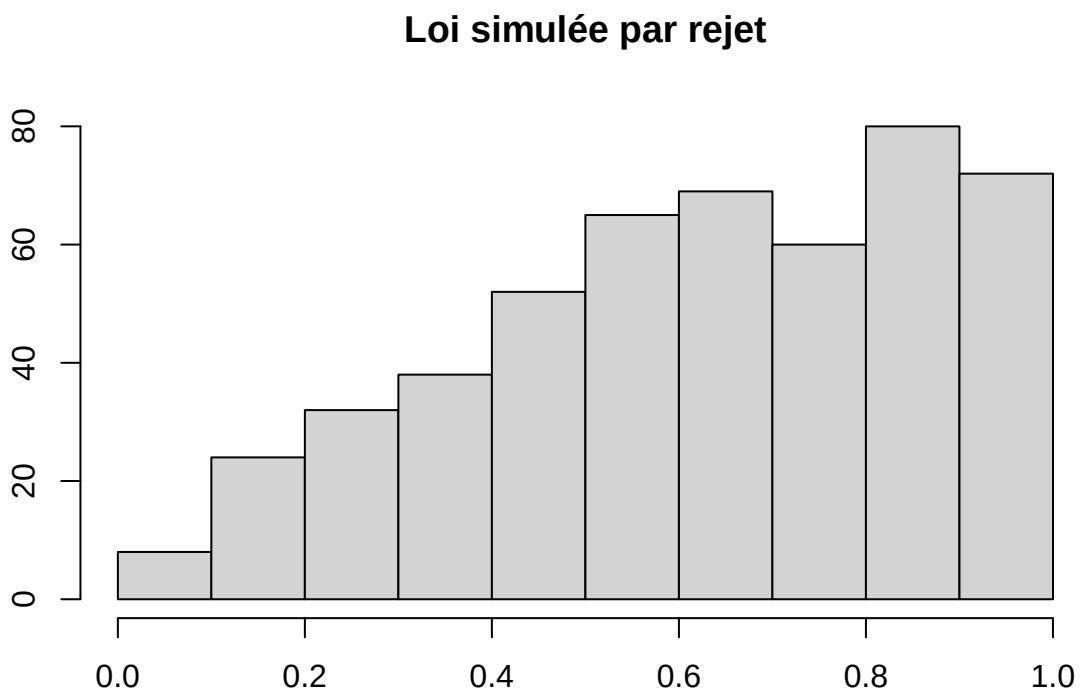


Afin de comparer ces deux approches, nous implémentons également la simulation par rejet:

```
Rejet <- function(n)
{
  c = 2/(log(2)^2)
  X = c()
  for (i in 1:n)
  {
    U = runif(1,min = 0,max = 1)
    Y = runif(1,min = 0,max = 1)
    while(U > log(1+Y)/(1+Y))
    {
      U = runif(1,min = 0,max = 1)
      Y = runif(1,min = 0,max = 1)
    }
    X <- c(X, Y)
  }
  return(X)
}
```

Testons là:

```
X3 = Rejet(n2)
hist(X3,xlab='', ylab='',main ="Loi simulée par rejet")
```



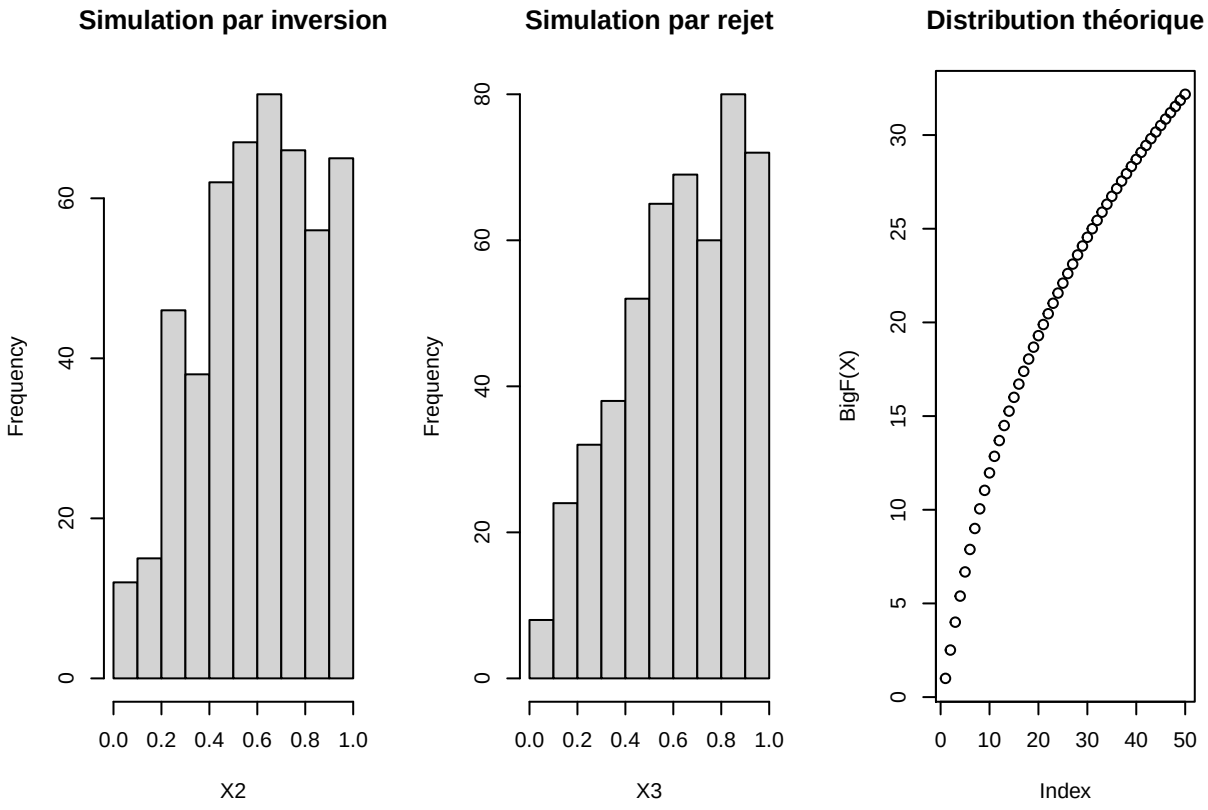
On s'intéresse par la suite à l'optimisation de chacune de ces méthodes. On y procède en comparant le temps de calcul pour simuler 1000 valeurs.

```
microbenchmark::microbenchmark(times=100,Rejet(1000),Inversion(1000))
```

```
## Unit: milliseconds
##             expr  min   lq mean median   uq max neval
##      Rejet(1000) 16.4 18.2 25.6   21.7 32.3  68   100
##  Inversion(1000)  3.3  3.7  5.2    3.8  4.7  25   100
```

Il est donc apparent que bien la simulation par rejet est valable pour beaucoup de fonctions de répartition, elle demeure nettement plus coûteuse et il en découle que dès lors que la méthode par inversion est applicable, il faudrait la privilégier. Enfin, on vérifie la proximité des valeurs simulées de la distribution de théorique en mettant côte à côte les histogrammes.

```
X_lin = seq(from = 0, to = 1, by =  0.1)
par(mfrow=c(1,3))
hist(X2, main='Simulation par inversion')
hist(X3, main='Simulation par rejet')
plot(BigF(X), main='Distribution théorique')
```



Partie 3: Application aux files d'attentes

On considère ici des files d'attentes au sens FIFO. Le temps écoulé entre deux arrivées de client et la durée de réponse suivent deux lois exponentielles de paramètres respectifs lambda et mu. On adopte un modèle Markovien M/M/1 pour modèliser notre système ce qui nous amène à étudier une Chaîne de Markov continue où un état correspond au nombre de clients actuellement dans le système.

Question 6

On implémente une fonction réalisant une simulation de file d'attente qui retourne deux tableaux, un contenant les temps d'arrivées et l'autre les temps de départ. Un exemple de situation simulée est indiqué ci-dessous:

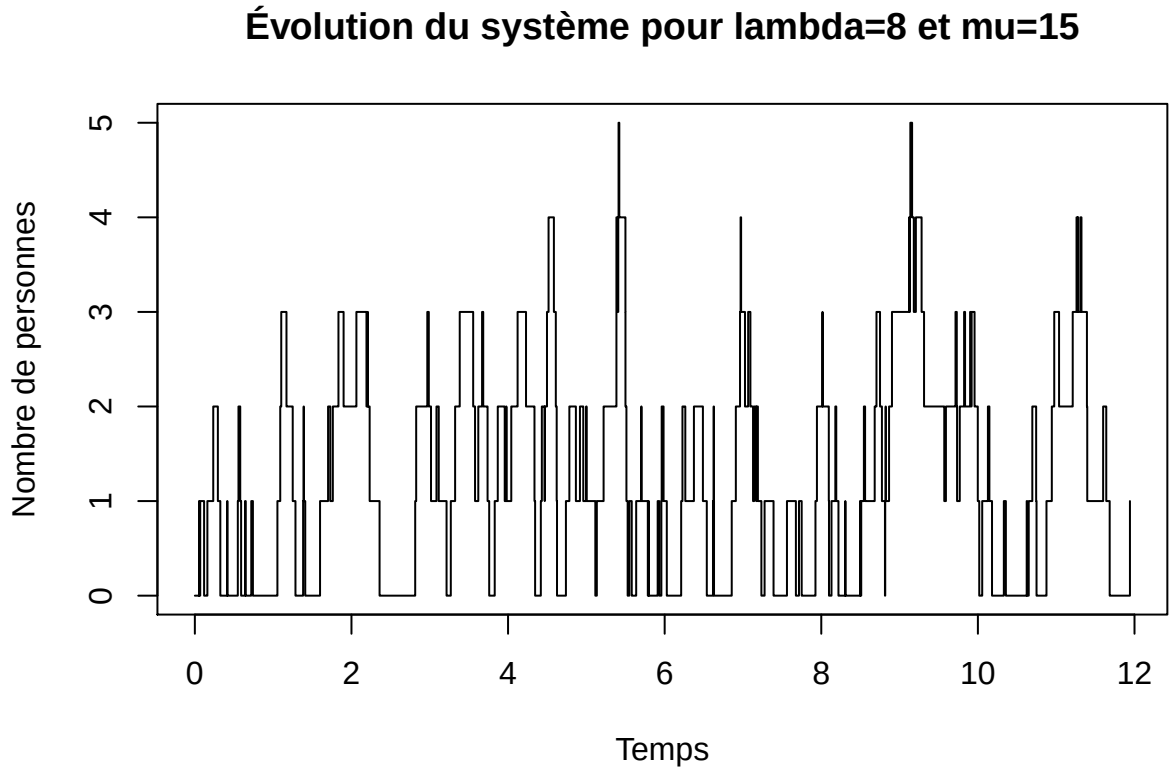
```
## [1] "Tableau d'arrivées"

##  [1] 0.32 0.71 0.94 0.96 1.51 1.55 1.67 2.31 2.38 2.40 2.49 2.58 2.87 2.89 3.19
## [16] 3.41 3.57

## [1] "Tableau de départs"

## [1] 0.37 1.13 2.16 2.58 3.15 3.49
```

##Question 7 À partir des données récupérées grâce à la fonction précédemment créée, on arrive à retourner l'évolution du nombre de clients dans le système grâce à la fonction suivante:

```
MM1Evolution <- function(queue)
{
  arrive <- queue[[1]]
  depart <- queue[[2]]
  nbarrive <- length(arrive)
  nbdepart <- length(depart)
  time <- matrix(nrow=nbarrive + nbdepart +1, ncol=1)
  attendees <- matrix(nrow=nbarrive + nbdepart +1, ncol=1)
  time[1] = 0
  attendees[1]= 0

  i <- 1
  j <- 1
  while ( i <= nbarrive && j <= nbdepart)
  {
    if (arrive[i] <= depart[j])
    {
      attendees[i+j] = attendees [i+j-1] + 1
      time [i+j] = arrive [i]
      i <- i+1
    }
    else
    {
      attendees[i+j] = attendees [i+j-1] - 1
      time [i+j] = depart [j]
      j <- j+1
    }
  }
  if (i <= nbarrive)
  {
    for (k in i:nbarrive)
    {
      attendees[k+j] = attendees[k+j-1] + 1;
      time [k+j] = arrive [k]
    }
  }
  else if (j <= nbdepart)
  {
    for (k in j:nbdepart)
    {
      attendees[k+i] = attendees[k+i-1] - 1;
      time [k+i] = depart [k]
    }
  }
  results <- list(time, attendees)
  return(results)
}
```

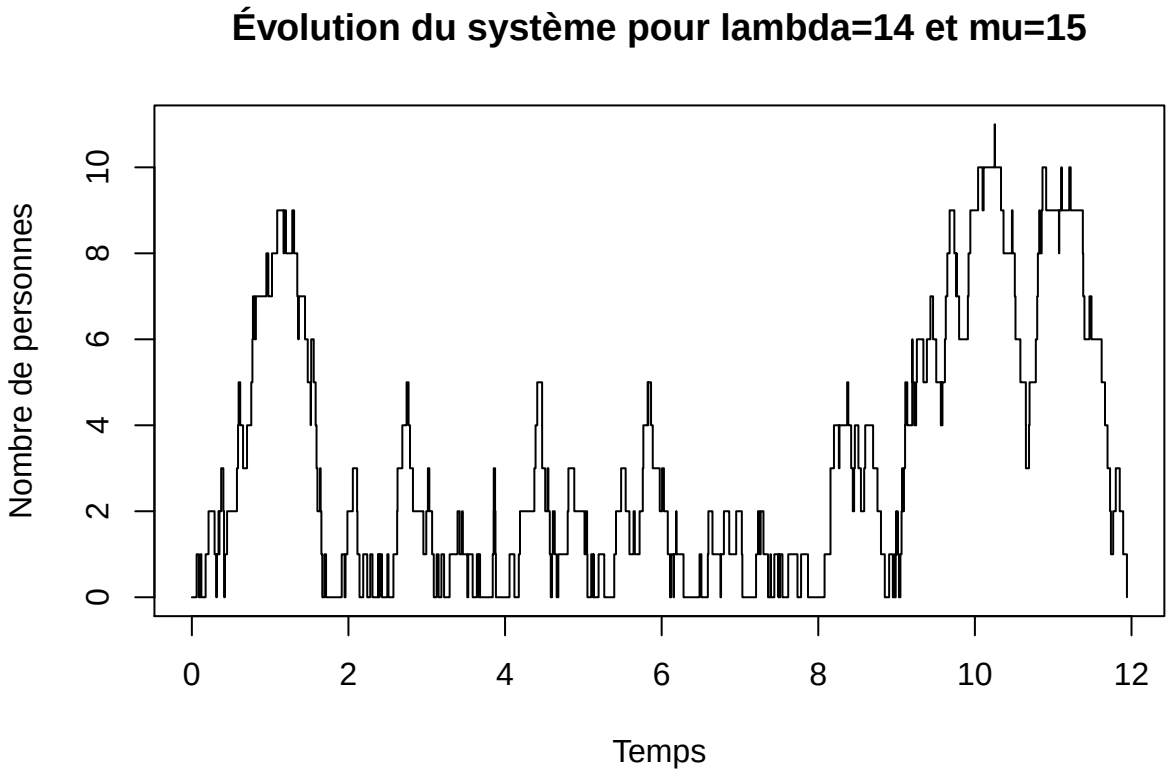
Pour faire en sorte qu'il y ait 8 clients en moyenne qui arrivent et 15 qui repartent par heure, et pendant 12 heures de fonctionnement, on obtient l'évolution suivante:

```
queue <- FileMM1(8,15,12)
results <-MM1Evolution(queue)
plot(results[[1]],results[[2]],'s', xlab='Temps', ylab = 'Nombre de personnes',
     main = 'Évolution du système pour lambda=8 et mu=15')
```

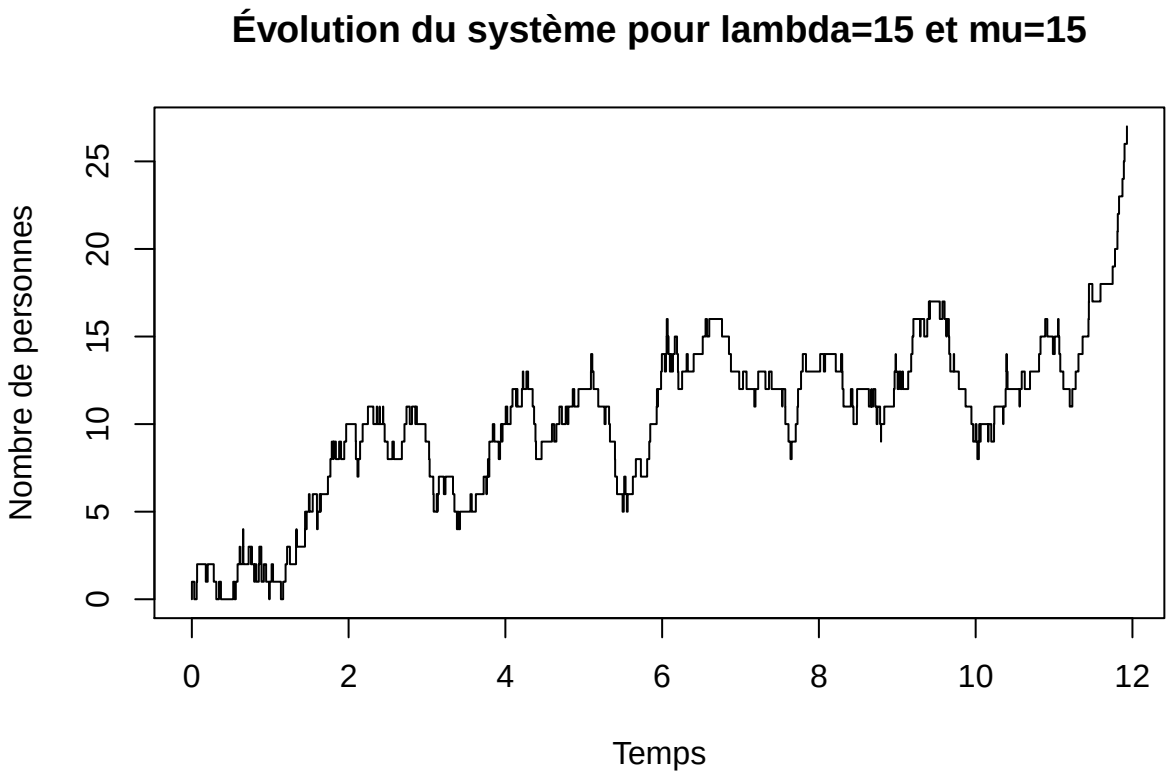


On observe alors un système qui fluctue, avec des petis pics, mais qui reste assez fluide et contrôlable dans le sens où il n'y a pas un moment où on arrive plus à répondre à la demande des client. On converge vers une moyenne, un régime stable, où on réussit à gérer l'ensemble des demandes entrantes. Pour différentes configurations, on obtient:

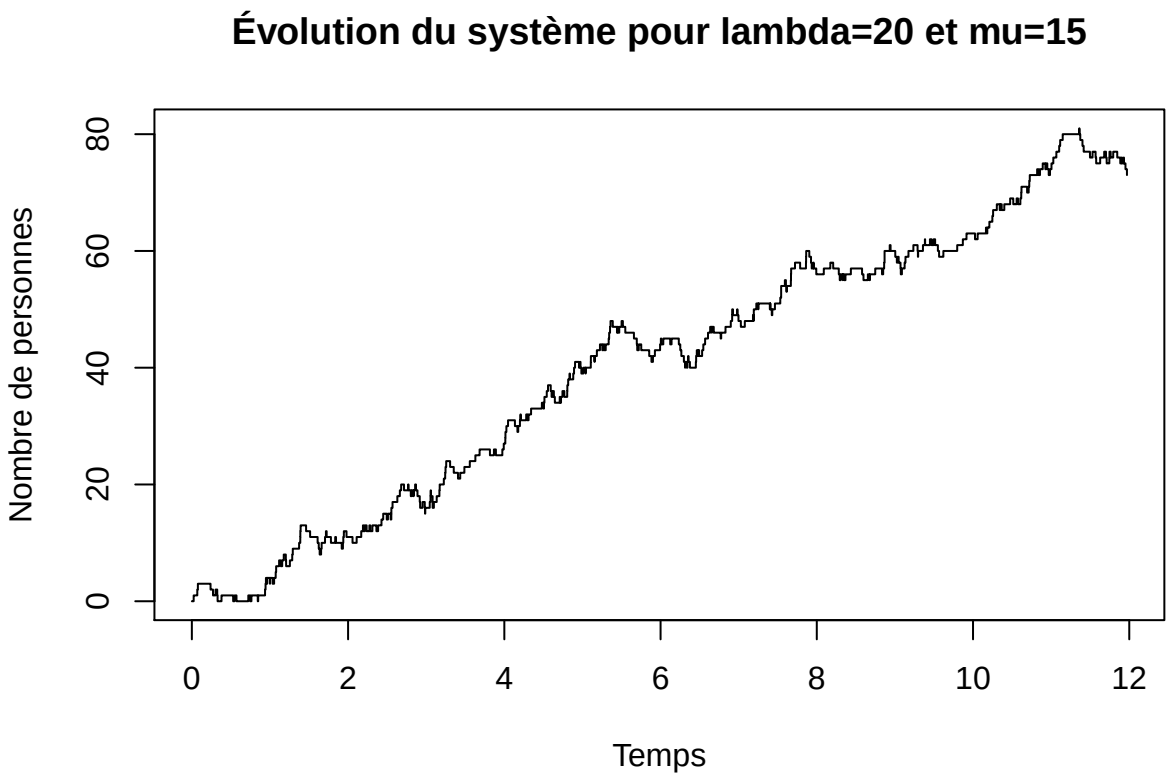
```
queue <- FileMM1(14,15,12)
results <-MM1Evolution(queue)
plot(results[[1]],results[[2]],'s', xlab='Temps', ylab = 'Nombre de personnes',
     main = 'Évolution du système pour lambda=14 et mu=15')
```




```
queue <- FileMM1(15,15,12)
results <-MM1Evolution(queue)
plot(results[[1]],results[[2]],'s', xlab='Temps', ylab = 'Nombre de personnes',
     main = 'Évolution du système pour lambda=15 et mu=15')
```




```
queue <- FileMM1(20,15,12)
results <-MM1Evolution(queue)
plot(results[[1]],results[[2]],'s', xlab='Temps', ylab = 'Nombre de personnes',
     main = 'Évolution du système pour lambda=20 et mu=15')
```



On remarque que plus on augmente la valeur de lambda en gardant fixée celle de mu, plus on évolue vers un système qui diverge, pour lequel il existe un moment où le nombre de personnes présentes dans le système augmente sans cesse, à quelques fluctuations locales en baisse qui sont négligeables devant l'évolution globale. Pour lambda à 14, on est arrive toujours à un régime stable, mais à partir de 15, on commence à diverger.

Question 8

On peut procéder à la vérification de la formule de Little pour lambda qui vaut 8 ou 14. Au-delà de 15, on est face à un système qui n'admet pas de régime stationnaire et tout le raisonnement construit autour d'une intensité du trafic inférieure à 1 perd son sens. On estime alors le temps moyens de présence d'un client dans le système:

```
AvgTime <- function(queue)
{
  arrive <- queue[[1]]
  depart <- queue[[2]]

  nbarrive <- length(arrive)
  nbdepart <- length(depart)
  avg <- 0
  for (i in 1:length(depart))
  {
    avg <- avg + depart[i] - arrive [i]
  }
  avg <- avg / length(depart)
```

```
  return(avg)
}
```

Le nombre moyen de clients dans le système est donné par:

```
AvgAttendance <- function(lambda, mu)
{
  alpha = lambda / mu
  E = (alpha) / (1 - alpha)
  return(E)
}
```

Pour lambda valant 8, on obtient:

```
queue <- FileMM1(8,15, 720)
temps <- AvgTime(queue)
N <- AvgAttendance(8, 15)
print(N/temps)
```

```
## [1] 7.8
```

Sur plusieurs essais réalisés sur nos machines, la valeur retournée est très proche de celle de lambda et on retrouve effectivement la formule de Little dans ce cas de figure. Et pour une valeur de 14 pour lambda:

```
queue <- FileMM1(14,15, 720)
temps <- AvgTime(queue)
N <- AvgAttendance(14, 15)
print(N/temps)
```

```
## [1] 19
```

En se fixant un lambda à 14, on s'approche dangereusement du cas limite et la formule de Little n'est pas tellement vérifiée pour l'intervalle de temps précédent. On décide de s'en affranchir en élargissant encore l'intervalle de temps étudié:

```
queue <- FileMM1(14,15, 2000)
temps <- AvgTime(queue)
N <- AvgAttendance(14, 15)
print(N/temps)
```

```
## [1] 15
```

Ce qui semble relativement remédier à cet "effet de bord".