title: "TP" authors: Isaline Foissey, Nour Eljadiri, Djouadi Celia output: pdf_document: default html_notebook: default

Tests de générateurs pseudo aléatoires

Question 1 : implémentation des algorithmes

Pour l'implémentation des différents générateurs, merci de vous référer au fichier `genrators.R` . On peut effectuer des tests afin de voir à quoi ressemblent les nombres générés par les différents générateurs. On prendra n = 10 afin de ne pas avoir des séquences beaucoup trop longues

```
# Une fenetre de l'explorateur de fichier s'ouvrira pour que vous choisissiez
generators.R
f <- file.choose()
f <- "generators.R"
source(file = f)
sequenceRANDU <- RANDU(graine = 69420, n = 10)
sequenceSM <- standardMinimal(graine = 69420, n = 10)
sequenceVN <- VonNeumann(graine = 69420, n = 10)[,1]
sequenceMT <- MersenneTwister(graine = 69420 , n = 10)[,1]
```

Les premières observations montrent que les nombres générées par le générateur de von neumann se retrouvent toujours entre 0 et 9999 (c'est la définition même de l'algoritme de ce générateur). Cependant, cela augmente les chances de collision entre les graines, et favorisera l'obtention de certaines valeurs au lieu d'autres.

Question 2 : tests visuels des générateurs

2.1 : historgramme de distributions des différents générateurs

```
test_visuel(100)
```

Commentaire

- **Von Neumann**

Le générateur de nombres aléatoires de Von Neumann produit plus de nombres entre 0 et 2000 que entre 2000 et 9999 en raison de l'étape de troncature dans l'algorithme. Lorsque le nombre au carré est inférieur à 1000, il conserve ses trois chiffres, donc le nombre généré reste dans l'intervalle de 0 à 999. En revanche, lorsque le nombre au carré est compris entre 1000 et 9999, la troncature supprime le premier et le dernier chiffre, ce qui donne un nombre entre 0 et 999.

Ainsi, plusieurs nombres au carré dans l'intervalle de 1000 à 9999 sont associés à la même valeur tronquée dans l'intervalle de 0 à 999. Par conséquent, il y a moins de nombres uniques dans l'intervalle de 2000 à 9999 par rapport à l'intervalle de 0 à 2000.

- **RANDU**

Le générateur RANDU fonctionne en appliquant de manière répétée une formule congruentielle linéaire, où chaque nombre est généré en multipliant le nombre précédent par un multiplicateur constant, puis en prenant le modulo du résultat. Le problème avec RANDU réside dans le choix de ses paramètres, qui entraînent des propriétés indésirables.

Un problème majeur est la présence d'une période courte. La séquence de nombres générés par RANDU se répète après un nombre relativement faible d'itérations. Cela signifie que le générateur va parcourir un ensemble limité de valeurs, ce qui entraîne une distribution non uniforme.

De plus, RANDU présente de mauvaises propriétés spectrales, ce qui signifie que les nombres générés ont tendance à se situer sur un nombre limité de plans ou d'hyperplans dans l'espace des nombres. Cet effet de regroupement introduit des motifs et des corrélations dans la séquence générée, s'éloignant ainsi d'une distribution uniforme.

- **Standard Minimal**

Il est conçu pour fournir une meilleure qualité de génération de nombres aléatoires par rapport à des générateurs plus simples comme RANDU.

Le générateur StandardMinimal utilise des paramètres spécifiques et des techniques de permutation pour améliorer les propriétés statistiques de la séquence de nombres générés. Il a une période beaucoup plus longue, ce qui signifie qu'il génère une séquence de nombres aléatoires qui se répète après un nombre beaucoup plus grand d'itérations. Cela contribue à une meilleure distribution des nombres aléatoires et réduit les effets indésirables tels que la corrélation et les biais.

- **Mersenne-Twister**

Le principal avantage du générateur Mersenne Twister est sa période extrêmement longue. Il peut générer des séquences de nombres aléatoires de qualité avec une période de 2^19937 - 1, ce qui signifie qu'il peut générer un nombre immense de nombres avant que la séquence ne se répète.

De plus, le générateur Mersenne Twister présente de bonnes propriétés statistiques. Il produit des nombres qui sont largement distribués et qui semblent indépendants les uns des autres. Il a une excellente uniformité, une faible corrélation et un bon mélange des bits, ce qui le rend adapté à une large gamme d'applications nécessitant une génération de nombres aléatoires de haute qualité.

2.2 : Traçage de $S_n = f(S_{n-1})$

```
tracage()
```

- **RANDU** A première vue, le traçage $S_n = f(S_{n-1}) pour RANDU ne montre aucun motif répétitif, ni cluster. Cependant, il est important de noter que RANDU est connu pour avoir des propriétés statistiques indésirables, telles qu'une période courte et une distribution non uniforme (en raison de sa fonction génératrice). Bien que cela ne soit pas clairement visible dans le graphique, ces problèmes peuvent avoir un impact sur la qualité des nombres pseudo-aléatoires générés par cet algorithme.
- **Standard Minimal** Tout comme RANDU, le traçage $S_n = f(S_{n-1}) pour Standard Minimal ne montre aucun motif répétitif, ni cluster. En effet, le générateur a été conçu pour améliorer la qualité des nombres aléatoires générés par rapport à des générateurs plus simples (RANDU). On peut voir alors que la distribution du nuage de points est plus homogène que celle de RANDU.

La faible présence de motifs/clusters pour ces deux algorithmes vient appuyer les distributions observées grâce aux différents histogrammes de la question précédente.

- **Von Neumann** La présence de clusters évidents dans la visualisation de Sn en fonction de S(n-1) pour le générateur de Von Neumann peut être interprétée comme un signe de non-uniformité dans la distribution des nombres pseudo-aléatoires générés. Ces clusters indiquent que les valeurs générées tendent à être regroupées dans certaines plages spécifiques plutôt que d'être uniformément réparties sur toute la plage de valeurs possibles.

D'un point de vue probabiliste, cela peut être problématique car cela signifie que certaines valeurs sont plus susceptibles d'être générées que d'autres. Cela peut introduire un biais dans les résultats d'une simulation ou d'une analyse statistique qui utilise ces nombres pseudo-aléatoires.

De plus, les clusters observés peuvent également indiquer une dépendance entre les nombres générés. Dans un générateur pseudo-aléatoire idéal, les valeurs successives devraient être indépendantes les unes des autres. Cependant, la présence de clusters suggère une corrélation ou une dépendance entre les nombres générés, ce qui peut affecter la propriété d'indépendance statistique souhaitée pour les applications probabilistes.

- **Mersenne-Twister** Dans le cas du générateur Mersenne-Twister, la visualisation de S_n en fonction de S_{n-1} montre une distribution uniforme des points sans la présence de clusters ou de motifs apparents. Cela indique que les nombres générés par le générateur Mersenne-Twister sont indépendants les uns des autres et qu'ils sont répartis de manière équilibrée sur toute la plage de valeurs possibles.

D'un point de vue probabiliste, cette distribution uniforme suggère que chaque nombre généré par le Mersenne-Twister a la même probabilité d'apparaître. Cela correspond à l'une des propriétés souhaitées pour un générateur pseudo-aléatoire de haute qualité.

De plus, la longue période du Mersenne-Twister `(2^19937 - 1)` contribue à assurer une faible corrélation entre les nombres générés successivement. Cela est crucial pour de nombreuses applications probabilistes qui dépendent de l'indépendance statistique des nombres aléatoires.

Test de fréquence monobit

La fonction binary qui nous est fournie permet de convertir tout nombre entier en séquence de 32 bits

Implémentation

On commence par implémenter la fonction de calcul du nombre de bits à 1. En résumé, la fonction "Frequency" convertit chaque élément du vecteur "x" en une séquence binaire, calcule une somme pondérée de cette séquence, puis applique un test statistique pour évaluer la fréquence observée de cette somme. La valeur de "p_value" renvoyée indique la probabilité de l'observation obtenue sous l'hypothèse nulle d'une distribution normale.

L'appel à la fonction se fait sous la forme `p_value <- Frequency(x, nb)`

Test de fréquence monobit

La première étape consiste à prendre 100 graines aléatoires entre 1 et 10000. On utilise la fonction `set.seed()` afin que la fonction `sample.int()` retourne la même séquence.

```
set.seed(555)
seeds <- sample.int(10000 , 100)
```

On procède ensuite à la génération des séquences grâce aux générateurs pseudo-aléatoires étudiés. Les vecteurs `freq_GENERATEUR` stockent les différentes p-valeurs retournées par la fonction `Frequency(x , nb)` pour chaque séquence aléatoire générée par une graine différente. L'implémentation de la fonction `Frequency()` , ainsi que toutes les fonctions qui implémentent un test statistique se trouvent dans le fichier `util.R`

```
frequences <- test_frequence(seeds)
```

La prochaine étape consiste à effectuer une étude de la qualité des différents générateurs. On commence d'abord par extraire le nombre de fois qu'une p-valeur est inférieure à notre seuil de signification (0.01), cela pourra nous donner une indication de combien de valeurs *supposées aléatoires* pouvons nous être surs de rejetter pour chaque générateur.

```
# Nombre de valeurs inferieures à 0.01
n_RANDU <- length(which(frequences$RANDU < 0.01))
n_SM <- length(which(frequences$SM < 0.01))
n_VonNeumann <- length(which(frequences$VN < 0.01))
n_MersenneTwister <- length(which(frequences$MT < 0.01))
```

Un autre indicateur interessant est la valeur moyenne de la p-valeur pour chaque séquence de fréquences monobit générée.

```
#Calcul de la p_value moyenne pour chacun des générateurs
avg_pvalue_RANDU <- mean(frequences$RANDU)
avg_pvalue_SM <- mean(frequences$SM)
avg_pvalue_VonNeumann <- mean(frequences$VN)
avg_pvalue_MersenneTwister <- mean(frequences$MT)
```

On remarque que Mersenne-Twister et StandardMinimal ont de bien meilleures performances que les autres algorithmes. Cela vient confirmer la distribution qu'on a plot plus haut, qui montre que les distributions de ces deux algorithmes sont celles qui se rapprochent le plus d'une loi uniforme.

Le nombre de p-value inférieures à 0,01 sont les suivants :

- Mersenne-Twister : 0
- RANDU : 54
- StandardMinimal : 2
- VonNeumann : 98

Cela suggère que pour les séquences générées :

Mersenne-Twister : Aucune p-valeur n'est inférieure à 0,01. Cela indique que la séquence générée par Mersenne-Twister semble être en conformité avec la distribution aléatoire équilibrée, du moins en termes de ce test spécifique.

RANDU : 54 p-valeurs sont inférieures à 0,01. Cela suggère que la séquence générée par RANDU présente un déséquilibre significatif et ne suit pas une distribution aléatoire équilibrée selon ce

test. Il peut y avoir un biais ou une structure non aléatoire dans la séquence.

StandardMinimal : 2 p-valeurs sont inférieures à 0,01. Cela suggère que la séquence générée par StandardMinimal présente également un léger déséquilibre, et ne suit pas une distribution aléatoire équilibrée selon ce test. Le générateur reste cependant de meilleure qualité que RANDU.

VonNeumann : 98 p-valeurs sont inférieures à 0,01. Cela indique que la séquence générée par VonNeumann présente un déséquilibre important et ne suit pas une distribution aléatoire équilibrée selon ce test.

Aussi, nous avons calculé la p_value moyenne pour chaque séquence générée :

- RANDU : 0.197
- StandardMinimal : 0.504
- Von Neumann : 0.014
- Mersenne-Twister : 0.493

Bien que la moyenne de ces p_value peut suggérer que Standard Minimal est meilleur que Mersenne-Twister, il faut garder en tête que le générateur de Mersenne-Twister n'a renvoyé aucune p_valeur inférieure au niveau de signification.

Test des runs

On reprend alors les tests sur les séquences générées précédemment

```
runs <- test_runs(seeds)

# Nombre de p-valeurs inferieures à 0.01
n_RANDU <- length(which(runs$RANDU < 0.01))
n_SM <- length(which(runs$SM < 0.01))
n_VonNeumann <- length(which(runs$VN < 0.01))
n_MersenneTwister <- length(which(runs$MT < 0.01))
#Calcul de la p_value moyenne pour chacun des générateurs
avg_pvalue_RANDU <- mean(runs$RANDU)
avg_pvalue_SM <- mean(runs$SM)
avg_pvalue_VonNeumann <- mean(runs$VN)
avg_pvalue_MersenneTwister <- mean(runs$MT)
```

Le nombre de p-value inférieures à 0,01 sont les suivants :

- Mersenne-Twister : 0
- RANDU : 43
- StandardMinimal : 3
- VonNeumann : 100

Cela suggère que pour les séquences générées :

Mersenne-Twister : Aucune p-valeur n'est inférieure à 0,01. Cela indique que la séquence générée par Mersenne-Twister semble être en conformité avec la distribution aléatoire équilibrée, du moins en termes de ce test spécifique.

RANDU : 43 p-valeurs sont inférieures à 0,01. Cela suggère que la séquence générée par RANDU présente un déséquilibre significatif et ne suit pas une distribution aléatoire équilibrée selon ce test. Il peut y avoir un biais ou une structure non aléatoire dans la séquence.

StandardMinimal : 3 p-valeurs sont inférieures à 0,01. Cela suggère que la séquence générée par StandardMinimal présente également un léger déséquilibre, et ne suit pas une distribution aléatoire équilibrée selon ce test. Le générateur reste cependant de meilleure qualité que RANDU.

VonNeumann : 100 p-valeurs sont inférieures à 0,01. Cela indique que la séquence générée par VonNeumann présente un déséquilibre important et ne suit pas une distribution aléatoire équilibrée selon ce test. L'hypothèse de la distribution des nombres uniforme est à rejeter.

Aussi, nous avons calculé la p_value moyenne pour chaque séquence générée :

- RANDU : 0.260
- StandardMinimal : 0.488
- Von Neumann : 0
- Mersenne-Twister : 0.508

Bien que la moyenne de ces p_value peut suggérer que Standard Minimal est meilleur que Mersenne-Twister, il faut garder en tête que le générateur de Mersenne-Twister n'a renvoyé aucune p_valeur inférieure au niveau de signification.

Ce test aussi montre que en moyenne, le générateur de Mersenne-Twister présente de meilleures performances que le générateur Standard Minimal. Cela est du au fait que les valeurs générées par `Standard Minimal` peuvent suivre des motifs.

Test d'ordre

```
orders <- test_order(seeds)

# Nombre de p-valeurs inferieures à 0.01
n_RANDU <- length(which(orders$RANDU < 0.01))
n_SM <- length(which(orders$SM < 0.01))
n_VonNeumann <- length(which(orders$VN < 0.01))
n_MersenneTwister <- length(which(orders$MT < 0.01))
```

Le nombre de p-value inférieures à 0,01 sont les suivants :

- Mersenne-Twister : 1
- RANDU : 3
- StandardMinimal : 1
- VonNeumann : 83

Mersenne-Twister : Le générateur Mersenne-Twister présente seulement 1 p-valeur inférieure à 0,01 lors du test d'ordre. Cela suggère que ce générateur produit des séquences de nombres aléatoires qui sont relativement bien réparties et ne montrent pas de structure ou de biais significatifs.

RANDU : Le générateur RANDU présente 3 p-valeurs inférieures à 0,01 lors du test d'ordre. Cela indique que les séquences générées par RANDU peuvent présenter des motifs ou des biais non aléatoires, ce qui les rend moins appropriées pour des applications nécessitant une génération de nombres véritablement aléatoires.

StandardMinimal : Le générateur StandardMinimal montre seulement 1 p-valeur inférieure à 0,01 lors du test d'ordre. Cela suggère qu'il peut produire des séquences de nombres aléatoires relativement bien réparties, similaires au Mersenne-Twister.

VonNeumann : Le générateur VonNeumann présente un nombre élevé de 83 p-valeurs inférieures à 0,01 lors du test d'ordre. Cela indique qu'il peut produire des séquences de nombres présentant des structures ou des biais significatifs, les rendant moins souhaitables pour des applications nécessitant une génération de nombres aléatoires de haute qualité.

Nous avons testé le générateur RANDU sur 100 graines différentes et que vous obtenez des résultats relativement bons lors du test d'ordre, cela pourrait être dû à la chance ou à des particularités des graines spécifiques que nous avons utilisées. RANDU peut, dans certains cas, générer des séquences qui semblent aléatoires ou qui passent certains tests statistiques, mais il reste un générateur de nombres pseudo-aléatoires de mauvaise qualité sur le long terme.

```
#Calcul de la p_value moyenne pour chacun des générateurs
avg_pvalue_RANDU <- mean(orders$RANDU)
avg_pvalue_SM <- mean(orders$SM)
avg_pvalue_VonNeumann <- mean(orders$VN)
avg_pvalue_MersenneTwister <- mean(orders$MT)
```

Aussi, nous avons calculé la p_value moyenne pour chaque séquence générée :

- RANDU : 0.48
- StandardMinimal : 0.49
- Von Neumann : 0.008
- Mersenne-Twister : 0.52

Mersenne-Twister reste le meilleur générateur en terme de qualité des nombres générés, tandis que Von Neumann est celui qui présente les plus mauvaises performances. Le générateur RANDU n'est pas réputé pour présenter de bons résultats lors du test d'ordre, mais plutôt pour échouer à ce test.

Application aux files d'attente

Question 6

Le fichier `files_d_attente.R` contient l'implémentation de la méthode FileMM1. En résumé, la fonction "FileMM1" simule la file d'attente M/M/1 en générant les temps d'arrivée des clients et en calculant les temps de départ correspondants, en tenant compte des taux d'arrivée et de service. La simulation s'arrête lorsque le temps de départ dépasse une limite fixée. La fonction renvoie une liste contenant les vecteurs des temps d'arrivée et de départ.

L'appel à la fonction se fait alors comme suit.

```
ma_list <- FileMM1(lambda = 10 , mu = 10 , D = 12)
```

Question 7

On commence par trouver le nombre de clients présents à chaque instant. Pour cela, on combine les deux vecteurs `arrivée` et `départ` qu'on trie par ordre croissant. A chaque arrivée, le nombre de client augmente de 1, et à chaque départ, ce nombre diminue de 1.

```
evolution_nb_clients <- function(arrivees , departs){
  #on crée deux data frame qui contiennent les heures d'arrivées et de départs
de clients ainsi qu'un tag valant -1 si l'horaire correspond à une heure de
départ et 1 si elle correspond à une heure d'arrivée
  a_df <- data.frame(Temps = arrivees, Tag = rep(1,length(arrivees)))
  d_df <- data.frame(Temps = departs, Tag = rep(-1,length(departs)))

  # On concatene ces deux data frame
  timeline <- rbind(a_df, d_df)

  # On trie le data frame dans l'ordre croissant et on fait une somme cumulée
des tags. Cela représente le nombre de clients dans la file à chaque arrivée de
client.
  timeline <- timeline[order(timeline$Temps), ]
  timeline$nb_clients <- cumsum(timeline$Tag)

  return (timeline)
}
```

```
liste_de_files <- generateFileMM1(duration = 12)
testLittle <- compare_nb_clients(liste_de_files = liste_de_files)
```

Commentaires :

On remarque que le nombre de clients en simultané reste acceptable dans les 2 premiers cas. Dans les cas où lambda et mu valent respectivement 20, 20 et 30,20 on remarque que le nombre de personne dans le magasin en simultané est croissant. La file d'attente deviens quasiment interminable.

Question 8 : Calcul du nombre de client moyen

Le calcul du nombre de clients moyen en attente se fait en prenant la moyenne pondérée par la durée d'attente. L'implémentation par dataframe permet d'utiliser la fonction `weighted.mean()`.
Exemple

```
mean1 <- weighted.mean(testLittle$lambda10$nb_clients, w =
c(0,diff(testLittle$lambda10$Temps)))
```

En résumé, la fonction "temps_presence" calcule la durée moyenne de présence des clients dans un système en soustrayant les temps d'arrivée des temps de départ, puis en prenant la moyenne de ces écarts.

```
temps_presence <- function(arrivee, depart) {
  delta <- c()
  for(i in seq_along(depart)) {
    delta <- c(delta, depart[i]-arrivee[i])
  }
  return(mean(delta))
}
```

Par la suite, on essaie de regrouper les différents résultats obtenus afin de les présenter sous forme de tableau.

```
means <- generate_means(testLittle)
average_wait_times <- generate_average_duration(liste_de_files)

results <- collect_results(means, average_wait_times)
```

Commentaire :

On s'aperçoit que le nombre de clients moyen en simultané est très élevé dans le cas 2 et 3. Les clients attendent plusieurs minutes en moyenne pour être pris en charge.

Après execution de nos algorithmes, on trouve que la loi de Little est presque vérifiée pour des valeurs élévées de D.

```
knitr::kable(results, format = "markdown", caption = "File M/M/1 , D = 12")
```

On remarque qu'on se rapproche de plus en plus de la formule de Little, plus le taux d'arrivée augmente.

En effectuant plusieurs tests en faisant varier la durée total du service, on remarque aussi qu'on se rapproche de la formule de Little.

Test avec D = 20h

```
liste_de_files2 <- generateFileMM1(duration = 20)
testLittle2 <- compare_nb_clients(liste_de_files2)

means2 <- generate_means(testLittle2)
avg_durations2 <- generate_average_duration(liste_de_files2)


results2 <- collect_results(means2, avg_durations2)
```

```
knitr::kable(results2, format = "markdown", caption = "File M/M/1 , D = 20")
```

En analysant les résultats présentés, on peut observer une situation intéressante. Le paramètre D, qui représente la durée maximale de service, a été augmenté. On constate que cette augmentation de D conduit à une diminution de l'écart relatif entre le nombre de client moyen calculé à l'aide de la formule de Little (En = lambda * Ew) et le nombre de client moyen réel.

Plus précisément, à mesure que D augmente, l'écart relatif diminue. Cela peut s'expliquer par le fait que lorsque D est plus grand, les clients ont davantage de temps pour être servis, ce qui réduit la probabilité qu'ils dépassent la limite de temps et soient forcés de quitter le système. Par conséquent, la différence entre le temps moyen de séjour estimé par la formule de Little et le temps de séjour réel devient plus faible.

Ces résultats indiquent que l'augmentation de la durée maximale de service a un impact positif sur la précision de la formule de Little pour estimer le temps moyen de séjour et le nombre moyen de clients.

Question bonus : Files M/M/n

Cas d'une file M/M/2

Considérons le cas où il y a n serveurs dans le système. Les lois d'arrivées et de départ seront de nouveau prises selon des lois exponentielles. Le modèle est alors noté M/M/n. Nous prendrons ici n = 2.

La simulation de la file d'attente se sera pas sensiblement différentes que la file M/M/1. La seule différence ici sera le fait que deux clients se font process en même temps. La première étape consiste à générer le vecteur des arrivées des clients de la meme manière pour la file M/M/1. Il faudra aussi générer les départs des deux premiers clients qui peuvent se calculer directement.

```
arrivees <- generate_arrivees(lambda, D)
departs <- c(arrivees[1] + rexp(1,mu) , arrivees[2] + rexp(1 ,mu))
```

La deuxième étape consiste à calculer les différents départs des clients. Plusieurs cas sont possibles:

- Si le client arrive et que au moins un des serveurs est libre, alors son traitement se fait automatiquement : `( arrivees\[i] + rexp(1,mu) )`
- Si les deux serveurs sont occupés, le client attend le premier serveur qui se libère, *i.e* le client dont le temps de départ est le plus petit : `( min(departs\[i-1],departs\[i-2]) + rexp(1,mu) )` On prend alors le max de ces deux instants

```
departs[i] <- max(arrivees[i],min(departs[i-1],departs[i-2])) + rexp(1,mu)
```

```
liste_filesMM2 <- generateFileMM2(duration = 12)
testLittleMM2 <- compare_nb_clients(liste_filesMM2)
```

On remarque directement ici que le nombre maximal de clients en attente est toujours inférieur à celui d'une file MM1. Cela est du au traitement simultané de deux clients en même temps.

On effectue alors la même étude que sur la file MM1, c'est à dire le calcul du nombre moyen de clients, ainsi que le temps moyen passé à attendre.

```
means <- generate_means(testLittleMM2)
average_wait_times <- generate_average_duration(liste_filesMM2)

resultsMM2 <- collect_results(means, average_wait_times)
# suppression de ces deux colonnes qui n'ont plus aucun sens since la formule de
Little ne s'applique plus
resultsMM2$LittleFormula <- NULL
resultsMM2$Ecart_Relatif <- NULL
```

Nous pouvons alors résumer les résultats dans le tableau suivant :

```
knitr::kable(resultsMM2, format = "markdown", caption = "File M/M/2 , D = 12")
```

Pour de faibles valeurs de λ, nous ne remarquons pas de différence significative. Cepedant, plus λ augmente, plus on voit que le nombre de client moyen, ainsi que le temps moyen d'attente diminue, ce qui est en accord avec notre intuition : *deux serveurs en même temps feront avancer la file deux fois plus vite*

PS : La fonction FileMMn a aussi été implémentée. Elle se trouve dans le fichier `files_d_attente.R`