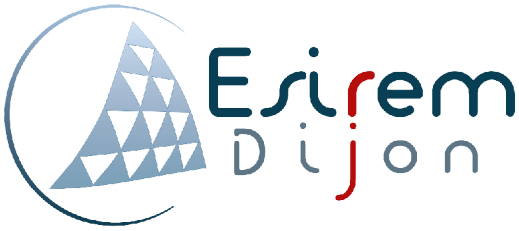
UBFC
UNIVERSITÉ
BOURGOGNE FRANCHE-COMTÉ



Collectionneur de vignettes

3ième année informatique et réseaux TD A
TALEB Nour Eddine
KIZIL Huseyin
ESIREM Dijon

Table des matières

Introduction

Dans le cadre de ce travail, nous explorons une activité populaire : la collection de vignettes. Cette activité consiste à rassembler un ensemble complet de vignettes uniques souvent distribuées aléatoirement dans des paquets de céréales par exemple. Notre étude se concentre sur le temps que peut mettre une personne pour compléter la collection. Combien de paquets de céréales une personne doit-elle acheter pour terminer sa collection de N vignettes différentes ?

Ce sujet nous intéresse pour son lien étroit avec les concepts mathématiques, en particulier la théorie des probabilités. En analysant les probabilités d'obtenir des vignettes déjà possédées ou nouvelles à chaque achat de paquet de céréales, en supposant que la personne achète 1 paquet par semaine, nous cherchons à établir des estimations théoriques pour le nombre moyen de paquets nécessaires pour achever la collection et donc la notion d'ésperances rentrera en jeu.

En outre, nous avons examiné des situations plus complexes, comme les échanges de doublons avec d'autres collectionneurs ou les offres spéciales des fabricants permettant l'acquisition de vignettes spécifiques en échange d'un certain nombre de vignettes (par exemple 10 vignettes contre 1 vignette spécifique). Il est clair que ces éléments vont influencer au niveau du temps de collecte des vignettes et c'est ce que nous allons démontrer durant notre étude.

L'objectif principal de notre étude est donc de développer un modèle mathématique solide pour estimer le temps moyen nécessaire à un collectionneur pour finaliser sa collection de vignettes. Nous analysons également diverses stratégies et conditions qui pourraient affecter ce temps.

Au fil de ce rapport, nous présentons notre approche mathématique, discutons des résultats théoriques obtenus à l'aide des lois de probabilités usuelles et analysons différentes situations pour mieux comprendre le problème du collectionneur de vignettes. Pour cela, nous avons réalisé un programme informatique en C++ et nous avons fait plusieurs simulations à l'aide de RStudio.

Les différents codes sont disponibles en annexe, à la fin du document.

Pour accéder aux codes sources sur GitHub, veuillez visiter : Lien GitHub vers le code source.

Modèle de base

2.1 Étude théorique

Lorsque l'on suppose que toutes les vignettes sont équiprobables, c'est-à-dire que chaque type de vignette a une probabilité de $\frac{1}{n}$ d'être obtenu lors de l'achat d'une boîte, on peut étudier la durée nécessaire pour compléter une collection de vignettes.

Soit T la variable aléatoire qui représente le nombre de paquets à acheter (et par extension, le nombre de semaines nécessaires) pour compléter la collection de vignettes.

L'obtention d'une nouvelle vignette dans un paquet k suit une loi géométrique de paramètre $p = \frac{n-(k-1)}{n}$, où n est le nombre total de vignettes à collecter. Ainsi, $T_{n,k}$, une variable aléatoire représentant le nombre de paquets à acheter pour obtenir une nouvelle vignette lorsque vous en avez déjà $k-1$ différentes, suit également une loi géométrique de paramètre p.

L'espérance de $T_{n,k}$ est donnée par : $E(T_{n,k}) = \frac{1}{p} = \frac{n}{n-(k-1)}$

Par conséquent, la variable aléatoire T qui représente le temps nécessaire pour compléter la collection entière est la somme de toutes les $T_{n,k}$ où k varie de 1 à n,

$$T = \sum_{k=1}^{n} T_{n,k}$$

En utilisant la linéarité de l'espérance, l'espérance de T est :

$$\begin{aligned}
E(T_n) &= \sum_{k=1}^{n} E(T_{n,k}) \\
&= \sum_{k=1}^{n} \frac{n}{n-(k-1)} \\
&= n\left(\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \ldots + \frac{1}{2} + \frac{1}{1}\right) \\
&= n\left(1 + \frac{1}{2} + \ldots + \frac{1}{n-2} + \frac{1}{n-1} + \frac{1}{n}\right) \\
&= n\sum_{k=1}^{n} \frac{1}{k}
\end{aligned}$$

En notant H_n le n-ième nombre harmonique,

$$E(T_n) = nH_n$$

En utilisant le développement asymptotique de H_n, on obtient :

$$E(T_n) = nH_n = n\ln(n) + \gamma n + \frac{1}{2} + o(1)$$

Où $\gamma \approx 0.577$ est la constante d'Euler-Mascheroni.[1]

2.2 Simulations

2.2.1 Simulation en C++

Notre solution consiste à simuler un certain nombre de paquets de céréales dans lesquels des vignettes sont présentes, puis à compter combien de paquets sont nécessaires pour remplir une collection séquentiellement. Après avoir répété cette procédure un certain nombre de fois, on calcule simplement la moyenne du nombre de paquets nécessaires pour remplir les albums. Pour ce faire, nous avons codé en langage C++ et nous avons réalisé les courbes à l'aide du logiciel Rstudio.

L'algorithme doit reproduire les conditions décrites dans le problème : ici, l'album comporte par exemple 500 vignettes, chaque paquet contient d'abord 1 vignette pour le cas simple, puis nous verrons avec 5 vignettes distinctes, et les vignettes seront considérées comme réparties uniformément parmi les paquets. Ses étapes sont les suivantes :

1. **Initialisation :** Un tableau `collected` représente les vignettes, initialisé à aucune vignette collectée.
2. **Boucle principale :** Une boucle `while` est utilisée pour simuler les semaines de collecte.
   - À chaque semaine, une nouvelle vignette est choisie aléatoirement en utilisant la fonction `rand() % collectionNumber`.
   - Si la vignette n'a pas encore été collectée (`collected[newVignette]` est `false`), elle est marquée comme collectée en mettant `collected[newVignette]` à `true`.
3. **Vérification de la fin :** La boucle continue jusqu'à ce que toutes les vignettes soient collectées.
4. **Libération de la mémoire :** La mémoire allouée dynamiquement pour le tableau est libérée.
5. **Retour :** Le nombre de semaines nécessaires pour collecter toutes les vignettes est renvoyé.

À noter qu'on fait un arrondi à l'entier inférieur vu qu'on utilise des variables de type "int" pour estimer le nombre de semaines.
Nous avons aussi créé des fonctions permettant de calculer les valeurs théoriques, que ce soit avec la somme harmonique ou avec le développement asymptotique, pour comparer les résultats trouvés.
En exécutant notre programme, voici ce qu'on obtient :
Dans un premier temps, nous entrons nos différentes valeurs afin d'obtenir nos résultats (voir figure 1 ci-dessous).

```
Entrez nombre de vignettes de votre collection (0 pour arreter programme)
Ici >  500
Entrez nombre de simulations (0 pour arreter programme)
Ici >  1000

-Simulation :
>> Il faut en moyenne 3409.1 semaines pour completer la collection de 500 vignettes sans echange.
-Valeur theorique :
>> Il faut en moyenne 3396.41 semaines pour completer la collection de 500 vignettes sans echange a l'aide de la formule theorique n*somme(1/k).
>> Il faut en moyenne 3396.41 semaines pour completer la collection de 500 vignettes sans echange a l'aide de l'approximation de la forumule theorique.
```

FIGURE 1 – Simulation et comparaison

Soit T la variable aléatoire représentant le nombre de paquets nécessaires pour compléter la collection. Nous nous intéressons à la valeur attendue de T, donc nous répétons simplement la procédure ci-dessus pour un grand nombre d'albums et prenons la moyenne arithmétique des valeurs finales du nombre de semaines retournées afin d'avoir des valeurs plus stables et précises. À titre d'exemple, nous avons exécuté

l'algorithme pour 1000 "albums" et avons obtenu 3409,1 comme estimation du nombre attendu de paquets nécessaires pour compléter un album avec 1 vignette dans un paquet. Il convient de souligner que, en tant qu'approximation stochastique, l'algorithme donne généralement une réponse différente à chaque exécution (la réponse 3410 mentionnée précédemment vient de l'arrondi à l'entier supérieur de 3409,1). En comparant cela à la valeur théorique de 3397 en prenant l'arrondi supérieur, on voit une petite différence.

2.2.2 courbe sous Rstudio

Nous avons réalisé une simulation du problème du collectionneur de vignettes en modélisant le processus d'achat de paquets de céréales. Nous avons suivi l'évolution du nombre de vignettes collectées au fil du temps, et les résultats ont été représentés graphiquement.

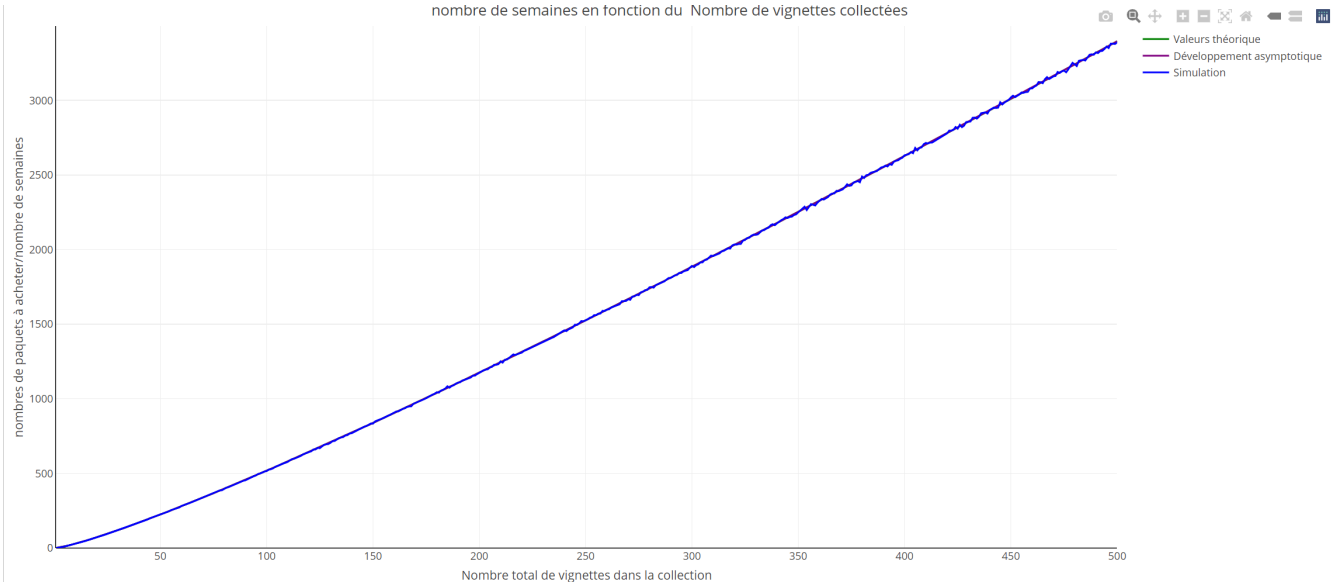


FIGURE 2 – Évolution du nombre de vignettes collectées au fil du temps

La Figure 2 illustre l'évolution du nombre de vignettes collectées en fonction du temps. À noter que dans cet exemple, il n'y a qu'une vignette dans le paquet de céréales. Nous pouvons tirer plusieurs observations et conclusions à partir de cette représentation graphique.

Tout d'abord, l'ascension initiale de la courbe indique une phase où de nombreuses vignettes uniques sont collectées rapidement au début de la période d'achat. Cela reflète la probabilité élevée d'obtenir des vignettes manquantes au début de la collection.

Ensuite, au fur et à mesure que le temps progresse, la courbe présente une pente moins prononcée, indiquant que la collecte de nouvelles vignettes devient plus lente. Cela pourrait correspondre à la phase où le collectionneur commence à accumuler des vignettes en double et trouve moins fréquemment de nouvelles vignettes.

Nous remarquons par ailleurs que la courbe est de la forme nH_n. Ces observations préliminaires nous inciteront à approfondir notre analyse, à examiner les stratégies utilisées et à comparer les résultats obtenus. Cette analyse détaillée sera présentée dans la section suivante.

2.3 Possibilité d'échange

2.3.1 Algorithme et code C++

Pour réaliser cette simulation, nous avons gardé le même algorithme (voir la section 2.2.1) en ajoutant la possibilité de faire des échanges. En effet, on peut obtenir une vignette manquante en échange de 10 doublons quelconques (voir la figure 3 ci-dessous).

```
49          int newVignette = rand() % collectionNumber;
50          if (!collected[newVignette]) {
51              collected[newVignette] = true;
52              //compter vignettes restantes
53              notCollectedCounter--;
54          } else
55              duplicatesVignette++;
56
57          //faire echange a la fin (si nombre de vignettes manquantes = doublons/10 on echange) et mettre fin
58          if (notCollectedCounter <= duplicatesVignette / 10) {
59              break;
60          }
```

FIGURE 3 – Code d'échange

Afin d'optimiser le code, nous avons laissé la possibilité d'échanger nos vignettes jusqu'à la fin, vu que plus on collecte, plus le nombre de semaines nécessaires (resp. nombre de paquets de céréales) devient important. Avec cette approche, on optimise au maximum notre simulation en réduisant ainsi le nombre de semaines.
Nous assurons cela en créant la variable "`notCollectedCounter`" qui sera initialiser par le nombre total de vignettes dans notre collection , et qu'on décrimente par 1 à chaque fois qu'on trouve une vignette manquante , sinon on incrémente la variable "`duplicatesVignette`" qui représente les vignettes doublons. On arrête la simulation si le nombres de vignettes manquantes est inférieures ou égales au nombres de vignettes doublons divisé par le prix d'échange qui est égal à 10 dans ce cas.

2.3.2 Comparaison entre cas normal et possibilité d'échanger 10 vignettes

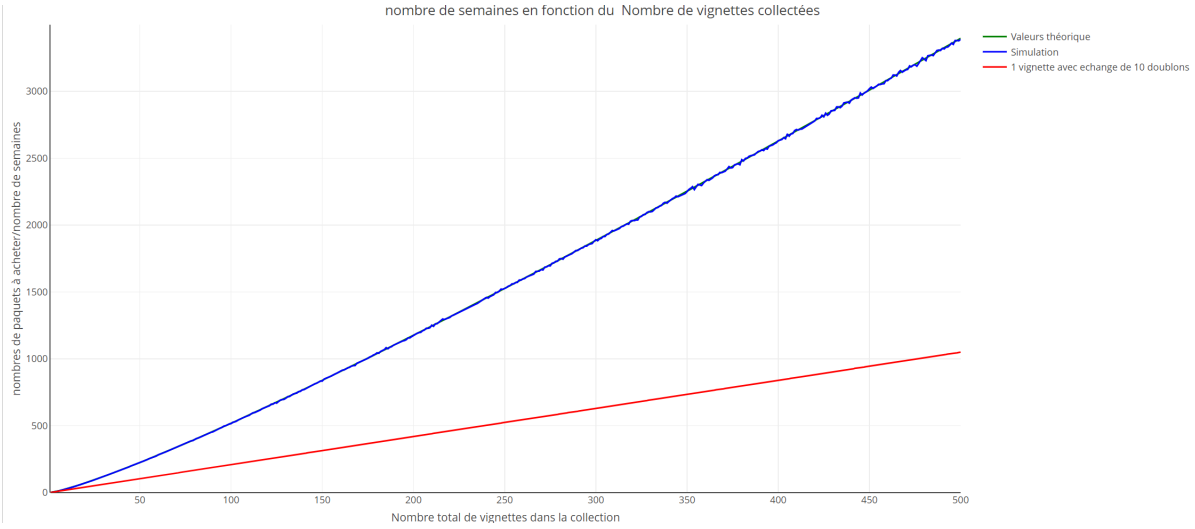


FIGURE 4 – Nombre de semaines en fonction du nombre de vignettes collectées

Le graphe de la figure 4 montre les courbes pour 1 vignette dans le paquet de céréales sans échange et avec échange.

Stratégie 1 : 1 Vignette dans le Paquet sans Échange
Temps de Collecte : 3500 semaines pour compléter la collection de 500 vignettes. Interprétation : Cette stratégie implique l'achat de paquets de céréales contenant une seule vignette, sans pratiquer d'échanges. Le temps de collecte élevé suggère que la complétion de la collection est ralentie par l'absence d'échanges.

Stratégie 2 : 1 Vignette dans le Paquet avec Échange de 10 Doublons
Temps de Collecte : 1070 semaines pour compléter la collection de 500 vignettes. Interprétation : Dans cette stratégie, chaque paquet de céréales contient une seule vignette, mais l'échange avec le fabricant nécessite 10 doublons. Le temps de collecte considérablement plus court indique que cette stratégie, malgré le coût en doublons, accélère significativement le processus de collecte.

Ce graphique souligne l'impact significatif de la stratégie d'échange avec le fabricant sur le temps de collecte. La stratégie avec échange de doublons semble être plus efficace dans ce contexte particulier, accélérant le processus de collecte malgré le coût en doublons. Cette conclusion doit être interprétée en tenant compte des objectifs spécifiques du collectionneur et de la gestion des doublons.

2.3.3 Influence du prix d'échange

Nous appelons ici le prix d'échange le nombre de vignettes nécessairex avant de pouvoir les échanger avec une nouvelle vignette que nous n'avons pas encore dans notre collection.

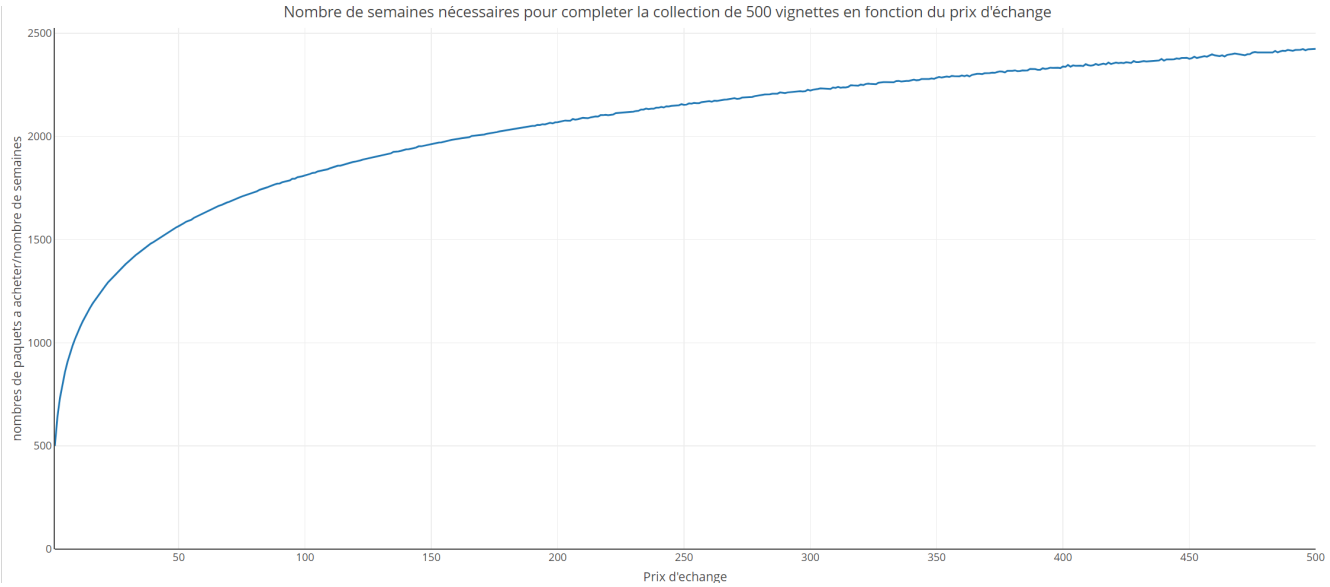


FIGURE 5 – Influence du prix d'échange

La collecte de vignettes peut être influencée par le prix d'échange comme le montre la courbe ci-dessus. En effet, on peut remarquer en particulier que jusqu'à 150 vignettes comme prix d'échange, la courbe augmente très rapidement en nombre de semaines puis l'augmentation est moins marquée par la suite. Cela est dû à l'augmentation du nombre de doublons nécessaires pour effectuer l'échange. Ainsi, plus le prix d'échange est élevé, plus le temps de collecte sera long par conséquent.

Généralisation : Plusieurs vignettes

3.1 Etude théorique et approximation

Dans ce chapitre, nous cherchons à trouver une formule approximative permettant d'estimer le nombre de semaines nécessaires pour le cas où l'on trouve un nombre de vignettes dans le paquet de céréales supérieur à 1. Et par la suite, nous ajoutons la possibilité d'effectuer des échanges.

3.1.1 Rappel

Dans le problème original du collectionneur de vignettes, chaque paquet a une seule vignette. Ainsi, le nombre de paquets T nécessaires pour compléter la collection est une somme de variables aléatoires géométriques avec une probabilité variable de trouver une nouvelle vignette. La valeur attendue de cette variable est simplement la somme des valeurs attendues de telles variables aléatoires géométriques :
$E(T) = n \sum_{k=1}^{n} \frac{1}{k}$.

La série à l'intérieur des parenthèses est la N-ième somme harmonique. Souvent, elle est approximée par $\ln(n)$. Cependant, une approximation différente est donnée par : $E(T) = n \sum_{k=1}^{n} \frac{1}{k} \approx n \ln(n) + \gamma n + \frac{1}{2}$,

où $\gamma \approx 0.57721$ est la constante d'Euler–Mascheroni.

3.1.2 Approximation

D'après l'article publié par rapport à ce sujet "The sticker collector's problem" [2], pour paquet qui contient 5 vignettes différentes, une approximation du nombre de semaines pour compléter la collection serait $E(T)/5$. Dans un premier temps la serie harmonique est souvent appriximer par ln(n), alors pour un nombre de vignettes n=649 on obtient $649 \cdot \ln(649)/5 \approx 840.51$.
Une autre approximation est donné par l'approximation de Euler-Mascheroni $(659 \ln(649) + \gamma \cdot 649 + \frac{1}{2})/5 \approx 915.53$.
Voici une comparaison sur l'exactitude des différentes methodes, par simulation de Monte Carlo, par l'approximation logarithmique , et par l'approximation de Euler-Mascheroni pour le cas ou le nombre total de vignettes dans la collection est 649 et le nombre de vignettes dans le paquet de céréales est 5.

n	exact	MC	log	E–M	error MC	error log	error E–M
1	4577.7	4577.6	4202.6	4577.7	0.0000	0.0819	0.0000
5	913.1	913.3	840.5	915.5	0.0002	0.0795	0.0027
10	455.0	455.5	420.3	457.8	0.0011	0.0764	0.0060
61	72.0	72.1	68.9	75.0	0.0016	0.0432	0.0421
62	70.8	70.8	67.8	73.8	0.0005	0.0426	0.0429
100	42.7	42.8	42.0	45.8	0.0024	0.0154	0.0725
500	5.3	5.3	8.4	9.2	0.0024	0.5858	0.7273
649	1.0	1.0	6.5	7.1	0.0000	5.4754	6.0534

FIGURE 6 – Tableau récapitulatif d'exactitude des différentes solutions.

On peut en déduire que l'approximation de Euler-Mascheroni est plus exacte que l'approximation logarithmique de la série harmonique.

3.2 Simulation

Pour la suite des simulations nous allons prendre en particulier le cas de deux vignettes dans le paquet céréales pour simuler le cas de vignettes multiples.

3.2.1 Algorithme et code C++

Nous avons garder le même principe du premier code en ajoutant la particularité ou on peut créer plusieurs vignettes d'un seul coup avant de passer à l'itération suivante. Il faut noter que les vignettes dans le paquets de céréales sont tous différentes. (Voir la figure 7 ci-dessous)

```
95      for (int i = 0; i < vignetteNumber; i++) {
96          int randomVignette;
97          bool isUnique; //Assurer que les vignettes trouvées dans le céréale soient différentes
98          do {
99              // Générer un nombre aléatoire entre 0 et collectionNumber - 1
100             randomVignette = rand() % collectionNumber;
101             // Vérifier si le nombre généré est déjà présent dans le tableau
102             isUnique = true;
103             for (int j = 0; j < i; j++) {
104                 if (newVignetteTable[j] == randomVignette) {
105                     isUnique = false;
106                     break;
107                 }
108             }
109         } while (!isUnique);
110         // Assigner le nombre aléatoire unique au tableau
111         newVignetteTable[i] = randomVignette;
```

FIGURE 7 – Code de vignettes multiples différentes

3.2.2 Courbe sous R

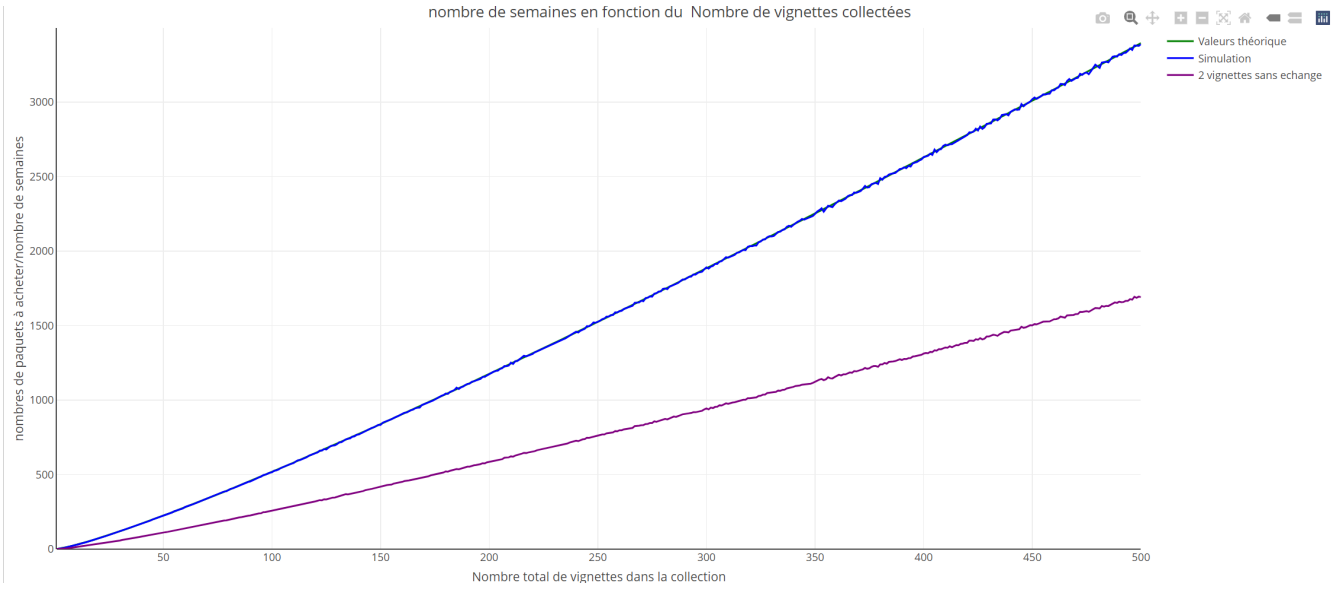


FIGURE 8 – Nombre de semaines en fonction du nombre de vignettes collectées.

La figure 8 est un graphe qui nous montre le cas de plusieurs vignettes dans la boite de céréale (qui est ici de 2 vignettes dans notre cas) sans échanges. On remarque que la courbe avec 2 vignettes dans la boite est en-dessous de la courbe bleue (courbe de base avec 1 vignette dans la boite). En effet, cela nous montre qu'on collecte toutes les vignettes plus rapidement avec 2 vignettes dans la boite. Ceci est normal car à chaque fois, nous avons 2 fois plus de chances d'avoir une nouvelle vignette car il y a 2 vignettes dans la boite cependant, au fil du temps, il est évident que nous allons collecter des doublons et donc il est nécéssaire d'étudier les autres cas dans les parties suivantes.

3.3 Plusieurs vignettes avec échange

Dans cette partie nous allons combiner la possibilité d'avoir 2 vignettes dans le paquets de céréales, et d'échanger 10 doublons quelconques contre une vignette manquante comme indiqué dans la section 2.3.1 .

3.3.1 Simulation et comparaison entre les différents cas

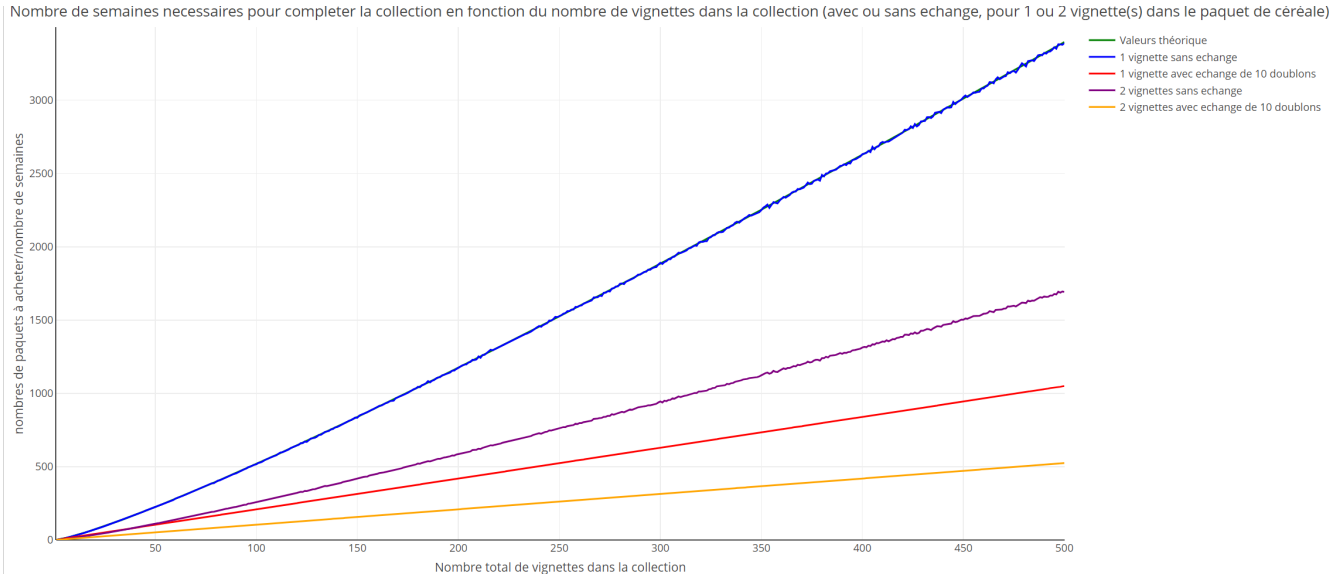


FIGURE 9 – Comparaison entre les différentes manières de collecte

Stratégie sans échange :
il est clair que la courbe bleue (1 vignette sans échange) est au-dessus de la violette (2 vignettes sans échange) car nous avons plus de chance de trouver une nouvelle vignette (non obtenues jusqu'à présent) avec 2 vignettes dans le paquet autrement dit, la collection peut être obtenue plus rapidement avec 2 vignettes dans le paquet.
Impact des échanges :
L'échange avec le fabricant offre une alternative pour obtenir des vignettes manquantes en échange d'un certain nombre de vignettes en double, généralement fixé à 10. Cette stratégie peut être particulièrement utile pour compléter rapidement la collection en ciblant des vignettes spécifiques. Par exemple, comme on peut l'observer sur la figure 9 , si nous prenons 500 comme nombre total de vignettes dans la collection, il faut 525 semaines pour la courbe jaune correspondant à 2 vignettes dans le paquet (avec échange), et 1692 semaines pour la courbe violet (sans échange). De même que pour la courbe rouge correspondant à 1 vignette dans le paquet (avec échange), il faut 1050 semaines pour compléter la collection alors qu'il faut 3389 semaines pour la courbe bleue (sans échange).

	Valeurs théoriques	1 vignette sans échange	1 vignette avec échange	2 vignette sans échange	2 vignette avec échange
Collection de 500 vignettes	33396.412	3389	1050	1692	525

TABLE 3.1 – Tableau récapitulatif des valeurs pour une collection de 500 vignettes

On peut voir que pour un N (axe des abcisses) assez grand, temps de collecte de vignettes change considérablement. Le temps est divisé presque par 2.5 si on échange nos doublons.
Si nous prenons N=100 (plus petit), on voit que pour la courbe 1 vignette sans échange, il faut 500 semaines alors que pour celle avec échange, il en faut 250. Donc le temps est quand-même divisé par 2.
Ainsi, on remarque que plus N est grand et plus la méthode d'échanges de doublons est efficace.

3.4 Determiner point de croisement

Dans cette partie, nous allons nous interesser à l'influence du prix d'échange sur le temps de collecte. En particulier, pour le cas où on a 1 vignette dans la boite de céréale avec possibilité d'échange (10 doublons quelconques contre 1), quel est le prix d'échange à prendre afin de pourvoir finir la collection dans la même temps si on a 2 vignettes dans la boite de céréale sans la possibilité d'échange....

3.4.1 Exemple avec 2 vignettes sans échange et 1 vignette avec echange

Remarque : afin de determiner le point de croisement nous allons ajouter comme conditions, en plus de la condition d'égalité, le cas ou la courbe en dessous (rouge) dépasse la courbe violette vu qu'on a une distribution ponctuelle des valeurs...

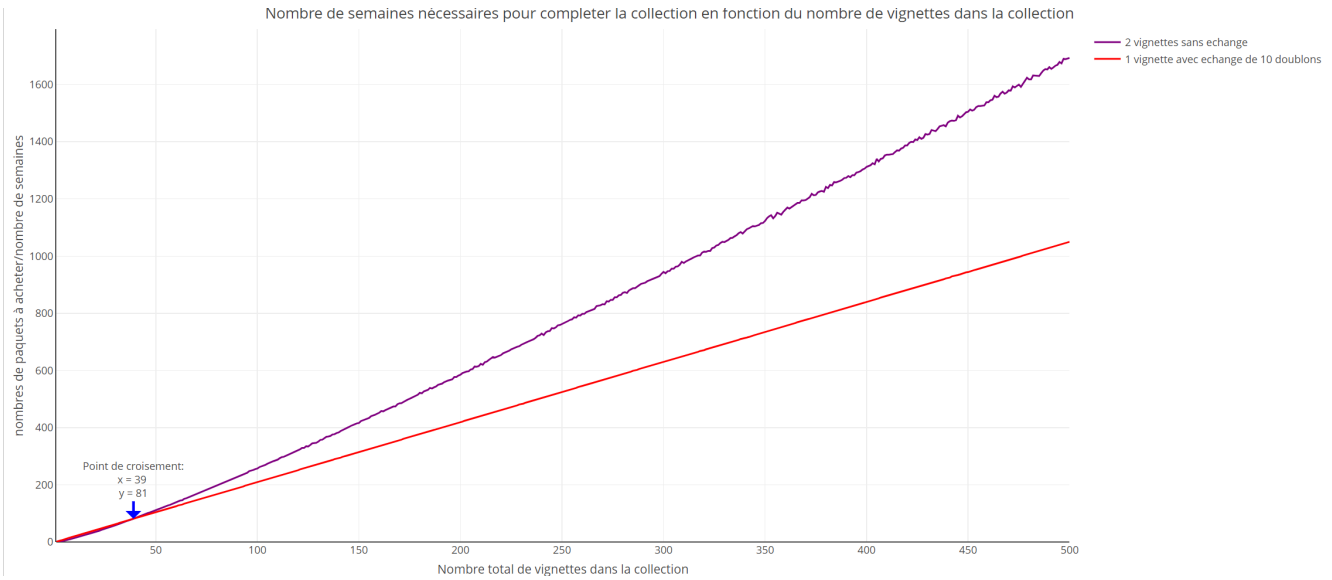


FIGURE 10 – Point de croisement

Point de croisement des courbes rouge et violet :
le point de croisement des coordonnées x=31 et y=81 où la courbe rouge est au-dessus de la courbe violet. On peut remarquer qu'avant ce point, la courbe rouge est au-dessus de la courbe violette. Ceci vient du fait que les premières vignettes sont plus facilement trouvable et vu que la courbe violette représente 2

vignettes dans le paquet et donc qu'il y a 2 fois plus de chance de trouver une nouvelle vignette, la courbe violet est plus rapide au début en terme de temps de collection. Mais, lorsque la collection augmente (et donc N augmente), on voit tout de suite le changement de rôle entre les 2 courbes. En effet, la violet passe au-dessus et diverge même s'il y a 2 fois plus de vignettes dans le paquet, si on considère qu'il n'est pas possible d'échanger, il est clair qu'on va prendre plus de temps à collectionner l'ensemble des vignettes alors qu'avec 1 vignette dans le paquet avec les doublons que nous obtenons, nous pouvons les échanger et obtenir une nouvelle vignette et donc la collection se termine plus vite.

3.4.2 Différents points de croissement

Pour les différentes valeurs du prix d'échange on observe que le point de croissement s'éloigne du 0 si on augmente le prix d'échange, et inversement il s'approche de 0 si on dimunue le prix d'échange. En outre, nous remarquons que lorsque l'on fait la moyenne des points de croisement, nous obtenons une droite linéaire croissante. Plus le prix d'échange est grand et plus le point de croisement se produira plus tard dans le sens où le croisement des deux courbes sera retardé. Ceci s'explique par le nombre de vignette très petit présent dans la boîte (qu'une seule) et donc si on augmente le prix d'échange, le temps va être allongé également pour la courbe rouge et donc le point de croisement sera retardé entre celle-ci et la courbe violet.

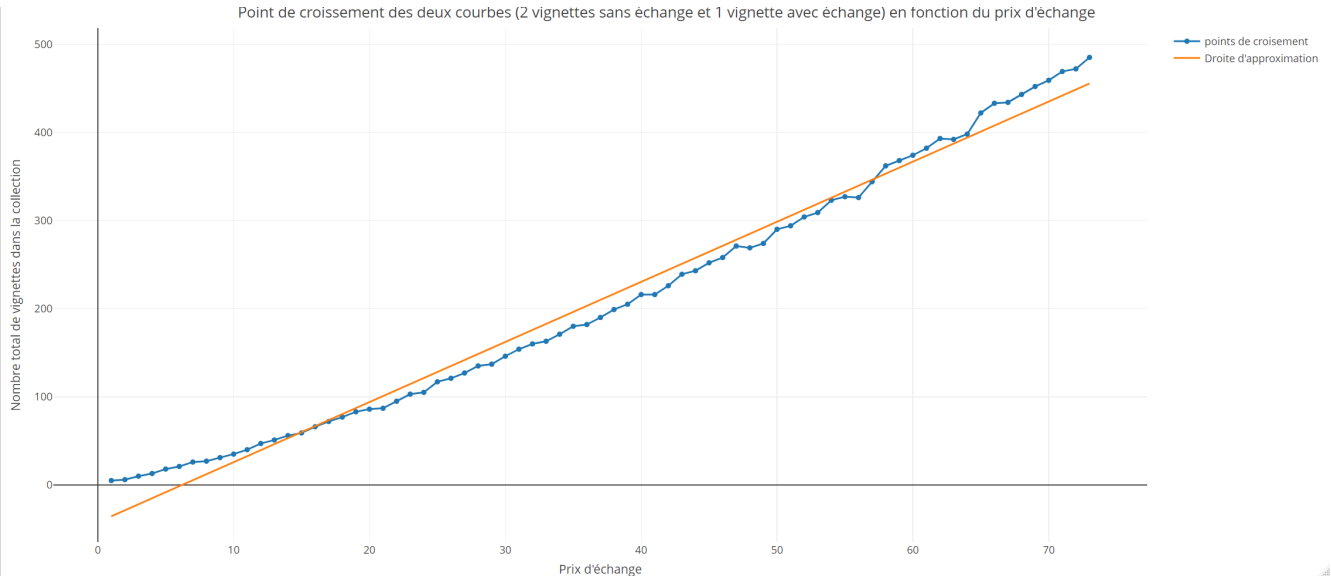


FIGURE 11 – Ensemble des points

3.5 Influence du prix d'echange sur le nombre de semaines

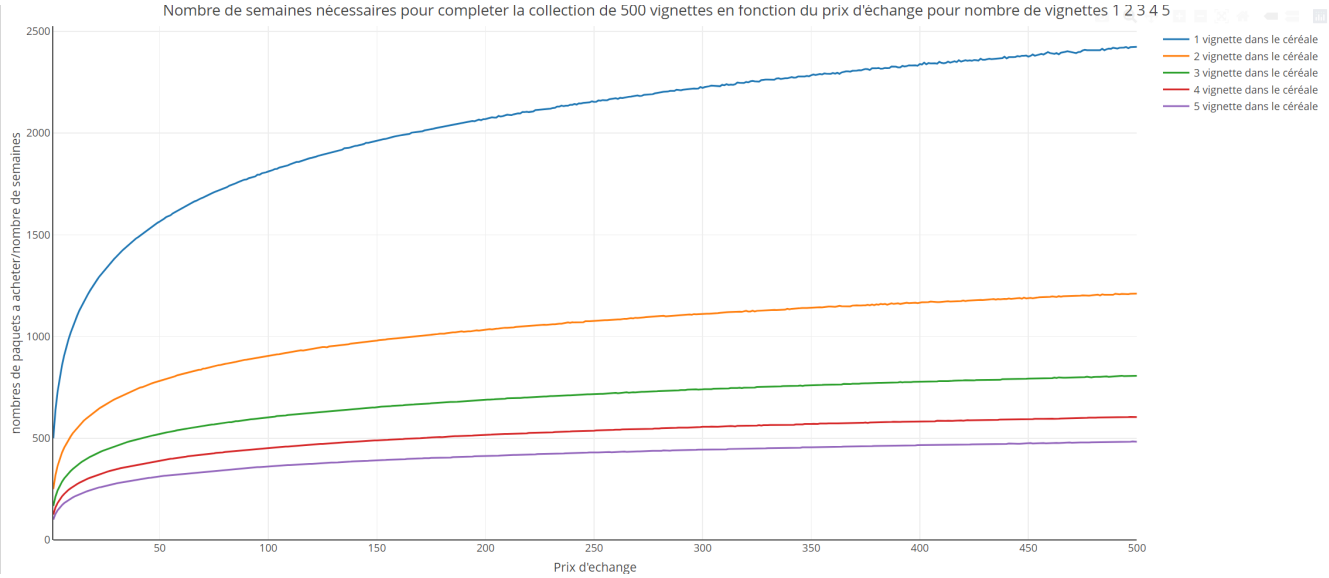


FIGURE 12 – Nombre de semaines en fonction du prix d'échange pour une collection de 500 vignettes

La collecte de vignettes peut être influencée par le prix d'échange.

Augmentation du Coût : Si le prix d'échange avec le fabricant (en termes de nombre de doublons requis) est élevé, le nombre de semaine pour obtenir la collection augmente également. Ainsi, cela peut décourager les collectionneurs d'opter pour cette stratégie, car l'accumulation rapide de doublons peut compenser les avantages de l'obtention de vignettes spécifiques. On voit d'ailleurs que les courbes augmentent assez vite au départ puis augmentent moins vite pour des valeurs de prix d'échange assez grandes.

Diminution du Coût : À l'inverse, un coût d'échange plus bas peut rendre cette stratégie plus attrayante, même si elle entraîne une accumulation plus rapide de doublons.

Conclusion

La présente étude a exploré de manière approfondie le problème du collectionneur de vignettes, en se concentrant sur différentes stratégies d'acquisition, d'échange et leurs impacts sur le temps nécessaire pour compléter la collection. Les résultats obtenus à partir des simulations ont fourni des observations précieuses sur l'efficacité relative de différentes approches.

La stratégie d'achat de paquets de céréales avec une vignette sans échange s'est avérée efficace pour minimiser le temps de collecte. Cette approche maximise les chances d'obtenir des vignettes manquantes tout en limitant l'accumulation de doublons.

L'échange avec le fabricant, bien que nécessitant un certain nombre de doublons par vignette, s'est révélé être une stratégie puissante pour accélérer la complétion de la collection. La gestion prudente des échanges est essentielle pour équilibrer les avantages de l'obtention de vignettes spécifiques avec les inconvénients de l'accumulation de doublons.

Les stratégies avec échange semblent généralement plus rapides dans le contexte de cette étude. Le choix entre les stratégies doit être guidé par la priorité accordée à la rapidité de la complétion par rapport à la gestion des doublons.

Cette étude a mis en lumière l'importance de considérer diverses stratégies dans le problème du collectionneur de vignettes.

Ainsi, le problème du collectionneur de vignettes est un exemple fascinant des défis probabilistes et d'optimisation qui peuvent émerger dans des situations du quotidien. Cette étude fournit une contribution significative à la compréhension de ces dynamiques et ouvre la porte à de futures recherches et explorations dans le domaine.

Table des figures

Liste des tableaux

Bibliographie

[1] Problème du collectionneur de vignettes. [Wikipédia] : `https://fr.wikipedia.org/wiki/ProblÃÎme_du_collectionneur_de_vignettes`.

[2] M. A. Diniz, D. Lopes, A. Polpo, and L. E. B. Salasar. The sticker collector's problem. *The College Mathematics Journal*, 47(4) :255–263, 2016. Published by the Mathematical Association of America.

Annexe : Codes de simulation

```h
#ifndef COUPON_COLLECTOR_S_SIMULATION_FUNCTIONS_H
#define COUPON_COLLECTOR_S_SIMULATION_FUNCTIONS_H

#include <iostream>
#include <cmath>

int simulateCollection(int collectionNumber);

int simulateCollectionWithExchange(int collectionNumber);

int simulateWithMultipleCollections(int collectionNumber, int vignetteNumber);

int simulateWithMultipleCollectionsWithExchange(int collectionNumber, int
    vignetteNumber);

double theoricalValueUsingHarmonicSerie(int collectionNumber);

double theoricalValueUsingAsymptoticDevelopment(int collectionNumber);

int simulateCollectionWithExchange(int collectionNumber, int Exchange);

int simulateWithMultipleCollectionsWithExchange(int collectionNumber, int
    vignetteNumber, int Exchange);

#endif //COUPON_COLLECTOR_S_SIMULATION_FUNCTIONS_H
```

```cpp
#include "Functions.h"

int simulateCollection(int collectionNumber) {

    bool *collected = new bool[collectionNumber]; // Allocation dynamique du tableau

    for (int i = 0; i < collectionNumber; i++) //initialiser tableau de vignettes
        collected[i] = false;

    int weeks = 0;
    bool allCollected = false;

    while (!allCollected) {
        weeks++;
        int newVignette = rand() % collectionNumber; //choisir un valeur alétoire qui
            represente une vignette
        if (!collected[newVignette])
            collected[newVignette] = true;

        //verifier si tout les vignettes sont collectées pour finir, si la nouvelle
            vignette est déja présente je verifie pas
        allCollected = true;
        for (int i = 0; i < collectionNumber; i++) {
            if (!collected[i]) {
                allCollected = false;
                break;
            }
        }
    }
    delete[] collected; // Libération de la mémoire allouée dynamiquement
    return weeks;
}
```

```
31
32 int simulateCollectionWithExchange(int collectionNumber) {
33
34     bool *collected = new bool[collectionNumber]; // Allocation dynamique du tableau
35
36     for (int i = 0; i < collectionNumber; i++)
37         collected[i] = false;
38
39     int weeks = 0;
40     int duplicatesVignette = 0;
41     bool allCollected = false;
42     int notCollectedCounter = collectionNumber;
43     while (!allCollected) {
44         weeks++;
45
46         int newVignette = rand() % collectionNumber;
47         if (!collected[newVignette]) {
48             collected[newVignette] = true;
49             //compter vignettes restantes
50             notCollectedCounter--;
51         } else
52             duplicatesVignette++;
53
54
55         //faire echange a la fin (si nombre de vignettes manquantes = doublons/10 on
               echange) et mettre fin
56         if (notCollectedCounter <= duplicatesVignette / 10) {
57             break;
58         }
59
60
61         //verifier si tout les vignettes collecter pour finir
62         allCollected = true;
63         for (int i = 0; i < collectionNumber; i++) {
64             if (!collected[i]) {
65                 allCollected = false;
66                 break;
67             }
68         }
69     }
70     delete[] collected; // Libération de la mémoire allouée dynamiquement
71     return weeks;
72 }
73
74 int simulateWithMultipleCollections(int collectionNumber, int vignetteNumber) {
75
76     bool *collected = new bool[collectionNumber]; // Allocation dynamique du tableau
77
78     for (int i = 0; i < collectionNumber; i++)
79         collected[i] = false;
80
81     int weeks = 0;
82     bool allCollected = false;
83
84     //exception si collectionNumber<vignetteNumber alors weeks = 1
85     if (collectionNumber < vignetteNumber) {
86         return 1;
87     }
88     int *newVignetteTable = new int[vignetteNumber]; // si on a plus d'une vignette
```

```
        dans la boite de céréales
89
90      while (!allCollected) {
91          weeks++;
92
93          for (int i = 0; i < vignetteNumber; i++) {
94              int randomVignette;
95              bool isUnique; //Assurer que les vignettes trouvées dans le céréale soient
                    différentes
96
97              do {
98                  // Générer un nombre aléatoire entre 0 et collectionNumber - 1
99                  randomVignette = rand() % collectionNumber;
100
101                 // Vérifier si le nombre généré est déjà présent dans le tableau
102                 isUnique = true;
103                 for (int j = 0; j < i; j++) {
104                     if (newVignetteTable[j] == randomVignette) {
105                         isUnique = false;
106                         break;
107                     }
108                 }
109             } while (!isUnique);
110
111             // Assigner le nombre aléatoire unique au tableau
112             newVignetteTable[i] = randomVignette;
113
114             // Marquer la vignette comme collectée
115             if (!collected[newVignetteTable[i]])
116                 collected[newVignetteTable[i]] = true;
117         }
118
119         //verifier si toutes les vignettes sont collectées
120         allCollected = true;
121         for (int i = 0; i < collectionNumber; i++) {
122             if (!collected[i]) {
123                 allCollected = false;
124                 break;
125             }
126         }
127     }
128     delete[] newVignetteTable;
129     delete[] collected; // Libération de la mémoire allouée dynamiquement
130     return weeks;
131 }
132
133 int simulateWithMultipleCollectionsWithExchange(int collectionNumber, int
        vignetteNumber) {
134
135     bool *collected = new bool[collectionNumber]; // Allocation dynamique du tableau
136
137     for (int i = 0; i < collectionNumber; i++)
138         collected[i] = false;
139
140     int weeks = 0;
141     int duplicatesVignette = 0;
142     bool allCollected = false;
143     int notCollectedCounter = collectionNumber;
144
```

```
145     if (collectionNumber < vignetteNumber) {
146         return 1;
147     }
148     int *newVignetteTable = new int[vignetteNumber]; //si on a plus d'une vignette dans
            la boite de céréales
149 
150     while (!allCollected) {
151         weeks++;
152 
153         for (int i = 0; i < vignetteNumber; i++) {
154             int randomVignette;
155             bool isUnique; //Assurer que les vignettes trouvées dans le céréale soient
                différentes
156 
157             do {
158                 // Générer un nombre aléatoire entre 0 et collectionNumber - 1
159                 randomVignette = rand() % collectionNumber;
160 
161                 // Vérifier si le nombre généré est déjà présent dans le tableau
162                 isUnique = true;
163                 for (int j = 0; j < i; j++) {
164                     if (newVignetteTable[j] == randomVignette) {
165                         isUnique = false;
166                         break;
167                     }
168                 }
169             } while (!isUnique);
170 
171             // Assigner le nombre aléatoire unique au tableau
172             newVignetteTable[i] = randomVignette;
173 
174             // Marquer la vignette comme collectée
175             if (!collected[newVignetteTable[i]]) {
176                 collected[newVignetteTable[i]] = true;
177                 notCollectedCounter--;
178             } else
179                 duplicatesVignette++;
180 
181         }
182         if (notCollectedCounter <= duplicatesVignette / 10) {
183             break;
184         }
185 
186         //verifier si toutes les vignettes sont collectées
187         allCollected = true;
188         for (int i = 0; i < collectionNumber; i++) {
189             if (!collected[i]) {
190                 allCollected = false;
191                 break;
192             }
193         }
194     }
195     delete[] newVignetteTable;
196     delete[] collected; // Libération de la mémoire allouée dynamiquement
197 
198     return weeks;
199 }
200 
201 //on utilisant série harmonique: N*sum(1/k) de 1 a N
```

```
202 double theoricalValueUsingHarmonicSerie(int collectionNumber) {
203     double somme = 0;
204     for (int i = 1; i <= collectionNumber; i++)
205         somme += 1.0 / i;
206     return collectionNumber * somme;
207 }
208
209 //En utilisant le développement asymptotique E(Tn)= n Hn =nln(n)+   n   +1/2+o(1) où
        0   .5772156649 est la constante d'Euler-Mascheroni.
210 double theoricalValueUsingAsymptoticDevelopment(int collectionNumber) {
211     return collectionNumber * (std::log(collectionNumber) + 0.57721) + 1. / 2;
212 }
213
214 int simulateCollectionWithExchange(int collectionNumber, int Exchange) {
215
216     bool *collected = new bool[collectionNumber]; // Allocation dynamique du tableau
217
218     for (int i = 0; i < collectionNumber; i++)
219         collected[i] = false;
220
221     int weeks = 0;
222     int duplicatesVignette = 0;
223     bool allCollected = false;
224     int notCollectedCounter = collectionNumber;
225     while (!allCollected) {
226         weeks++;
227
228         int newVignette = rand() % collectionNumber;
229         if (!collected[newVignette]) {
230             collected[newVignette] = true;
231             //compter vignettes restantes
232             notCollectedCounter--;
233         } else
234             duplicatesVignette++;
235
236
237         //faire echange a la fin (si nombre de vignettes manquantes = doublons/10 on
              echange) et mettre fin
238         if (notCollectedCounter <= duplicatesVignette / Exchange) {
239             break;
240         }
241
242
243         //verifier si tout les vignettes collecter pour finir
244         allCollected = true;
245         for (int i = 0; i < collectionNumber; i++) {
246             if (!collected[i]) {
247                 allCollected = false;
248                 break;
249             }
250         }
251     }
252     delete[] collected; // Libération de la mémoire allouée dynamiquement
253     return weeks;
254 }
255
256 int simulateWithMultipleCollectionsWithExchange(int collectionNumber, int
      vignetteNumber, int exchange) {
257
```

```
258     bool *collected = new bool[collectionNumber]; // Allocation dynamique du tableau
259
260     for (int i = 0; i < collectionNumber; i++)
261         collected[i] = false;
262
263     int weeks = 0;
264     int duplicatesVignette = 0;
265     bool allCollected = false;
266     int notCollectedCounter = collectionNumber;
267
268     if (collectionNumber < vignetteNumber) {
269         return 1;
270     }
271     int *newVignetteTable = new int[vignetteNumber]; //si on a plus d'une vignette dans
        la boite de céréales
272
273     while (!allCollected) {
274         weeks++;
275
276         for (int i = 0; i < vignetteNumber; i++) {
277             int randomVignette;
278             bool isUnique; //Assurer que les vignettes trouvées dans le céréal soient
                différentes
279
280             do {
281                 // Générer un nombre aléatoire entre 0 et collectionNumber - 1
282                 randomVignette = rand() % collectionNumber;
283
284                 // Vérifier si le nombre généré est déjà présent dans le tableau
285                 isUnique = true;
286                 for (int j = 0; j < i; j++) {
287                     if (newVignetteTable[j] == randomVignette) {
288                         isUnique = false;
289                         break;
290                     }
291                 }
292             } while (!isUnique);
293
294             // Assigner le nombre aléatoire unique au tableau
295             newVignetteTable[i] = randomVignette;
296
297             // Marquer la vignette comme collectée
298             if (!collected[newVignetteTable[i]]) {
299                 collected[newVignetteTable[i]] = true;
300                 notCollectedCounter--;
301             } else
302                 duplicatesVignette++;
303
304         }
305         if (notCollectedCounter <= duplicatesVignette / exchange) {
306             break;
307         }
308
309         //verifier si toutes les vignettes sont collectées
310         allCollected = true;
311         for (int i = 0; i < collectionNumber; i++) {
312             if (!collected[i]) {
313                 allCollected = false;
314                 break;
```

```
315                 }
316             }
317         }
318         delete[] newVignetteTable;
319         delete[] collected; // Libération de la mémoire allouée dynamiquement
320 
321         return weeks;
322 }
```

Pour accéder aux codes sources sur GitHub, veuillez visiter : Lien GitHub vers le code source.