TP #2.2 — Approche expérimentale du problème de collecteur de coupons

Le but de cet exercice est d'analyser expérimentalement le *problème du collecteur de coupons*. Ce problème (illustré par la Figure 1) consiste à collectionner *tous* les coupons d'un ensemble fini, où chaque coupon s'obtient typiquement par un tirage aléatoire lors de l'achat d'un paquet de *bubble gum* ou d'une boîte de céréales.

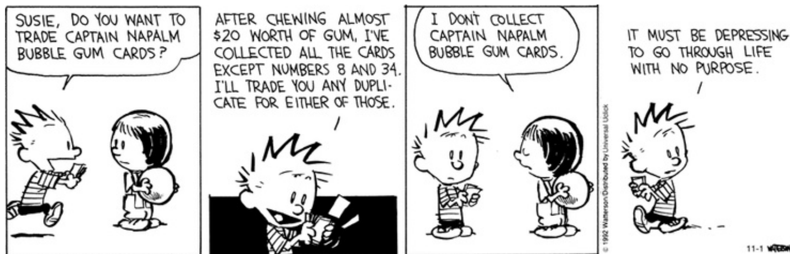

FIGURE 1 – Problème du collecteur de coupons, version *Calvin & Hobbes* (par Bill Watterson).

Un peu plus formellement, on considère un ensemble fini de coupons $\mathcal{C}$, un ensemble de coupons collectionnés $\mathcal{G}$ initialement vide, et tant que $\mathcal{G} \neq \mathcal{C}$ on tire uniformément un élément c de $\mathcal{C}$ que l'on ajoute à $\mathcal{G}$ s'il ne s'y trouve pas déjà. Les tirages se font « avec remise », ce qui veut dire que l'on peut tirer plusieurs fois le *même* coupon (ce qui nous embête plus qu'autre chose car cela n'augmente pas le nombre de coupons différents en notre possession, et les coupons en trop ne peuvent même pas servir de monnaie d'échange si personne d'autre n'en veut à la récré). L'objectif est alors de déterminer en fonction de la cardinalité n de $\mathcal{C}$ combien de tirages seront *espérés* (au sens mathématique) avant d'obtenir tous les coupons (autrement dit, combien de boîtes de céréales faudra-t'il s'attendre à acheter en moyenne).

1. Qu'en pensez vous ?

Vous analyserez peut-être ce problème l'année prochaine en cours de mathématiques, mais nous allons dès maintenant tenter de le résoudre (partiellement) de façon expérimentale en simulant un grand nombre de tirages, pour des valeurs de n variées.
Afin d'implémenter une telle simulation, nous allons tout d'abord devoir définir une façon de représenter les ensembles (éventuellement « incomplets ») de coupons. Pour cela, on assimile les n coupons de $\mathcal{C}$ aux entiers $[\![0, n-1]\!]$ entre 0 et $n-1$

(tous deux inclus), et l'on choisit de représenter un sous-ensemble par un tableau de booléens dont les seuls éléments à `true` sont aux indices de valeurs égales aux éléments présents dans le sous-ensemble. Ainsi, l'ensemble vide correspond à un tableau tout à `false` et l'ensemble « complet » de tous les coupons à un tableau tout à `true` ; on donne également les quelques exemples supplémentaires ci-dessous :

```
bool a[4] = {false, false, true, true};// représente {2, 3}
bool b[4] = {true, false, true, false};// représente {0, 2}
bool c[4] = {false, true, false, false};//  représente {1}
```

2. Quelle est la cardinalité de l'ensemble des parties (l'ensemble de tous les sous-ensembles) d'un ensemble de cardinalité finie $n > 0$?
3. Montrez que l'on peut ainsi représenter tous les sous-ensembles de n coupons par un tableau de type `bool [<B>]`, avec `<B>` une constante littérale de valeur *supérieure* ou égale à n.
4. Écrivez une fonction C de signature :

   ```
   void false_fill(size_t nc, bool coupons[nc])
   ```

   qui initialise son argument *coupons* tout à `false` (c'est à dire, à une représentation d'un ensemble vide de coupons).

   On « rappelle » pour cela que bien que les arguments tableaux soient passés par valeur (comme tous les autres arguments de fonctions en C), une modification des *éléments* d'un tel argument à l'intérieur d'une fonction *persiste au delà de l'exécution de la fonction*. N'hésitez pas à aller consulter les notes de cours ou à m'appeler en cas de question à ce sujet.
5. Écrivez une fonction C de signature :

   ```
   uint64_t collect(size_t nc, bool coupons[nc])
   ```

   qui simule *une fois* la collection de *nc* coupons, et compte et renvoie combien de boîtes de céréales ont dû être achetées (et l'on espère, consommées) avant de tous les obtenir. Cette fonction nécessite le tirage de nombres aléatoires, et vous êtes libres de les effectuer avec la méthode que vous souhaitez tant qu'elle produit de suffisamment bons résultats. Par défaut, on conseille d'utiliser :

   ```
   rand() % nc;
   ```

   pour tirer un nombre « aléatoire » dans l'intervalle $[\![0, nc[\![$. Cette approche (suffisante ici) est en fait très mauvaise pour de multiples raisons ; si vous souhaitez mieux faire, allez consulter la section relative à la génération d'aléa dans les notes de cours, mais n'y consacrez pas trop de temps.
6. Écrivez une fonction C de signature :

   ```
   void test_collect(void)
   ```

   qui définit un tableau de type `bool [100]` permettant de représenter les sous-ensembles des ensembles de 1 à 100 coupons, et appelle & affiche le résultat de *collect* pour toutes ces tailles d'ensembles.

   Un exemple de sortie pouvant être produite par cette fonction est :

```
test_collect: it took 1 cereal boxes to get my 1 coupons
test_collect: it took 3 cereal boxes to get my 2 coupons
test_collect: it took 6 cereal boxes to get my 3 coupons
test_collect: it took 13 cereal boxes to get my 4 coupons
test_collect: it took 10 cereal boxes to get my 5 coupons
test_collect: it took 22 cereal boxes to get my 6 coupons
[...]
test_collect: it took 444 cereal boxes to get my 95 coupons
test_collect: it took 488 cereal boxes to get my 96 coupons
test_collect: it took 401 cereal boxes to get my 97 coupons
test_collect: it took 413 cereal boxes to get my 98 coupons
test_collect: it took 528 cereal boxes to get my 99 coupons
test_collect: it took 558 cereal boxes to get my 100 coupons
```

Pour que les simulations effectuées soient statistiquement significatives (le seront-elles ?), on souhaite effectuer plusieurs appels à *collect* pour chaque nombre de coupons, et moyenner les résultats. Cette moyenne n'ayant *a priori* pas de raison d'être un entier, on utilisera un type `double` pour la calculer. Ce type (dont nous reparlerons plus en détails plus tard) permet de représenter des approximations des nombres dits « réels » ; la division entre `double`s se fait en utilisant le même opérateur de division « / » que pour les types entiers.

7. Écrivez une fonction C de signature :

   ```
   void stats(size_t nc, bool coupons[nc], uint64_t nexpe)
   ```

   qui effectue *nexpe* simulations du problème du collecteur de coupons à *nc* coupons et affiche la moyenne du nombre de boîtes de céréales nécessaires pour obtenir tous les coupons ainsi que le rapport entre cette moyenne et *nc*, et le nombre de simulations effectuées. Les messages affichés ressembleront donc à :

   ```
   stats: on average, it took 461.129213 cbs to get my 89 coupons
   (ratio 5.181227) [178 experiments]
   ```

   REMARQUES :

   — Le spécifieur de conversion pour l'affichage des nombres de type `double` avec *printf* est `%f`.

   — Afin de s'éviter tout tracas lié aux conversions entre types entiers et non entiers (parfois un peu subtiles), on conseille d'utiliser des `double`s pour toutes les variables « de calcul » définies dans la fonction, y compris celles pouvant être représentées exactement par des entiers.

8. Utilisez votre fonction *stats* pour calculer et afficher le nombre moyen de boîtes de céréales nécessaires à la collection de 1 à 1000 coupons. On conseille de réaliser (par exemple) $2n$ expériences pour un nombre n de coupons.
9. Pouvez-vous conjecturer une formule donnant le nombre espéré de boîtes en fonction du nombre n de coupons ? Réponse à la page suivante.

On peut montrer que l'espérance du nombre de boîtes nécessaires à la collection de n coupons vaut $n \times H_n$, où $H_n := \sum_{i=1}^{n} 1/i$ est le $n$ème *nombre harmonique*. La fin de cet exercice consiste à vérifier l'adéquation entre ce calcul théorique et nos expériences. Pour cela, il va être nécessaire de calculer les 1000 premières valeurs de H_n.

10. Estimez grossièrement le coût total (par exemple en nombre approché d'opérations arithmétiques) d'un calcul naïf des k premières valeurs de H_n, en fonction de k.

Bien que ce coût soit acceptable (il serait loin d'être dominant dans notre programme actuel), on peut l'améliorer nettement en exploitant le fait que : 1) H_{n+1} se calcule aisément en fonction de H_n ; 2) on peut stocker les valeurs de H_n dans un tableau ; 3) on a réellement envie de calculer toutes les k premières valeurs de H_n.

11. Écrivez une fonction C de signature :

```
void hn_fill(size_t hn, double h[hn])
```

qui écrit `0.` à l'indice 0 de h, et calcule (approximativement) et écrit H_i aux indices $0 < i < hn$.

Si « bien » écrite, le coût de cette fonction est proportionnel à hn : elle permet donc de calculer les hn premières valeurs de H_n pour un coût *total* de $\approx hn$. On dit alors que le coût *amorti* par valeur calculée est constant, ce qui est nettement mieux que l'approche naïve.

12. Modifiez votre fonction *stats* afin qu'elle accepte un argument supplémentaire *hnc* donnant une approximation de H_{nc}, qu'elle affiche alors en comparaison du rapport entre le nombre moyen de boîtes et le nombre de coupons. Les messages affichés ressembleront donc à :

```
stats: on avg, it took 1575.503891 cbs to get my 257 coupons
(ratio 6.130365, H257 ~ 6.128236) [514 exps]
```

13. Réalisez à nouveau les expériences de la Question 8 en affichant cette information supplémentaires. Que pensez-vous ici de l'adéquation entre théorie et simulation ?