Sorbonne Université **M1 Mathématiques**
2022–2023 **MU4M056 : Programmation en C++**

TP noté 2 : Urnes

Les *urnes* sont des systèmes dynamiques stochastiques simples qui ont une grande richesse de comportement. L'objectif de ce TP est d'implémenter en C++ différents types d'urnes, et d'observer leur dynamique.

Les fichiers fournis doivent conserver les mêmes noms et les fichiers rendus doivent être compilables sans erreur, quitte à commenter les parties de code qui ne marchent pas.

On apportera un soin particulier à l'indentation du code, à la *const correctness*, c'est à dire au fait par exemple d'indiquer explicitement `const` pour les méthodes ne devant pas changer l'objet sur lequel elles sont appelées, et à la minimisation des copies inutile des (gros) objets.

On utilisera autant que faire ce peut les algorithmes de la bibliothèque standard, tels que `std::accumulate` , `std::generate` , `std::copy` , `std::transform` , `std::fill` ,...

Remarque : dans le texte ci-dessous, le symbole **(T)** *indique un fichier à télécharger sur Moodle avant le début du partiel.*

1 Introduction mathématique et implémentation

1.1 Description mathématique

Une *urne* est un récipient qui dans notre contexte contiendra des *boules* de couleur. Le nombre total de *couleurs* possibles n est fixé. Les couleurs sont représentées par les entiers de 0 à $n-1$. À chaque instant, l'évolution du contenu de l'urne se fait par une *règle de mise à jour* : on tire une boule uniformément dans l'urne, et suivant la couleur i tirée on « ajoute » un nombre $c_j^{(i)}$ de boules de la couleur j dans l'urne, pour tout $j \in \{0, \dots, n-1\}$.

Nous allons considérer deux grands types d'urnes :

— Les urnes de Polya, pour lesquelles après avoir regardé la couleur de la boule tirée et l'avoir remise dans l'urne, on rajoute une boule de la même couleur. On a alors $c_j^{(j)} = 1$ et $c_j^{(i)} = 0$ pour $i \neq j$;

— Les urnes de Friedman, pour lesquelles la quantité de boules ajoutées de couleur *différente* de celle tirée est strictement positive. On a alors $c_j^{(i)} > 0$ si $i \neq j$.

1.2 Implémentation

On prend le parti ici de séparer l'urne à proprement parler qui sera un modèle de classe `Urne` , et la façon dont le nombre de boules de chaque couleur évolue, qui pourra prendre plusieurs formes, mais qui a une interface unique, et qui est représentée par le type générique `Update` , qu'on discute plus en détails dans la section 2.2.

Une urne sera représentée par le modèle de classe (*class template*) `Urne` avec un paramètre de type, `Entier` (qui pourra être `int` , `long` , `unsigned` , etc. suivant les applications) et un paramètre entier `n` représentant le nombre de couleurs. Chaque instanciation possède deux champs privés `contenu` et `contenu0` de type `std::array<Entier,n>` , représentant des tableaux de taille fixe. La case numéro j de `contenu0` (resp. `contenu`) donnera le nombre de boules de couleur j à l'instant initial (resp. à l'instant courant).

Les constructeurs seront *toujours* écrits de sorte que les tableaux `contenu` et `contenu0` soient initialisés *de la même façon*.

En plus des constructeurs, ce modèle de classe possède :

— une méthode `nb_boules()` qui renvoie le nombre total de boules dans l'urne ;

— une méthode `reset()` qui réinitialise l'urne dans son état initial ;

— une méthode `fraction` qui prend en paramètre un entier j (supposé entre 0 et `n-1`) et qui renvoie la proportion de boules de couleur j dans l'urne ;
— un modèle de méthode `maj` , qui prend en argument un objet de type générique `Update` , représentant la règle de mise à jour, et un générateur de nombre pseudo-aléatoires de type générique `RNG` . Cette méthode ne renvoie rien, mais modifie en conséquence le tableau `contenu` ;
— un modèle de méthode `maj_p_fois` qui applique un certain nombre de fois la mise à jour.

1.3 Quelques mots sur `std::array`

La structure de données `std::array` est définie dans l'entête `<array>` de la bibliothèque standard. Elle est très similaire à `std::vector` à bien des égards : on peut accéder aux éléments avec les crochets. Les deux ont des itérateurs de début et fin, comme les autres conteneurs de la STL.

Les grandes différences pertinentes ici sont les suivantes : d'abord, au lieu d'un seul type passé en paramètre avec les crochets `<>` , on en a un autre, entier, qui correspond à la taille. Contrairement aux vecteurs, les `std::array` ont une taille qui doit être déterminée à la compilation, et ne peut pas être changée en cours de programme. Un objet de la classe `std::array<int,5>` est un tableau d'entiers de taille 5. Il n'y a pas de constructeur explicite pour cette classe. Il ne sera en particulier pas possible d'utiliser une liste d'initialisation des membres de la classe `Urne` pour directement construire correctement `contenu` et `contenu0` . Il faudra écrire le code nécessaire dans le corps du constructeur.

Les `std::array` ont un constructeur par copie, et un opérateur d'affectation qui permet de recopier dans un tableau les valeurs d'un autre tableau de même taille.

2 Écriture de la classe modèle

Télécharger les fichiers initiaux depuis Moodle.

2.1 Constructeurs, premières méthodes et affichage

1. Dans le fichier `urne.hpp` , remplacer les `XXX` pour terminer la définition du modèle de classe `Urne` et écrire un constructeur par défaut qui initialise tous les éléments des tableaux `contenu0` et `contenu` à la valeur 1. On pourra utiliser l'algorithme `std::fill` qui prend trois arguments : un itérateur de début de domaine dans un conteneur, un itérateur de fin, et une valeur `x` , et remplit le domaine entre les deux itérateurs avec `x` .

2. Écrire la méthode `nb_boules` qui ne prend pas d'argument et qui renvoie le nombre total de boules dans le tableau `contenu` . On pourra utiliser l'algorithme `std::accumulate` qui a une version avec trois arguments :
— un itérateur de début de domaine dans un conteneur,
— un itérateur de fin
— une valeur initiale
et qui renvoie la valeur initiale à laquelle on a ajouté la somme de toutes les valeurs lues dans le domaine. Il est défini dans `numeric` .

3. Écrire la méthode `fraction` telle que `u.fraction(j)` renvoie la proportion de boules de couleur `j` , c'est-à-dire le nombre de boules de couleur `j` dans l'urne `u` divisé par le nombre total de boules.

4. Écrire la méthode `reset` qui restaure le tableau `contenu` à son état initial. Cette méthode ne prend pas d'arguments et ne renvoie rien.

5. Écrire un opérateur d'affichage `operator<<` permettant d'écrire dans un flux sortant les informations sur une urne. De préférence, on souhaite que cet opérateur, pour une instanciation donnée de la classe, soit ami uniquement avec cette instanciation (version *introvertie* d'amitié). La sortie attendue pour l'urne à 2 couleurs de type `int` construite par défaut est

```
2
1    1
```

6. Dans le fichier `test_urne.cpp` ***(T)***, écrire deux lignes pour tester le code des questions précédentes, pour déclarer une urne à 2 couleurs `urne2` de type `int`, construite par défaut, et l'afficher sur le terminal.

7. Modifier légèrement la définition du modèle de classe pour que si la valeur de `n` n'est pas précisé, c'est la valeur 2 qui est utilisée.

2.2 Procédures de mise à jour

Nous allons maintenant écrire le modèle de méthode `maj`. Précisons ce que l'on attend de ses deux arguments : `f` de type générique `Update`, et `G` de type générique `RNG`.

- le type générique `RNG` représente un générateur de nombres aléatoires. En particulier, on s'attend à ce que si `U` est une loi de probabilité définie dans la bibliothèque `<random>`, alors `U(G)` renvoie des réalisations de variables aléatoires indépendantes de loi `U`.
- le type générique `Update` représente n'importe quel « objet » `f` tel que la syntaxe suivante `f(j,v)` a un sens, avec `v` un `std::array` de taille n passé en référence (non constante) et `j` un entier entre 0 et $n-1$ (on supposera que cet encadrement est toujours vérifié au moment de l'appel pour éviter d'écrire de test dans les `f`). Suivant la valeur de `j`, `f` modifiera les valeurs de `v`, souvent en les incrémentant (en ajoutant des boules).

La méthode `maj` doit effectuer les tâches suivantes :

- définir une loi de probabilité `random_color` qui donne la couleur d'une boule tirée uniformément dans l'urne. Pour `random_color`, on pourra utiliser le modèle de classe `std::discrete_distribution<int>` de la STL. Son constructeur avec pour arguments l'itérateur de début et l'itérateur de fin d'un domaine de taille m de conteneur (dont les coefficients sont supposés positifs) fournit une loi sur les entiers de 0 à $m-1$ avec des probabilités proportionnelles aux valeurs lues dans le domaine. C'est ce qu'on veut ici avec comme domaine le conteneur `contenu` en entier.
- appeler `f` avec les paramètres `j` (donné par une réalisation de `random_color`), et `contenu`.

8. Déclarer dans le modèle de classe ce modèle de méthode, et écrire le code correspondant dans le fichier entête à l'extérieur de la classe.

9. Écrire un modèle de méthode `maj_p_fois` qui prend en argument une règle de mise à jour `f`, un entier `p`, et un générateur de nombres pseudo-aléatoires `g`, et qui applique `maj(f,g)` `p` fois.

3 Urnes de Polya

La règle de mise à jour pour une urne dite *de Polya* est qu'à chaque étape le nombre de boules de la couleur tirée augmente de 1.

3.1 Urnes de Polya à 2 couleurs

10. Définir dans la fonction `main` un générateur de nombres pseudo-aléatoires `g`, et une lambda fonction `updatePolya2` qui prend comme argument un entier j (supposé égal à 0 ou 1), et un `std::array<int,2>` passé par référence, et qui augmente de 1 la case j de ce tableau.

Un exercice d'application de la théorie des martingales nous dit que la proportion de boules de couleur 0 dans une urne de Polya initialisée avec deux boules de couleurs différentes converge p.s. vers une loi uniforme sur $[0, 1]$ lorsque le nombre de mises à jour tend vers l'infini. Nous allons vérifier numériquement dans les deux questions suivantes que la distribution de cette proportion au bout d'un grand nombre de mises à jour est proche, pour certains critères, de la loi uniforme.

11. Déclarer dans la fonction `main` un entier `size_prop` égal à 10000, et un vecteur `prop` de `double` de taille `size_prop`. Remplir chaque case du vecteur avec le résultat de `urne2.fraction(0)` après l'avoir remise à son état initial, et l'avoir fait évoluer 1000 fois. Le résultat aléatoire pour chaque case doit être indépendant des précédents.

12. Toujours dans la fonction `main`, trier ensuite le vecteur `prop` et afficher les *déciles* de la distribution empirique calculée à la question précédente, c'est à dire les cases d'indice `size_prop/10*j` du vecteur trié avec `j` $\in \{0, \dots, 9\}$. Vérifier que c'est proche de ce qu'on attend de la loi uniforme (c'est à dire `j` /10).

3.2 Urnes de Polya à plusieurs couleurs

13. Dans le fichier `regles.hpp` ***(T)***, écrire un modèle de fonction globale `updatePolyaN` qui prend comme arguments un entier (supposé entre 0 et $n-1$) et un `std::array<Entier,n>` passé en référence, et qui augmente de 1 la case j de ce tableau.

14. Dans la fonction `main`, déclarer `urne7`, urne de taille 7 à coefficients `long`, la mettre à jour mise à jour avec une version de `updatePolyaN`, et afficher son contenu.

4 Urnes de Friedman

Pour les urnes dites *de Friedman* que nous allons considérer, on supposera que la règle de mise à jour permet d'ajouter :

— q boules de la couleur tirée,

— r boules de chaque couleur non tirée.

La règle que nous écrirons sera spécifique pour les urnes de taille 2, et sera représentée par une classe `Friedman2`, dans le fichier `regles.hpp` Cette classe a deux champs entiers `q` et `r` privés, et un constructeur à deux entiers `q0` et `r0` qui initialisent par copie `q` et `r` respectivement.

15. Écrire l'opérateur d'évaluation `operator()` pour cette classe qui prend comme argument un entier `j` (0 ou 1) et un `std::array<int,2>` en référence. Cet opérateur effectue la mise à jour suivant la règle indiquée au début de cette section. Souvenez-vous où on doit écrire le code d'une méthode de classe sans *template*.

16. Dans la fonction `main`, déclarer un objet `updateF` de la classe `Friedman2` avec $q = 7$ et $r = 2$.

17. Reprendre la fin de la section sur les urnes de Polya à deux couleurs et adapter le code ici pour afficher les déciles de la distribution empirique de la fraction de boules de couleur 0. Conclure que c'est plausible que cette fraction tende p.s. vers 1/2. (C'est effectivement le cas, même si $q > r$!).