TD12 - Algorithmes probabilistes

Exercice 1

Le *mélange de Knuth* permet de mélanger, sur place, les éléments d'un tableau. On rappelle ici son principe : on parcourt le tableau de la gauche vers la droite et, pour chaque élément à l'indice i, on l'échange avec un élément situé à un indice tiré aléatoirement entre 0 et i *inclus*.

Question 1. Écrire en C une fonction `void knuth_shuffle(int a[], int n)` qui réalise cet algorithme pour un tableau d'entiers `a` de taille `n`. On rappelle qu'on tire un entier aléatoire entre 0 et `n` -1 avec `rand() % n`. On pourra définir une la fonction `swap` qui échange deux éléments d'un tableau.

Question 2. Montrer que le mélange de Knuth est un bon mélange au sens où la probabilité que l'élément initialement dans la case i se retrouve au final dans la case j est exactement $\frac{1}{n}$ où n est la taille du tableau.

Exercice 2

Supposons que l'on veuille choisir k valeurs parmi n, sous l'hypothèse $1 \leqslant k \leqslant n$. Pour fixer les idées, les valeurs sont données dans un tableau a de taille n et on souhaite renvoyer un tableau de taille k avec les valeurs sélectionnées. Il convient d'être précis quant à la spécification du problème.

- L'ordre dans le tableau renvoyé n'est pas significatif. En particulier, les éléments peuvent ne pas être ordonnés comme dans le tableau initial.
- Le tableau a peut contenir des doublons mais on considère pour autant tous les éléments de a comme distincts. Dit autrement, tout se passe comme si on sélectionnait k *indices* parmi n pour renvoyer ensuite les éléments de a situés à ces indices. Si par exemple $a = [1, 1, 2]$, alors on renvoie $[1, 1]$ une fois sur trois et $[1, 2]$ deux fois sur trois.

Pour $k = 1$, le problème est simple : il suffit de tirer un indice dans $[0, n[$. Pour $k \geqslant 2$, le problème est plus complexe. Après avoir choisi un premier élément, le choix devient plus délicat. Que faire si on retombe sur un élément déjà choisi ? L'ignorer et recommencer ? Cela va-t-il terminer dans un temps raisonnable ? Et quand bien même cela terminerait rapidement, obtient-on un tirage équitable ? Un algorithme simple et efficace résout ce problème. Il définit un tableau r de taille k qui contiendra le résultat final.

1. On initialise le tableau r avec les k premières valeurs de a, c'est-à-dire qu'on pose $r = [a_0, a_1, \dots, a_{k-1}]$.
2. Pour i de k à $n-1$, on tire au hasard un entier j entre 0 et i inclus. Si $j < k$, on remplace r_j par a_i.

Question 1. Écrire une fonction `sampling : int -> int array -> int array` qui implémente cet algorithme avec une complexité linéaire en la taille de son tableau argument.

Question 2. En utilisant l'invariant pour la boucle `for` suivant :

$$\textit{pour tout entier } j \in [\![0, i[\![, \mathbb{P}\left[a_j \textit{est sélectionné}\right] = k/i$$

montrer que tous les éléments du tableau a ont la même probabilité d'être sélectionnés.

Exercice 3

Écrire une fonction `random_element : 'a list -> 'a` qui renvoie un élément tiré au hasard dans une liste supposée non vide. Le tirage doit être équiprobable et la fonction doit effectuer *un unique parcours de la liste*. En particulier, calculer la longueur puis utiliser `List.nth` n'est pas une option.

Exercice 4

Soit n un entier naturel non nul et trois matrices A, B, C d'entiers, de tailles $n \times n$. On souhaite vérifier si le produit AB est égal à C.

Question 1.

□ **1.1.** Écrire un programme qui effectue le produit matriciel AB et qui teste si AB est égal à C.

□ **1.2.** En supposant les opérations arithmétiques élémentaires de complexité constante, quelle est la complexité du programme précédent ?

Question 2. Afin d'améliorer cette complexité, on adopte une *approche probabiliste* du problème. Pour vérifier si AB est égal à C, on adopte l'*algorithme de Freivalds*.

- Générer un vecteur aléatoire $X \in \{0, 1\}^n$.
- Calculer le vecteur $Z = A \times (BX) - CX$.

- Si Z est le vecteur nul, renvoyer `true` sinon renvoyer `false`.

On cherche à majorer la probabilité que Z soit nul, notée $\mathbb{P}[Z = 0]$.
□ **2.1.** Si $AB = C$, que vaut exactement $\mathbb{P}[Z = 0]$? Justifier votre réponse.
□ **2.2.** Dans la suite, on suppose que $AB \neq C$ et on pose $D = AB - C$. On désigne par d_{ij} les entiers de D. Justifier l'existence d'au moins un entier d_{ij} non nul.
□ **2.3.** On désigne par x_i les entiers de X. On pose $z_i = \sum_{k=1}^{n} d_{ik}x_k$ que l'on met sous la forme $z_i = d_{ij}x_j + y_i$.Que vaut la probabilité conditionnelle $\mathbb{P}[z_i = 0 | y_i = 0]$.
□ **2.4.** Majorer la probabilité conditionnelle $\mathbb{P}[z_i = 0 | y_i \neq 0]$.
□ **2.5.** En déduire que $\mathbb{P}[z_i = 0] \leqslant 1/2$.
□ **2.6.** Puis que $\mathbb{P}[Z = 0] \leqslant 1/2$
□ **2.7.** Conclure sur l'intérêt de cette approche probabiliste.
□ **2.8.** Écrire un programme qui met œuvre l'algorithme de Freivalds.

Exercice 5

Un *filtre de Bloom* est une structure de données qui réalise un ensemble et fournit deux opérations : ajouter un élément et tester la présence d'un élément. Cette dernière opération doit donner un résultat correct pour les éléments qui ont été ajoutés à l'ensemble mais elle peut donner un résultat incorrect pour les autres éléments. Plus précisément, lors du test de la présence d'un élément dans un ensemble, un filtre de Bloom permet de savoir :

- avec certitude l'absence d'un élément (il ne peut pas y avoir de faux négatif) ;
- avec une certaine probabilité la présence d'un élément (il peut y avoir des faux positifs).

La *taille* d'un filtre de Bloom est *fixe* et *indépendante du nombre d'éléments contenus*, ce qui en fait une structure très compacte. L'inconvénient est toutefois qu'il y a d'autant plus de faux positifs qu'il y a d'éléments dans la structure. Le principe du filtre est le même que pour le hachage.

Un filtre de Bloom utilise un tableau de m booléens et k fonctions de hachage $h_1, \dots, h_k$ qui envoient les éléments sur $0, \dots, m-1$. Quand on ajoute un élément x, on met à `true` les booléens aux indices $h_1(x), \dots, h_k(x)$. Quand on teste la présence de x, on renvoie `true` si et seulement si *tous* les booléens aux indices $h_1(x), \dots, h_k(x)$ sont à `true`.

Question 1. Proposer une implémentation en C d'un filtre de Bloom pour des chaînes de caractères, où les paramètres k et m sont passés en arguments au constructeur. Pour la fonction de hachage h_i, on pourra s'inspirer de la fonction ci-dessous où la constante 31 sera remplacée par un entier tiré au hasard au moment de la construction.

```
int hash(char *k) {
  int h = 0;
  char c;
  while ((c = *k++) != 0)
    h = 31 * h + c;
  return h;
}
```

Question 2. Tester empiriquement l'efficacité d'un tel filtre, pour différentes valeurs des paramètres k et m. Par exemple, ajouter tous les mots d'un dictionnaire dans un filtre, puis, pour chaque mot w du dictionnaire, tester la présence d'un mot wc où c est un caractère qui n'apparaît dans aucun mot (par exemple un caractère non alphabétique comme `'\n'`). Compter les faux positifs et commenter le résultat.