TP #4 — Détection de cycles

Le thème de ce TP est la notion de cycle dans une permutation, ou une fonction finie en général.
Le premier exercice vise à implémenter un algorithme de génération de permutation uniformément aléatoire, puis à déterminer les cycles d'une permutation, et d'utiliser ces deux fonctions pour évaluer expérimentalement certaines propriétés statistiques sur les cycles des permutations uniformes.
Le second exercice demande d'implémenter un algorithme de détection de cycle dans le *graphe fonctionnel* d'une fonction finie quelconque, puis à l'appliquer à la recherche de collision dans une telle fonction.

Exercice 1. *Cycles de permutations*
On décrit l'algorithme suivant pour tirer uniformément au hasard une permutation des entiers $0, \ldots, n-1$ (c'est à dire une fonction bijective $\pi : [\![n]\!] \to [\![n]\!]$) :

- On initialise un tableau T de n entiers où chaque élément est égal à son index (commençant à 0) ; autrement dit, T est tel que `T[i] = i` pour i entier de 0 à $n-1$
- Pour i de 0 à $n-2$ (tous deux inclus) :
  - On tire uniformément un nombre j dans $[\![i, n-1]\!]$
  - On échange `T[i]` avec `T[j]` (on remarque que l'on peut avoir $i = j$; dans ce cas cet échange ne fait rien)
- On renvoie le tableau T ainsi obtenu

On prétend que si les tirages des indices sont uniformes, alors cet algorithme tire bien une permutation uniformément au hasard (c'est à dire que pour toute permutation, la probabilité qu'elle soit produite par l'algorithme est $1/n!$).

1. Écrivez une fonction Python `shuffle(T)` qui implémente cet algorithme en utilisant une `list` pour représenter `T`.
   On rappelle l'existence d'une fonction `randint` dans le module `random`. Faites néanmoins attention aux spécifications de cette fonction.
2. Testez votre fonction.
3. *(Optionnel.)* Prouvez que cet algorithme génère bien une permutation uniforme.

Un *cycle d'ordre k* d'une permutation π est une suite de k valeurs $c_0, \ldots, c_{k-1}$ telles que pour $0 \leq i < k$ l'on a $\pi(c_i) = c_{(i+1) \bmod k}$ (où $x \bmod y$ désigne l'unique reste positif de la division entière de x par y).

4. Écrivez une fonction Python `cycles(P)` qui prend en entrée une `list` représentant une permutation (dans le même sens que précédemment) et renvoie un dictionnaire contenant des couples (clefs, valeurs) de la forme `x : (x, c)` où `x` est un certain entier $\in [\![0, n-1]\!]$ représentant le cycle de `P` le contenant, et `c` l'ordre de ce cycle.
Attention : chaque cycle ne doit être « comptabilisé » qu'une fois : la somme des `c`'s pour toutes les entrées du dictionnaire doit valoir n. Cependant, les représentants des cycles peuvent être quelconques.
Par exemple, soit `P` représentée par :

```
[7, 0, 2, 8, 9, 3, 4, 5, 1, 6]
```

`cycles(P)` peut indifféremment renvoyer :

```
{0: (0, 6), 2: (2, 1), 4: (4, 3)}
```

ou

```
{3: (3, 6), 2: (2, 1), 6: (6, 3)}
```

Conseils : il peut être utile d'utiliser un second dictionnaire dans votre fonction pour déterminer efficacement si une valeur a déjà été comptabilisée dans un certain cycle.

5. Écrivez une fonction Python `statFP` qui utilise vos fonctions `shuffle` et `cycles` pour vérifier expérimentalement que le nombre espéré de *points fixes* (de cycles d'ordre 1) d'une permutation tirée uniformément au hasard est égal à 1.
6. Testez votre fonction avec des permutations de tailles variées.
7. *(Optionnel.)* Prouvez la propriété ci-dessus (indice : utilisez la linéarité de l'espérance avec des variables aléatoires bien choisies).
8. *(Optionnel.)* Montrez qu'il existe $(n-1)!$ permutations de n éléments qui sont *cycliques*, c'est à dire formées d'un unique cycle d'ordre n.
9. Écrivez une fonction Python `statCyclic` qui évalue expérimentalement la propriété ci-dessus.
10. Testez votre fonction avec des permutations de tailles variées.
11. Écrivez une fonction Python `statLarge` qui constate expérimentalement le fait que la probabilité qu'une permutation uniforme possède un cycle contenant plus de la moitié des éléments (c'est à dire d'ordre supérieur à $n/2$ pour une permutation de n éléments) est *constante* en fonction de n (et donc en fait de toute autre quantité).
12. Testez votre fonction avec des permutations de tailles variées.

Exercice 2. *Détection de cycle et collisions* ⋆⋆

Soit une fonction $F : [\![n]\!] \to [\![n]\!]$, on définit son *graphe fonctionnel* comme un ensemble de n *sommets* $0, \ldots, n-1$ reliés par des *arcs*, tels qu'il existe un arc $i \to j$ ssi. $F(i) = j$. On dit d'un tel arc qu'il est *sortant* pour i et *entrant* pour j. F étant une fonction, un sommet ne peut avoir qu'au plus un arc sortant, mais il peut avoir plusieurs arcs entrants.

On définit un *cycle* de F de la même façon que pour une permutation. F étant de domaine finie, elle admet au moins un cycle.

On définit une *collision* (non triviale) pour F comme une paire $i, j \neq i \in [\![0, n-1]\!]^2$ telle que $F(i) = F(j)$. F possède (au moins) une collisions ssi. elle n'est pas une permutation.

On peut remarquer que F possède une collision ssi. elle possède au moins un point qui n'appartient pas à un cycle.

Si F possède une collision, celle-ci peut s'exprimer en terme du graphe fonctionnel comme un sommet possédant (au moins) deux arcs entrants. De façon plus spécifique, F contient alors au moins un sommet *appartenant à un cycle* (possiblement d'ordre 1) possédant au moins deux arcs entrants, dont l'un est celui de son prédécesseur dans le cycle.

En reformulant, si F possède une collision, alors elle contient un cycle $x_0, \ldots, x_{k-1}$ tel qu'il existe x_i dans ce cycle avec :

— $F(x_{(i-1) \bmod k}) = x_i$
— un certain y (ne faisant pas partie du cycle) tel que $F(y) = x_i$

La forme alors obtenue en considérant le cycle et la « queue » formée par y est souvent comparée à celle de la lettre ρ.

De tout ce qui précède, on peut déduire une stratégie pour chercher une collision dans une fonction non bijective : on choisit un point de départ x_0, on itère F sur x_0 (autrement dit, on suit l'unique chemin du graphe fonctionnel défini par l'arc sortant de x_0, l'arc sortant de $F(x_0)$ etc.) et :

— si l'on atteint à nouveau x_0 la collision trouvée est triviale et l'on recommence avec un autre point de départ (ce cas est peu probable pour une fonction uniforme, qui est celui qui nous intéressera par la suite)
— si l'on est en mesure de détecter que ce chemin possède un cycle qui ne contient **pas** x_0, alors puisque nous avons abouti à ce cycle en partant de x_0, il *existe* un sommet du cycle possédant un arc entrant depuis un sommet qui *ne se trouve pas sur le cycle* mais qui est un *successeur* de x_0. Soit k l'ordre du cycle, on a alors $F^o(x_0) = F^{k+o}(x_0)$ (où $F^i(x)$ dénote l'application de F composée i fois avec elle même, appliquée à x) pour un certain entier $o > 0$ (le cas $o = 0$ correspond au cas précédent où x_0 appartient à un cycle et où la collision trouvée est triviale), ce qui constitue une collision non triviale.

L'intérêt de cette stratégie est que l'on dispose d'algorithmes efficaces pour *détecter* un cycle (et calculer son ordre) dans le graphe fonctionnel d'une fonction F quelconque, et notamment d'algorithmes utilisant très peu de mémoire. Puisque

les autres étapes de la stratégie ne nécessitent pas non plus de mémoire, cela implique que l'on peut trouver une collision d'une fonction qui en possède une (ou déterminer qu'elle n'en possède pas) en *stockant* seulement un **très petit nombre** de ses évaluations (ce qui n'est pas *a priori* évident).
On propose d'implémenter une telle stratégie en utilisant l'algorithme suivant pour détecter la présence d'un cycle :

- on fixe un point de départ `s` (par exemple `0`)
- on initialise une structure de donnée de *pile* à la pile vide
- tant qu'un cycle n'a pas été détecté :
  - on calcule `ns = F(s)`
  - on retire de la pile tous les éléments s'y trouvant éventuellement qui sont strictement plus grands que `ns`
  - si l'élément au sommet de la pile est maintenant égal à `ns`, on a détecté un cycle qui contient `ns`
  - sinon on ajoute `ns` en sommet de pile, on affecte `s = ns`, et l'on recommence

On peut remarquer que cet algorithme peut en principe utiliser beaucoup de mémoire, typiquement si les sommets du graphe fonctionnel sont visités par valeur croissante (ce que l'on ne peut pas éviter *a priori*). Cependant, si appliqué à une fonction dont toutes les images ont été tirées uniformément et indépendamment au hasard, le nombre *moyen* d'éléments stockés dans la pile restera « petit ». Il est aussi possible de garantir un bon comportement *espéré* pour n'importe quelle fonction en conjuguant celle-ci avec une permutation uniforme, ce que nous nous abstiendront de faire (ou d'analyser) ici.

1. Écrivez une fonction Python `genF(n)` qui crée et renvoie une `list` de n entiers tirés uniformément et indépendamment au hasard entre 0 et $n-1$ (tous deux inclus).
2. Testez votre fonction.
3. Écrivez une fonction Python `cycleFind(F)` qui prend en entrée une représentation de fonction telle que renvoyée par `genF` et qui implémente l'algorithme de détection de cycle ci-dessus. Cette fonction devra renvoyer un point x appartenant à un cycle, le nombre d'itération de la boucle qui a été effectué avant de détecter le cycle, et l'ordre du cycle.
4. Prouvez la terminaison et la correction de votre fonction.
5. Testez votre fonction.
6. Écrivez une fonction Python `colFind(F)` qui implémente la stratégie de recherche de collision décrite ci-dessus, en utilisant votre fonction `cycleFind` pour la détection de cycle.
7. Testez votre fonction.
8. Constatez une conséquence du « paradoxe des anniversaires » : pour une fonction F uniformément aléatoire de domaine $[\![0, n-1]\!]$, vous pouvez trouver

une collision en évaluant $\approx \sqrt{n}$ fois F.