Université de Bordeaux
L2 Informatique/Maths-Informatique

Probabilités, Statistiques, Combinatoire : TM2

Combinatoire : génération exhaustive (2)

Cette feuille d'exercices constitue la suite de la feuille TM1 ; elle n'a d'intérêt que si vous avez déjà fait au moins les questions 1 à 3 de l'exercice 1 de cette feuille.

Vous pouvez télécharger depuis la page du cours le fichier `gen.py`, qui contient plusieurs versions des fonctions demandées dans le TM1.

5.1 Permutations

Exercice 5.1
Permutations

Reprenons l'exercice sur la génération des permutations de $[[1, n]]$.

1. Écrire une fonction `SequencesSousDiagonales(n)`, qui retourne la liste de toutes les *séquences sous-diagonales* de longueur n. Une séquence sous-diagonale de longueur n est une séquence $(s_1, s_2, \ldots, s_n)$ d'entiers, telle que pour chaque $1 \leq i \leq n$, on ait $1 \leq s_i \leq i$ (le nom vient du fait que si on "dessine" la séquence en mettant un point de coordonnées (i, s_i) pour chaque i, tous les points sont compris entre l'axe horizontal et la diagonale).

   Indication : pour obtenir les séquences sous-diagonales de longueur $n \geq 1$, on peut procéder de la manière suivante : prendre chaque séquence sous-diagonale de longueur $n-1$, et compléter chaque séquence de n manières différentes, en ajoutant chacun des entiers de 1 à n. (Ceci mène naturellement à une fonction récursive ; il faut bien entendu traiter correctement un cas de base)

2. Le nombre de séquences sous-diagonales de longueur n est naturellement $n!$, comme celui des permutations de $[[1, n]]$. Écrire une fonction `SeqSousDiagVersPerm(s)`, qui réalise une bijection entre l'ensemble des séquences sous-diagonales de longueur n et celui des permutations de $[[1, n]]$; *i.e.* votre fonction prendra en entrée une séquence sous-diagonale et retournera (de manière bijective) une permutation.

   Indication : la description d'une telle bijection faisait l'objet de l'exercice 7 de la feuille TD1. Une possibilité consiste à partir de la permutation $t = (1, 2, \ldots, n)$, et à parcourir la séquence sous-diagonale s élément par élément ; lorsqu'on lit le i-ème élément s_i, on échange dans t le i-ème élément avec le (s_i)-ème. Attention aux indices qui commencent à 0 en `Python`.

3. Un **point fixe** d'une permutation est un élément de l'ensemble de départ qui est égal à son image : $x = f(x)$. Nos permutations étant représentées par la séquence des images de $1, 2, \ldots n$, un point fixe devient un entier i tel que $s_i = i$.

   Écrire une fonction `ComptePointsFixes(s)` qui reçoit une permutation et retourne son nombre de points fixes.

   Utiliser vos fonctions pour expérimenter et proposer une réponse générale à la question suivante : en fonction de n, combien y a-t-il au total de points fixes parmi les $n!$ permutations de $[[1, n]]$?

Exemple : pour $n = 2$, il y a 2 permutations, $(1, 2)$ et $(2, 1)$. La première a 2 points fixes, la seconde en a 0 ; donc pour $n = 2$, la réponse est $2 + 0 = 2$.

Vous pouvez aussi écrire des fonctions pour déterminer combien, parmi les $n!$ permutations de $[[1, n]]$, il y en a qui ont k points fixes (en faisant varier k de 0 à n).

5.2 Itérateurs

Jusqu'à présent, vous avez écrit vos fonctions de telle sorte qu'elles retournent des listes d'objets. Cela engendre des listes très longues : il y a par exemple 16384 séquences binaires de longueur 16, et 3628800 permutations de $[[1, 10]]$; cela peut provoquer des problèmes de mémoire assez facilement.

Or, le plus souvent, on n'a pas besoin d'avoir simultanément en mémoire tous les objets d'une classe : on se contenterait de les passer en revue, par exemple pour les compter, ou pour mesurer certaines statistiques (comme, par exemple, le nombre de points fixes dans les permutations).

Le langage `Python` fournit un mécanisme pratique pour ce genre de chose : il est possible de produire des *itérateurs* (ou *générateurs*) au lieu de listes. La construction `range(n)` de `Python` ne fournit pas une liste, mais un itérateur sur l'ensemble des valeurs de 0 à $n - 1$; et le `for` de `Python` fonctionne par défaut avec des itérateurs (si on lui donne une liste, il construit automatiquement un itérateur sur les éléments de la liste).

Un itérateur peut s'écrire exactement comme on écrirait une fonction, à ceci près qu'on y utilise l'instruction `yield` : chaque utilisation de `yield` produit une valeur que fournira l'itérateur, un peu comme si on mettait un `return`, à ceci près que les variables locales sont conservées et que, lorsqu'on demande la valeur suivante, l'exécution reprend juste après le `yield`.

Vous trouverez deux exemples d'itérateurs pour les séquences binaires dans le fichier `gen.py`, l'un récursif, l'autre non.

Un générateur peut s'utiliser dans un `for`, comme s'il s'agissait d'un `range` :

```
c = 0
for s in genWords(5):
  c = c + 1
print(c)
```

Inspirez-vous de ces deux exemples pour écrire des versions "itérateurs" des fonctions de génération exhaustive de la feuille TM1 comme de celle-ci.