Université de Bordeaux
L2 Informatique/Maths-Informatique

Probabilités, Statistiques, Combinatoire : TM2

Combinatoire : génération exhaustive (2)

`http://www.labri.fr/perso/duchon/Enseignements/Probas/`

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 `tm2.py`, qui contient plusieurs versions de certaines des fonctions demandées dans le TM1, et les déclarations à compléter pour cette séance.

5.1 Permutations

Exercice 5.1
Permutations

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

1. Reprendre la fonction `ToutesSeqSousDiagoRec(n)` qui était donnée dans le TM1. Écrivez-en une version non récursive si ce n'est déjà fait ; cette fonction devrait bien entendu retourner la même liste de tuples, éventuellement dans un ordre différent.
   Indication : deux pistes sont possibles ; l'une consiste à adapter, de manière non récursive, l'algorithme de la fonction fournie, qui consiste, pour obtenir la liste de toutes les séquences sous-diagonales de longueur n, à ajouter tour à tour, à chaque séquence sous-diagonale de longueur $n-1$, chacun des entiers de 1 à n ; l'autre, plus difficile, consiste à adapter la méthode proposée à la dernière question de l'exercice 1.2 du TM1 : écrire les séquences sous-diagonales dans l'ordre lexicographique, en identifiant la "première" et en passant de chaque séquence à la "suivante" par un algorithme à mettre au point.
2. Le nombre de séquences sous-diagonales de longueur n est naturellement $n!$, comme celui des permutations de $[[1, n]]$. Écrire une fonction `SeqVersPerm(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, si ce n'est déjà fait, une fonction `PointsFixes(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).

4. Un **cycle de longueur** 2 dans une permutation s est la donée de deux entiers i et j, distincts, tels que j apparaisse en position i dans s, et i apparaisse en position j. Par exemple, dans $s = (3, 4, 1, 2)$ il y a deux cycles de longueur 2, car 3 apparaît en position 1 et 1 apparaît en position 3, mais également 2 apparaît en position 4, et 4 apparaît en position 2.

   Écrire une fonction `DeuxCycles(s)` qui retourne le nombre de cycles de longueur 2 dans la permutation s. **Indication :** compter les éléments i qui sont dans de tels cycles, mais avec un j plus grand que i ; il y en a autant que de cycles de longueur 2. En effet, pour un entier i donné, s'il est dans un tel cycle, il est facile de déterminer quel est l'autre entier impliqué : c'est l'entier en position i dans s.

   Utiliser vos fonctions pour *deviner* une formule donnant, en fonction de n, le nombre total de cycles de longueur 2 dans l'ensemble des permutations de $[[1, n]]$.

Exercice 5.2
Mots de Dyck

Dans cet exercice, on va représenter les mots de Dyck comme des séquences binaires, en traitant la valeur 0 comme la lettre b (pas descendant dans les chemins) et la valeur 1 comme la lettre a (pas montant dans les chemins).

1. Écrire une fonction `EstDyck(s)`, qui retourne `True` si la séquence binaire s est un mot de Dyck, et `False` sinon.
2. En réutilisant les fonctions écrites lors du TM1, écrire (cela ne doit pas prendre plus de quelques lignes !) une fonction `TousMotsDyck(n)`, qui retourne la liste de tous les mots de Dyck de longueur $2n$. Vous connaissez une formule donnant le nombre de tels mots en fonction de n ; vérifiez au moins que votre fonction donne le bon nombre de mots pour n allant de 0 à 10.
3. Un **pic** dans un mot de Dyck est une lettre a, immédiatement suivie d'une lettre b (dans le chemin : un pas montant suivi d'un pas descendant, d'où le nom). Écrire une fonction `Pics(s)`, qui retourne le nombre de pics du mot de Dyck passé en paramètre.
4. Utiliser vos fonctions pour déterminer combien il y a, au total, de pics dans l'ensemble des mots de Dyck de longueur $2n$, pour n allant jusqu'à 9. Proposer une formule donnant le nombre moyen de pics dans les mots de Dyck de longueur $2n$; **indication :** pour les longueurs que vos fonctions vous permettent d'explorer, écrire le nombre total de pics, divisé par le nombre de mots de Dyck, comme une fraction irréductible, et essayez de *deviner* une formule générale.
5. **Plus difficile :** chercher à *prouver* la formule que vous avez devinée ; en reprenant la définition du codage d'un arbre binaire complet, comment obtient-on un pic ? combien, au total, y a-t-il de feuilles dans les arbres binaires complets de taille n ?

5.2 Itérateurs (générateurs)

Cette partie est à considérer comme plus avancée : on se propose d'explorer une façon plus efficace de faire grosso modo la même chose que jusqu'à présent, en utilisant une construction spécifique du langage `Python` *que sont les générateurs.*

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 si on souhaite, par exemple, explorer l'ensemble de toutes les permutations de $[[1, 13]]$ ou de tous les mots de Dyck de longueur 30.

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 nombre de pics dans les mots de Dyck). Pour ces applications, il est amplement suffisant d'être capable de "voir" chaque objet une seule fois : si on peut le faire sans en manipuler la liste complète, le problème de mémoire devient seulement un problème de temps, sensiblement moins aigu.

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 est un mécanisme qui va permettre à un programme de fournir un ensemble de valeurs une par une, à la demande ; au lieu de forcément calculer à l'avance la liste de toutes les valeurs, le programme va calculer chaque valeur une fois que la précédente a été traitée.

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`.

Par exemple, un générateur pour les entiers de 0 à $n-1$ (équivalent à `range(n)`) pourrait s'écrire ainsi :

```
def my_range(n):
  k = 0
  while (k<n):
    yield(k)
    k = k+1
```

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 genSeqBin(5):
  print(s)
  c = c + 1
print(c)
```

(On peut aussi récupérer la liste de toutes les valeurs retournées avec `list(genSeqBin(5))`, mais dans ce cas pas la peine de se fatiguer à écrire un générateur !)

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

Cette méthode est en particulier efficace lorsque les objets à générer sont naturellement un sous-ensemble d'un ensemble plus gros, qui sont définis par une condition facile à tester. Par exemple, un générateur pour les mots de Dyck d'une longueur donnée est beaucoup plus facile à écrire à partir d'un générateur pour les séquences binaires et de la fonction `EstDyck(s)`, qu'à écrire directement. Il y a un surcoût : le nombre de séquences binaires de longueur $2n$ est 4^n, bien supérieur au nombre de mots de Dyck de longueur $2n$, C_n ; mais ce surcoût n'est que d'un facteur d'ordre [1] $n\sqrt{n}$, somme toute assez acceptable.

Pour vous en convaincre, écrivez directement un générateur pour les mots de Dyck de longueur $2n$ (ou "juste" une fonction qui en renvoie la liste).

1. Totalement hors programme : C_n est proche de $\frac{4^n}{n\sqrt{\pi n}}$.