Université de Bordeaux
L2 Informatique/Maths-Informatique

Probabilités, Statistiques, Combinatoire : TM1

Combinatoire : génération exhaustive

On se propose d'écrire des fonctions permettant la *génération exhaustive* d'objets combinatoires : étant donnée une famille, on va produire, un par un, tous les objets de la famille, pour une taille donnée.

Remarque sur la façon de représenter les objets : les objets que l'on va chercher à représenter dans nos programmes sont sous forme de séquences. Il peut sembler naturel de les représenter sous forme de *listes* PYTHON, mais, **il est très fortement recommandé d'utiliser des tuples plutôt que des listes**. Un *tuple* est similaire à une liste, à ceci près qu'il n'est pas modifiable. Quelques opérations utiles sur les tuples :

- on peut former un tuple en écrivant une séquence finie entre parenthèses, comme dans `(1,2,3)`
- on peut former un tuple à partir d'une liste, comme dans `t=tuple(L)` (et une liste à partir d'un tuple, avec `L=list(t)`)
- on peut concaténer des tuples (pour former un nouveau tuple) avec l'opérateur `+`
- on peut former un tuple vide avec `()`, et un tuple de longueur 1 (cas particulier) en mettant une virgule après l'élément, comme dans `(1,)`

5.1 Génération de listes

Exercice 5.1

Coefficients binomiaux : parties d'un ensemble On se propose, étant donnés deux entiers n et k (avec $0 \leq k \leq n$), de produire toutes les parties de $\{1, 2, \ldots, n\}$ ayant exactement k éléments. On sait qu'il y en a $\binom{n}{k}$; pour peu que k ne soit pas trop grand, ce nombre est bien inférieur à 2^n (nombre de toutes les parties de $\{1, \ldots, n\}$.

1. Les parties de l'ensemble $\{1, \ldots, n\}$ sont en bijection naturelle avec les suites de longueur n dont les éléments valent chacun 0 ou 1 : la valeur 1 code un élément présent, la valeur 0 un élément absent, de sorte que par exemple, pour $n = 4$, la partie $\{2, 3, 4\}$ serait codée par la séquence $(0, 1, 1, 1)$.

   Écrire une fonction `SequencesBinaires(n)`, qui retourne une liste dont les éléments sont (dans n'importe quel ordre) les 2^n séquences binaires de longueur n. Deux pistes pour cela :
   - Vous pouvez écrire une fonction qui convertit un entier entre 0 et $2^n - 1$ en une séquence de n chiffres binaires, correspondant au codage binaire d'un entier ; votre fonction `SequencesBinaires` se contentera alors de former la liste des résultats de la conversion de ces entiers ;
   - Un peu plus malin : vous pouvez commencer par la séquence ne comportant que des 0, et passer de séquence en séquence par une transformation qui correspond à incrémenter un entier en binaire ; ainsi, pour $n = 4$, la première séquence serait

$(0,0,0,0)$, puis $(0,0,0,1)$, etc ; et pour $n = 8$, la séquence $(0,1,0,0,1,1,1,1)$ serait suivie par $(0,1,0,1,0,0,0,0)$. Il est conseillé de travailler sur une liste, et de seulement convertir la liste en tuple pour la formation de la liste des séquences.

2. Écrire une fonction `SequenceBinaireVersPartie(s)`, qui met en oeuvre la bijection classique entre séquences binaires et parties : prenant en entrée une séquence binaire de longueur n, elle retournera la partie de $\{1,\dots,n\}$ correspondante ; chaque partie sera mise sous la forme d'un tuple dans lequel les éléments seront dans l'ordre croissant (ainsi, par exemple, l'ensemble $\{1,5,4\}$ sera représenté par le tuple `(1,4,5)`).
3. En combinant vos deux fonctions, écrire une fonction `Parties(n,k)` qui retourne une liste dont les éléments (dans l'ordre de votre choix) sont les $\binom{n}{k}$ parties de $\{1,\dots,n\}$ à k éléments, présentés de la même manière que précédemment.
4. Votre fonction précédente procède probablement en construisant les 2^n parties, et en ne conservant que celles de taille k ; or, par exemple, pour $k = 3$, le nombre de parties à 3 éléments est d'ordre $n^3/6$, bien plus petit que 2^n. On va donc chercher à améliorer la complexité de votre fonction, en ne générant *que* les parties à k éléments.

   Pour cela, on va les engendrer dans un ordre particulier, appelé l'ordre *lexicographique* : la première partie sera $(1,2,\dots,k)$; la suivante sera $(1,2,\dots,k-1,k+1)$, puis $(1,2,\dots,k-1,k+2)$, et ainsi de suite jusqu'à $(1,2,\dots,k-1,n)$, puis on passe à $(1,2,\dots,k-2,k,k+1)$, etc. La dernière sera $(n-k+1,n-k+2,\dots,n)$. **Ici, il est indispensable de réfléchir un stylo à la main : comment passe-t-on d'une partie à la suivante ?**

   Vous allez donc écrire deux fonctions : une fonction `PremierePartie(n,k)`, qui retourne la première partie de $\{1,\dots,n\}$ à k éléments pour l'ordre lexicographique ; et une deuxième fonction `PartieSuivante(n,p)`, qui retourne la partie qui suit la partie p dans l'ordre lexicographique. Idéalement, la complexité de votre fonction `PartieSuivante` devrait être $O(k)$ (k étant la longueur du tuple p).
5. À partir des deux fonctions de la question précédente, écrire une fonction `ToutesParties(n,k)` qui retourne la liste de toutes les parties de $\{1,\dots,n\}$ à k éléments, mais de manière beaucoup plus efficace (l'appel `ToutesParties(30,3)` devrait être quasi instantané).

Exercice 5.2

Permutations L'un des exercices de la feuille de TD1 vous proposait de décrire une bijection entre les permutations de $\{1,\dots,n\}$ et les séquences de n entiers dont le k-ème est compris entre 1 et k.

Programmer une de ces constructions, sous la forme d'une fonction prenant en entrée une séquence d'entiers $(i_1,\dots,i_n)$, telle que pour tout $1 \leq k \leq n$ on ait $1 \leq i_k leq k$, et qui retourne la permutation correspondante, sous la forme d'une séquence contenant une fois chacun des entiers de 1 à n.

Écrire également une fonction qui produit la liste de toutes les permutations de $[[1,n]]$.

Utiliser vos fonctions pour répondre à la question suivante : un *point fixe* d'une permutation s de $[[1,n]]$ est un entier k tel que l'on ait $s(k) = k$; et une permutation est un *dérangement* si elle n'a aucun point fixe. Par exemple, $(3,2,4,1)$ a 2 comme point fixe, mais $(2,3,4,1)$ est un dérangement. Combien y a-t-il de dérangements sur $[[1,10]]$?

Exercice 5.3

Mots de Dyck En vous inspirant de la fin de l'exercice 1, écrivez deux fonctions pour produire tous les mots de Dyck d'une longueur donnée $2n$, dans un ordre bien choisi (typiquement, lexicographique) ; utiliser 1 et 0 (entiers) à la place des lettres a et b. Pas d'indication pour cet exercice plus difficile – si vous vous y prenez bien, la complexité du passage d'un mot de Dyck au suivant doit être $O(n)$.

Vous pouvez écrire une fonction `SeqDyckVersString(s)`, qui convertit un mot de Dyck (sous forme de séquence de 0 et de 1) en une chaîne de caractères utilisant les lettres a et b, afin d'obtenir des affichages plus usuels.