Université de Bordeaux
L2 Informatique/Maths-Informatique

Probabilités, Statistiques, Combinatoire : TM4

Combinatoire : génération exhaustive

On se propose cette semaine 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 ; l'espoir est qu'on va faire ça en temps à peu près proportionnel au nombre des objets, mais pas plus.

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.

   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.

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 $\{1, \ldots, n\}$ 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 $\{1, \ldots, n\}$ 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 que l'on veut ; 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 $(1, 2, \ldots, k)$; la suivante sera $(1, 2, \ldots, k-1, k+1)$, puis $(1, 2, \ldots, k-1, k+2)$, et ainsi de suite jusqu'à $(1, 2, \ldots, k-1, n)$, puis on passe à $(1, 2, \ldots, k-2, k, k+1)$, etc. La dernière sera $(n-k+1, n-k+2, \ldots, n)$. **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 $\{1, \ldots, n\}$ 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 $\{1, \ldots, n\}$ à k éléments, mais de manière beaucoup plus efficace (l'appel `ToutesParties(30,3)` devrait être quasi instantané).

Exercice 5.2

Permutations et mélange d'une liste L'un des exercices de la feuille de TD5 vous proposait de décrire une bijection entre les permutations de $\{1, \ldots, n\}$ et les séquences de n entiers dont le k-ème est compris entre 1 et k.

Utiliser cette construction pour produire une fonction `Melange(L)`, qui prend en entrée une liste L (*a priori*, ne contenant que des valeurs distinctes) et qui retourne une nouvelle liste comportant exactement les mêmes valeurs, mais mélangées dans un ordre aléatoire – de telle sorte que, si la liste est de longueur n, les $n!$ listes possibles aient chacune probabilité $1/n!$ d'apparaître. Votre fonction devrait faire $n-1$ appels à `randint` : une fois `randint(1,2)`, une fois `randint(1,3)`, etc., jusqu'à `randint(1,n)`. Idéalement, la complexité de votre fonction devrait être $O(n)$.

Si vous avez un jour programmé l'algorithme `QuickSort`, comparez la complexité expérimentale de `QuickSort` sur la liste comportant tous les entiers de 1 à n dans l'ordre croissant (`list(range(1,n+1))`), et sur une liste comportant les mêmes valeurs, bien mélangées (`Melange(list(range(1,n+1)))`). Des valeurs de n de l'ordre de 1000 ou 10000 ne devraient pas poser de problème.

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. 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 $O(n)$.