TM 1

Combinatoire : génération de séquences et de permutations

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 (mots, permutations). 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,)`
- Comme avec une liste, on peut accéder (mais en lecture seulement) à l'élément d'indice k d'un tuple `t` par `t[k]` ; comme avec une liste, les indices commencent à 0.

Remarque sur l'indexation des séquences : Dans le cours, on suit la tradition en combinatoire des mots qui consiste à indexer les éléments d'une séquence à partir de 1 : la première lettre du mot w est notée w_1, la deuxième est notée w_2, et ainsi de suite. PYTHON, comme la plupart des langages de programmation, indexe les éléments des listes et des tuples à partir de 0 : dans la liste `[1,2,5,14]`, l'élément qui a pour valeur 14 est l'élément d'indice 3 et non 4.

1.1 Génération de séquences

Exercice 1.1

Écriture binaire d'un entier ; séquences binaires

Pour une "taille" k donnée, les 2^k nombres entiers compris entre 0 et $2^k - 1$ peuvent se coder par des séquencess binaires de longueur k sur l'alphabet $\{0, 1\}$. Ainsi, par exemple, pour $k = 3$, l'entier 1 peut se coder par $(0, 0, 1)$, et 3 par $(0, 1, 1)$.

1. Écrire une fonction `EntierVersBinaire(k,n)`, qui retourne sous forme de tuple le codage binaire de longueur k de l'entier n. Votre fonction peut ne pas retourner de valeur bien définie si n n'est pas compris entre 0 et $2^k - 1$.
2. Écrire la fonction "réciproque" `BinaireVersEntier(t)`, qui retourne l'entier codé par le tuple t (l'entier k n'est pas fourni : il est obtenu comme la longueur du tuple fourni).
3. Vérifier expérimentalement que vos deux fonctions sont bien inverse l'une de l'autre.
4. En utilisant les fonctions précédentes, écrire une fonction `ToutesSequencesBinaires(k)` qui retourne la liste de toutes les séquences binaires (chacune écrite sous la forme d'un tuple) de longueur k.

   Par exemple, `ToutesSequencesBinaires(2)` devrait retourner `[(0,0),(0,1),(1,0),(1,1)]` (ou une autre liste contenant les mêmes 4 tuples dans un autre ordre).

5. Écrire une fonction `NbConsecutifs(t)` qui retourne le nombre maximum de 1 consécutifs dans le tuple t passé en entrée. Ainsi, votre fonction devrait retourner 0 si $t = (0, 0, 0, 0, 0)$ (ou tout autre tuple ne comportant que la valeur 0), et 3 si $t = (0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0)$.
6. Utiliser vos fonctions pour répondre à la question suivante : parmi les $2^{20} = 1048576$ séquences binaires de longueur 20, combien contiennent au moins 3 fois consécutivement la valeur 1 ? 5 fois ? 10 fois ? Plus généralement, pour chaque valeur de k de 1 à 20, à partir de quelle valeur de ℓ (en fonction de k) y a-t-il moins de la moitié des séquences binaires de longueur k qui contiennent au moins ℓ fois consécutivement la valeur 1 ? Par exemple pour $k = 4$, parmi les 16 séquences de longueur 4, 1 a 4 1 consécutifs, 2 en ont 3, et 5 en ont 2 ; la réponse est donc 3 car $1 + 2 < 8$ mais $1 + 2 + 5 \geq 8$.
7. **Version récursive :** Écrire une fonction `ToutesSequencesBinairesRec(k)`, qui retourne la même liste de valeurs que la fonction `ToutesSequencesBinaires(k)` (éventuellement dans un autre ordre), mais qui fonctionne de manière récursive : pour "fabriquer" les séquences de longueur k, on commence (pour $k > 1$) par "fabriquer" les séquences de longueur $k - 1$; puis, à partir de chaque séquence de longueur $k - 1$, on en obtient deux de longueur k : une en ajoutant un 0, une autre en ajoutant un 1.

   Vous pouvez vous inspirer pour cela de la fonction donnée en exemple à l'exercice sur les séquences sous-diagonales.

Exercice 1.2

Parties d'un ensemble L'objectif est maintenant, étant donnés deux entiers n et k (avec $0 \leq k \leq n$), de générer toutes les parties de $[[1, n]]$ composées d'exactement k éléments – on sait qu'il en existe $\binom{n}{k}$. Les parties seront représentées sous la forme de tuples, avec les éléments dans l'ordre croissant : ainsi, l'ensemble $\{1, 3, 6\}$ sera représenté par le tuple PYTHON `(1,3,6)`.

Dans un premier temps, on réutilisera les fonctions écrites au premier exercice. Cela implique toutefois de construire 2^n objets, pour n'en retenir que $\binom{n}{k}$; si, par exemple, k est "petit" et n "grand", c'est très inefficace ; dans un second temps, on cherchera donc à produire uniquement les parties de taille k fixée sans passer par l'ensemble de toutes les parties.

1. Écrire une fonction `SequenceVersPartie(t)`, qui prend en entrée une séquence binaire (sous la forme de celles de l'exercice précédent), et la "convertit" en une partie de $[[1, n]]$ (où n est la longueur de t), représentée sous la forme d'un tuple. Il s'agit de repérer, dans t, les positions où on trouve la valeur 1 ; attention au fait que les indices vont de 0 à $n - 1$, et on souhaite des valeurs allant de 1 à n.
2. En réutilisant la fonction précédente et celles écrites à l'exercice précédent, écrire une fonction `Parties(n,k)`, qui retourne la liste de toutes les parties de $[[1, n]]$ contenant k éléments. Par exemple, pour $n = 5$ et $k = 2$, votre fonction devrait retourner (éventuellement dans un autre ordre) les 10 tuples $(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)$.
3. **Cette question demande plus de réflexion ; il est conseillé d'y réfléchir hors TM (et loin du clavier).** Il est probable que votre fonction `Parties` mette plusieurs secondes à calculer la liste des 1140 parties de taille 3 de $[[1, 20]]$ ($n = 20$, $k = 3$), tout simplement parce qu'elle commence par générer une liste de plus d'un million de tuples pour n'en filtrer qu'un peu plus de 1000. Générer toutes les parties de taille 2

de $[[1, 30]]$ (quelques centaines, mais parmi plus d'un milliard) serait prohibitif. Nous allons chercher un moyen plus efficace de procéder, et faire en sorte que la complexité de votre fonction ne soit dominée "que" par la longueur de la liste à retourner.

Nous allons engendrer les $\binom{n}{k}$ parties de taille k dans un ordre bien défini qui est appelé *l'ordre lexicographique* pour les séquences de longuer k fixée, et qui est défini ainsi : si u et v sont deux séquences d'entiers, toutes deux de longueur k, et différentes (c'est-à-dire qu'il existe un indice $j \leq k$ tel que $u_j \neq v_j$), alors si, en prenant pour j le *plus petit* entier entre 1 et k tel que l'on ait $u_j \neq v_j$, on déclare que la suite u est "plus petite" que la suite v (noté $u < v$) si $u_j < v_j$; sinon, on déclare que la suite v est plus petite que u. Ainsi, on a $(1, 4, 12) < (1, 5, 6)$. (Si on remplace les entiers par des lettres, c'est l'ordre du dictionnaire, à ceci près que dans la description ci-dessus, on ne dit pas comment comparer deux séquences qui ne sont pas de la même longueur.)

La méthode itérative pour la génération des parties de taille k de $[[1, n]]$ va ainsi consister en les briques suivantes :

— On identifie la "première" séquence : celle qui est plus petite que toutes les autres ; et la "dernière" séquence, celle qui est plus grande que toutes les autres.

— On identifie comment, à partir d'une séquence u, on peut définir la *séquence suivante* : l'unique séquence v qui est plus grande que v, mais qui est plus petite que toutes les autres séquences plus grandes que u ; une seule séquence n'a pas de suivante : la dernière (la "plus grande").

À titre d'exemple, pour l'ordre lexicographique des séquences de 4 entiers entre 1 et 20, la suivante de $(2, 8, 19, 20)$ est $(2, 9, 10, 11)$.

— On n'a plus qu'à traduire ces définitions en deux fonctions `PremierePartie(n,k)` et `Suivante(n,s)`, et générer la liste de toutes les parties de taille k se fera avec une simple boucle :

```
L=[]
s = PremierePartie(n,k)
while (s!=None):
  L.append(s)
  s = Suivante(n,s)
```

Ici on a supposé que l'appel `Suivante(n,s)` retourne la valeur spéciale `None` quand s est la dernière partie.

Mettez en œuvre cette méthode pour la génération de toutes les parties de taille k de $[[1, n]]$. Si vous vous y prenez bien, la complexité de votre code devrait être $O(k\binom{n}{k})$ (la complexité de la fonction `Suivante(n,s)` devrait être $O(|s|)$).

1.2 Génération de permutations

Exercice 1.3

Permutations L'un des exercices de la feuille de TD1 vous proposait de décrire une bijection entre les permutations de $[[1, n]]$ et les "séquences sous-diagonales" de longueur n ; les suites de n entiers dont, pour chaque entier $1 \leq k \leq n$, le k-ème est compris entre 1 et k. Par exemple, les séquences sous-diagonales de longueur 3 sont (dans l'ordre lexicographique) $(1, 1, 1), (1, 1, 2), (1, 1, 3), (1, 2, 1), (1, 2, 2), (1, 2, 3)$

L'objectif de cet exercice est d'obtenir, pour un entier n donné (pas trop grand ; $n = 11$ est un peu la limite, 11! étant proche de 40 millions), la liste des $n!$ permutations de $[[1, n]]$.

1. Pour cela, une méthode est d'écrire une fonction qui prend en entrée une séquence sous-diagonale s quelconque, et lui applique l'une des transformations bijectives donnant une permutation, décrites dans l'exercice de TD. Il restera alors à écrire une fonction qui produit la liste de toutes les séquences sous-diagonales de longueur n, mais c'est facile. Voici une version récursive ; vous écrirez vous-mêmes une version non récursive.

```
def ToutesSeqSousDiagoRec(n):
  if (n<=0):
    return [()]
  else:
    L = ToutesSeqSousDiagoRec(n-1)
    M = []
    for s in L:
      for j in range(1,n+1):
        M.append(s+(j,))
    return M
```

   La génération de toutes les permutations de taille n peut alors prendre la forme suivante :

```
L = ToutesSeqSousDiago(n)
M = []
for s in L:
  M.append(SeqVersPerm(s))
```

   Ou même, en une ligne pour les experts de PYTHON : `M = [SeqVersPerm(s) for s in ToutesSeqSousDiago(n)]`

   Résultat, pour écrire une fonction qui retourne la liste de toutes les permutations de $[[1, n]]$, il n'y a plus qu'à produire la fonction `SeqVersPerm`. À vous de jouer.

2. Un *point fixe* d'une permutation est un entier (entre 1 et n) qui est envoyé sur lui-même par la permutation. Dans notre représentation des permutations par des tuples, c'est un entier j qui apparaît en j-ème position dans le tuple (attention aux problèmes d'indices ! les indices PYTHON commencent à 0, les nôtres à 1). Ainsi, $(4, 2, 3, 1)$ a deux points fixes (2 et 3) ; $(6, 4, 5, 1, 3, 2)$ n'a aucun point fixe.

   Écrire une fonction `NbPointsFixes(s)`, qui prend en entrée une permutation et retourne le **nombre** de ses points fixes. Ainsi, `NbPointsFixes((4,2,3,1))` devrait retourner 2, et `NbPointsFixes((6,4,5,1,3,2))` devrait retourner 0.

3. En combinant vos fonctions, répondre aux questions suivantes, pour n allant de 1 à 11 (plus si vous le pouvez) :
   (a) quel est le nombre total de points fixes dans l'ensemble des $n!$ permutations de $[[1, n]]$?
   (b) combien y a-t-il de permutations *sans* point fixe parmi les $n!$ permutations de $[[1, n]]$? (ou, numériquement : quelle *proportion* des $n!$ permutations de $[[1, n]]$ sont sans point fixe ?) (Une permutation sans point fixe est aussi appelée un *dérangement*)