L. DIDIER & R. CABANE

La combinatoire des parties

Présentation et objectifs

Dans le programme (spécialité Terminale)

Contenus Nombre des parties d'un ensemble à n éléments. Lien avec les n -uplets de {0,1}, les mots de longueur n sur un alphabet à deux éléments, les chemins dans un arbre, les issues dans une succession de n épreuves de Bernoulli.	Combinaisons de k éléments d'un ensemble à n éléments : parties à k éléments de l'ensemble. Représentation en termes de mots ou de chemins. **Exemple d'algorithme** Génération des parties à 2, 3 éléments d'un ensemble fini.

Situation déclenchante

Lundi

– *(le chef de service) Alex, je voudrais former une équipe avec quelques membres du service qui s'entendent vraiment bien. Peux-tu me faire la liste de toutes les équipes possibles ? Peu importe le nombre d'équipiers.*
– *Votre liste va être longue, non ?*
– *Peu importe…*
– *Bien chef.*

Mardi

– *Alex, tu avais raison, il y en a trop….*
– *Chef, on pourrait décider que l'équipe est formée de quatre personnes*
– *Bonne idée, fais-moi ça au plus vite.*
– *Oui chef.*
– *Et fais-moi une proposition d'équipe à partir de ta nouvelle liste, n'importe laquelle.*

Il s'agit donc ici de parties (ou sous-ensembles) d'un ensemble à n éléments, par exemple $E=[0,1,\ldots,n-1]$, non pour les dénombrer (voir la fiche sur le triangle de Pascal) mais pour les énumérer c'est-à-dire en faire une liste soit complète soit limitée aux parties à k éléments (combinaisons).

- La représentation informatique d'un ensemble peut se faire de diverses manières, la plus simple étant fournie par les listes, en convenant de ne pas prêter attention à l'ordre des éléments. Pour plus de facilité dans la saisie comme l'affichage, les chaînes de caractères sont commodes d'autant qu'elles peuvent être parfois traitées avec la même syntaxe que les listes[1] .
- Manipulations de listes : voir l'appendice 1 pour plus de détails sur les syntaxes appropriées.
- Les sous-ensembles seront donc pris comme des sous-listes ou des sous-chaînes (selon le contexte), et la collection des sous-ensembles sera formée comme une liste de listes ou une liste de chaînes. Par sous-liste d'une liste `L` on entend une liste formée de certains éléments distincts de `L` (dans le même ordre que dans `L`), et par sous-chaîne d'une chaîne de caractères `s` on entend une chaîne formée de certains caractères de `s`, dans le même ordre. Le respect de l'ordre nous évitera d'avoir des « doublons » comme `"ab"` vis-à-vis de `"ba"`.
- Enfin, au lieu de produire une liste des parties, nous pouvons aussi chercher à en tirer une au hasard, en veillant à ce que ce tirage soit « équiprobable ».

1 Attention : au contraire des listes, les chaînes ne sont pas « mutables » en Python, au sens où on ne peut modifier leurs éléments.

L. DIDIER & R. CABANE

Un outil essentiel : la récursivité

Pour ces diverses tâches, il est essentiel de pouvoir se servir du principe de récursivité, admis par le langage Python : une fonction peut s'appeler elle-même, sous réserve de ne pas aboutir à une « boucle infinie ». Voir l'appendice 1 pour plus de détails.

Objectifs

1. Écrire une fonction Python récursive opérant sur une chaîne de caractères distincts `s` et renvoyant une liste contenant toutes les sous-chaînes de `s`. L'idée pourrait être de combiner toutes les sous-chaînes contenant le premier caractère de `s` avec toutes les sous-chaînes ne le contenant pas.
2. Écrire une fonction Python récursive opérant sur une chaîne de caractères distincts `s` et renvoyant une liste contenant toutes les sous-chaînes de longueur `k` de `s`.
3. Écrire une fonction Python récursive opérant sur une liste de caractères distincts `L` et renvoyant une sous-liste de longueur `k` de `L` choisie au hasard.

L. DIDIER & R. CABANE

Fiche méthode

Objectif 1 : la liste des parties

La logique à suivre est de considérer le premier élément `s[0]` de l'ensemble `s` (ici pris comme une chaîne de caractères) et de distinguer les parties contenant cet élément de celles qui ne le contiennent pas.

Une fois que ce choix est effectué, il n'y a plus qu'à « recommencer de même » sur les éléments restants, ce qui se note `s[1:]` ; nous avons ici une fonction récursive !

```
ÉDITEUR : PARTIES
LIGNE DU SCRIPT 0010
def parties(s):
  if s=='':
    return ['']
  if len(s)==1: # cas simple
    return ['',s] # sortie
  t=s[1:] # s sauf 1er élt
  L=parties(t) # appel récursif
  L2=[s[0]+u for u in L]+[u for
     u in L]
  return L2
```

On utilise ici des « listes en compréhension », syntaxe concise permettant de créer le résultat à partir des éléments d'une liste.

Pour que cela s'achève, il faut traiter préalablement les cas où l'ensemble possède 0 ou 1 élément (l'ensemble à 1 élément admet deux parties, la partie vide et la partie pleine). Le principe de récursivité montre, au passage, que le nombre de parties double quand on ajoute un élément à l'ensemble ; en d'autres termes, le nombre de parties d'un ensemble à n éléments est 2^n.

```
PYTHON SHELL
>>> parties("")
['']
>>> parties("a")
['', 'a']
>>> parties("ab")
['a', 'ab', '', 'b']
>>> parties("abc")
['ab', 'abc', 'a', 'ac', 'b', 'b
c', '', 'c']
```

Objectif 2 : la liste des combinaisons

Commençons par un cas très simple, suggéré par le programme : lister les sous-ensembles de taille 2 d'un ensemble donné. Cela peut se faire de manière très simple en Python grâce au principe des « listes en compréhension », voir ci-contre.

```
PYTHON SHELL
>>> L=[1,2,3,4]
>>> [[x,y] for x in L for y in L
 if x<y]
[[1, 2], [1, 3], [1, 4], [2, 3],
 [2, 4], [3, 4]]
```

On pourrait procéder de même pour les parties à trois éléments, mais la commande commence à être pénible à écrire, aussi convient-il de chercher une réponse plus générale. Si on s'autorise à écrire une fonction doublement récursive, on peut suivre un peu la même idée que pour le listage de toutes les parties : celles qui contiennent le premier élément et celles qui ne le contiennent pas.

Il faut prendre garde à l'initialisation et bien prévoir les deux cas extrêmes : lorsqu'on ne cherche que des parties vides (en ce cas, on renvoie le vide sous la forme `['']`), et lorsqu'on ne cherche que la partie pleine (en ce cas, on renvoie `s` pour seule réponse).

```
ÉDITEUR : PARTIES
LIGNE DU SCRIPT 0041
def combin(k,s):
  if k==0 or k>len(s):
    return ['']
  if k==len(s): # cas simple
    return [s]  # sort direct
  t=s[1:] # s sauf 1er élt
  L1=combin(k-1,t) # appel réc
  L2=combin(k,t)
  return [s[0]+u for u in L1]+L2
```

Détail à surveiller : le symbole `+` désigne ici la concaténation (mise bout à bout) dans deux contextes différents :

- l'expression `s[0]+u` est une chaîne de caractères, commençant par `s[0]` et continuant avec `u`,
- et l'opération `+` qui suit concatène deux listes.

Nous reviendrons plus loin sur l'inefficacité de ce codage (défi « modérer la récursivité »).

Objectif 3 : une combinaison au hasard

Pour tirer une partie de `k` éléments d'un ensemble donné (ici décrit par une liste `L`), nous pouvons aussi procéder de manière récursive. S'il s'agit de tirer une partie vide (`k=0`), c'est immédiatement réglé.

Sinon, on tire au hasard un élément de `L`. Ce choix est effectué par l'instruction `randint(0,len(L)-1)` ; puis on le retire de `L` (grâce à l'appel `L.pop`).

```
ÉDITEUR : PARTIES
LIGNE DU SCRIPT 0067
def combalea(k,L):
    if k==0:
        return []
    ch=L.pop(randint(0,len(L)-1))
    K=combalea(k-1,L)
    return sorted([ch]+K)
```

Si `L` est une liste comme `[2,3,5]`, `L.pop(1)` fait deux choses :
– retirer le deuxième élément de `L`
– renvoyer cet élément (ici, 3).

Si `L` est une liste, l'appel `sorted(L)` renvoie une liste triée ayant les mêmes éléments que `L`.

Il ne reste plus qu'à tirer `k-1` éléments de la liste restante, ce que est réalisé par l'appel récursif `combalea(k-1,L)`.

Pour fournir la réponse, on met l'élément tiré au hasard en tête (ce que fait l'instruction `[ch]+K`) puis on trie le tout (appel de la fonction `sorted`) pour éviter de présenter comme différentes des réponses ne se distinguant que par l'ordre.

Une fois que l'idée est formulée, il n'y a aucune difficulté à reprendre l'algorithme et à le programmer de manière itérative (non récursive) : on accumule de proche en proche les éléments tirés (au hasard) de `L` dans une liste `C` initialement vide ; il ne reste plus qu'à renvoyer cette liste, triée.

```
ÉDITEUR : PARTIES2
LIGNE DU SCRIPT 0063
def combaleait(k,L):
    C=[]
    for i in range(k):
        r=randint(0,len(L)-1)
        C.append(L.pop(r))
    return sorted(C)
```

Pour aller plus loin

Et la représentation binaire ? (objectif 4)

Il s'agit de ce qui est évoqué au programme par le « lien avec les n-uplets de {0,1} ». L'idée est la suivante. Tout nombre entier m peut se décomposer en une somme de puissances de 2, de manière analogue à la représentation décimale avec les puissances de 10. Les exposants de ces puissances sont appelés les « bits[2] » de m ; ainsi, on a $11=2^3+2+1$. En choisissant certains des n bits de rangs $0,1,\cdots,n-1$, on peut représenter tous les entiers de 0 à 2^n-1 ; c'est ainsi que le choix d'un sous-ensemble de $\{0,1,\ldots,n-1\}$ revient à choisir un entier inférieur ou égal à 2^n-1 et à déterminer ses « bits », c'est-à-dire à trouver sa représentation binaire.

Un nouvel **objectif** apparaît alors : il s'agit d'écrire une fonction Python non récursive renvoyant une liste de toutes les parties (sous-listes) de $\{0,1,\ldots,n-1\}$, en s'appuyant sur la représentation binaire de l'entier n.

Objectif 4 : codage en Python

Commençons par la recherche des bits d'un nombre entier `m`. Le bit le plus aisé à trouver est le bit « de poids faible » c'est-à-dire associé à $1=2^0$ dans l'écriture binaire de `m` : ce bit vaut 1 si m est impair et 0 sinon. Pour accéder aux autres bits, il suffit de diviser `m` par 2 : cela fait perdre le bit de poids faible et décale tous les autres bits. La fonction ci-contre réalise ce principe.

C'est ainsi que le choix du nombre 13, dont les bits ont pour rangs 0, 2 et 3, est associé au choix de la partie $\{0,2,3\}\subset\{0,1,2,3\}$. En récupérant les ensembles de bits de tous les entiers compris entre 0 et 2^n-1 on aura la liste de toutes les parties possibles, ce que la fonction `partbin` réalise.

Les parties apparaissent dans un ordre peu intuitif, mais ce n'est qu'une question de tri.

ÉDITEUR : PARTIES
LIGNE DU SCRIPT 0021

```
def bits(m): # liste bits de m
    L=[] # réceptacle réponse
    k=0
    while m>0:
        if m % 2==1: # impair ?
            L.append(k)
        k=k+1 # bit suivant
        m=m//2 # décalage
    return L
```

PYTHON SHELL

```
>>> bits(13)
[0, 2, 3]
```

ÉDITEUR : PARTIES
LIGNE DU SCRIPT 0032

```
def partbin(n):
    R=[] # réceptacle réponse
    for k in range(2**n):
        R.append(bits(k))
    return R
```

PYTHON SHELL

```
>>> partbin(4)
[[], [0], [1], [0, 1], [2], [0,
2], [1, 2], [0, 1, 2], [3], [0,
3], [1, 3], [0, 1, 3], [2, 3], [
0, 2, 3], [1, 2, 3], [0, 1, 2, 3
]]
```

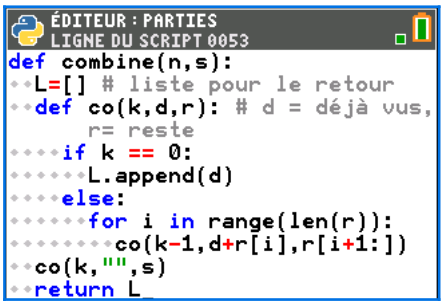
2 Abréviation anglaise pour « binary digits », c'est-à-dire chiffres binaires.

Défi : peut-on modérer un peu la récursivité ?

La récursivité présente l'avantage de la simplicité mais l'inconvénient d'amener parfois une « explosion » de la consommation de mémoire, très inefficace. C'est le cas dans la fonction `combin` ci-dessus : lors de l'appel `combin(2,"abcd")`, se déclenchent les appels `combin(1,"bcd")` et `combin(2,"bcd")` ; le premier va appeler `combin(1,"cd")` et le second de même : il y a un « doublon ». Dans l'appel `combin(2,"abcde")` les doublons sont bien plus fréquents…

Une fonction « simplement récursive » (qui s'appelle elle-même une seule fois) serait bien préférable. Nous en présentons un exemple ci-dessous.

Les lecteurs sont invités à en analyser le comportement en décrivant pas à pas (sur papier !) les actions qui se suivent lors de l'appel `combin(2,"abcd")`.



Indication : nous avons ici une « définition de fonction à l'intérieur d'une fonction ». Cette manière de procéder est tout à fait légitime en Python ; la fonction interne `co` (qui est récursive) va pouvoir traiter la variable `L` comme « externe » mais modifiable.