Exercices de révision et d'approfondissement

1 Exercices préliminaires

Exercice 1. Niveau 1 : écrire une fonction de nom `deuxième` qui admet trois arguments entiers `a,b` et `c` et qui renvoie la valeur médiane. Exemple :

```
>>> deuxième(3, 6, 2)
    3

>>> deuxième (1, 5, 5)
    5
```

Exercice 2. Niveau 1 : écrire une fonction de nom `somme` admettant un argument entier `n` et qui renvoie la somme $S_n = \sum_{i=1}^{n} \sum_{j=i}^{n} \frac{1}{1+i+j}$

Ecrire alors une fonction `premierIndice` qui admet un argument `v` (nombre entier ou flottant, peu importe) et qui renvoie le premier entier `n` tel que $S_n > v$. On fera bien sûr appel à la fonction précédente. Ici, on ne cherchera pas à être astucieux pour limiter le nombre d'appels à la fonction `somme`.

2 Arithmétique (un petit peu)

Exercice 3. Niveau 1.5. Etant donné n un naturel non nul, on dit que n est **parfait** si et seulement si il est égal à la somme de ses diviseurs propres. Par exemple $6 = 1+2+3$ est parfait.

1. Ecrire une fonction d'entête `def sommeDiv(n):` qui admet en argument un entier naturel non nul et qui renvoie la somme de ses diviseurs propres.

2. Ecrire une fonction `def estParfait(n):` qui admet en argument un entier naturel n non nul et qui renvoie `True` si n est parfait, et `False` sinon.

3. Ecrire une fonction `def parfaits(n):` qui admet en argument un entier naturel n non nul et qui renvoie la liste ordonnée des nombres parfaits entre 1 et n. (au sens large)

Exercice 4. Niveau 2 :

1. écrire une fonction d'entête `convertit(n, b)` qui prend en paramètre un entier naturel `n` et une base `b` (élément de $\mathbb{N}^*$) et qui renvoie une liste formée des chiffres (présentés sous forme d'entiers) de l'écriture en base `b` de `n`. Par exemple :

   ```
   >>> convertit(25, 13)
       [1, 12]
   ```

   On rappelle que la méthode repose sur des divisions euclidiennes répétées : par exemple en base 10, l'entier 152 lorsqu'on en écrit la division euclidienne par 10 devient : $152 = 10 \times 15 + 2$ et on voit que le reste donne le chiffre des unités, et on repart du quotient 15 qu'on divise encore par 10 pour avoir le chiffre des dizaines et ainsi de suite (jusqu'à ce que le quotient soit nul).

2. Modifier la fonction précédente pour qu'au lieu de renvoyer une liste, elle renvoie une chaîne de caractères. On supposera la base au plus égale à 36 et on écrira le chiffre 10 avec un A, 11 avec B et ainsi de suite jusqu'à, éventuellement, le chiffre 35 représenté par Z.

   ```
   >>> convertit2(25, 13)
       "1C"
   ```

   Indication : la fonction `chr` admet un argument entier, et renvoie le caractère associé (selon le codage unicode) à l'entier fourni en argument. La fonction `ord` réalise la fonction inverse : à un caractère, elle renvoie le code correspondant :

   ```
   >>> ord("C")
       67
   >>> chr(69)
       "E"
   >>> ord("0")
       48
   ```

Sans surprise, les codes des lettres A,B,C,…,Z se suivent, comme se suivent les codes des caractères numériques 0,1,…,9.

Exercice 5. Niveau 2 : Ecrire la fonction réciproque d'entête `deBaseB(s, b)` qui admet deux arguments : une chaîne `s` qui représente un entier écrit selon la base `b` (entre 2 et 36) et qui renvoie l'entier ainsi représenté :

```
>>> deBaseB("1C", 13)
    25
```

3 Combinaisons

Exercice 6. Niveau 1. Ecrire de deux manières (une version itérative et une version récursive) une fonction `def factorielle(n):` qui admet en argument un naturel n et qui renvoie la factorielle $n!$

Exercice 7. Niveau 1.5 Etant donnés des entiers $0 \leqslant p \leqslant n$, vous connaissez la combinaison $\binom{n}{p}$ par l'expression $\frac{n!}{p!(n-p)!}$.

Vous connaissez sans doute aussi la fameuse relation de Pascal qui énonce que pour tous n et p naturels :

$$\binom{n+1}{p+1} = \binom{n}{p} + \binom{n}{p+1}$$

Connaissant de plus les valeurs particulières : $\binom{n}{0} = \binom{n}{n} = 1$ cela permet de construire le triangle de Pascal donnant toutes les combinaisons sans avoir à calculer des factorielles :

$n=0$	1				
$n=1$	1	1			
$n=2$	1	2	1		
$n=3$	1	3	3	1	
$n=4$	1	4	6	4	10

1. Ecrire une première fonction d'entête `def comb(n, p):` qui fait appel à la fonction `factorielle` de l'exercice précédent.

2. Ecrire à l'aide de la relation de Pascal une fonction récursive d'entete `def combRec(n, p):` qui admet deux paramètres naturels n et p et calcule la combinaison $\binom{n}{p}$. On pourra supposer, sans le vérifier, que $0 \leqslant p \leqslant n$.

   Indication : comme toujours avec une fonction récursive, n'oubliez pas le test d'arrêt, c'est-à-dire des conditions selon lesquelles la réponse est immédiatement connue sans qu'il soit nécessaire de s'invoquer soi-même. (si $p=0$ ou $p=n$…)

   Bien sûr on teste. Y compris avec de grandes valeurs telles que `combRec(30, 15)` (au-delà, ce ne sera pas très raisonnable…)

3. Vous l'aurez constaté, pour de grandes valeurs, la fonction précédente est fort lente, ce qui s'explique par le fait que certains calculs sont effectués plusieurs (voire de très nombreuses) fois. Il est une autre formule très classique sur les combinaisons qui peut en accélérer singulièrement le calcul : pour tous naturels non nuls n et p :

   $$\binom{n}{p} = \frac{n}{p}\binom{n-1}{p-1}$$

   En déduire une seconde fonction récursive d'entête `def combRec2(n, p):` qui réalise le calcul de $\binom{n}{p}$ à l'aide de la formule précédente. Afin de calculer sur des entiers, on utilisera la division entière `//` (mais on fera attention à l'ordre des calculs, qui a alors son importance : `(2*3)//6` s'évalue en 1, mais `(2//6)*(3//6)` s'évalue en 0 par exemple).

Exercice 8. Niveau 1.5 On s'intéresse ici à la parité des coefficients combinatoires. Pour visualiser quelles sont les combinaisons paires et celles qui sont impaires, on va en quelque sorte dessiner un triangle de Pascal, mais au lieu d'écrire les valeurs des combinaisons, on écrira une étoile (`'*'`) pour toute combinaison impaire, et une combinaison paire sera représentée par un espace. Vous allez donc écrire une fonction qu'on nommera `triangle` et qui admet un unique argument : un entier `n`. Il n'y aura pas à proprement parler de valeur de retour (si ce n'est donc la valeur `None`) mais l'exécution de la fonction conduira à l'affichage de $n+1$ lignes (qu'on pourrait numéroter de 0 à n...), la première comptant 1 seul caractère, la seconde 2 jusqu'à la $n+1$-ième qui comptera $n+1$ caractères.

Par exemple, pour $n = 3$, l'affichage devra être :

```
*
**
* *
****
```

(écrivez les quatre premières lignes du triangle de Pascal, entourez les coefficients impairs et vous comprendrez...)
Testez ! (En faisant appel à `combRec2` de l'exercice précédent, $n = 200$ conduit à un affichage quasi-immédiat, tandis qu'on ne peut aller très loin avec `combRec`)

4 Parcours de liste

Exercice 9. Niveau 1 : écrire une fonction de nom `maximum`, admettant un unique argument de type liste (supposée non vide) et renvoyant la plus grande valeur de cette liste. Bien sûr, on s'interdira de faire appel à la fonction `max` de python !

Niveau 1.5 : reprendre la fonction précédente pour qu'elle renvoie non pas la plus grande valeur, mais l'indice de celle-ci. On pourra nommer `indiceMaximum` cette seconde fonction.

Exercice 10. Niveau 1.5 : écrire une fonction de nom `secondMaximum`, admettant un unique argument de type liste qu'on supposera comporter au moins deux éléments et qui renvoie la seconde plus grande valeur de cette liste. (Elle peut être égale à la plus grande, si cette plus grande valeur est présente deux fois).

Exercice 11. Niveau 1 : écrire une fonction d'entête `indiceDe(L, v)` qui admet pour arguments une liste `L`, et une valeur `v`, et qui renvoie le premier indice où `v` figure dans `L` s'il y est présent, et `None` s'il n'en fait pas partie. (`None`, et non pas la chaîne `'None'` : ne soyez pas surpris en testant votre fonction que rien ne s'affiche dans la console, c'est bien là le comportement attendu.)

Exercice 12. Niveau 2 : écrire une fonction d'entête `indicesDe(L, v)` qui admet pour arguments une liste `L`, une valeur `v` et qui renvoie la liste (éventuellement vide) de tous les indices où `v` figure dans `L`. Par exemple :

```
>>> indicesDe([2,1,6,1,3,1], 1)
    [1,3,5]

>>> indicesDe([2, 7, 4], 3)
    []
```

Exercice 13. Niveau 2 : écrire une fonction d'entête `sousListeCroissante(L)` qui admet pour argument une liste (qu'on supposera être de nombres, mais cela marcherait aussi pour une liste de chaînes, de tuples de nombres ou de tout type en python muni d'une relation d'ordre) et qui renvoie la plus grande longueur d'une sous-liste croissante (au sens large) formée de termes consécutifs de `L`. Par exemple :

```
>>> sousListeCroissante([1, 5, 6, 2, 5, 8, 1, 2, 0])
    3

>>> sousListeCroissante([4, 6, 1, 4, 6])
    3
```

puis reprendre la fonction précédente pour qu'elle renvoie non pas la longueur d'une plus grande sous-liste croissante, mais une telle sous-liste. Si plusieurs sous-listes sont possibles, on tâchera de renvoyer la première rencontrée (par un parcours de gauche à droite de notre liste bien entendu) :

```
>>> sousListeCroissante2([1, 5, 6, 2, 5, 8, 1, 2, 0])
    [1, 5, 6]

>>> sousListeCroissante([4, 6, 1, 4, 6])
    [1, 4, 6]
```

Exercice 14. Niveau 4 : écrire une fonction d'entête `sousListeCroissante(L)` qui admet pour argument une liste et qui renvoie la plus grande longueur d'une sous-liste croissante (au sens large) formée de termes non forcément consécutifs de `L` (mais tout de même d'indices croissants, il ne s'agit pas de trier en amont `L` pour donner comme réponse `len(L)`...). Par exemple :

```
>>> sousListeCroissante([1, 3, 2, 4, 3, 5])
    4
```

puis reprendre la fonction précédente pour qu'elle renvoie une telle sous-liste (d'indices dans `L` minimaux, ou n'importe laquelle après tout !) :

```
>>> sousListeCroissante2([1, 3, 2, 4, 3, 5])
    [1, 3, 4, 5]
```

Question subsidiaire : évaluer, selon la longueur `n` de `L`, la complexité de la votre fonction `sousListeCroissante`.

Exercice 15. Niveau 3 : écrire une fonction d'entête `tri(L)` qui, comme son nom l'indique, trie la liste `L` et renvoie la liste triée dans le sens croissant (peu importe ce qu'elle contient pourvu que ses éléments puissent être comparés : des nombres, des chaînes de caractères etc).

Il est permis de modifier ou non `L`.

On précisera la complexité, selon la longueur de `L`, de la fonction de tri implémentée. Si cela correspond à un tri standard (insertion, à bulles, sélection, fusion ou rapide), le préciser.

Exercice 16. Niveau 3 : écrire une fonction `dichotomie(L, v)` qui prend en argument une liste `L` supposée triée (non, on ne la trie pas à nouveau, c'est déjà fait !) et qui renvoie un indice où figure l'objet `v` s'il est présent dans `L`, et `None` dans le cas contraire.

On tiendra compte de ce que `L` est triée pour obtenir un algorithme de complexité logarithmique selon la longueur de `L`. (Reprenez le texte du TP si nécessaire)

Dans l'idéal : on cherchera à renvoyer le premier indice où figure `v` (ce n'est pas si évident. Comme on l'a vu en TP, formuler un invariant de boucle en amont aide à concevoir un tel algorithme)

5 Annexe : rappels de syntaxe

- N'oubliez pas que si l'opérateur d'affectation en python est `=`, l'affectation n'est pas du tout symétrique. Ce que traduit l'affectation `x = 2` est ce qu'en langage naturel on écrirait : $x \leftarrow 2$. Si on écrit `2 = x`, c'est une erreur de syntaxe, car `2` n'est pas un nom de variable valide.

- Ne pas confondre `=` (affectation) et `==` (test d'égalité)

- Il y a deux manières d'itérer sur les termes d'une liste :

  - par valeurs : `for v in L:`

    Par exemple si `L=[2, 5, 'x']`, ceci conduit à une boucle constituée de trois itérations, où `v` vaudra successivement `2`,`5` et `'x'`.

  - par indices : `for i in range(len(L)):`

    avec la même liste que précédemment, il y aura également trois itérations, et `i` vaudra successivement 0, 1 et 2, ce qui permet d'accéder aux termes de `L` par `L[0]`, `L[1]` et `L[2]`.

  (En toute rigueur, il y a en fait une troisième manière d'itérer qui permet de manier en même temps les indices et les termes de L : `for i, v in enumerate(L):` où, toujours sur le même exemple, `(i, v)` vaudra successivement `(0, 2)`, `(1, 5)` et `(2, 'x')`.)

- Chaînes de caractères : pour les chaînes de caractères comme pour les listes, `+` est l'opérateur de concaténation : `'hello'+'world'` conduit à `'helloworld'`.