TP 11
Révisions et consolidation 2

Exercice 1. *Pour le nouvel an*
La fin d'année approche. Écrire une fonction récursive `decompte(n)` qui affiche un décompte puis souhaite la bonne année. Par exemple avec $n = 5$:

```
>>> decompte(5)
5
4
3
2
1
Bonne année !!!
```

Exercice 2. *Rendu de monnaie*
On souhaite écrire une fonction `rendu(n)` qui prend en argument un entier n représentant une somme en euros et qui nous affiche comment il faut arriver à cette somme avec des billets et des pièces. Pour simplifier disons qu'on utilise uniquement pour l'instant des billets de 5 euros et des pièces de 1 et 2 euros. On voudrait par exemple obtenir les résultats suivants :

```
>>> rendu(13)
Billet de 5
Billet de 5
Pièce de 2
Pièce de 1
```

1. Au brouillon, proposer une formulation récursive du problème.
2. Écrire la fonction récursive `rendu(n)`. La fonction produit des affichages comme ci-dessus mais ne renvoie rien.

Exercice 3. *Classique classique*

1. Écrire une fonction `arrangement(n, k)` qui calcule le nombre d'arrangements, défini pour $n \in \mathbb{N}$ et $k \in \mathbb{N}$ par
$$A_n^k = \begin{cases} 0 & \text{si } k > n \\ \frac{n!}{(n-k)!} = n \times (n-1) \times \cdots \times (n-k+1) & \text{sinon} \end{cases} \tag{1}$$
en utilisant une boucle `for`. Remarquons que le nombre A_n^0 doit être égal à 1.
2. Écrire la même fonction mais cette fois-ci sans boucle et de façon récursive. Étonnamment, la récursivité porte sur l'entier k et non pas n.
3. Selon le *paradoxe des anniversaires*, le nombre de façons d'attribuer à k personnes leur date d'anniversaire parmi 365 jours de façon à ce qu'au moins deux personnes soient nées le même jour est $365^k - A_{365}^k$ (c'est le complémentaire d'un nombre d'arrangements : attribuer à k personnes des dates toutes différentes). La probabilité qu'au moins deux personnes aient la même date d'anniversaire est donc
$$p_k = \frac{1}{365^k}\left(365^k - A_{365}^k\right) = 1 - \frac{A_{365}^k}{365^k} \tag{2}$$
Écrire une boucle qui affiche, pour k de 1 à 50, le nombre k et la probabilité p_k.
On pourra aussi écrire une fonction `seuil(p)` prenant en argument un nombre réel $p \in [0, 1]$ et renvoyant le plus petit $k \geqslant 1$ tel que $p_k \geqslant p$, c'est à dire le nombre minimal de personne au-delà duquel la probabilité que parmi eux deux d'entre eux ait la même date d'anniversaire soit au moins p.

Remarque 1. On trouve les fonctions suivantes dans le module `math`, à importer avec `import math` :

1. `factorial(n)` : la factorielle de n.

2. `perm(n, k)` : nombre d'arrangements de k objets parmi n, appelé en anglais *nombre de permutations*.
3. `comb(n, k)` : coefficient binomial $\binom{n}{k}$, appelé en anglais *nombre de combinaisons*.

Si on peut se permettre de les utiliser parfois, il s'agit d'une question très très classique de savoir les ré-écrire. Pour que le programme soit efficace, il ne faut pas calculer $\frac{n!}{(n-k)!}$ en calculant la factorielle de chacun de ces deux termes puis en divisant — cela fait apparaitre des nombres très très grands alors que beaucoup de termes se simplifient dans la fraction — mais l'écrire comme un produit ci-dessus. Dans le paradoxe des anniversaires, il est bien plus efficace (rapidité et précision des calculs à virgule flottante) d'écrire $p_k = 1 - \frac{365}{365} \times \frac{364}{365} \times \cdots \times \frac{365-k+1}{365}$ et de calculer ces termes itérativement.

Exercice 4. *Palindromes*

On rappelle qu'un mot est un *palindrome* s'il se lit aussi bien de gauche à droite que de droite à gauche, par exemple le mot KAYAK ou le prénom ANNA. On souhaite écrire une fonction `est_palindrome(s)` qui prend en argument une chaine de caractères `s` et qui renvoie `True` si `s` est un palindrome et `False` sinon. De plus, on souhaite que la fonction soit récursive et utilise des sélections de tranches de `s`. La condition d'être un palindrome se formule récursivement à partir du premier caractère `s[0]`, du dernier caractère `s[-1]`, et du mot restant `s[1:-1]`.

1. Au brouillon, proposer une formulation récursive du problème. Attention à la condition d'initialisation : que se passe-t-il si le mot de départ était de longueur paire ? Et s'il était de longueur impaire ?
2. Écrire la fonction récursive `est_palindrome(s)`.

Exercice 5. *Une petite parenthèse...*

On s'intéresse aux expressions écrites uniquement avec des parenthèses ouvrantes et fermantes. Parmi celles-ci, certaines sont dites *bien parenthésées* quand les parenthèses sont correctement emboitées, par exemple `"((()(()))"` ou bien `"()(()))"`, mais d'autres sont dites *mal parenthésées* comme `")()("` ou bien `"(()()"`. L'algorithme pour tester si un mot est bien parenthésé est le suivant :

1. On initialise une variable p à 0,
2. On lit les caractères uns par uns,
3. À chaque parenthèse ouvrante on augmente p de 1, à chaque parenthèse fermante on diminue p de 1. Ainsi, p compte le nombre de parenthèses qui ont été ouvertes mais n'ont pas été refermées jusque là.

Alors :

1. Quelle(s) condition(s) peut-on dégager pour qu'une expression soit bien parenthésée ?
2. Écrire une fonction itérative `est_bien_parenthésée(s)` qui prend en argument une chaine de caractères, de type `str`, et qui renvoie `True` si `s` est bien parenthésée et `False` sinon.

Les mots bien parenthésés à n paires de parenthèses ouvrantes-fermantes sont dénombrés par les *nombres de Catalan* C_n. On a notamment $C_1 = 1$ avec le seul mot `"()"`, $C_2 = 2$ (les mots `"()()"` et `"(())"`). On considère que $C_0 = 1$.

3. Vérifier que $C_3 = 5$ en donnant les 5 mots bien parenthésés formés de 3 paires de parenthèses ouvrantes-fermantes.
4. Justifier que les nombres de Catalan vérifient la relation de récurrence $C_n = \sum_{k=0}^{n-1} C_k \times C_{n-1-k}$.
5. En déduire la valeur de C_4.
6. Pour calculer les nombres de Catalan avec cette formule, est-ce une meilleure idée d'écrire
   (a) Une fonction récursive ?
   (b) Une fonction itérative qui renvoie uniquement C_n ?
   (c) Une fonction itérative qui renvoie la liste des n premiers nombres de Catalan ?

   Justifier soigneusement et écrire cette fonction `catalan(n)`.

Exercice 6. *Permutations*

Écrire une fonction récursive `permutations(L)` qui prend en argument une liste `L` et qui renvoie la liste de toutes les permutations possibles de `L`.

On rappelle qu'un ordre naturel est, pour chacun des éléments de `L`, de placer cet élément en tête et de l'ajouter à toutes les permutations possibles du reste de la liste. On pourra utiliser l'opération de concaténation `+` sur les listes et les sélections de tranches `L[a:b]` pour réaliser ces opérations.

On doit par exemple avoir :

```
>>> permutations(["A", "B", "C"])
[['A', 'B', 'C'], ['A', 'C', 'B'], ['B', 'A', 'C'], ['B', 'C', 'A'],
['C', 'A', 'B'], ['C', 'B', 'A']]
```

Un autre ordre naturel consiste à récupérer seulement le premier élément de `L`, et l'insérer à toutes les positions possibles dans toutes les permutations du reste de la liste. Cela nécessite un seul appel récursif (permuter `L[1:]`) et ne donne à la fin pas les permutations dans le même ordre.