TP 13
Révisions et consolidation 2

Exercice 1 *Nouvel an*

1. Écrire une fonction **récursive** `decompte(n)` qui affiche un décompte puis souhaite la bonne année. Par exemple avec $n = 5$ on veut le résultat suivant :

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

2. Que se passe-t-il si dans le programme on inverse l'ordre des lignes `print` et `decompte(n-1)` ?

Exercice 2 *Classique classique*

1. Écrire une fonction `factoriel(n)`, **de façon récursive**, qui prend en argument un entier $n \in \mathbb{N}$ et renvoie la valeur de $n!$.

2. Écrire une fonction `arrangement(n, k)` qui calcule le nombre d'arrangements, défini pour $n \in \mathbb{N}$ et $k \in \mathbb{N}$ par

$$A_n^k = \begin{cases} 0 & \text{si } k > n \\ 1 & \text{si } k = 0 \\ \frac{n!}{(n-k)!} = n \times (n-1) \times \cdots \times (n-k+1) & \text{sinon} \end{cases}$$

en utilisant une boucle `for`.

3. Écrire la même fonction mais cette fois-ci sans boucle et **de façon récursive**. Étonnamment, il est plus facile de faire porter la récursivité l'entier k que sur n…

4. 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 de : attribuer à k personnes des dates toutes différentes parmi 365. La probabilité qu'au moins deux personnes aient la même date d'anniversaire est donc

$$p_k = \frac{1}{365^k}\left(365^k - A_{365}^k\right) = 1 - \frac{A_{365}^k}{365^k}$$

Écrire une boucle qui affiche, pour k de 1 à 50, le nombre k et la probabilité p_k.

5. Bonus : 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 personnes au-delà duquel la probabilité que parmi eux deux d'entre eux aient la même date d'anniversaire soit au moins p.

Il est en fait bien plus efficace (à la fois en rapidité, et en 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 avec une boucle.

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

- `factorial(n)` : la factorielle de n.
- `perm(n, k)` : nombre d'arrangements de k objets parmi n, appelé en anglais *nombre de permutations*.
- `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.

Exercice 3 *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(m)` qui prend en argument une chaine de caractères `m` et qui renvoie `True` si `m` est un palindrome et `False` sinon. De plus, on souhaite que la fonction soit récursive. La condition d'être un palindrome se formule bien de façon récursive à partir du premier caractère `m[0]`, du dernier caractère `m[-1]`, et du mot restant (tranche) `m[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(m)`.

Exercice 4 *Le compteur d'anagrammes (d'après annale DS)*
On souhaite écrire un programme pour dénombrer tous les anagrammes d'un mot. Pour cela, on a besoin d'une fonction `factoriel(n)` et de compter combien de fois apparait chaque lettre. Pour simplifier, on suppose que nos mots sont écrits uniquement avec les 26 lettres de l'alphabet en minuscule (pas d'accents, pas de majuscule, pas d'autres signes de ponctuation) et on donne la variable Python `alphabet = "abcdefghijklmnopqrstuvwxyz"`. Il est alors pratique de numéroter les lettres à partir de 0, ainsi `a` est la lettre 0 et `z` est la lettre 25.

1. Écrire, si ce n'est pas déjà fait, la fonction `factoriel(n)`.
2. Écrire une fonction `numero(x)` qui prend en argument un caractère seul `x` et renvoie son numéro en tant que lettre de l'alphabet.
3. Écrire alors une fonction `compte(m)` qui prend en argument une chaine de caractères `m` et qui renvoie une liste `C` de longueur exactement 26, telle que `C[j]` est le nombre de fois où la lettre numérotée `j` apparait dans `m`.
4. En déduire la fonction `anagrammes(m)` qui renvoie le nombre d'anagrammes du mot `m`.

   On rappelle qu'on l'obtient à partir de la factorielle de la longueur du mot, divisée par le produit des factorielles des nombres de fois que chaque lettre apparait. Comme $0! = 1$ il est cohérent de considérer que les lettres qui n'apparaissent pas apparaissent en fait 0 fois, ainsi ce ne sont pas des cas à traiter à part.
5. À partir des fonctions précédentes, écrire une fonction `sont_anagrammes(m, s)` qui renvoie `True` si les mots `m` et `s` sont bien anagrammes l'un de l'autre, et `False` sinon.

Exercice 5 *Mots de Fibonacci*
On s'intéresse aux suites de n chiffres `0` ou `1` telles qu'il n'y ait pas deux `1` consécutifs. Nous avons vu en TD que ces suites sont obtenues de deux façons :
- Soit à partir d'une suite de longueur $n-1$, à laquelle on rajoute comme premier terme un `0`,
- Soit à partir d'une suite de longueur $n-2$, à laquelle on rajoute `10` au début.

Ainsi le nombre u_n de telles suites vérifie la relation de Fibonacci $u_n = u_{n-1} + u_{n-2}$.

Mais le but cette fois-ci est de produire la liste de tous les mots qu'on peut obtenir, c'est une *liste* de *chaines de caractères*. Écrire cette fonctions `suites(n)` de façon récursive.

On obtient par exemple :

```
>>> suites(5)
['00000', '00001', '00010', '00100', '00101', '01000', '01001', '01010', '10000',
'10001', '10010', '10100', '10101']
```

Ici il y a bien 13 mots, et le nombre 13 est bien dans la suite de Fibonacci.

Exercice 6 (*) *Générer les anagrammes*

1. Écrire une fonction `anagrammesAB(a ,b)` qui renvoie la *liste* de tous les anagrammes qu'on peut produire avec seulement les lettres A et B, en utilisant a fois la lettre A et b fois la lettre B.

   Récursivement, ces anagrammes sont tous obtenus en prenant la lettre A et en la concaténant à tous les anagrammes possibles avec autant de B mais $a-1$ lettres A, ou bien en démarrant par B concaténé à tous les anagrammes possibles avec autant de A mais $b-1$ lettres B.

2. Plus généralement, écrire une fonction `anagrammes(C)` qui prend en argument une liste `C` de longueur 26 (comme dans l'exercice 4), donnant combien de fois doit apparaitre chaque lettre de l'alphabet, et qui renvoie la liste de tous les anagrammes possibles avec ce compte de lettres.

Exercice 7 (**) *Permutations*

Écrire une fonction `permutations(n)` qui renvoie une liste de toutes les permutations possibles de l'ensemble $[\![1, n]\!]$. On doit par exemple avoir

```
>>> permutations(3)
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

Récursivement, un ordre naturel consiste à récupérér la liste des permutations de $[\![1, n-1]\!]$ et à insérer, dans chacune de ces permutations, le nombre n à chacune des positions possibles ; l'ordre obtenu pour l'énumération des permutations dépend de l'ordre des opérations effectuées. Pour « insérer » on pourra utiliser à la fois les tranches `L[a:b]` et l'opération de concaténation `+` entre listes.