CH01 - Exercices

Activité préparatoire

Te rappelles-tu ce qu'est une poignée de main ? Voici comment WIKIPÉDIA définit cette coutume « pré-Covid 19 » :
Une poignée de main est un geste de communication effectué le plus souvent en guise de salutation mais qui peut également être une signification de remerciement ou d'accord.

On se pose la question suivante : « Quand n personnes se rencontrent, si chacun serre la main des autres une seule fois, combien cela fait-il de poignées de main en tout ? ».
On décide de noter $f(n)$ ce nombre.

1. a. Que valent $f(2)$, $f(3)$, $f(4)$, $f(5)$?

   b. Que valent logiquement $f(1)$ et $f(0)$?

2. Supposons que l'on connaisse $f(10)$. Une 11^{e} personne arrive. Combien doit-on ajouter à $f(10)$ pour obtenir $f(11)$?

3. Pour tout $n \in \mathbb{N}^*$, déduis-en ce que vaut $f(n)$ à partir de $f(n-1)$.

4. À partir du résultat précédent, écrit en PYTHON la fonction `f` qui :

   - en entrée prend un `int n` positif ;
   - renvoie le nombre de poignées de mains lors de la rencontre de `n` individus.

   Indice : rien n'interdit à `f` de « s'appeler elle-même » !

Exercice 1

En s'inspirant de la fonction factorielle, coder en PYTHON de manière récursive la fonction `somme` qui

- en entrée prend un entier naturel `n` ;
- renvoie zéro si `n` vaut zéro ;
- renvoie `n + somme(n - 1)` sinon.

On vérifiera que `somme(1000)` vaut 5 050. Expliquer ce que calcule `somme(n)`.

Exercice 2

Programmer la fonction `sommes_cubes()` en Python de manière récursive, et tester cette fonction.
`sommes_cubes(n)` devra renvoyer, pour $n \in \mathbb{N}$

$$\sum_{k=0}^{n} k^3$$

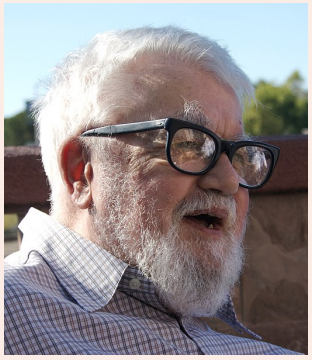
C'est-à-dire la somme des cubes des entiers naturels de 0 à n.

Exercice 3 : récursion imbriquée

La fonction f_{91} de McCarthy est définie sur $\mathbb{N}$ par :

$$f_{91}(n) = \begin{cases} n - 10 & \text{si } n > 100 \\ f_{91}\left(f_{91}(n+11)\right) & \text{sinon} \end{cases}$$

Programmer f_{91} en PYTHON et vérifier que pour tout entier naturel n inférieur ou égal à 101, $f_{91}(n)$ vaut... 91.



John McCarthy, informaticien, lauréat du prix Turing en 1971.

Exercice 4 : fonction puissance

Coder en PYTHON la fonction `puissance` qui

- en entrée prend un `int` positif `n` et un `float` `x` ;
- renvoie la valeur de `x**n` calculée récursivement (cas de base et cas récursif à trouver soi-même).

Exercice 5 : récursion mutuelle

On considère les deux suites a et b définie par

$$a(n) = \begin{cases} 1 & \text{si } n = 0 \\ n - b(a(n-1)) & \text{sinon} \end{cases} \quad \text{et} \quad b(n) = \begin{cases} 0 & \text{si } n = 0 \\ n - a(b(n-1)) & \text{sinon} \end{cases}$$

Programmer a et b en PYTHON, puis conjecturer pour quelles valeurs de n on a $a(n) \neq b(n)$ (ces valeurs sont en relation avec une suite déjà rencontrée).

Exercice 6 : palindromes

Un `str` est un palindrome si on peut le lire à l'envers comme à l'endroit. Par exemple « kayak », « Un radar nu » ou «!a!bcb!a!» sont des palindromes.
Écrire une fonction récursive `palindrome` qui :

- en entrée prend un mot (un `str`);
- renvoie `True` si c'est un palindrome et `False` sinon;
- procède récursivement
  - si le mot a une lettre ou bien deux lettres pareilles, c'est un palindrome;
  - sinon on regarde si les lettres du début et de la fin sont les mêmes. Si ce n'est pas le cas, ce n'est pas un palindrome. Si c'est le cas alors il faut regarder si le sous-mot restant est un palindrome.

Rappels : Si `s` est un `str`

- `s[0]` et `s[-1]` sont respectivement son premier caractère et son dernier caractère.
- `s[p:q]` renvoie la sous chaîne allant de `s[p]` à `s[q-1]`.

Exercice 7* : nombres de Catalan

Considérons l'opération « puissance ». appliquée à trois nombres, par exemple 2, 3 et 4, pris dans cet ordre. Il y a plusieurs manière de procéder :

- $2^{(3^4)} = 2^{81} = 2417851639229258349412352$
- $(2^3)^4 = 8^4 = 4096$

Pour 3 nombres, il y a donc 2 manières de placer les parenthèses pour effectuer les opérations.
Et pour 4 nombres ? Pour 5 ? Pour simplifier l'écriture on peut noter $a * b$ au lieu de a^b.
Alors avec n lettres on peut reformuler l'exemple précédent ainsi : quel est le nombre de manières (noté C_n) de placer des parenthèses autour des lettres de sorte que

Eugène Catalan, mathématicien franco-belge du XIX^{e} siècle.

- on n'ait jamais plus de deux termes non parenthésés : pas de choix à faire;
- on ne mette jamais des parenthèses autour d'un seul terme : pas de parenthèses inutiles.

Les premiers cas sont simples :

- $C_1 = 1$: il n'y a qu'une manière d'écrire a.
- $C_2 = 1$ aussi : une seule possibilité : $a * b$.
- $C_3 = 2$, on l'a vu : $a * (b * c)$ et $(a * b) * c$.
- Pour calculer C_4, on peut classer les parenthésages suivant la dernière opération $*$ à faire :
  - après a : il y a $a * ((b * c) * d)$ et $a * (b * (c * d))$.
  - « entre b et c » : $(a * b) * (c * d)$.
  - juste avant d : $((a * b) * c) * d$ et $(a * (b * c)) * d$.

  Finalement cela fait 5 possibilités, $C_4 = 5$ et on remarque que $C_4 = C_1C_3 + C_2C_2 + C_3C_1$.

Cela se généralise : pour tout n entier supérieur à 2 on a

$$C_n = C_1C_{n-1} + C_2C_{n-2} + \ldots + C_{n-1}C_1$$

En effet, on commence par choisir la position de la dernière opération à effectuer : puisqu'il y a n lettres il y a $n - 1$ choix de positions possibles, chacun scindant le mot de n lettres en un mot de p lettres et un autre de $n - p$, qui donnent donc lieu respectivement à C_p et C_{n-p} parenthésages indépendants, donc C_pC_{n-p} parenthésages en tout. En considérant toutes les valeurs de p possibles on arrive à

$$C_n = \sum_{p=1}^{n-1} C_pC_{n-p} \qquad (*)$$

Ce qui permet de calculer $C_5 = 14$, $C_6 = 42$, $C_7 = 132$, $C_8 = 429$, $C_9 = 1430$, $C_{10} = 4862$…

Travail à faire :

Programmer la fonction `catalan` en PYTHON :

– Le cas de base est pour `n` valant `1`, où la fonction renvoie `1` ;

– Si `n` est plus grand, alors on calcule `catalan(n)` récursivement à l'aide de (*).

On pourra utiliser la fonction `sum`.
Par exemple `sum(n for n in range(10))` calcule la somme $0 + 1 + \ldots + 9$.

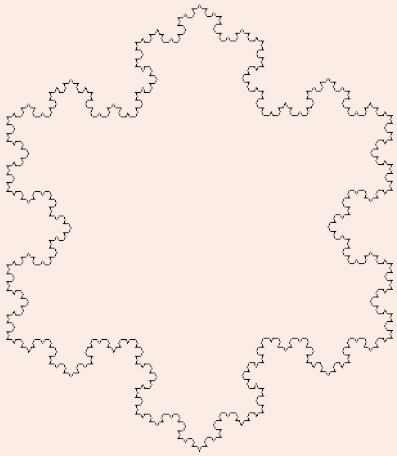
Exercices supplémentaires

Exercice 8 : approximation d'un flocon de Von Koch

On part d'un segment (étape 0) que l'on coupe en 3 parties égales. Sur le segment du milieu on construit un triangle équilatéral puis on enlève ce segment. On obtient alors une itération du procédé de Von Koch. On peut ensuite répéter indéfiniment ce procédé.

Les 4 premières étapes du procédé.

Lorsque l'on part d'un triangle équilatéral auquel on applique ce procédé une infinité de fois, l'objet obtenu s'appelle un *flocon de Von Koch*. Il a la particularité d'avoir une aire finie mais un périmètre infini.



Une approximation d'un flocon de Von Koch.

Pour dessiner, on va utiliser le module `turtle` de PYTHON.

1. Regarde bien le micro-tutoriel pour comprendre le fonctionnement de base de `turtle`.
2. Coder la fonction `koch` qui
   - en entrée prend un `float x` (longueur du segment de base) et un `int n` (nombre d'itérations);
   - ne renvoie aucune valeur mais dessine le processus itéré n fois, de manière récursive.

Exercice 9 : fonction puissance améliorée

Soit x un nombre réel et n un entier positif alors on a

$$x^n = \begin{cases} (x^{n/2})^2 & \text{si } n \text{ est pair} \\ x \times (x^{(n-1)/2})^2 & \text{sinon} \end{cases}$$

1. En se basant sur cette observation, coder une fonction `puissance_amelioree` qui a les mêmes spécifications que la fonction `puissance` vue dans un exercice précédent.
2. Créer pour ces deux fonctions une variable `nb_appels` pour comptabiliser le nombre d'appels récursifs et comparer ces nombres dans `puissance(2,128)` et `puissance_amelioree(2,128)`.

Exercice 10 : algorithme d'Euclide récursif

Soient n et p deux entiers strictement positifs. On note pgcd$(a;b)$ le plus grand entier qui divise à la fois a et b.
Par exemple

- $1050 = 2 \times 3 \times 5 \times 5 \times 7$;
- $770 = 2 \times 5 \times 7 \times 11$;
- le pgcd de ces deux nombres est donc $2 \times 5 \times 7 = 70$.

Pour trouver le pgcd de deux nombres on peut, comme dans l'exemple précédent, les décomposer en produit de facteurs premiers et prendre le produit de tous les facteurs communs, mais on peut aussi utiliser l'algorithme d'Euclide :
Soient a et b deux entiers strictement positifs, on suppose que $a \leqslant b$

- on écrit la division euclidienne de a par b : $a = q \times b + r$ avec $r < b$;
- si un entier divise a et b alors on peut facilement montrer qu'il divise aussi r, de sorte que pgcd$(a;b)$ = pgcd$(b;r)$.
- si $r \neq 0$ alors on recommence alors en prenant a égal à b et b égal à r ;
- si $r = 0$ alors le pgcd des nombres de départ est le dernier b qu'on a utilisé.

Voyons la méthode sur un exemple :

- on divise 1050 par 280 : $1050 = 1 \times 770 + 280$
- on divise 770 par 280 : $770 = 2 \times 280 + 210$
- on divise 280 par 210 : $280 = 1 \times 210 + 70$
- on divise 210 par 70 : $210 = 3 \times 70 + 0$
- le reste est nul, le dernier diviseur est 70 et c'est le pgcd des deux nombres de départ.

Programmer cette fonction `pgcd` de manière récursive.