Chapitre 1

Récursivité

1 Le formidable principe de la récursivité

1.1 S'appeler et se terminer

On dit qu'une fonction est **récursive** lorsque dans sa définition on appelle (une ou plusieurs fois) cette même fonction.

Cela pose un problème : si la fonction ne cesse de s'appeler, comment cela peut-il se terminer ?

Exemple 1

> La fonction suivante est incorrecte :
>
> ```
> def f():
>     return f()
> ```
>
> l'appel `f()` provoque *a priori* une boucle infinie.

Exemple 2

> ```
> def f(n : int):
>     return f(n-1)
> ```
>
> Quelle que soit la valeur de `x`, l'appel `f(x)` provoque également *a priori* une boucle infinie : `f(10)` appelle `f(9)` qui appelle `f(8)` *et cætera*.

Syntactic Sugar

« sucre syntaxique » d'un langage de programmation : ensemble des règles de syntaxe qui ont été ajoutées pour rendre le code plus facile à lire et à écrire.

```
def f(n : int) -> int:
    return 0 if n <= 0 else f(n - 1)
```

1.2 Un premier exemple digne d'intérêt

Soit $n \in \mathbb{N}$. *Factorielle n* est le produit de tous les entiers non-nuls inférieurs ou égaux à n, et se note $n!$.

- $1! = 1$;
- $2! = 1 \times 2 = 2$;
- $10! = 1 \times 2 \times \ldots \times 10 = 3\,628\,800$;
- $0! = 1$ par convention.

À l'aide d'une boucle for

Python

```
def factorielle( n : int) -> int:
result = 1
for i in range(1, n + 1):
    result *= i
return result
```

L'algorithme est dit **itératif** car il utilise une boucle.

De manière récursive

Pseudocode

```
fonction factorielle(n : entier naturel) -> entier naturel
    si n = 0 alors
        renvoyer 1
    sinon
        renvoyer n * factorielle(n -1)
    fin si
```

D'une certaine manière, c'est plus « élégant ».

Exercice 1

- Programmer la fonction `factorielle` en PYTHON de manière récursive, et tester cette fonction en vérifiant que `factorielle(10)` renvoie bien 3 628 800.
- Mettre du *Syntactic Sugar* là-dedans !

2 Formidable, mais pourquoi ?

Principe

- On veut écrire une fonction `f` qui résout un problème dépendant d'un entier naturel `n` ;
- on examine le cas ou `n` vaut 0 (ou 1), correspondant à un problème très simple, que l'on sait résoudre ;
- on suppose que l'on sait résoudre le problème pour un entier `n-1`, à l'aide de la fonction `f`, on regarde alors les opérations à effectuer pour passer de ce problème au problème de taille `n` ;
- on programme alors la fonction `f` de manière récursive.

2.1 L'exemple des poignées de main

Nous l'avons traité en activité préparatoire :

Python

```
def f(n : int) -> int
    return 0 if n == 0 else n - 1 + f(n - 1)
```

2.2 Un exemple de récursion double

La célèbre suite de Fibonacci, notée F, est définie ainsi :

- Ses deux premiers termes F_0 et F_1 valent 1;
- On construit chaque terme suivant en faisant la somme des deux précédents.

De proche en proche, on calcule :

- $F_2 = 1 + 1 = 2$
- $F_3 = 2 + 1 = 3$
- $F_4 = 3 + 2 = 5$
- *et cætera*

Et plus généralement

$$F_n = \begin{cases} 1 & \text{si } n = 0 \text{ ou } n = 1 \\ F_{n-1} + F_{n-2} & \text{sinon} \end{cases}$$

Pseudocode

```
fonction fibonacci(n : entier naturel) -> entier naturel
    si n < 2 alors
        renvoyer 1
    sinon
        renvoyer fibonacci(n - 1) + fibonacci(n - 2)
    fin si
```

Exercice 2

> Programmer la fonction `fibonacci` en PYTHON et vérifier que `fibonacci(30)` vaut 1 346 269.

2.3 Remarques

- La récursivité est un des concepts *fondamentaux* de l'informatique.
  On dit que c'est un **paradigme de programmation**.
- Certains algorithmes se programment naturellement de manière récursive.
- Certaines **structures de données** se définissent également de manière récursive.

3 La « magie » a un coût

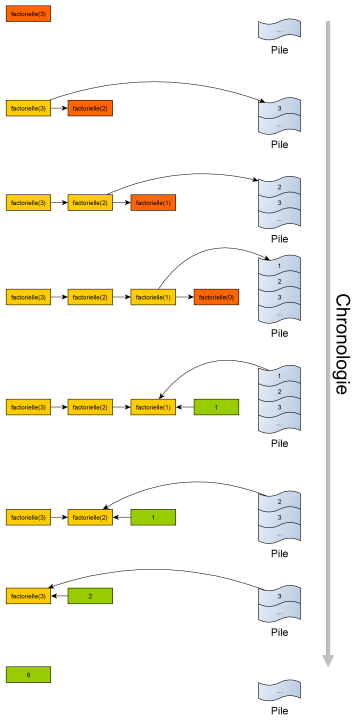
3.1 La pile d'appels

Que se passe-t-il réellement en machine lorsqu'on évalue la fonction récursive `factorielle(3)` ?

PYTHON possède une **pile** : c'est une structure de données simple, qui permet d'empiler et de dépiler des données un peu comme on empile des assiettes.

Lors de chaque appel récursif, une valeur est empilée en attendant le résultat de l'appel.
Lorsqu'il y a beaucoup d'appels récursifs *imbriqués*, la taille de la pile augmente.
Pour éviter qu'elle sature, PYTHON fixe la limite des appels récursifs *imbriqués* à 999. Dès que l'on dépasse cette limite on obtient un message d'erreur :

```
RecursionError: maximum recursion depth exceeded in
comparison
```



On peut fixer la taille de la pile Pour aller jusqu'à 10 000 appels récursifs au maximum (par exemple).

Python

```python
import sys

sys.setrecursionlimit(10_000)
```

Exercice 3

Reprendre la fonction récursive `fibonacci` et utiliser le site `http://pythontutor.com/visualize.html` pour visualiser la pile d'appels lors de l'évaluation de `fibonacci(5)`.

Quel commentaire peut-on faire ?

4 Exercices

Exercices de base

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 4

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)`.

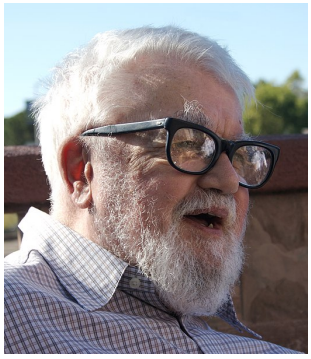
Exercice 5

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 6 : récursion imbriquée



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

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.

Exercice 7 : 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 8 : 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 9 : 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]`.

Avec de l'arithmétique

Définition : division euclidienne dans $\mathbb{N}$

Soient A et B deux entiers naturels, et $B \neq 0$. Il existe deux nombres uniques Q et R (vérifiant $0 \leqslant R < B$) tels que l'on puisse écrire

$$A = Q \times B + R$$

C'est exactement la division que l'on a apprise à l'école primaire (celle où l'on s'arrête aux nombres entiers) :

$$\begin{array}{c|c} A & B \\ \hline R & Q \end{array}$$

- A est appelé le *dividende*;
- B est le *diviseur*;
- Q est le *quotient*;

- R est le *reste*, il est *impérativement* plus petit que B.

En PYTHON on obtient `Q` en évaluant `A // B` et `R` en évaluant `A % B`, cette dernière opération se lit « `A` modulo `B` ».Voici un exemple

Python

```
>>> 22 // 7
3
>>> 22 % 7
1
```

Exercice 10

1. En PYTHON, écrire une fonction `units_digit` qui
   - en entrée prend un `int` positif;
   - renvoie un `int` qui est son chiffre des unités.
2. De même écrire une fonction `hundreds_digit` pour le chiffre des centaines.
3. De même pour une fonction `thousands_digit` qui renvoie le chiffre des milliers.

Exercice 11

Écrire une fonction récursive `decimal_length` basée sur `//` et/ou `%` qui

- en entrée prend un `int` positif;
- renvoie un `int` qui est le nombre de chiffres de l'écriture décimale de ce nombre.

Exercice 12

En s'inspirant de l'exercice précédent : Écrire une fonction récursive `binary_length` basée sur `//` et/ou `%` qui

- en entrée prend un `int` positif;
- renvoie un `int` qui est le nombre de chiffres de l'écriture binaire de ce nombre.

Exercice 13

On considère le procédé suivant :

- soit $m \in \mathbf{N}$ un entier écrit en écriture décimale $m = (a_p \cdots a_1 a_0)_{10}$,
  par exemple $m = 31\,976$;

- on « coupe » cette écriture en deux au niveau des unités : avec m on forme $m_1 = (a_p \cdots a_1)_{10}$
  et $m_2 = (a_0)_{10}$,
  pour notre exemple $m_1 = 3\,197$ et $m_2 = 6$;

- On calcule $m' = m_1 - 2m_2$,
  pour notre exemple cela donne $m' = 3\,197 - 2 \times 6 = 3185$

1. à l'aide de `//` et/ou `%` écrire une fonction `f` qui
   - en entrée prend un `int` positif `m` ;
   - en sortie renvoie `m'`.
2. En fait, la fonction f donne un critère de divisibilité par 7 pour un entier m :
   - si $m \leqslant 70$ alors s'il appartient à -7 ; 0 ; 7 ; 14 ; 21 ; 28 ; 35 ; 42 ; 49 ; 56 ; 63 ; 70, m est divisible par 7, sinon il ne l'est pas ;
   - sinon on regarde si $f(m)$ est divisible par 7.

   Pour notre exemple on obtient

   $$31\,976 \mapsto 3\,197 - 2 \times 6 = 3\,185 \mapsto 318 - 2 \times 5 = 308 \mapsto 30 - 2 \times 8 = 14$$

   et on en conclut qu'il est divisible par 7.

   Programmer une fonction récursive `is_divisible_by_7` qui
   - en entrée prend un `int` positif ;
   - en sortie renvoie `True` ou `False` selon que l'entier est divisible par 7.

   Cette fonction utilisera la fonction `f` définie précédemment.

Exercice 14 : algorithme d'Euclide récursif

Soient n et p deux entiers strictement positifs. On note $\text{pgcd}(a;b)$ 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 $a \leqslant b$

1. on écrit la division euclidienne de a par b : $a = q \times b + r$ avec $r < b$;
2. si un entier divise a et b alors on peut facilement montrer qu'il divise aussi r, de sorte que $\text{pgcd}(a;b) = \text{pgcd}(b;r)$.
3. si $r \neq 0$ alors on recommence alors en prenant a égal à b et b égal à r;
4. 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.

À savoir faire absolument

Exercice 15 : lire un code

On considère la fonction suivante :

```
def mystery3(lst: list) -> bool:
    """lst est une liste d'int non vide"""
    if len(lst) > 1:
        if lst[0] > lst[1]:
            return False
        else:
            return mystery3(lst[1:])
            # on rappelle que lst[1:] désigne la liste composée
            # de lst[1],lst[2], etc jusqu'au dernier élément de
              ↪ lst
    else:
        return True
```

1. Calculer `mystery3([5])`;
2. Calculer `mystery3([5, 7])`;
3. Calculer `mystery3([5, 1])`;
4. Calculer `mystery3([5, 10, 8])`;
5. Que fait donc cette fonction ?

Exercice 16 : produire un code récursif

1. Écrire une fonction récursive `maxi` qui
   - en entrée prend une liste d' `int`;
   - renvoie le maximum de cette liste.

   Vous pouvez utiliser la fonction `max`.
2. Écrire une fonction récursive `reverse` qui
   - en entrée prend un `str`;
   - renvoie ce `str` « à l'envers »

   Par exemple, `reverse("salut")` devra renvoyer `"tulas"`.

Excercices supplémentaires

Exercice 17* : 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 (*).

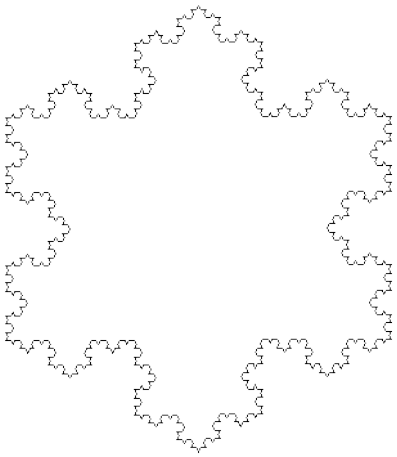
On pourra utiliser la fonction `sum`.
Par exemple `sum(n for n in range(10))` calcule la somme $0 + 1 + \ldots + 9$.

Exercice 18* : 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 19* : 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)`.