TP : Récursivité

Instruction générale : hormis pour les exercices corrigés collectivement, vous ferez valider votre travail par l'enseignant.

1 Récursivité

1.1 Récursivité

Une **fonction récursive** (ou un algorithme récursif) est une fonction dont la définition contient un appel à elle-même.

Exemple : La fonction suivante permet d'obtenir la somme de tous les entiers de 1 à n : $1+2+\cdots+n$.

```
def somme(n : int) -> int :

    if n == 0 :
           return 0
    else :
           return n + somme(n-1)
```

On peut voir ici que la fonction `somme` s'appelle elle-même dans le second `return`.

Remarque : La récursivité est très utilisée dans le paradigme fonctionnel car elle permet de remplacer les boucles.

1.2 Terminaison et cas de base

Une fonction récursive s'appelant elle-même, on retrouve une problématique commune avec la boucle `while` : la question de la **terminaison** des appels récursifs. En effet, il y a ici un fort risque de régression à l'infini. Une fonction récursive doit donc toujours comporter un **cas de base** qui met fin à la récursion.

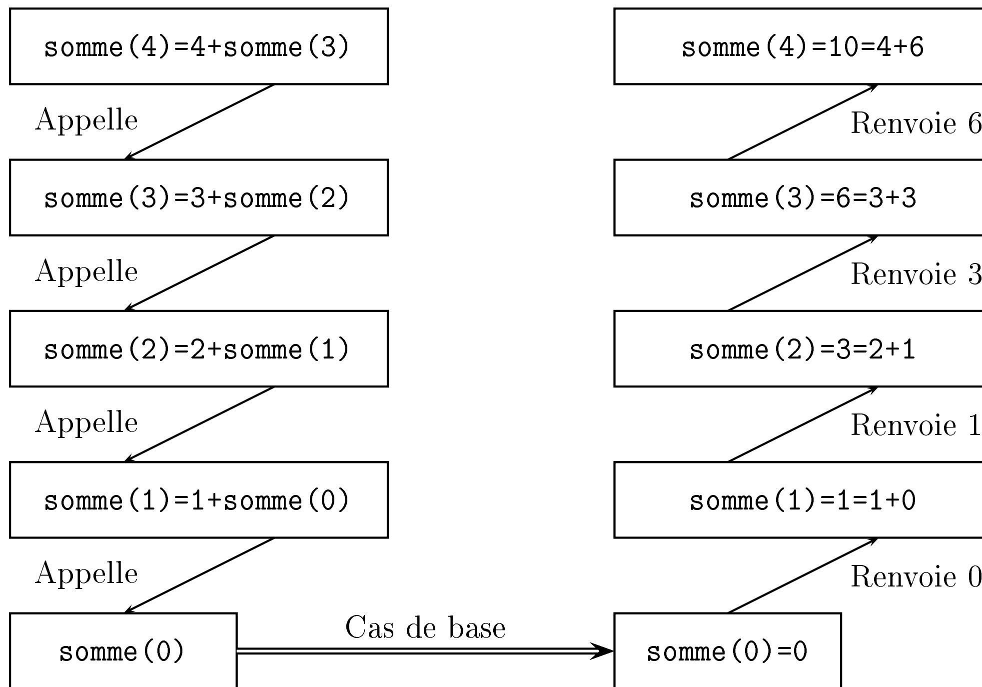
Bonne pratique : lorsque l'on commence à programmer une fonction récursive, on programme en premier le cas de base.

Exemple : en reprenant l'exemple précédent, on a pour cas de base :

```
if n == 0 :
    return 0
```

Le test ici sert précisément à déterminer si on est dans ce cas de base ou non. Si oui, la fonction renvoie 0 ; sinon, elle fait appel à elle-même jusqu'à revenir au cas de base.

Si on appelle la fonction `somme` avec $n = 4$, on obtient alors le processus ci-dessous.



1.3 Méthode de programmation d'une fonction récursive

La programmation d'une fonction de façon récursive s'effectue selon les étapes suivantes :

1. Déterminer les paramètres de la fonction comme on le ferait pour une fonction ordinaire.
2. Vérifier si l'algorithme est sécable (divisible) en une fonction effectuant des opérations à partir d'appels plus simples à cette même fonction.
3. Déterminer le cas de base permettant l'arrêt de la récursion.
4. Reconstituer la valeur de retour à partir des appels récursifs.

Exemple : on reprend la somme des entiers de 1 à n. En l'observant, on remarque que :

$$S(n) = \underbrace{1 + 2 + \cdots + n - 1}_{=S(n-1)} + n = S(n-1) + n.$$

Autrement dit, la suite S est définie par récurrence. Récurrence mathématique et récursivité informatique sont deux notions très proches. De manière générale, toute suite définie par récurrence peut se traduire en une récursion informatique : il suffit pour cela d'utiliser la formule de récurrence pour la récursion et le cas de base correspond au(x) premier(s) terme(s) de la suite.

2 Exercices

2.1 Un peu de mathématiques

Exercice 1. [Factoriel] On définit le factoriel d'un nombre entier n comme le produit des entiers de 1 à n. On le note $n!$ et par convention $0! = 1$.

$$n! = 1 \times 2 \times \cdots \times n.$$

Programmer une fonction `factoriel` calculant récursivement le factoriel d'un nombre entier.

Exercice 2. [Puissance] Programmer une fonction `puissance` calculant récursivement la puissance n d'un nombre réel x.

Exercice 3. [Méthode de Héron] La méthode Héron (mathématicien grec de l'antiquité) permet d'approcher la valeur numérique de la racine carrée d'un nombre réel positif a. Il s'agit d'une suite définie par récurrence convergeant vers $\sqrt{a}$:

$$\left\{\begin{array}{rcl} x_{n+1} & = & \dfrac{x_n + \frac{a}{x_n}}{2},\ n \in \mathbb{N}^* \\ x_0 & > & 0. \end{array}\right.$$

Le premier terme x_0 doit être non nul sans plus de contrainte. Toutefois, il est judicieux de le prendre le plus proche possible de $\sqrt{a}$; on pourra par exemple prendre sa partie entière $E(\sqrt{a})$ (comment la déterminer sans connaître $\sqrt{a}$?).

Écrire une fonction récursive mettant en œuvre la méthode de Héron. Elle devra prendre en entrée un réel positif dont on veut calculer la racine carrée et un entier n afin de calculer le n-ième terme de la suite.

Exercice 4. [Partitions d'un entier] Une partition d'un entier est une écriture de cet entier sous la forme d'une somme d'entiers qu'ils lui sont inférieurs ou égaux (une partition peut ne contenir qu'un seul entier). On ne tiendra pas compte de l'ordre des sommes.

Partitions de 3 : on a trois partitions :

1. 3=3 ;
2. 3=2+1 ;
3. 3=1+1+1.

Partitions de 4 : on a cinq partitions :

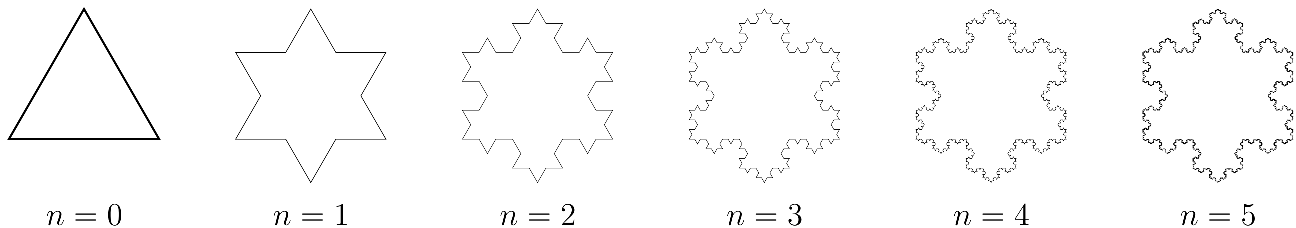
1. 4=4 ;
2. 4=3+1 ;
3. 4=2+2 ;
4. 4=2+1+1 ;
5. 4=1+1+1+1.

Programmer une fonction récursive déterminant les partitions d'un entier naturel.

2.2 Un peu de fractales

Exercice 5. [Le flocon de Koch] Le flocon de Koch est une figure fractale, i.e. un motif se répétant à l'infini à partir d'un triangle équilatéral. Chacun des côtés du triangle se voit découper en trois sections égales et on construit sur la section centrale un nouveau triangle équilatéral avant de la supprimer. La figure ci-dessous représente ces opérations sur l'un des cotés du triangle.

En réitérant à plusieurs reprises ce processus sur chacun des nouveaux côtés, on obtient la suite de figures suivantes (avec n nombres de fois où le processus a été exécuté).



À partir de la quatrième figure, on reconnaît la forme d'un flocon de neige que l'on nomme alors flocon de Koch. Le but de cet exercice est de programmer une fonction récursive traçant des flocons de Koch à l'aide du module `turtle`.

```
import turtle

turtle.forward(distance) # fait avancer la tortue d'une certaine distance
turtle.right(angle) # fait tourner la tortue à droite d'un certain angle en degré
turtle.left(angle) # fait tourner la tortue à gauche d'un certain angle en degré
```

1. Programmer une fonction permettant de tracer récursivement un côté d'un flocon de Koch. Elle devra prendre en entrée la longueur initiale du côté (celle du triangle équilatéral) et le nombre d'étapes de modification.
2. Programmer une fonction traçant à partir de la précédente fonction un flocon de Koch.

Exercice 6. [Les flammes du dragon] Les flammes du dragon sont une figure fractale s'obtenant à partir d'une liste de booléens et des instructions suivantes dans le module Turtle.

1. Avancer d'une distance fixe.
2. Si `booléen = True`, tourner à gauche, sinon à droite (ou inversement).
3. Réitérer sur l'intégralité de la liste.

La liste des booléens s'obtient récursivement de la façon suivante :

Initialisation : Le premier booléen est `False`.

Hérédité : Pour passer de la liste d'ordre n à celle d'ordre $n+1$, il faut concaténer à la liste d'ordre n la liste `[False]` puis le symétrique de la dite liste dans laquelle on inverse les valeurs booléennes.

Ordre 0 : `liste_0 = [False]`.

Ordre 1 : on prend le symétrique de `liste_0 = [False]` et on inverse sa valeur, ce qui donne `[True]`. On effectue alors la concaténation suivante :

```
liste_1 = liste_0 + [False] + symetrique(inversion(liste_0)) = [False,
False, True].
```

Ordre 2 : on prend le symétrique de `liste_1 = [False, False, True]`, ce qui donne

```
[True, False, False].
```

On inverse ensuite cette liste, obtenant ainsi

```
[False, True, True].
```

Reste alors à effectuer la concaténation de `liste_1` avec `[False]` et la liste ci-dessus pour obtenir

```
liste_2 = [False, False, True, False, False, True, True].
```

Ordre n+1 : si la liste d'ordre n est déjà construite, on obtient celle d'ordre $n+1$ en faisant

```
liste_n + [False] + symetrique(inversion(liste_n)).
```

Programmer une fonction `flammes_du_dragon` affichant dans `turtle` la figure fractale construite à partir des règles ci-dessus. Pour un rendu optimal, on affichera les segments tracés sur fond noir avec une couleur aléatoirement choisie parmi trois (jaune, orange, rouge par exemple).

2.3 Un peu de chaînes de caractères

Exercice 7. [Palindrome] Un mot est un palindrome s'il possède une symétrie de lecture, autrement dit s'il se lit de la même façon en partant depuis la fin que depuis le début. Par exemple, « kayak » est un palindrome mais pas « crocodile ». Écrire une fonction récursive déterminant si un mot est un palindrome ou non.

Exercice 8. [Recherche] Programmer une fonction récursive déterminant si un caractère appartient à une de caractères ou non. On pourra dans un deuxième temps faire en sorte que la fonction renvoie en plus la position de la première apparition du caractère si celui-ci est présent (*indication :* on pourra considérer une valeur par défaut pour l'un des arguments de ladite fonction).

Exercice 9. [Anagrammes] Deux mots sont des anagrammes s'il est possible d'écrire l'un en utilisant tous les caractères de l'autre. Par exemple, « chien » et « niche ». Il est possible de déterminer si deux chaînes de caractères sont des anagrammes après les avoir triées. Toutefois, le but de cet exercice est de le faire avec une approche récursive :

- Si les deux chaînes ont longueur 1, on renvoie `True` ou `False` en fonction de l'égalité des deux chaînes.
- Sinon, on teste la présence du premier caractère de la première chaîne dans la seconde.
  - S'il y est présent, on réitère le test sur les deux chaînes desquelles on aura retiré le caractère précédemment testé.
  - Sinon, on renvoie `False`.

Programmer une fonction testant si deux chaînes de caractères sont des anagrammes ou non.

3 Ressources supplémentaires

- Vidéo sur la récursivité.
- Vidéo sur la récursivité et les fractales.
- Vidéo sur la récursivité et les tours de Hanoï.