SEQ2 - La récursivité et ses utilisations

B - Cours - La récursivité

1 - Exemple : les moutons 🐑

Considérons la définition suivante d'un troupeau de moutons :

troupeau = *Si la taille du troupeau est de 1* : un mouton 🐑

Sinon : un mouton 🐑+ un troupeau

Appliquons la définition successivement sur un troupeau de 4 moutons *(on ne sait pas a priori qu'il y en a 4**)* :

Codons-le en Python, avec une fonction *troupeau* qui évalue le nombre de moutons (le nombre d'éléments) dans la liste Python *t* :

```
def troupeau(t):
    if len(t) == 1:
        return 1
    else:
        return 1 + troupeau(t[1:len(t)])
```

2 - Définition

Une **fonction récursive** est une fonction qui s'appelle elle-même. Comme nous l'avons vu avec les moutons, cela peut suffire à la définir !

Pour éviter que cette fonction s'appelle à l'infini, il faut :

- qu'elle *ait* un **état trivial** qui amène à son arrêt,
- qu'elle *se ramène* vers cet état trivial.

Une fonction récursive s'appelle donc elle-même sur un problème « plus petit », et qui se réduit progressivement.

3 - Fonctionnement d'une pile d'exécution

Cas général

Lors de l'appel d'une fonction, le système sauvegarde différents paramètres comme ses arguments et les variables utilisées à l'intérieur de celle-ci. C'est son ***contexte d'exécution***. Cela permet d'interrompre l'exécution d'une fonction lorsque celle-ci en appelle d'autre, et de reprendre son exécution ensuite.

Exemple :

```
def h(x) :
    return x + 1

def g(x)
    return h(x) * 2

def f(x) :
    return g(x) + 1
```

Représentation de la pile pour l'appel de f(5) :

Cas récursif

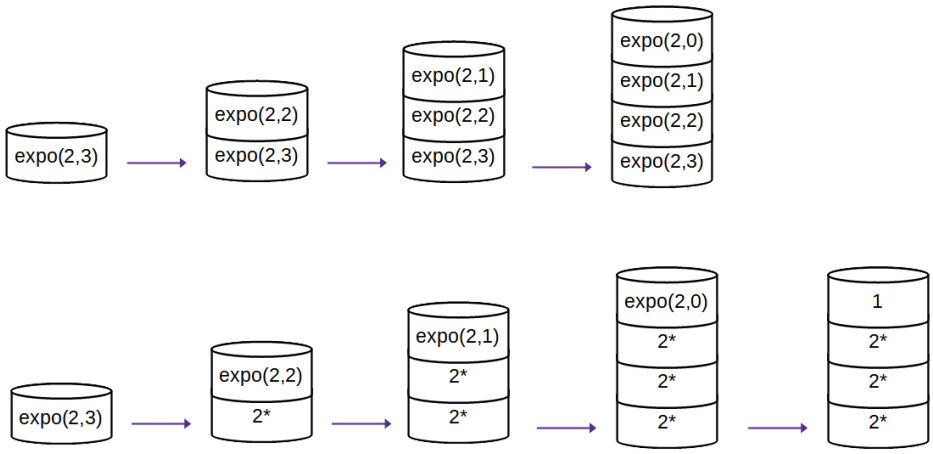
Une **pile d'exécution** est donc utilisée lors de l'utilisation d'une fonction récursive pour stocker le contexte d'exécution de ses différents appels.

Prenons un exemple classique du calcul de a^n :

```
def expo(a, n) :
    if n == 0 :
        return 1
    else :
        return a*expo(a, n-1
```

Écrire le calcul de expo(2,3) :

Le schéma suivant explique le processus en terme de pile d'exécution :



La pile d'exécution a une **taille maximale**. Lorsque la pile **déborde** (par défaut au bout de 1000 appels – *stack overflow*), Python renvoie le message suivant :

```
RecursionError: maximum recursion depth exceeded while calling a Python object
```

4 - Les paradigmes

Le paradigme **récursif** s'oppose généralement au paradigme **itératif**. Ce dernier est une sous-catégorie du **paradigme impératif** et fait référence à l'utilisation de boucles.

Avec la récursivité, on peut se passer de boucles. C'est cette méthode qui les remplace lorsque l'on programme en utilisant le paradigme **.....**.

B - Exercices - La récursivité

Cherchez vos solutions par écrit avant de les implémenter.

Exercice 1 :

On veut trouver le maximum d'une liste Python d'entiers positifs.

1. En proposer une version itérative *maximum_it*, qui renvoie la valeur du maximum.
2. Nous voulons en trouver une version récursive.
   a. Le cas trivial correspond au cas où la longueur de la liste Python est de 1. Quel est alors le maximum ?
   b. Sur quel paramètre se fera l'appel récursif ? *(il doit progressivement ramener l'algorithme vers le cas trivial)*
   c. En déduire une version récursive *maximum_rec*.

Exercice 2 :

La suite de Fibonacci est définie comme suit :

- u0 = 0 et u1 = 1
- Pour tout n entier > 1 : u(n) = u(n-1) + u(n-2)
  1. Écrire une fonction récursive *fib_rec* qui donnera le nième terme de la suite de Fibonacci. Cette fonction prendra en paramètre l'entier n.
  2. Représenter les appels récursifs de fib_rec(5) sous la forme d'une structure hiérarchique, puis sous la forme d'une pile.

B - TD - La récursivité : Les tours de Hanoï

Introduction

Un problème classique se résolvant particulièrement bien avec la récursivité est celui des **tours de Hanoï**. C'est un jeu de stratégie, dont l'objectif est de déplacer une tour complète composée de disques de taille différente d'une tige à une autre. On dispose pour cela d'une tige intermédiaire, et on ne peut déplacer qu'un seul disque à la fois, sachant qu'un disque ne peut pas être disposé sur un disque de taille inférieure.

1. Résolution théorique

Résolution pour 2 disques :

Résolution pour n disques :

Assimilons le disque rouge utilisé précédemment à une pile de disques. Résoudre ce problème pour *n* disques consiste à effectuer les opérations suivantes :

Ces trois étapes sont récapitulées dans la partie gauche du tableau ci-dessus.

Envoyer les n-1 premiers disques sur la place du milieu	**Résoudre le problème pour n-1 disques avec le départ ..... et l'arrivée .....**
Envoyer le disque n à droite	Envoyer le disque n à droite
Envoyer les n-1 premiers disques sur la place de droite	Résoudre le problème pour n-1 disques avec le départ ..... et l'arrivée .....

Être capable de déplacer une pile d'une tige vers une autre consiste à résoudre le problème initial en partant d'une certaine tige et en arrivant sur une autre. Compléter la partie droite du tableau ci-dessus avec les tiges de départ et d'arrivée.

Adapté de Stephen VAN ZUIJLEN

2. L'algorithme

Pour résoudre le problème pour *n* disques, on fait appel au problème pour *n-1* disques (2 fois). Il est donc pertinent d'utiliser un algorithme récursif.

Compléter celui-ci, écrit sous la forme de pseudo-code ci-dessous :

```
fonction Hanoi(n, dep, inter, arr)
ENTREES :
    n : entier représentant le nombre de disques à déplacer
    dep, inter, arr : caractères indiquant respectivement les sommets de départ,
 intermédiaire et arrivées
    Si n est égal à 0 Alors
        retourner Rien
    Sinon
        Hanoi(.....)
        afficher le déplacement depuis *dep* jusqu'à *arr*
        Hanoi(.....)
    Fin si
```

3. Implémentation en Python

Implémenter cette fonction en Python, et la rédiger ci-dessous :

N.B. : pour l'affichage, on peut utiliser la ligne suivante :

```
print("Déplacer le disque ", n, "de la tige ", dep, " vers la tige ", arr)
```

4. Affichage du jeu

Pour un meilleur affichage des différentes étapes, complétez le code fourni par votre professeur.

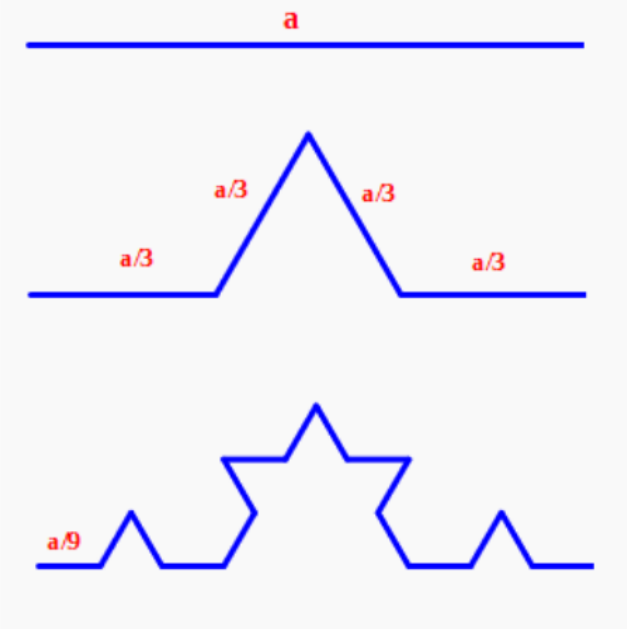
Adapté de Stephen VAN ZUIJLEN

B - TP 2 - La récursivité : dessiner des flocons

Introduction

Le flocon de Von Koch est une des premières courbes fractales à avoir été décrite. Elle a été inventée en 1904 par le mathématicien suédois Helge von Koch. Voici une partie de ce flocon :

1. Méthode de construction



- On commence par un segment de longueur *a*.
- On coupe ce segment en 3 parties égales.
- On remplace le segment central par un triangle équilatéral de côté *a/3*.
- Chaque segment de longueur *a/3* est lui-même découpé en 3 parties égales (chacune de longueur *a/9*).
- On remplace chaque partie centrale par un triangle équilatéral de côté *a/9*.
- Etc.

Il est décidé à l'avance combien de fois le redécoupage se fera.

Les triangles dessinés sont des triangles équilatéraux : vous pouvez en déduire les angles qu'il faudra utiliser.

2. Utiliser *turtle*

Le module *turtle* de Python permet de réaliser des figures.

Le code suivant permet de tracer les lignes de bases qui seront utiles à ce TP. Analysez-le et testez-le.

```
import turtle as t

# déplace la tortue aux coordonnées
t.penup()
t.goto(-100,0)
t.pendown()
t.hideturtle() # on cache la tortue
t.speed(0) # on accélère la tortue
t.color('blue')
t.pensize(3)
t.forward(100)
t.left(60)
t.forward(100)
t.right(120)
t.forward(100)
t.left(60)
t.forward(100)
```

_____

Adapté de Stephen VAN ZUIJLEN

3. L'algorithme

Compléter l'algorithme suivant, qui permet de tracer une partie du flocon :

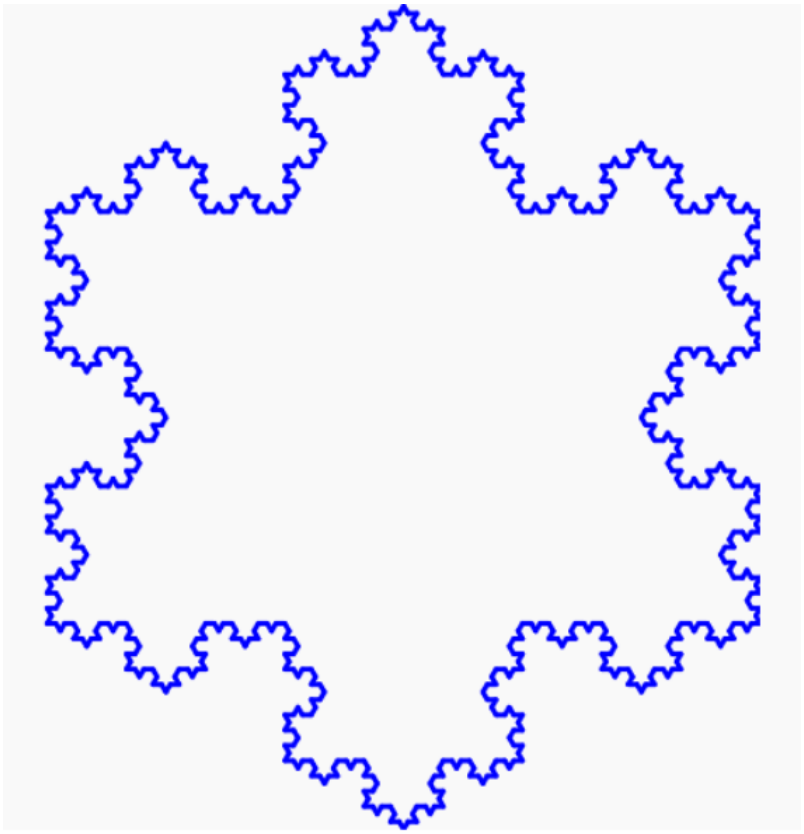
```
fonction *fractale*(n, cote):
    ENTREES :
        *n* : entier indiquant le nombre d'itération à faire
        *cote* : la longueur du segment initial

    Si n est égal à 0 alors
        On trace le segment de longueur cote
    Sinon
        On appelle la fonction fractale avec les paramètres __________
        On tourne de _____ degrés sur la _______________

    ____________________________________________________
    ____________________________________________
    ____________________________________________________
    ____________________________________________
    ____________________________________________________
    Fin si
```

4. Implémentation

1. Implémenter l'algorithme en Python.
2. Écrire la fonction *flocon*, qui permet d'afficher le flocon en entier :



Adapté de Stephen VAN ZUIJLEN