0.1 Introduction

0.1.1 Activités

Activité 1 Testez le code suivant avec différentes valeurs. Que fait la fonction `mystere` ?

```
def mystere(x, n):
    if n==1:
        return x
    else:
        return x*mystere(x, n-1)
```

Questions:

1. Comment peut-on être sûr que l'appel à la fonction `mystere` s'arrête ?
2. Lancer le debugger et observer la *pile* d'appel des fonctions.
3. Que renvoie cette fonction ?

Activité 2 Voici un script, étudiez-le et répondez aux questions posées:

```
import turtle as t
couleurs = ["blue", "green",
            "yellow", "orange",
            "red", "purple"]
def dessin(i):
    if i < 180:
        t.color(couleurs[i%6])
        t.forward(i)
        t.right(59)
        dessin(i+1)
```

Questions:

1. Que peut-on dire de l'appel de fonction dans la fonction `dessin` ?
2. Comment peut-on être sûr que l'appel à la fonction `dessin` s'arrête ?
3. Avec quelle valeur de i appeler la fonction `dessin()` ?
4. Tapez ce code et observez la pile d'appel de ces fonctions avec un interpréteur et un debugger.

Activité 3 La suite de Fibonacci est la suite numérique définie par:

$$\begin{cases} F_0 = F_1 = 1 \\ \forall n \geqslant 2,\ F_{n+2} = F_{n+1} + F_n. \end{cases}$$

1. Calculer F_2, F_3, F_4, F_5.
2. Ecrire une fonction itérative `fibonacci(n)` qui renvoie la valeur de F_n.
3. Cette relation de récurrence exprime le nombre u_n à l'ordre n en fonction des nombres u_{n-1} et u_{n-2} , respectivement à l'ordre n-1 et n-2. Ainsi, on définit u_n directement en fonction de u_{n-1} et de u_{n-2}. Cette écriture est compacte et très expressive: elle ne fait notamment pas intervenir de boucle comme dans le code de la fonction que vous avez programmée ci-dessus.

   Il serait donc intéressant de définir la fonction `fibonacci()` de manière aussi simple dans le langage algorithmique qu'en mathématiques.

   Ecrire une fonction récursive `fibo_rec(n)` permettant d'afficher F_n.

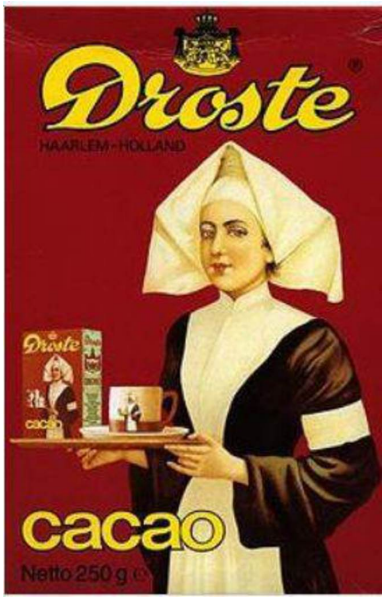
0.2 La récursivité

Un paradigme de programmation En programmation, nombreux sont les problèmes qu'on résout en répétant plusieurs fois des séquences d'instructions.

On peut aborder certains problèmes simplement en résolvant un sous problème de même nature, mais plus simple.

Cette méthode de résolution s'appelle la récursivité. La possibilité d'écrire des algorithmes récursifs donne une souplesse considérable et parfois décisive. Il peut être très délicat d'écrire une méthode itérative pour résoudre un problème car sa nature est récursive.

Effet Droste Le nom *Droste* vient de l'entreprise éponyme néerlandaise, dont le logotype a longtemps été une religieuse à cornette portant un plateau sur lequel était posé un paquet de chocolat Droste, figurant lui-même une religieuse à cornette.



Boîte de chocolat Droste

Dans cet effet, on montre une image à l'intérieur de laquelle apparaît l'image entière, l'image réduite contenant à son tour l'image, et ainsi de suite. Cette succession d'images est récursive.(*wikipedia*)

0.3 Définition

On appelle **récursive** toute fonction ou algorithme qui s'appelle elle-même. Ce concept est très proche de celui de la récurrence en mathématique.

Les algorithmes récursifs permettent de travailler sur des structures de données définies récursivement comme les chaînes de caractères, les listes ou les arbres.

Exemple Les règles de dérivation sur les fonctions numériques d'une variable réelle font appels à la récursivité. Pour dériver une somme, il est nécessaire de **faire appel à la fonction dérivée** de u et v.

Règle de dérivation

- (u+v)=u+v
- (ku)'=ku'
- (uv)=uv+uv
- $\frac{u}{v} = \frac{u'v - v'u}{v^2}$

fonction `derivee`

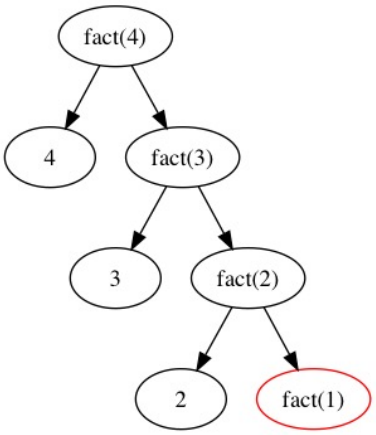
1.**Si** f est une fonction élémentaire de base
2. Renvoyer sa dérivée
3.**Sinon si** $f = u + v$
4. Renvoyer `derivee`(u)+`derivee`(v)
5.**Sinon si** $f = uv$
6. Renvoyer `derivee`(u)×v+u×`derivee`(v)
7. etc....

Exemple L'exemple le plus connu est celui du calcul de la factorielle. Rappelons que:

$$n! = n \times (n-1) \times (n-2) \times ..... \times 2 \times 1.$$

L'expression de $(n-1)!$ apparait clairement dans celle de $n!$. On peut donc simplifier le problème du calcul de $n!$ en faisant appel au calcul de $(n-1)!$ et ainsi de suite. L'algorithme ci-dessous calcule $n!$ récursivement.

```
1  def fact(n):
2      """
3      retourne la factorielle de n.
4      :param n (int):
5      :CU: n>=0:
6      """
7      if n ==1:
8          return 1
9      else:
10         return n*fact(n-1)
```



Arbre des appels pour `fact(5)`

Le cas `n==1` en rouge sur l'arbre des appels est appelé **cas terminal**.

Le premier bloc (lignes 7 et 8) constitue ce que l'on appelle la base récurrente qui correspond à la situation que l'algorithme sait traiter sans faire appel à lui-même. C'est l'écriture informatique de l'initialisation dans un raisonnement par récurrence.

Le second appelé autodéfinition concerne la partie de l'algorithme qui fait référence à lui même, ici un unique appel récursif (ligne 10). Cela correspond à l'hérédité dans un raisonnement par récurrence en mathématiques.

Voici l'ensemble des appels, avec la phase de remontée, une fois le cas terminal atteint:

```
factorielle(5)
       └── 5*factorielle(4)
                  └── 5*(4*factorielle(3))
                             └── 5*(4*(3*factorielle(2)))
                                        └── 5*(4*(3*(2*factorielle(1))))
                                                   5*(4*(3*(2*(1))))
                                        5*(4*(3*(2)))
                             5*(4*(6))
                  5*(24)
       120
```

Nous pouvons donc résumer l'écriture d'un programme récursif simple comme cela:

```
if (cas simple):
    (solution immédiate)
else:
    (solution récursive,
    impliquant un cas plus simple que le problème original)
```

Exercice

1. Ecrire les appels de la fonction factorielle pour n=4..
2. Implémenter cette fonction et observer la *pile* d'appel avec le debugger de Thonny.

Note : Cette liberté d'écriture offerte au programmeur repose sur l'existence d'une **pile**. À chaque appel de la fonction, un bloc d'activation contenant les paramètres d'appel de la fonction, ses

variables locales et sa valeur de retour est empilé. La fonction travaille toujours avec les variables du bloc d'activation au sommet de la pile.

Si elle est appelée à nouveau récursivement, un nouveau bloc d'activation est **empilé** et la fonction travaillera avec les variables de ce nouveau bloc et ainsi de suite *jusqu'à ce que sa valeur de retour soit instanciée et qu'elle termine son exécution*, moment où le bloc d'activation est **dépilé**.

0.4 Règle de mise en oeuvre

L'idée est de trouver un énoncé récursif ou par récurrence de résolution du problème, c'est-à-dire un énoncé qui fasse référence au problème lui-même.

Tout comme pour une récurrence, ce ou ces appels récursifs se font sur des instances plus petites du ou des paramètres de l'algorithme, pour la factorielle *n1* au lieu de *n*. Comme l'algorithme sait traiter le cas de base, la suite des appels récursifs sur des instances de plus en plus petites aboutira inexorablement à un appel sur la ou les instances de la base récurrente ce qui achèvera le processus.

Il est donc important de respecter les règles suivantes pour construire un algorithme récursif:

1. Tout algorithme récursif doit distinguer plusieurs cas dont l'un au moins ne doit pas contenir d'appels récursifs.

   Les cas non récursifs d'un algorithme récursif sont appelés *cas de base*. Les conditions que doivent satisfaire les données dans ces cas de bases sont appelées *conditions de terminaison*.

   Exemple: Cet algorithme calcule-t-il $n!$?

   ```
   def fact2(n):
       return n*fact2(n-1)
   ```

2. Tout appel récursif doit se faire avec des données plus proches de données satisfaisant les conditions de terminaison.

   Il faut s'assurer que la situation de terminaison est atteinte après un nombre fini d'appels récursifs. La preuve de terminaison d'un algorithme récursif se fait en identifiant la construction d'une suite strictement décroissante d'entiers positifs ou nuls.

   Exemple: Dans le cas de factorielle, il s'agit simplement de la suite des valeurs du paramètre.

   Exemple: Cet algorithme calcule-t-il $n!$?

   ```
   def fact3(n):
       if n ==1 :
           return 1
       else:
           return fact3(n+1)/(n+1)
   ```

Note: Ce qui distingue fondamentalement une méthode itérative d'une méthode récursive, c'est la nécessité ou non de conserver une trace de chaque calcul à toutes les étapes de la méthode.

0.5 Types de récursivité

0.5.1 Récursivité simple ou linéaire

Un appel récursif est dit simple ou linéaire si la fonction ne contient qu'un seul appel récursif à elle-même. Chaque cas qu'il distingue se résout en au plus un appel récursif. L'algorithme de calcul de n! ci-dessus est récursif simple.

0.5.2 Récursivité multiple

Un algorithme récursif est multiple si l'un des cas qu'il distingue se résout avec plusieurs appels récursifs. C'est le cas de l'algorithme de dérivation donnée ci-dessus. La fonction `derivee` est appelée plusieurs fois. L'algorithme récursif du calculs des coefficient binomiaux en est un exemple (Exercice 4), celui du calcul des termes de la suite de **Fibonacci** également. Un algorithme à récursivité multiple ne peut pas être terminal.

0.6 Complexité d'un algorithme récursif

Noton T_n le nombre d'opérations nécessaires pour évaluer le coût sur un problème de taille n. On cherche une relation de récurrence sur T_n et on pourra utiliser le tableau suivant:

Relation de récurrence	Complexité
$T_n = T_{n-1} + \Theta(1)$	$\Theta(n)$
$T_n = T_{n-1} + \Theta(n)$	$\Theta(n^2)$
$T_n = 2T_{n-1} + \Theta(1)$	$\Theta(2^n)$
$Tn = T_{n/2} + \Theta(1)$	$\Theta(log_2(n))$

Sur la première ligne, le cas général comporte un nombre d'opération constant à traiter. L'algorithme naïf de Fibonacci relève du troisième cas.

0.7 Exercices

Les fonctions données dans cette série d'exercices sont des fonctions récursives. Vous ne pouvez pas utiliser les fonctions intégrées `int()` et `bin()`.

Exercice 1:somme des carrés La somme des carrés des premiers entiers est: $S_n = 1^2 + 2^2 + ... + (n-1)^2 + n^2$. En utilisant la formule de récurrence:

$$S_1 = 1 \quad et \quad S_n = S_{n1} + n^2$$

qui calcule S_n par un algorithme récursif, programmez une fonction `somme_carres_rec(n)` qui retourne la somme des carrés de 1 à n.

Cas terminal: $n = 1$ pour lequel $S_1 = 1$.

Cas général: pour $n \geqslant 2$, correspond au calcul de la somme S_n écrite sous la forme $S_n = S_{n1} + n^2$.

Exercice 2: inverser une liste Programme une fonction récursive `inverser(liste)` qui inverse l'ordre des éléments d'une liste. Exemple:

```
>>>inverser([1,2,3,4,5])
[5,4,3,2,1].
```

Le principe à suivre est le suivant: on extrait le premier élément, on inverse le reste de la liste, enfin on rajoute le premier élément à la fin de liste obtenue.

Cas terminal: Si la longueur n de la liste est 1 (ou 0) : renvoyer la liste sans modification (il n'y a rien à inverser). $n = 1$ pour lequel $S_1 = 1$.

Cas général:

1. On note x_0 le premier élément de liste.
2. On note `fin_liste` le reste de la liste (la liste sans x_0).
3. On effectue un appel récursif `inverser(fin_liste)` qui renvoie une liste .
4. On ajoute à ce résultat l'élément x_0 en queue de liste.
5. On renvoie cette liste.

Exercice 3: nombre d'or La suite (U_n) définie par la relation de récurrence ci-dessous calcule une valeur du nombre d'or φ:

$$\begin{cases} u_0 & = 1 \\ u_{n+1} & = 1 + \dfrac{1}{u_n} \end{cases}$$

Programmez une fonction récursive `nb_dor(n)` prenant un entier n en paramètre et retournant la valeur correspondante de u_n.

Exemple (calcul de u_{10}):

```
>>> nb_dor(5)
1.625
```

Exercice 4:compte à rebours

1. Écrire une fonction récursive `compte(n)` prenant en paramétre un entier et affichant chaque nombre jusque zéro de la façon suivante :

   ```
   >>>compter(5)
   "5, 4, 3, 2, 1, Décollage !"
   ```

2. Écrire une fonction récursive `compte_v2(n)` prenant en paramétre un entier et renvoyant la chaine de caractère `"5, 4, 3, 2, 1, Décollage !"`

Exercice 6 Voici trois exemples, donner la réponse à la question "est-ce que le mot donné est un palindrome ?", justifier.

- RADAR → ADA → D
- SERPES → ERPE → RP
- ELLE → LL → " "

Ecrire une fonction `palindrome(mot:str)->bool` retournant `True` si la chaine de caractère entrée est un palindrome.

Exercice 7 Dans cet exercice, le cas général et le cas terminal sont donnés. Soit une liste $[x_0, x_1, ..., x_{n-1}]$

Programmer une fonction `max_liste` qui retourne le maximum de cette liste.

Cas terminal: Si la liste ne contient qu'un seul élément, c'est le maximum.

Cas général: Soit x_0 le premier terme.

- On note M_1 le maximum du reste de la liste calculé par un appel récursif.
- On retourne $M = max(x_0, M_1)$.

Exercices sur les listes Programmer un algorithme récursif renvoyant la longueur d'une liste. Interdiction d'utiliser la fonction `len`

Exercice 9:binaire Dans cet exercice les fonctions prennent en paramètre un entier n positif et retournent un entier. Distinguez clairement le ou les cas de base, le cas général avant de coder votre fonction..

1. Ecrivez une fonction `dec2bin(nb:int)->list` récursive qui convertit un entier donné en base 10 vers la base 2.
2. Réalisez une fonction récursive `taille_bin`. Celle-ci calcule le nombre de chiffres dans l'écriture binaire de l'entier naturel en paramètre.

Exemples

```
>>>taille_bin(0)
1
>>>taille_bin(1023)
10
>>>taille_bin(1024)
11
```

3. Réalisez la fonction `poids_bin`. Celle-ci calcule récursivement le nombre de chiffre 1 dans l'écriture binaire de l'entier naturel naturel n.

Exemples

```
>>>poids_bin(0)
0
>>>poids_bin(1)
1
>>>poids_bin(2)
1
>>>poids_bin(255)
8
```

Exercice A:puissance récursive Réalisez la fonction `puissance_rec`, celle-ci calcule récursivement x^n. Elle renvoie donc un flottant. $(x \in \mathbb{R}, n \in N)$

Exemples

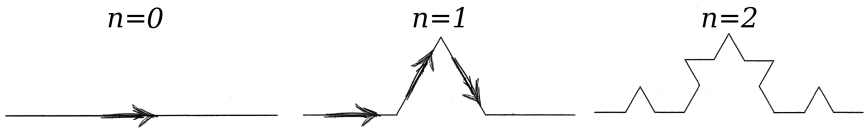
```
>>>puissance_rec(10, 0)
1
>>>puissance_rec(10, 1)
10
>>>puissance_rec(2, 10)
1024
```

Exercice B: figure récursive Testez le programme ci-dessous, vous l'étudierez attentivement. Que dessine-t-il ?

```
import turtle as t
t.speed(100)
t.width(2)
def fractal(length=100):
  if length>10:
      for _ in range(5):
          t.forward(length)
          fractal(length*0.3)
          t.left(144)

fractal(200)
exitonclick()
```

Exercice D:Courbe de Von Koch La courbe de Von Koch est une figure fractale. A chaque étape, chaque segment est remplacé par 4 nouveaux segments de même longueur suivant le schéma suivant:



Courbe de Von Koch pour n=1, n=2, n=3.

Ecrire une fonction récursive `flocon( , n)` qui trace cette courbe à l'étape n, étant une longueur donnée.

Cas terminal: Si n=0, tracer un segment de longueur .

Cas général: On effectue 4 appels récursifs. Attention aux angles qui ne sont pas indiqués!

- Tracer la courbe d'ordre n-1, avec la longueur $\frac{}{3}$. Tourner à gauche.
- Tracer la courbe d'ordre n-1, avec la longueur $\frac{}{3}$. Tourner à droite.
- Tracer la courbe d'ordre n-1, avec la longueur $\frac{}{3}$. Tourner à gauche.
- Tracer la courbe d'ordre n-1, avec la longueur $\frac{}{3}$.

Exercice E:Algorithme de Horner Programmer une fonction récursive `horner()` prenant en paramètre un nombre binaire de type `list` et retournant sa valeur en décimal.

Exercice F: Les tours de Hanoï Les tours de Hanoï (originellement, la tour d'Hanoï) sont un jeu de réflexion imaginé par le mathématicien français Édouard Lucas, et consistant à déplacer des disques de diamètres différents d'une tour de *départ* à une tour d'*arrivée* en passant par une tour *intermédiaire*, et ceci en un minimum de coups, tout en respectant les règles suivantes :

- on ne peut déplacer plus d'un disque à la fois ;
- on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.

On suppose que cette dernière règle est également respectée dans la configuration de départ. (*Wikipedia*)

Le but est de déplacer n disques de la tour de départ vers la tour d'arrivée.

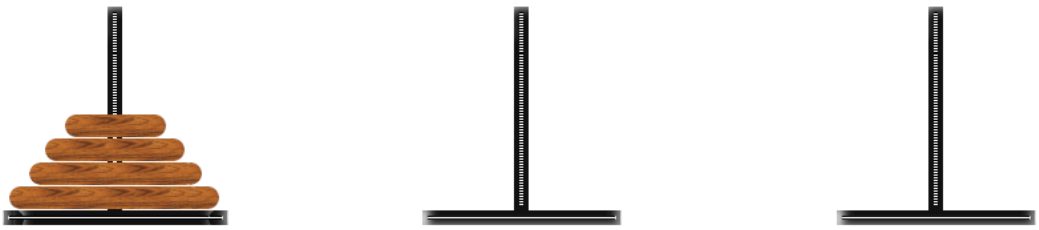


Figure 1: Le but est d'amener les disques sur la troisième tour

1. Résoudre le problème avec 3 disques.
2. On veut utiliser un programme récursif pour résoudre le problème. On suppose que l'on sait procéder pour 3 disques. Montrer avec un schéma que l'on peut résoudre le problème avec 4 tours.
3. Compléter le programme suivant :

```
def hanoi(n, depart, arrivee, intermediaire):
    """

    Affiche les déplacements à effectuer pour déplacer
    n disques de depart vers arrivee
    """
    if n>0:
        #déplace de la tour de départ vers la tour intermédiaire
        hanoi(.....................)
        print(depart ,"-->", arrivee)
        #déplace de la tour intermédiaire vers la tour finale
        hanoi(.....................)
n = int(input("Nombre de disques : "))
hanoi(n, "A", "C", "B")
```

Exemple :

```
>>> hanoi(2, "A", "B", "C")
Nombre de disques : 2
A --> B
A --> C
B --> C
```

0.8 Compléments

0.8.1 Récursivité terminale

Un algorithme récursif simple est terminal lorsque l'appel récursif est la dernière chose effectuée. Il n'y a pas de "calcul en attente". L'avantage est qu'il n'y a rien à mémoriser dans la pile, ce mode de rédaction permet une économie des ressources mémoires.

Ce n'est pas le cas de l'agorithme `fact`. Généralement ce genre d'algorithme peut facilement être transformé en une boucle.

Exemple Dans un langage de programmation fictif :

```
def recursion_terminale(n):
    ...
    return recursion_terminale(n-1)
```

```
def recursion_non_terminale(n):
    ...
    return n+ recursion_non_terminale(n-1)
```

`récursionNonTerminale()` n'est pas une récursion terminale car sa dernière instruction est une composition faisant intervenir l'appel récursif. Dans le premier cas, aucune référence aux résultats précédents n'est à conserver en mémoire, tandis que dans le second cas, tous les résultats intermédiaires doivent l'être.(*wikipedia*)

Exemple d'algorithme récursif terminal Prédicat de présence d'un caractère dans une chaîne : Un caractère `c` est présent dans une chaîne `s` non vide, s'il est le premier caractère de `s` ou s'il est présent dans les autres caractères de `s`.

```
def est_present(c,s):
    if s == '' :
        return False
    elif c == s[0]:
        return True
```

```
    else:
        return est_present(c, s[1:])
```

0.8.2 Récursivité croisée ou mutuelle

Deux fonctions peuvent s'appeler l'une l'autre, on parle alors de récursivité croisée ou d'algorithmes mutuellement récursifs. On peut étendre cette définition à un nombre quelconque d'algorithmes.

Exemple Deux fonctions nommées `pair` et `impair` déterminant la parité ou l'imparité d'un entier.

```
def pair(n):
    if n == 0:
        return True
    else:
        return impair(n-1)
```

```
def impair(n):
    if n == 0:
        return False
    else:
        return pair(n-1)
```

0.9 Références

- *Python au lycée: Tome 2 (Arnaud Bodin)*:`http://exo7.emath.fr/`
- `https://fr.wikipedia.org/wiki/Recursivite`