TP n° 11 : Récursivité

Travail à faire avant la séance :

— lire et tester le code de la partie 1 Introduction.

— chercher les questions **Q.1** à **Q.3**.

1 Introduction

En mathématiques, en informatique, en biologie, mais aussi dans notre quotidien, nous faisons souvent face à des situations où un problème doit être résolu en utilisant une méthode de résolution qui est répétée plusieurs fois.

— Dans **l'itération**, la recherche de la solution est séquentielle ;

— dans **la récursion**, la méthode s'appelle elle-même.

Comme nous le faisons depuis le début de l'année, le traitement **itératif** est mis en œuvre à l'aide de boucles `for` ou `while`.

La **récursivité** est un concept plus délicat à mettre en œuvre. Pour autant elle est un processus fondamental impossible à éviter : l'auto-reproduction, qui constitue le fondement de toute vie, est un processus récursif. Le sociologue Edgar Morin suggère que « les individus d'aujourd'hui seraient les produits d'un système que les individus avant eux ont reproduit ». Ces nouveaux individus produisent à leur tour la société, soit un système qui engendre d'autres individus qui engendrent la société, etc. ». Ainsi, Edgar Morin se distingue par l'emploi régulier de formules faisant référence à la récursivité, comme « Nous faisons le langage qui nous fait ».

Concernant les langages de programmation, la plupart portent aujourd'hui la programmation récursive, toutefois, ce n'est pas le cas de certains langages « anciens » tels que COBOL, FORTRAN, BASIC etc. Par contre, en programmation fonctionnelle (LISP, CAML etc.) ou en programmation logique (PROLOG), les programmes sont toujours récursifs.

On dit que ce sont deux **paradigmes** différents de programmation : la programmation **itérative** et la programmation **récursive**.

1.1 Cas d'école : calcul de la factorielle

> Une fonction **récursive** est une fonction qui **s'appelle elle-même**.

Par exemple si $u_n = n!$ les termes de cette suite se calculent par récurrence :

$$\begin{cases} u_0 &= 1 \\ u_n &= n \times u_{n-1} \end{cases}$$

On traduit tout simplement cette formule de récurrence en **une fonction qui s'appelle elle-même** :

```
def fact(n):
    if n == 0 :
        return 1   # cas dit "de base"
    else :
        return n * fact(n-1)   # appel récursif
```

On teste :

```
>>> fact(5)
120
>>> fact(10)
3628800
```

> Cela fonctionne très bien ! On voit tout de suite l'intérêt de la programmation récursive : le code est beaucoup **plus simple** et **plus élégant**.
> **Il n'y a pas de boucle**.

On peut visualiser l'exécution du calcul de `fact(5)` sur le site suivant : pythontutor.com.
Il faut cliquer sur « Visualize Execution ». pour lancer le programme, puis cliquer sur Next pour dérouler les étapes.
La petite flèche rouge à gauche indique l'instruction qu'est en train de lire l'interpréteur : à chaque fois on teste si on est arrivé au cas de base et sinon on lance un nouvel appel de la fonction.

> Que se passe-t-il ?

On voit que l'interpréteur **gère tout tout seul** : les différents appels sont sauvegardés dans une structure mémoire appelée **pile** (car les données sont empilées comme pour une pile d'assiettes) et lorsqu'on arrive au cas de base il ne reste plus qu'à **dépiler** les données (donc c'est la dernière donnée empilée qui est dépilée la première, comme avec les assiettes).

> Le programmeur n'a rien à faire. C'est pratique !

1.2 Autres exemples

On peut aussi avoir une **récursivité d'ordre 2**, par exemple avec la suite de Fibonacci définie par :

$$\begin{cases} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{cases}$$

La fonction récursive est toute simple :

```python
def fibo(n) :
    if n == 0 :   # premier cas de base
        return 0
    elif n == 1 : # second cas de base
        return 1
    else :
        return fibo(n-1) + fibo(n-2)  # appel récursif dans le cas général
```

On teste :

```
>>> fibo(10)
55
```

On peut avoir une **récursivité croisée** entre deux fonctions :

```
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)
```

On teste :

```
>>> pair(5)
False
>>> pair(4)
True
>>> impair(5)
True
```

On peut aussi manipuler des variables de type `list`. Pour cela on rappelle que si `L` est une liste alors :

— `L[-1]` est son dernier élément,

— `[L[-1]]` est donc une liste composée d'un seul élément qui est le dernier élément de `L`,

— `L[:-1]` désigne la liste `L` privée de son dernier élément.

Pour créer la liste « miroir » d'une liste `L` on prend son dernier élément `L[-1]`, on le place en premier et on concatène ensuite avec la liste miroir de `L[:-1]`. Cette remarque donne le code suivant :

```
def miroir(L) :
    if L == [] :  # cas de base : la liste vide est égale à sa version miroir
        return L
    else :
        return [L[-1]] + miroir(L[:-1])  # appel récursif : + est la concaténation
```

On teste :

```
>>> miroir([1, 2, 6, 4])
[4, 6, 2, 1]
```

Malheureusement tout n'est pas aussi simple.

— Il faut être sûr que la suite des appels récursifs amène à un des cas de base sinon cette suite va être infinie et la pile mémoire va saturer (erreur : `RecursionError: maximum recursion depth exceeded in comparison`).

— Même sans cela la quantité de mémoire consommée peut être exponentielle. Par exemple, à cause de la récursivité d'ordre 2, le calcul de `fibo(10)` demande 177 appels récursifs et celui de `fibo(100)` en demande 1 146 295 688 027 634 168 201, soit environ 10^{21}. Autant dire que le calcul est impossible même sur les ordinateurs les plus puissants.

> La résolution de ces problèmes sera étudiée en seconde année.
> Pour cette année on se contente de remarquer que la récursivité donne un code **très simple** et **très élégant**, proche des mathématiques.

Pour mémoire voici les versions **itératives** des calculs de la factorielle et de la suite Fibonacci :

```
def fact2(n) :
    u = 1
    for k in range(1, n+1) : # boucle non exécutée si n=0
        u = k * u
    return u

def fibo2(n) :
    u, v = 0, 1
    if n == 0 :
        return u
    elif n == 1 :
        return v
    else:
        for k in range(2, n+1) :
            u, v = v, u+v
            # si on n'utilise pas l'affectation simultanée
            # il faut une variable auxiliaire (voir TP4)
        return v
```

Le code est beaucoup moins élégant mais le calcul de `fibo2(100)` ne prend que quelques milisecondes.

2 Exercices

2.1 Premières fonctions récursives

Pour écrire une fonction récursive on respectera le squelette suivant :

```
def fonction(variable) :
    if ..... :  # cas de base (il peut y en avoir plusieurs)
        return ............
    else :   # appel récursif
        return .........
```

Q.1 Écrire une fonction récursive `somme(n)` qui calcule la somme des entiers de 1 à n : $1+2+\cdots+n$. (`somme(100)` doit renvoyer 5050).

Q.2 Écrire une fonction récursive `sommeListe(L)` qui calcule la somme des éléments d'une liste de nombres `L` (si la liste est vide la fonction renvoie 0).
Code pour tester :

```
>>> L = [ k for k in range(101) ]   # liste des entiers de 1 à 100
>>> print(sommeListe(L)) #  doit afficher 5050
```

Q.3 La méthode de Héron permet d'obtenir des valeurs approchées de racines carrées. Elle est basée sur la convergence de la suite suivante définie par :

$$\begin{cases} u_0 &= a \\ u_{n+1} &= \dfrac{1}{2}\left(u_n + \dfrac{a}{u_n}\right) \end{cases}$$

Cette suite converge vers $\sqrt{a}$ (vu en cours de mathématiques).

Proposez une solution **itérative** `heronIter(a, n)` et une solution **récursive** `heronRec(a, n)` de la méthode de Héron. Ces fonctions ont pour arguments `a` et le nombre d'itérations `n`. Attention : dans la version récursive un seul appel récursif suffit pour calculer u_n : si l'on écrit une solution avec deux appels récursifs comme la définition de la suite tend à le suggérer, le nombre d'appels récursifs croît exponentiellement quand n augmente : le code ne fonctionne pas quand n est trop grand.

Code pour tester :

```
>>> print(heronIter(2, 100))
>>> print(heronRec(2, 100))
>>> from math import sqrt
>>> print(sqrt(2))  # pour vérifier
```

Q.4 Écrire une fonction récursive `concatene(L1,L2)` qui renvoie la concaténation des listes `L1` et `L2`.

On n'utilisera pas l'opérateur de concaténation `+` *mais on pourra utiliser* `L.append(x)` *qui ajoute l'élément* `x` *à la fin de la liste* `L` *(sans renvoyer de valeur).*

Code pour tester :

```
>>> print(concatene([1,2,3], [4,5,6,7]))   # affiche  [1,2,3,4,5,6,7]
```

3 Évaluation de polynômes

3.1 Version naïve

Si $P(x) = \sum_{k=0}^{N} a_k x^k$ est un polynôme et x_0 un réel on souhaite calculer la valeur de $P(x_0)$.

Le polynôme P sera représenté en Python par la liste `a = [a0, a1, ...,aN]`

Q.5 Proposer une fonction **itérative** `evalPolynome(a, x0)` qui calcule la valeur de $P(x_0)$.

Code pour tester :

```
>>> evalPolynome([5, -4, 3, -2, 1], 2.5)  #  affiche 21.5625
```

3.2 Méthode de Horner

Le nombre de produits est de $N-1$ pour le calcul de x_0^N, de $N-2$ pour le calcul de x_0^{N-1}, etc. Soit finalement un coût calculatoire qui est de $(N-1)+(N-2)+\cdots+1+0 = \dfrac{N(N-1)}{2}$. La charge de calcul est donc quadratique, elle croît asymptotiquement comme N^2, le carré du degré du polynôme soit un coût en $O(N^2)$.

D'une façon générale, la charge calculatoire vient des calculs des puissances successives de x_0. En effet, la multiplication nécessite un temps de traitement bien supérieur à l'addition ou à l'affectation. Pour diminuer la charge de calcul nous allons utiliser la *méthode de Horner*.

Elle repose sur la factorisation successive par x_0 :

$$P(x_0) = a_0 + x_0 \times (a_1 + a_2 x_0^1 + \cdots + a_{N-1} x_0^{N-2} + a_N x_0^{N-1})$$

L'opération est répétée une seconde fois :

$$P(x_0) = a_0 + x_0 \times \left(a_1 + x_0 \times (a_2 + a_3 x_0^1 + \cdots + a_{N-1} x_0^{N-3} + a_N x_0^{N-2})\right)$$

etc

À la $(N-1)$-ième factorisation, nous avons l'expression suivante :

$$P(x_0) = a_0 + \underbrace{x_0 \times}_{1^{\text{ère}}\text{ fact.}}\Big(a_1 + \cdots + x_0 \times \big(a_{N-3} + x_0 \times \big(a_{N-2} + \underbrace{x_0 \times (a_{N-1} + a_N x_0)}_{(N-1)^{\text{ième}}\text{ fact.}}\big)\big)\Big)$$

Il apparaît une récurrence du type :

$$\begin{cases} y_N &= a_N \\ y_{N-k} &= a_{N-k} + x_0 \times y_{N-k+1} \quad \text{pour } k = 1, \ldots, N \end{cases}$$

La valeur de $P(x_0)$ étant y_0.

En comparaison avec l'algorithme naïf, on voit donc que le nombre de produits est réduit à N. La charge de calcul pour évaluer le polynôme en x_0 est maintenant proportionnel au degré du polynôme N soit linéaire c'est-à-dire en $O(N)$. En conséquence, cet algorithme est beaucoup plus efficace que le précédent.

Outre l'efficacité de cet algorithme, cette méthode peut limiter la manipulation de nombres très grands et permettre d'éviter ainsi les dépassements de capacité lorsque les traitements sont effectués par un ordinateur. Elle présente aussi un intérêt pour la conversion de base de numération, la division euclidienne, l'estimation des zéros d'un polynôme (méthode dichotomique), etc.

Q.6 En utilisant la formule :

$$P(x_0) = a_0 + x_0 \times (a_1 + a_2 x_0^1 + \cdots + a_{N-1} x_0^{N-2} + a_N x_0^{N-1})$$

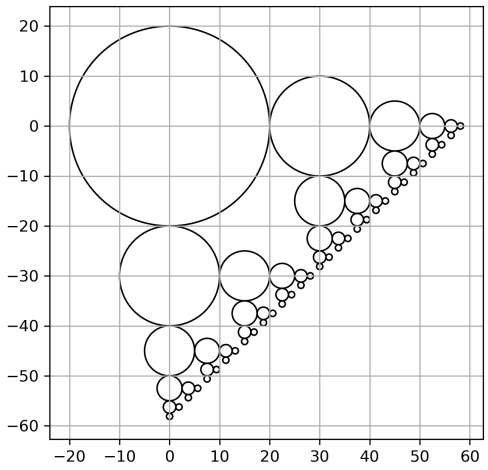
proposez une fonction **récursive** `horner(a, x0)` pour l'évaluation de $P(x_0)$ en appliquant un schéma de Horner.

Code pour tester :

```
>>> print(horner([5, -4, 3, -2, 1], 2.5)) # affiche 21.5625
```

4 Fractales

La figure ci-contre peut être décrite de façon récursive. Elle est formée d'un cercle et de deux copies de ce cercle ayant subi une réduction d'un facteur 2. Ces deux autres cercles sont tangents au périmètre du cercle initial tel que les lignes des centres sont parallèles aux axes du repère. Ils deviennent à leur tour « cercle initial » pour poursuivre la figure.
Ce processus de construction est récursif et peut être généré avec le code python donné ci-dessous (et à télécharger sur Moodle).

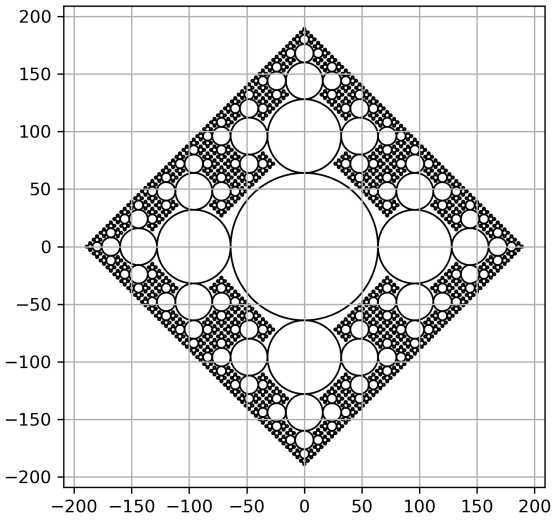



```
from matplotlib import pyplot as plt

plt.figure()
F=plt.gca()  # Récupère les axes courants pour pouvoir dessiner dessus par la suite.

def cercle(x,y,r):
    # cercle de centre (x,y) et de rayon r
    cir=plt.Circle([x,y],radius=r,fill=False)
    # ajout du cercle à la figure :
    F.add_patch(cir)

def CerclesRec(x,y,r):
    # construction récursive de la figure
    cercle (x,y,r)  # le cas de base est appelé dans tous les cas
    if r>1:   # appels récursifs
        CerclesRec(x+3*r/2,y,r/2)
        CerclesRec(x,y-3*r/2,r/2)

# appel de la fonction CerclesRec
CerclesRec(0,0,20)

# pour placer toute la figure dans un repère orthonormé :
plt.axis('scaled')

# affichage de la figure
plt.grid('on')
plt.show()
```

Q.7 Qu'est ce qui garantit que cette fonction ne s'appellera qu'un nombre fini de fois ?

Q.8 Dans l'appel initial, si l'on change `CerclesRec(0,0,20)` par `CerclesRec(0,0,64)`, ou par `CerclesRec(0,0,4)` qu'obtiendra-t-on ?

Q.9 On veut maintenant obtenir la figure ci-contre. Pour cela, on adapte le code précédent selon la trame donnée page suivante (et dans le fichier à télécharger sur Moodle). On utilise en particulier un paramètre supplémentaire pour définir la fonction récursive `CerclesRec(x, y, r, voisin)` (`x` : abscisse du centre, `y` : ordonnée du centre, `r` : rayon du cercle, `voisin` : position du voisin) où `position` est l'une des chaînes de caractères : `'haut'`, `'bas'`, `'droite'`, `'gauche'`.




```python
# Code à compléter
import matplotlib.pyplot as plt
F = plt.gca()

def cercle(x, y, r):
    """Trace un cercle de centre (x, y) et de rayon r"""
    cir = plt.Circle([x, y], radius=r, fill=False)
    # ajout du cercle à la figure :
    F.add_patch(cir)

def CerclesRec(x, y, r, voisin):
    """Construit récursivement la figure pour un cercle de rayon
    r dont la position du voisin (le cercle de rayon 2r) est situé
    à la position indiquée par la chaîne de caractères voisin"""
    # On trace un cercle de rayon r centré sur (x, y)
    cercle(x, y, r)
    # Et si le rayon de ce cercle est assez grand (supérieur à 1)
    # on trace ses plus proches voisins :
    if r > 1:
        # Si le voisin du cercle de rayon r est en haut
        if voisin == 'haut':
            # On trace un cercle de rayon r/2 à sa droite,
            # dont le voisin (le cercle de rayon r) est
            # à gauche.
            CerclesRec(x+3*r/2, y, r/2, "gauche")
            # et il faut encore tracer deux cercles :
            #
            #
        elif voisin == 'bas':
            # à compléter
        elif voisin == 'gauche':
            # à compléter
```

```
        elif voisin == 'droite':
            # à compléter
        else:
            # Si le cercle de rayon r n'a pas de voisin, c'est le
            # cercle de rayon le plus grand de la figure, il faut
            # tracer quatre cercles pour l'entourer

# appel de la fonction CerclesRec
CerclesRec(0, 0, 64, "")

plt.axis('scaled') # pour placer toute la figure dans un repère orthonormé
plt.grid('on')
plt.show()
```

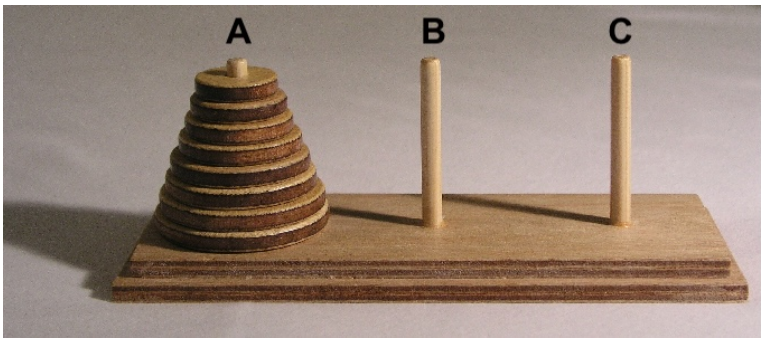
On trouvera d'autres exemples de fractales sur cette page : https://mathcpge.org/index.php/maths-pcsi/decouverte-des-fractales .

5 Les tours de Hanoï

Dans le problème des tours de Hanoï, on veut déplacer n disques de diamètres différents empilés du plus grand au plus petit sur une tour de départ A jusqu'à une tour d'arrivée C, en s'aidant d'une tour intermédiaire B, en respectant les deux règles suivantes :

— on ne peut déplacer qu'un seul disque à la fois,

— un disque ne peut être placé que sur un disque de plus grand diamètre ou sur une tour vide.



Q.10 Écrire une fonction récursive `hanoi` prenant en argument un nombre de disques `n` et les noms de trois tours, et affichant le détail des opérations à effectuer pour résoudre le problème. Pour `n=3` disques, numérotés du plus petit au plus grand, l'exécution de `hanoi(3, 'A', 'C', 'B')` devra afficher le texte suivant :

```
Déplacer le disque 1 de A vers C
Déplacer le disque 2 de A vers B
Déplacer le disque 1 de C vers B
Déplacer le disque 3 de A vers C
Déplacer le disque 1 de B vers A
Déplacer le disque 2 de B vers C
Déplacer le disque 1 de A vers C
```

6 Énumération récursive des permutations d'une liste

Pour déterminer toutes les permutations d'une liste `L` à `n` éléments, on peut procéder de la manière suivante :

- Si la liste n'a qu'un élément (ou aucun), elle est sa seule permutation.
- Sinon :
  - on construit d'abord toutes les permutations commençant par `L[0]` :
    - on génère toutes les permutations de la liste `L` privée de `L[0]`,
    - on construit toutes les listes commençant par `L[0]` suivi d'une des permutations à `n-1` éléments précédentes ;
  - on construit ensuite toutes les permutations commençant par `L[1]` :
    - on génère toutes les permutations de la liste `L` privée de `L[1]`,
    - on construit toutes les listes commençant par `L[1]` suivi d'une des permutations à `n-1` éléments précédentes ;
  - etc. jusqu'à `L[n-1]`.

Q.11 Écrire une fonction `permutations` qui renvoie la liste de toutes les permutations d'une liste donnée en argument. Par exemple, `permutations([1, 2, 3])` doit renvoyer
`[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]`