Récursivité

Activité 05

Partie 1 : un nouveau paradigme de programmation

En mathématiques, certains objets se construisent par récurrence. Par exemple, considérons la suite de Fibonacci, définie par $f_0 = 0$, $f_1 = 1$ et $f_{n+2} = f_{n+1} + f_n$. Pour calculer le n-ème terme de la suite de Fibonacci en Python, le premier programme convient : il consiste à construire petit à petit les termes de la suite, en retenant deux valeurs successives dans deux variables locales.

```python
def fibo(n):
    if n <= 1:
        return n
    a,b = 0,1
    for i in range(n-1):
        a,b = b,a+b
    return b
```

Il faut toutefois reconnaître que ce premier programme n'est pas forcément « naturel » au premier regard. En informatique, il est toutefois possible de définir des fonctions qui s'appellent elles-mêmes : on appelle cela une fonction récursive. Ci-contre, vous trouvez un exemple de fonction récursive calculant la suite de Fibonacci.

```python
def fibo(n):
    if n <= 1:
        return n
    else:
        return fibo(n-1) + fibo(n-2)
```

Exercice 1

1. Recopier les fonctions précédentes, et les tester pour `n = 10`.
2. Tester pour `n = 38` : que remarquez-vous ?
3. La suite de Tribonacci est la suite définie par $T_0 = 0, T_1 = T_2 = 1$ et $T_{n+3} = T_{n+2} + T_{n+1} + T_n$ pour tout $n \geqslant 0$. Écrire une fonction récursive `tribo(n)` qui calcule le n-ème terme de la suite. Pour tester, $T_{10} = 149$.

Exercice 2

1. La fonction ci-contre ne fonctionne pas : tester la fonction, et expliquer pourquoi elle ne fonctionne pas.
2. La suite de Catalan est la suite définie par $C_0 = 1$ et $C_{n+1} = \sum_{k=0}^{n} C_k \times C_{n-k}$.
   Écrire une fonction récursive `catalan(n)` qui calcule le n-ème terme de la suite, en faisant bien attention aux bornes. Pour tester, $C_{10} = 16\,796$.

```python
def fonction(n):
    if n <= 1:
        return 0
    else:
        return fonction(n)**2
```

Exercice 3

La récursivité ne s'applique pas qu'aux entiers.
Écrire une fonction `maximum(tab)` qui calcule le maximum d'un tableau selon le schéma récursif suivant : si le tableau est de longueur 1, on renvoie son seul élément. Sinon, on appelle `maximum` sur la moitié gauche et la moitié droite de `tab` ; puis on renvoie le plus grand des deux résultats obtenus.

Partie 2 : récursivité et dessin

Nous allons utiliser une bibliothèque graphique appelée `turtle` pour réaliser des figures particulières.

Exercice 4 : prise en main de `turtle`

1. Importer la bibliothèque `turle`.
2. Le tracé de dessins de cette bibliothèque repose sur l'idée suivante : une tortue, initialiement positionnée en (0,0), avance dans le plan en tenant un stylo. Si vous lui dites d'avancer, elle avance tout droit; si vous lui dites de tourner, elle pivote sur place. À l'aide de la syntaxe décrite à droite, écrire une fonction `tracer_carre(n)` qui trace un carré de largeur `d`.
3. Écrire une fonction `grille(n,m,d)` qui tracer une grille dont les cases sont des carrés de largeur `d`, avec `n` cases de hauteur de `m` cases de largeur.

`forward(n)`	fait avancer la tortue de `n` pixels
`left(n)`	fait tourner la tortue sur sa gauche de l'angle `n` (en degrés)
`right(n)`	sur sa droite
`penup()`	fait lever le stylo
`pendown()`	fait reposer le stylo
`speed(0)`	donne à la tortue sa vitesse de déplacement maximale
`hideturtle()`	cache le curseur de la tortue
`goto(a,b)`	déplace la tortue aux coordonnées `(a,b)`

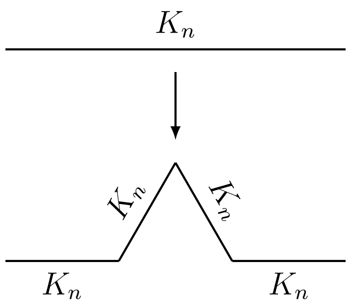
Nous allons tracer des fractales avec `turtle`.

Exercice 5

La ligne de Koch se construit récursivement à partir d'une suite de figures géométriques $(K_n)_{n\geqslant 0}$:

— K_0 est un segment;

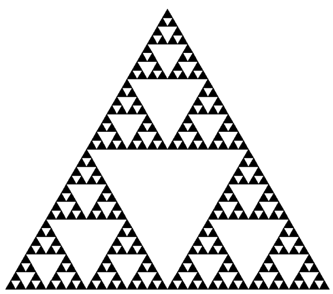
— pour tracer K_{n+1}, on trace 4 copies de K_n selon la ligne brisée suivante.

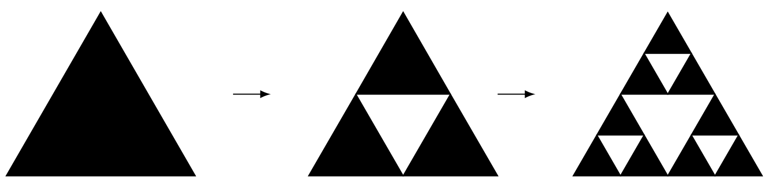


Écrire une fonction `koch(n,d)` qui renvoie la `n`-ème itération de la ligne de Koch de longueur `d`. Puis écrire une fonction `super_koch(n,d)` qui renvoie la superposition de ces figures l'une sur l'autre.

Exercice 6

Écrire une fonction `sierpinski(n,d)` qui trace le triangle de Sierpinski d'ordre `n` et de largeur `d` : à l'ordre 0, il s'agit du triangle équilatéral.
Pour remplir une zone de couleur, on utilise les commandes `begin_fill()` et `end_fill()` : on commence un begin, on trace une courbe fermée, et on arrête le remplissage avec le end.





Partie 3 : algorithmes classiques

Exercice 7

L'algorithme d'Euclide est considéré comme l'un des algorithmes les plus anciens connus en mathématiques : il sert à calculer le PGCD de deux entiers positifs. Il repose sur l'idée suivante (a et b sont deux entiers, $a \geqslant b$) :

— $\mathrm{PGCD}(a, 0) = a$ pour tout $a \in \mathbb{N}$;

— $\mathrm{PGCD}(a, b) = \mathrm{PGCD}(b, a \bmod b)$.

Écrire une fonction `euclide(a,b)` permettant d'obtenir le PGCD de a et b selon l'algorithme d'Euclide.

Exercice 8 : tri fusion

Nous allons étudier un nouvel algorithme de tri, appelé le tri fusion. Son principe est le suivant :

— si le tableau est petit, il est trié ;

— sinon, on coupe le tableau à trier en deux et on appelle récursivement notre fonction sur les deux moitiés, qu'on suppose donc triées ;

— on fusionne les deux moitiés pour récupérer toutes les données du tableau dans le bon ordre.

1. Tester l'algorithme à la main sur le tableau `[4,1,3,2]`.
2. Écrire une fonction `fusion(tab1,tab2)` qui prend deux tableaux triés, et renvoie leur fusion. Par exemple, `fusion([1,4,5],[2,3,6])` renvoie `[1,2,3,4,5,6]`.
3. Écrire enfin la fonction `tri_fusion(tab)` qui trie `tab` selon le tri fusion.

Cette méthode est appelée « diviser-pour-régner » : pour résoudre un problème, on le découpe en morceaux indépendants, puis on recolle les morceaux après.

Exercice 9 : exponentiation rapide

À l'échelle électronique de l'ordinateur, l'exponentiation n'est pas une opération élémentaire : il faut donc une méthode efficace pour obtenir l'exponentiation à partir de la multiplication.

1. Écrire une fonction `expo_naive(a,b)` qui calcule a^b en utilisant la formule $a^b = a \times a \times a \times \ldots \times a$.
2. Justifier que :
   — si b est pair, $a^b = (a^{b/2}) \times (a^{b/2})$;
   — si b est impair, $a^b = (a^{b/2-1}) \times (a^{b/2-1}) \times a$.
3. Exploiter ce résultat pour écrire une fonction récursive `expo_rapide(a,b)` qui permet de calculer a^b.
4. Tester les deux algorithmes pour calculer $2^{1\,000\,000}$: l'exponentiation rapide est sensiblement plus rapide.

Exercice 10 : énumération des sous-ensembles

En informatique, il est parfois nécessaire d'énumérer exhaustivement tous les sous-ensembles pour vérifier un résultat. Ici, on considérera qu'un ensemble est un tableau `tab` : ses cases sont les éléments de notre ensemble.
Écrire une fonction `sous_ens(tab)` qui renvoie un (très grand) tableau, dont chaque case est un sous-ensemble de `tab` (et qui les contient tous). Par exemple, `sous_ens([1,2])` renverra `[[], [1], [2], [1, 2]]`.

Exercice 11

Un mot est un palindrome s'il peut se lire identiquement de gauche à droite et de droite à gauche, comme `kayak` et `hannah`.

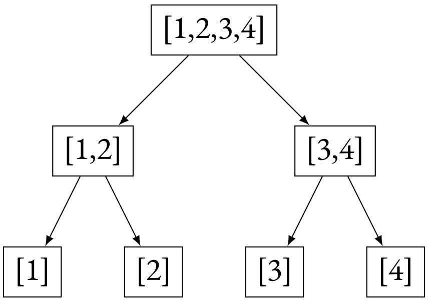
1. Écrire une fonction `est_pal(mot)` **non récursive** qui renvoie `True` si l'entrée est un palindrome.
2. Écrire une seconde fonction `est_pal_rec(mot)`, **récursive**, qui renvoie la même chose.

Partie 4 : arbre d'appels récursifs

Lorsqu'on exécute une fonction récursive, les appels faits à la fonction se font dans un ordre précis, dicté par le code de la fonction.

Considérons le code ci-contre. Lors de l'exécution de `somme_rec([1,2,3,4])`, on fait des appels à la fonction sur des entrées plus petites. On peut voir ces appels sous la forme d'un **arbre** :

```python
def somme_rec(tab):
    n = len(tab)
    if n == 0:
        return 0
    elif n == 1:
        return tab[0]
    else:
        s1 = somme_rec(tab[:n//2])
        s2 = somme_rec(tab[n//2:])
        return s1+s2
```



Un rectangle représente un appel à la fonction, et il pointe vers les sous-appels faits lors de son exécution. Tout en bas, on trouve les exécutions qui ne font pas d'appels récursifs : ils correspondent aux cas de base.

Exercice 12

1. Tracer l'arbre d'appels récursifs de `expo_rapide(2,6)` (voir exercice 9).
2. Tracer l'arbre d'appels récursifs de `fibo(5)` (voir exercice 1). En déduire pourquoi la méthode récursive est particulièrement peu efficace dans ce cas-là.
3. Tracer l'arbre d'appels récursifs de `fonction(5)` (voir exercice 2) : que remarquez-vous ?
4. On retourne sur `fibo` : chaque appel récursif demande au moins une opération élémentaire. Montrer que si on note C_n le nombre d'opérations nécessaires pour exécuter `fibo(n)`, alors $C_{n+2} \geqslant C_{n+1} + C_n$, et en déduire qu'il faut au moins un nombre exponentiel en `n` d'opérations pour exécuter `fibo(n)`.

Pour la prochaine fois :

— Écrire une fonction récursive `somme(n)` qui renvoie la somme des entiers de 0 à n inclus.

— Écrire une fonction récursive `miroir(texte)` qui renvoie le miroir de `texte`.

— Écrire une fonction récursive `nb_occurrences(elem,tab)` qui renvoie le nombre d'occurrences de `elem` dans `tab`.

— Écrire une fonction récursive `coef_binom(k,n)` qui calcule $\binom{n}{k}$.

— Écrire une fonction récursive `multip_rapide(a,b)` qui renvoie $a \times b$ en utilisant exclusivement des additions, sur le modèle de l'exponentiation rapide.