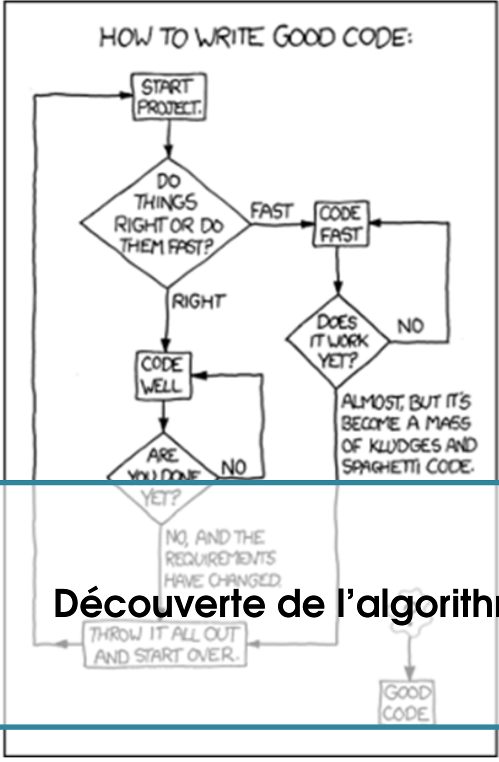
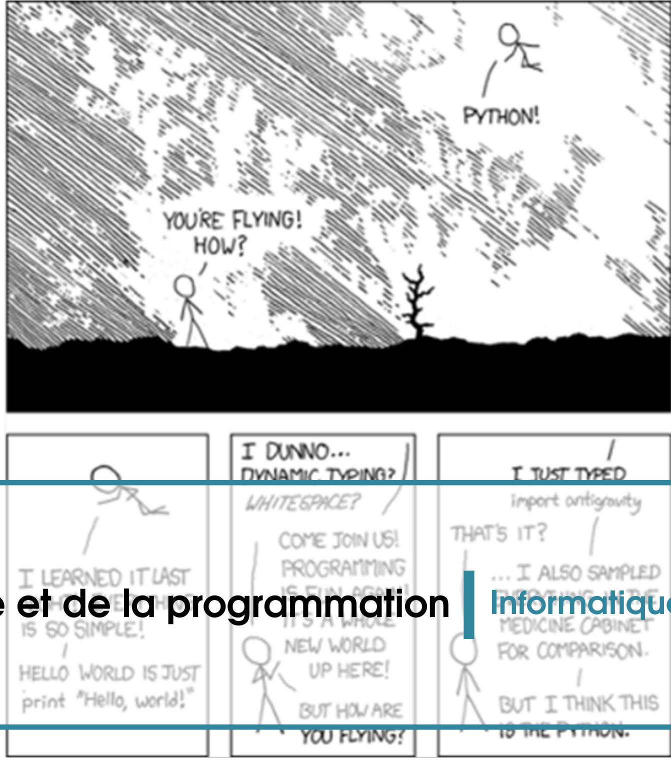




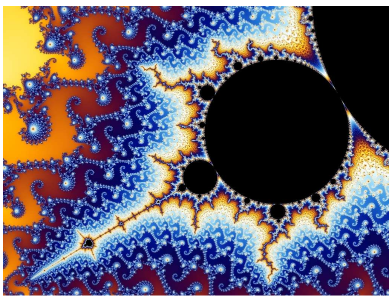
Cours

Chapitre 4

Récursivité

Savoirs et compétences :

- ❐ *Programmation récursive.*



1 Définitions

Définition Fonctions récursives Une fonction récursive est une fonction qui s'appelle elle-même.
On appelle récursion l'appel de la fonction à elle-même.

La programmation récursive est un paradigme de programmation au même titre que la programmation itérative. Un programme écrit de manière récursive peut être traduit de manière itérative, même si dans certains cas, cela peut s'avérer délicat.

Méthode
- Une fonction récursive doit posséder une condition d'arrêt (ou cas de base).
- Une fonction récursive doit s'appeler elle-même (récursion).
- L'argument de l'étape de récursion doit évoluer de manière à se ramener à la condition d'arrêt.

Les avantages et inconvénients d'un algorithme récursif :
- simplicité de l'écriture récursive dans certains cas ;
- l'algorithme peut sembler plus aisé lors de sa lecture ;
- comme pour les algorithmes itératifs, il faut prêter attention à sa terminaison en utilisant un variant de boucle (à voir ultérieurement) ;
- comme pour les algorithmes itératifs, il est aussi nécessaire de vérifier la correction de l'algorithme en utilisant un invariant de boucle (à voir ultérieurement) ;
- les complexités algorithmiques temporelle et spatiale (à voir ultérieurement) d'un algorithme récursif peuvent être plus coûteuses que celles d'un algorithme itératif.

2 Suites définies par récurrence

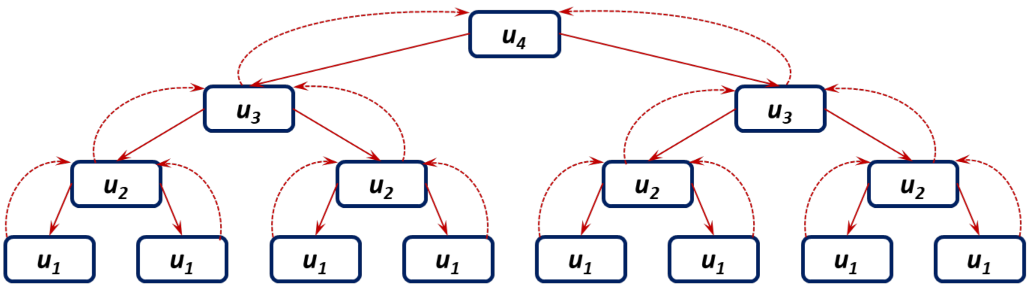
Les suite définies par récurrence pour lesquelles $u_n = f(u_{n-1}, u_{n-2}, \ldots)$ sont des cas d'application directs des fonctions récursives.

Par exemple, soit la suite u_n définie par récurrence pour tout $n \in \mathbb{N}^*$ par $\begin{cases} u_1 = 1 \\ u_{n+1} = \dfrac{u_n + 6}{u_n + 2} \end{cases}$. Il est possible de calculer le n^{e} terme par un algorithme itératif ou un algorithme récursif.

```
def un_it (n : int) -> float :
    if n == 1 :
        return 1
    else :
        u = 1
        for i in range(2,n+1):
            u = (u+6)/(u+2)
        return u
```

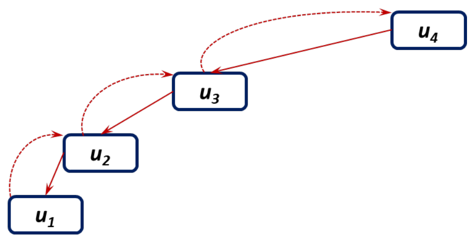
```
def un_rec (n : int) -> float :
    if n == 1 :
        return 1
    else :
        return (un_rec(n-1)+6)/(un_rec(n-1)+2)
```

La figure suivante montre que dans le cas de l'algorithme récursif proposé plusieurs termes sont calculés à plusieurs reprises ce qui constitue une perte de temps et d'espace mémoire.

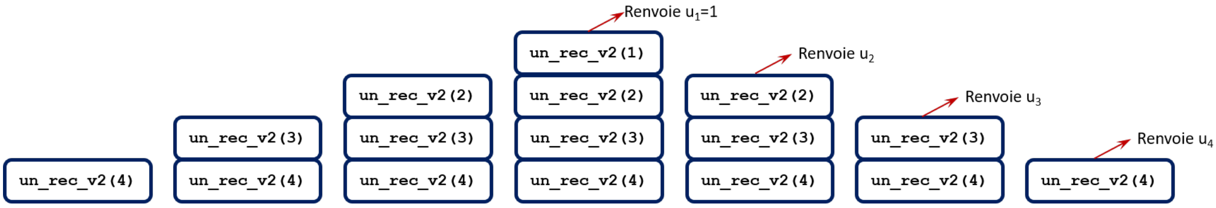


Une autre formulation de l'algorithme récursif permet très simplement de diminuer le nombre de termes calculés.

```
def un_rec_v2 (n : int) -> float :
    if n == 1 :
        return 1
    else :
        v = un_rec_v2(n-1)
        return (v+6)/(v+2)
```



Dans la même idée que les graphes présentés ci-dessus, il est possible de représenter la pile des appels récursifs, c'est à dire la succession des appels qui vont être faits pour calculer le nième terme d'une suite.



3 Slicing de tableau ou de chaînes de caractères

Il est possible d'agir par récursivité sur des listes ou sur des chaînes de caractère. Pour cela, il peut être nécessaire d'utiliser le slicing. Pour rappel, `a = ch[i:j]` permet d'affecter à `a` la liste (ou le chaîne de caractéres) constitué des éléments de `i` (inclus) à `j` (exclus). En écrivant `a = ch[i:]` on affecte à `a` tous les éléments du ième au dernier.

Ainsi, pour réaliser la somme des éléments d'une liste `L` par récursivité :

- commençons par déterminer le cas terminal : si la taille de la liste vaut 1, on renvoie `L[0]` ;
- sinon (si la taille vaut `n`), on peut choisir de renvoyer `L[0]+somme(L[1]+L[2]+...+L[n-1])`.

Cela se traduit ainsi.

```
def somme(L:list) -> float :
    if len(L)==1 :
        return L[0]
    else :
        return L[0]+somme(L[1:])
```

Pour une chaîne de caractères, si on souhaite renvoyer son miroir (par exemple, le miroir de `abc` serait `cba`) :

- si la chaîne de caractère a une taille de 1, on renvoie le caractère ;
- sinon, on renvoie la concaténation de `miroir(ch[1:])+ch[0]`.

```
def miroir(ch:str) -> str :
    if len(ch)==1 :
        return ch[0]
    else :
        # Attention le + désigne la concaténation
        return miroir(ch[1:])+ch[0]
```

4 Algorithmes dichotomiques – Diviser pour régner

Les algorithmes dichotomiques se prêtent aussi à des formulations récursives. Prenons comme exemple la recherche d'un élément dans une liste triée. L'algorithme de gauche propose une version itérative. L'algorithme de droite une version récursive.

```
def appartient_dicho(e : int , t : list) -> bool:
    """Renvoie un booléen indiquant si e est
    dans t. Préconditions : t est un tableau
    de nombres trié par ordre croissant e est
    un nombre"""
    # Limite gauche de la tranche où l'on recherche e
    g = 0
    # Limite droite de la tranche où l'on recherche e
    d = len(t)-1
    # La tranche où l'on cherche e n'est pas vide
    while g <= d:
        # Milieu de la tranche où l'on recherche e
        m = (g+d)//2
        pivot = t[m]
        if e == pivot: # On a trouvé e
            return True
        elif e < pivot:
            # On recherche e dans la partie gauche de la tranche
            d = m-1
        else:
            # On recherche e dans la partie droite de la tranche
            g = m+1
    return False
```

```
def appartient_dicho_rec(e : int , t : list) -> bool:
    # Limite gauche de la tranche où l'on recherche e
    g = 0
    # Limite droite de la tranche où l'on recherche e
    d = len(t)-1
```

```
    # La tranche où l'on cherche e n'est pas vide
    # Milieu de la tranche où l'on recherche e
    m = (g+d)//2
    if t==[]:#e n'est pas dans t
        return False
    elif t[m]==e:#e est dans t
        return True
    elif t[m]<e:#On recherche àdroite
        return appartient_dicho_rec(e, t[m+1:])
    elif t[m]>e:#On recherche àgauche
        return appartient_dicho_rec(e, t[:m])
```

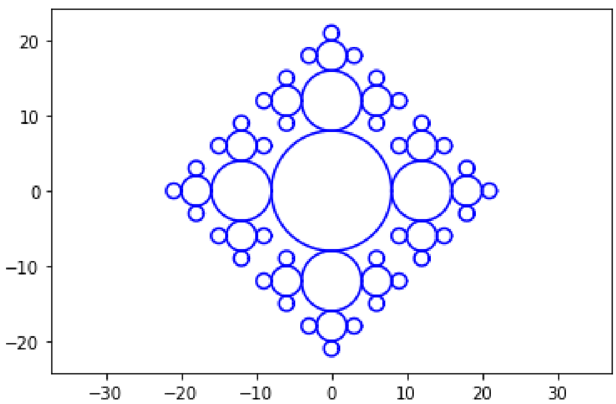
5 Tracer de figures définies par récursivité

Un grand nombre de figures peuvent être tracées en utilisant des algorithmes récursifs (flocon de Koch, courbe de Peano, courbe du dragon *etc.*).

Ci-dessous un exemple de figure définie par récursivité où à chaque itération un cercle va se propager vers le haut, le bas, la gauche et la droite. À chaque itération, le rayon de cercle est divisé par 2.

```
import matplotlib.pyplot as plt
import numpy as np
def cercle(x,y,r):
    theta = np.linspace(0, 2*np.pi, 100) #des points régulièrement espacés dans l'intervalle
        [0,2pi]
    X = [x+r*np.cos(t) for t in theta] #abscisses de points du cercle C((x,y),r)
    Y = [y+r*np.sin(t) for t in theta] #ordonnées de points du cercle C((x,y),r)
    plt.plot(X,Y,"b") #tracé sans affichage
```

```
def bubble(n, x, y, r, d):
    cercle(x, y, r)
    if n > 1:
        if d != 's':
            bubble(n-1,x,y+3*r/2,r/2,"n")
        if d != 'w':
            bubble(n-1,x+3*r/2,y,r/2,"e")
        if d != 'n':
            bubble(n-1,x,y-3*r/2,r/2,"s")
        if d != 'e':
            bubble(n-1,x-3*r/2,y,r/2,"w")
bubble(4,0,0,8,"")
plt.axis("equal")
plt.show()
```



Application 01

Applications – Bases

Savoirs et compétences :

- .

Exercice 1 – Multiplication

Question 1 *Ecrire une fonction itérative* `mult(n:int, p:int)->int` *permettant de multiplier deux entiers (sans utiliser* `*` *).*

Question 2 *Ecrire une fonction récursive* `mult_rec(n:int, p:int)->int` *permettant de multiplier deux entiers (sans utiliser* `*` *).*

Question 3 *Énoncer un variant de boucle et montrer la terminaison de l'algorithme.*

Question 4 *Énoncer un invariant de boucle et montrer la correction de l'algorithme.*

Question 5 *Donner et justifier la complexité temporelle de la fonction* `mult`.

Question 6 *Donner et justifier la complexité spatiale de la fonction* `mult`.

Exercice 2 – Exponentiation

Question 1 *Ecrire une fonction itérative* `expo(x:int, n:int)->int` *permettant de calculer x^n (sans utiliser* `**` *).*

Question 2 *Ecrire une fonction récursive* `expo_rec(x:int, n:int)->int` *permettant de calculer x^n (sans utiliser* `**` *).*

Question 3 *Énoncer un variant de boucle et montrer la terminaison de l'algorithme.*

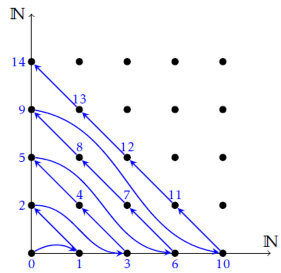
Question 4 *Énoncer un invariant de boucle et montrer la correction de l'algorithme.*

Question 5 *Donner et justifier la complexité temporelle de la fonction* `puiss`.

Question 6 *Donner et justifier la complexité spatiale de la fonction* `puiss`.

Exercice 3 –

On démontre que sur l'ensemble $\mathbb{N}\times\mathbb{N}$ est dénombrable en numérotant chaque couple $(x,y)\in\mathbb{N}^2$ suivant le procédé suggéré par la figure ci-dessous.



Question 1 *Rédiger une fonction récursive qui retourne le numéro du point de coordonnées (x,y).*

Question 2 *Rédiger la fonction réciproque, là encore de façon récursive.*

Exercice 4 –

Question 1 *Prouver la terminaison de la fonction G de Hofstadter, définie sur $\mathbb{N}$ de la façon suivante :*

■ **Python**

```python
def g(n) :
    if n== 0 :
        return 0
    return n-g(g(n-1))
```

Exercice 5 –

On suppose donné un tableau $t[0,..,n-1]$ (contenant au moins trois éléments) qui possède la propriété suivante : $t_0 \geq t_1$ et $t_{n-2} \leq t_{n-1}$. Soit $k \in [|1,n-2|]$; on dit que t_k est un minimum local lorsque $t_k \leq t_{k-1}$ et $t_k \leq t_{k+1}$.

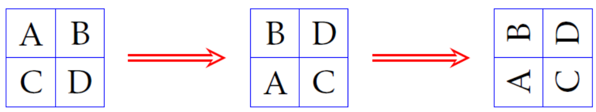
Question 1 *Justifier l'existence d'un minimum local dans t.*

Question 2 *Il est facile de déterminer un minimum local en coût linéaire : il suffit de procéder à un parcours de tableau. Pourriez-vous trouver un algorithme récursif* *qui en trouve un en réduisant le coût logarithmique ?*

Exercice 6 –

Les processeurs graphiques possèdent en général une fonction de bas niveau appelée *blit* (ou transfert de bloc) qui copie rapidement un bloc rectangulaire d'une image d'un endroit à un autre.

L'objectif de cet exercice est de faire tourner une image carrée de $n \times n$ pixels de 90°dans le sens direct en adoptant une stratégie récursive : découper l'image en quatre blocs de tailles $n/2 \times n/2$, déplacer chacun des ces blocs à sa position finale à l'aide de 5 *blits*, puis faire tourner récursivement chacun de ces blocs.



On supposera dans tout l'exercice que n est une puissance de 2.

Question 1 *Exprimer en fonction de n le nombre de fois que la fonction blit est utilisée.*

Question 2 *Quel est le coût total de cet algorithme lorsque le coût d'un blit d'un bloc k × k est en $\mathcal{O}(n^2)$?*

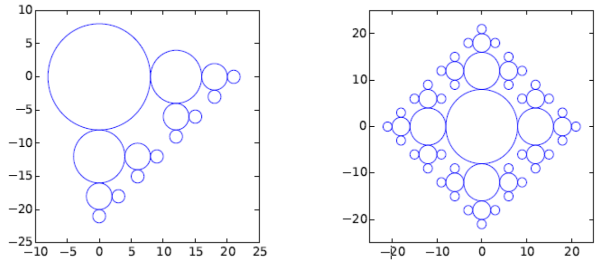
Question 3 *Et lorsque ce coût est en $\mathcal{O}(n)$?*

Question 4 *En supposant qu'une image est représentée par une matrice numpy n × n, rédiger une fonction qui adopte cette démarche pour effectuer une rotation de 90°dans le sens direct (on simulera un blit par la copie d'une partie de la matrice vers une autre en décrivant ces parties par le slicing).*

Exercice 7 –

On suppose disposer d'une fonction *circle([x, y], r)* qui trace à l'écran un cercle de centre $(x; y)$ de rayon r.

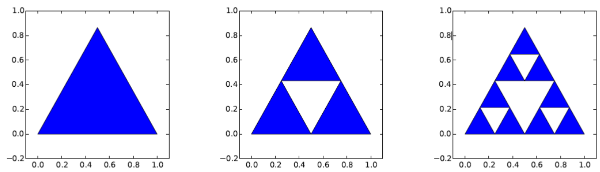
Question 1 *Définir deux fonctions récursives permettant de tracer les dessins présentés figure suivante (chaque cercle est de rayon moitié moindre qu'à la génération précédente).*



Exercice 8 –

On suppose disposer d'une fonction polygon((xa, ya), (xb, yb), (xc, yc)) qui trace le triangle plein dont les sommets ont pour coordonnées $(x_a; y_a)$, $(x_b; y_b)$, $(x_c; y_c)$.

Question 1 *Définir une fonction récursive permettant le tracé présenté figure suivante (tous les triangles sont équilatéraux).*



Exercice 9 –

Question 1 *Écrire une fonction récursive qui calcule a^n en exploitant la relation : $a^n = a^{n/2} \times a^{n/2}$.*

Question 2 *Écrire une fonction qui utilise de plus la remarque suivante :*

$$n/2 = \begin{cases} n/2 & \text{si } n \text{ est pair} \\ n/2+1 & \text{sinon} \end{cases}.$$

Question 3 *Déterminer le nombre de multiplications effectuées dans le cas où n est une puissance de 2, et majorer simplement ce nombre dans le cas général.*

Exercice 10 – Fonction 91 de McCarthy

On considère la fonction récursive suivante :

■ Python
```python
def f(n) :
    if n>100 :
        return n-10
    return f(f(n+11))
```

Question 1 *Prouver sa terminaison lorsque $n \in \mathbb{N}$ et déterminer ce qu'elle calcule (sans utiliser l'interpréteur de commande).*