Nous verrons dans ce chapitre ce qu'est la récursivité, son utilité et ses limites. Dans ce chapitre, vous mettrez en application ces concepts au travers d'exemples simples, les exemples plus complexes seront traités au travers des autres TPs de l'année (tris, graphes, etc.)

I Les fonctions récursives

1 Rappels : suites arithmético-géométriques

expl : la suite récursive $\begin{cases} u_{n+1}=0{,}8u_n+4 \\ u_0 \quad =-2 \end{cases}$, cherchons son expression explicite.

Recherche du point fixe : ...
Suite auxiliaire : ...

Résolution : ...

2 La récursivité

On dit qu'une fonction f est récursive lorsque son exécution provoque un ou plusieurs appels de f elle-même. Il y a alors un appel ..... et des appels .....

Comme nous le verrons, une récursivité mal gérée peut très vite accaparer l'ensemble des ressources de l'ordinateur et ainsi le paralyser. Ainsi pour se protéger, les langages de programmation des années 50-60 ne permettaient pas la récursivité mais de nos jours la grande majorité des langages l'acceptent.

expl : cherchons à calculer la factorielle $n! = \prod_{i=1}^{n} i$

Une possibilité serait :

```
def factorielle(n):
    result=1
    for i in ...
        ...
    ...
```

Néanmoins puisque la même opération est appliquée, on pourrait proposer une fonction récursive. En posant $u_n = n!$, la factorielle se définit par la récurrence $u_n = \begin{cases} \ldots & \text{si n=0} \\ \ldots\ldots\ldots & \text{sinon} \end{cases}$, et ainsi :

```
def factorielleR(n):
    if ...
        ...
    else :
        ...
```

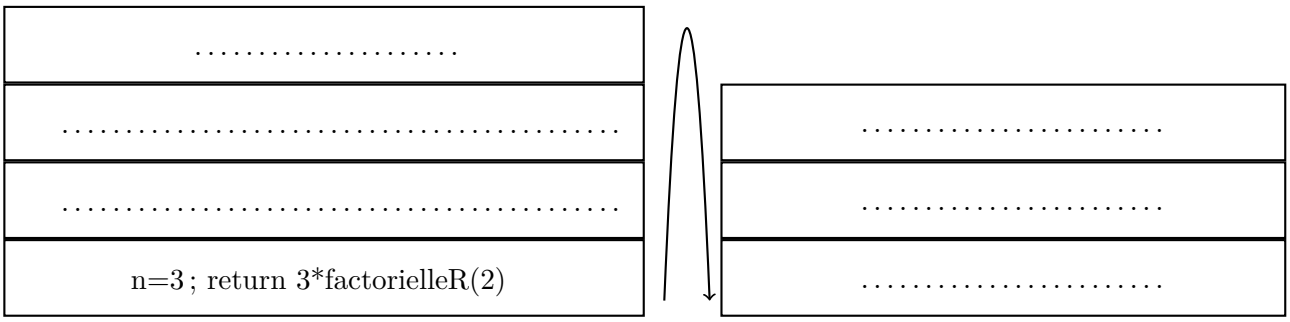
La fonction `factorielleR` va multiplier par n le résultat que renverra la même fonction mais avec le paramètre n-1 et ainsi de suite. *In fine*, on a bien calculé la factorielle de n.

3 Pile d'exécution d'une fonction récursive

Lorsque l'ordinateur exécute un programme, il met en mémoire le code des instructions ainsi que l'état de certaines variables. Il en va de même lors d'appels de fonction, le code est chargé ainsi que les paramètres. Puis une fois la fonction évaluée, cette mémoire est libérée.

On peut donc représenter les exécutions comme une pile d'instructions et de valeurs. Lorsqu'une fonction est appelée, la fonction ainsi que ses paramètres sont empilés, puis une fois exécuter, la pile est dépilée.

Si on appelle `factorielleR(3)`, que comporte la pile ?



On peut dès lors remarquer que si la fonction s'appelle elle-même de nombreuses fois, la pile va très vite augmenter et pourra certainement .....

Rq : en python, la récursivité est limitée à 1 000 appels.

4 Terminaison d'une fonction récursive

Comment être certain que la fonction s'arrêtera ?

On utilise en général la descente de Fermat (ou descente infinie) afin de montrer que la fonction se termine (*i.e.* qu'il y aura un nombre fini d'instructions exécutées).

expl : considérons la fonction `factorielleR` définie précédemment.

On considère le paramètre n qui est un entier donc ..... Puisque $factorielleR(0)$ se termine et renvoie 1, il suffit de vérifier que ..... Ce qui est le cas, donc la fonction `factorielleR` n'effectuera qu'un nombre fini d'instructions et se terminera.

expl : la terminaison de la suite de Syracuse n'a toujours pas été démontrée. On rappelle que cette suite est définie par u_0 entier et $u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$.

5 Correction d'une fonction récursive

Comment être certain que la fonction renverra le bon résultat ?

On utilise en général un résonnement par récurrence pour montrer qu'elle renvoie bien le bon résultat.

expl : définissons H_n l'hypothèse de récurrence .....

H_0 est .....

Supposons H_n vérifiée, ...

.....

H_n est donc vraie quelque soit n entier et la fonction récursive `factorielleR` est .....

6 Complexité d'une fonction récursive

La complexité a été vue en première année. Il existe principalement deux types de complexité : .....

.....

Dans nos approches, on considérera la complexité ..... comme la complexité arithmétique en comptant le nombre d'opérations faites.

La complexité ..... sera considérée comme la place mémoire nécessaire à l'exécution du programme.

expl : `factorielle` nécessite
. unités de mémoire (.....)
... opérations (.....)

expl : `factorielleR` nécessite
. unités de mémoire (.....)
le coût temporel est donné par la suite T_n

où $\begin{cases} T_0 = 1 \\ T_n = 3 + T_{n-1} \end{cases}$ (un test, une multiplication et une soustraction, puis l'appel de `factorielleR`)

Ce qui donne un coût temporel .....

7 Conclusion

◇ **avantages :**

(a) comme pour les suites définies de manière récurrente, à partir d'un problème il est facile d'écrire une fonction récursive et parfois difficile à écrire de manière itérative,

(b) une fonction récursive est en général plus facile à comprendre que son homologue itérative,

(c) les algorithmes "divisés pour régner" (*cf.* chapitre des tris) sont très efficaces.

◇ **inconvénients :**

(a) les appels récursifs sont gourmands en mémoire,

(b) la complexité temporelle explose très vite si plusieurs appels récursifs sont faits au sein de la même fonction,

(c) lorsque deux solutions équivalentes itérative et récursive existe, il est préférable de choisir l'itérative.

II Utilisation de la récursivité

1 La méthode de Héron

Traitons ensemble un cas particulier de la méthode de Newton pour trouver la racine carrée d'un nombre : la méthode de Héron.

La méthode est attribuée à Héron d'Alexandrie (I^{er}S ap JC) mais était connu des babyloniens (VI^{e}S av JC). Cette méthode est très efficace car à chaque itération, 2 décimales sont calculées. Afin d'extraire la racine carrée d'un nombre positif a, on définit par récurrence la suite u_n par $\begin{cases} u_0 = 1 \\ u_{n+1} = \frac{1}{2}\left(u_n + \frac{a}{u_n}\right) \end{cases}$.

Définissons la suite définie par $\begin{cases} u_0 = 1 \\ u_{n+1} = \frac{1}{2}\left(u_n + \frac{5}{u_n}\right) \end{cases}$ qui approxime.....

On peut alors définir trois fonctions qui satisfassent le calcul des termes de cette suite.

```
def SuiteU(n):
    if n==0: return 1
    else: ...

def SuiteV(n):
    if n==0: return 1
    else:
        ...
        ...

def SuiteW(n):
    w=1
    for i in ...
        ...
    ...
```

La fonction `suiteW` est définie de manière itérative avec une boucle `for` donc se termine. En étudiant les variations du paramètre n, celui-ci décroit strictement dans les fonctions `SuiteU` et `SuiteV`, les deux fonctions se terminent donc.

Les fonctions sont également correctes en considérant l'hypothèse de récurrence H_n .....
.....

Il y a par contre une différence de coûts entre chacune des fonctions.
En effet, le coût mémoire est donné par les relations :

$\begin{cases} MU_0 = \ldots\ldots \\ MU_n = \ldots\ldots\ldots\ldots\ldots\ldots \end{cases}$, $\begin{cases} MV_0 = \ldots\ldots \\ MV_n = \ldots\ldots\ldots\ldots\ldots\ldots \end{cases}$ et $MW_n = \ldots\ldots$

Ce qui donne

Fonction	SuiteU	SuiteV	SuiteW
Coût mémoire	...	...	...

Le coût temporel est donné par les relations suivantes (on compte la comparaison du `if`) :

$$\begin{cases} TU_0 = \ldots\ldots \\ TU_n = \ldots\ldots\ldots\ldots\ldots\ldots \end{cases}, \quad \begin{cases} TV_0 = \ldots\ldots \\ TV_n = \ldots\ldots\ldots\ldots\ldots\ldots \end{cases} \quad \text{et } TW_n = \ldots\ldots$$

Ce qui donne

Fonction	SuiteU	SuiteV	SuiteW
Coût temporel	...	...	...

On voit clairement que la fonction `SuiteU` avec ces deux appels récursifs explose sa complexité ..... et sa complexité .....

Résumé

exemples	**relation de récurrence**	**coût temporel**	**durée (n=250)**	**durée (n=10 000)**
def f(n) **:** **return** f(n//2)			$\sim$ 24 ns	$\sim$ 40 ns
def f(n) **:** **return** f(n-1)			$\sim 1\mu$s	$\sim 100\ \mu$s

def f(n) **:**
  for i **in range(n) :**
  a=i
  return f(n//2)				

def f(n) **:** **return** f(n//2)+f(n//2)				

def f(n) **:**
  for i **in range(n) :**
  a=i
  return f(n//2)+f(n//2)
			$\sim 6\mu$s	$\sim 920\mu$s

def f(n) **:**
  for i **in range(n) :**
  a=i
  return f(n-1)
			$\sim 625\mu$s	$\sim$ 1s

def f(n) **:** **return** f(n-1)+f(n-1)			$\sim 10^{68}$ans (!)	...

Rq : pour les ordres de grandeurs des durées, on a utilisé le $log_{10}()$ et considéré qu'une opération = 10 ns.
Rq : L'univers serait âgé de $\sim 10^{10}$ années et il contiendrait un total de $\sim 10^{80}$ atomes.

2 Exercices

Exo 1 : Déterminer la forme explicite de la suite définie par $u_0 = 20$ et $u_{n+1} = \frac{1}{7}u_n + 12$.

Exo 2 : Pour chacune des fonctions suivantes,

1. déterminer ce qu'elles font (faites des essais)
2. (facultatif) calculer les coûts spatiaux et temporels (pour mystere2, prendre $2m \approx n$).

```
from math import sqrt
def mystere1(m):      # m est un entier naturel
    if m==2 : return True
    if m==1 or m%2==0: return False
    return mystereR1(m,3)
def mystereR1(m, n):
    if n>sqrt(m) : return True
    elif m%n==0 :  return False
    return mystereR1(m,n+2)
```

```
# m et n sont deux entiers naturels avec m<=n
def mystere2(m, n):
    if m==0 or m==n : return 1
    return mystere2(m-1,n-1)+mystere2(m,n-1)
```

```
# n est un entier naturel
def mystere3(n):
    if n==0: return 0
    mystere3(n//2)
    print (n%2,end='')  # end='' permet de ne pas aller a la ligne a la fin
       du print
```

Exo 3 : Écrire une fonction récursive qui vérifie qu'un mot est un palindromme.

Exo 4 : On souhaite calculer le pgcd de deux nombres entiers p et q non nuls à l'aide de l'algorithme d'Euclide.
On rappelle ci-contre les étapes de cet algorithme pour trouver : pgcd(10 ;24)=2.

p	q	r
24	10	4
10	4	2
4	2	0

1. appliquer l'algorithme d'Euclide pour trouver le pgcd de 35 et 126,
2. en utilisant le reste de la division euclidienne (noté p % q), proposer une solution itérative,
3. en n'utilisant cette fois que les propriétés suivantes (*pgcd(p,q)=pgcd(q,p) ; pgcd(p,q)=pgcd(p-q,q)* et *pgcd(p,0)=p*), proposer une solution récursive.

Exo 5 : On souhaite calculer les termes de la suite de Fibonacci. On rappelle qu'elle est définie par $f_0 = 0$, $f_1 = 1$ et la relation $\forall n \in \mathbb{N}^*$, $f_{n+1} = f_n + f_{n-1}$.

Pour la suite, vous pouvez soit répondre à l'aide d'un algorithme soit à l'aide d'une implémentation en python.

1. proposer une fonction récursive naïve (`fiboN`) qui permette de calculer f_n,
2. proposer une fonction itérative (`fiboI`) qui permette de calculer f_n,

   Afin de garder un calcul récursif, on peut faire une mémoïsation des dernières valeurs f_n calculées. Ces valeurs vont alors être passées en paramètres.
3. Compléter la fonction suivante (ou proposer votre propre solution) qui permet un seul appel réccursif

```
def fibo(n,a=0,b=1):
    if xxx : return a
    elif n==xxx: return b
    else : return fibo(xxx)
```

4. préciser le coût temporel (un $\mathcal{O}$ suffit) de chacune des fonctions. On considèrera que $fiboN(n-1)$ et $fiboN(n-2)$ possèdent le même coût.

Exo 6 : sur tout l'exercice on n'utilisera pas l'opérateur puissance ('**').

1. Proposer un algorithme récursif naïf pour le calcul de la puissance n d'un nombre x.
2. Proposer un algorithme itératif naïf qui calcule x^n.
3. Considérant les relations $\begin{cases} x^{2n} &= (x^n)^2 \\ x^{2n+1} &= x(x^n)^2 \end{cases}$, vous proposerez une fonction plus efficace pour calculer x^n.

Expliquons un peu les relations données sur un exemple avec (x,n)=(x,11).
Puisque l'on a $11 = 5\times2+1 = (2\times2+1)\times2+1 = ((1\times2+0)\times2+1)\times2+1 = (((0\times2+1)\times2+0)\times2+1)\times2+1$, on a aussi $x^{11} = \left(x^5\right)^2 \times x = \left(\left(x^2\right)^2 \times x\right)^2 \times x = \left(\left((x\times1)^2\right)^2 \times x\right)^2 \times x = \left(\left((1\times x)^2\right)^2 \times x\right)^2 \times x$
L'appel récursif 'remonte' alors le calcul de la puissance à partir de la parenthèse la plus 'profonde' jusqu'à la dernière multiplication de 'surface'.

La fonction remonte donc les étapes de calculs de la manière suivante :

étape de calcul	k	result
debut		1
$0\times2+1$	1	x
$1\times2+0$	2	x^2
$2\times2+1$	5	x^5
$5\times2+1$	11	x^{11}

Proposer votre solution ou compléter la fonction `puissanceR` (*cf.* xxx) ci-dessous.

```
def puissanceR(x,n):
    if n==0: return 1
    else :
        r= puissanceR(x,n//2)
        if n % 2 == 0: return xxx
        else : return xxx
```

4. De la même manière, on souhaite proposer un algorithme itératif qui utilise les relations du **3.**. Il va donc falloir diviser successivement n par 2, se souvenir du reste de chaque division et effectuer le calcul ensuite.
Proposer votre solution ou bien compléter le code suivant (*cf.* xxx).

```
def puissanceI(x,n):
    p=[]# une pile est cree
    result=1
    while(n>0):# cette boucle permet de simuler l'appel recursif a l'aide d'
        une pile
        p.append(xxx)
        n = xxx
    while(len(p)>0):# on va calculer le resultat en depilant la pile
        reste = p.pop(-1)# on regarde le dernier reste en depilant
        if( xxx ):
            result = xxx
        else :
            result = xxx
    return xxx
```

5. pour les 2 algorithmes naïfs
   (a) montrer qu'ils se terminent et qu'ils sont corrects.
   (b) calculer le coût mémoire et temporel (un $\mathcal{O}$ suffit).
6. préciser le coût mémoire et temporel du 3ème algorithme (`puissanceR`) (un $\mathcal{O}$ suffit).

Exo 7 : On pourra compléter le fichier `CPGE-Info-Recursivite-Exo-Fractals.py`

L'ensemble de Cantor est un sous-ensemble fermé de $\mathbb{R}$, d'intérieur vide, non dénombrable. C'est, au même titre que le flocon de Von Koch, le tapis de Sierpinski, l'éponge de Menger, les ensembles de Julia et l'ensemble de Mandelbrot, un fractal.

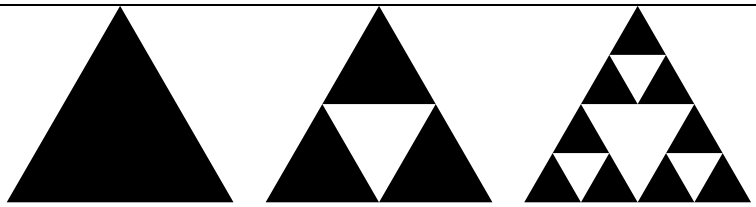
Il est défini de manière récursive partant d'un intervalle I, on le subdivise en 3 et on retire le sous-intervalle central. On réitère le procédé avec les intervalles qui restent et ce une infinité de fois.

1. proposer une fonction qui dessine de manière récursive l'ensemble de Cantor en fonction du nombre d'itérations n. On pourra s'inspirer du code ci-dessous qui donne l'image de droite (rq : pour être précis, il faudra modifier l'épaisseur de ligne (linewidth)).

```
import matplotlib.pyplot as plt
plt.plot([0,.33],[0,0],color='k',linewidth=5)
plt.plot(visible=False)
plt.plot([.67,1],[0,0],color='k',linewidth=5)
plt.show()
```


2. On dispose de la fonction `polygon([`x_a`, `y_a`], [`x_b`, `y_b`], [`x_c`, `y_c`])` qui trace et rempli le triangle dont les sommets ont pour coordonnées (x_a,y_a), (x_b,y_b), (x_c,y_c).

   Définir une fonction récursive qui trace le triangle de Sierpinski en fonction du nombre d'itérations n (*cf.* figure, les triangles y sont tous équilatéraux).

triangle_sierpinski(n)	n=1	n=2	n=3
résultat			

Pour l'implémentation sous python, on pourra compléter le code suivant (*cf.* xxx) :

```
import matplotlib.pyplot as plt
from numpy import sqrt

def polygon(*args):
    X,Y = [],[]
    for arg in args:
        X.append(arg[0])
        Y.append(arg[1])
    plt.fill(X, Y, 'k')

def triangle_sierpinski(n,a=[0,0],b=[1,0],c=[.5,sqrt(3)/2]): # triangle
    equilateral de cote 1
    if n == 1:
        xxx
    else:
        xxx # correspond a plusieurs lignes
    plt.show()
```