# RÉCURSIVITÉ

## 1 Le principe

Regarder attentivement le programme suivant.

```
def depart(n):
    if n == 0:
        print("partez !")
        return None
    else:
        print(n)
        depart(n-1)
```

La fonction `depart` s'appelle elle-même ! Et oui. Si dans la console, on entre `depart(3)`, alors :

- `depart(3)` va afficher 3 puis appeler `depart(2)`.
- `depart(2)` va afficher 2 puis appeler `depart(1)`.
- `depart(1)` va afficher 1 puis appeler `depart(0)`.
- `depart(0)` affiche `partez !` et ne fait rien d'autre : fin de la procédure.

**Définition 9.1**

Une fonction est dite récursive si elle peut s'appeler elle-même lors de son exécution.

**Exercice 1.** Que se passe-t-il si on exécute `depart(-1)` ? Et `depart(3.5)` ? Modifier la fonction pour éviter ces problèmes, et tester.

Il est primordial que la fonction se termine toujours et ne s'appelle pas à l'infini. Il faut donc une *condition d'arrêt*. Une fonction récursive commencera donc généralement par un `if <condition> :` suivi d'une commande `return`.

## 2 Itératif vs récursif

Quand une opération doit être répétée plusieurs fois, on peut le faire de manière *itérative* (avec une boucle `for` ou `while`) ou récursive.

**Exemple.** Voici un exemple classique pour calculer la factorielle d'un entier naturel :

```
def factoIter(n):       # version itérative
    p = 1
    for k in range(n):
        p = p*(k+1)
    return p
```

```
def factoRecu(n):       # version récursive
    if n==0:
        return 1
    else:
        return n*factoRecu(n-1)
```

**Exemple.** Autre classique : le calcul du pgcd de deux entiers. Voici une version itérative :

```
1 def pgcdIter(a,b):      # version itérative
2     if b == 0:
3         return a
4     while b != 0:
5         a, b = b, a%b
6     return a
```

Cette version itérative n'est pas très claire... On va essayer d'en écrire une version récursive. Pour cela, on utilise le fait que, si $a \in \mathbb{Z}$ et $b \in \mathbb{Z}^*$,

$$a \wedge b = b \wedge r \qquad \text{avec } r \text{ le reste de la division euclidienne de } a \text{ par } b$$

Cette propriété est la base de l'algorithme d'Euclide.

**Exercice 2.** Écrire une fonction `pgcdRecu`, qui est une version récursive de la fonction `pgcdIter` ci-dessus.

## 3 Récursivité et complexité

Le coût d'un algorithme récursif est lié au nombre d'appels récursifs. On peut l'obtenir par une relation de récurrence. Par exemple, pour la fonction `factoRecu`, si on note $c_n$ le coût pour l'appel `factoRecu(n)`, on a :

- $c_0 = 1$ car on réalise juste la comparaison l. 2.
- $c_n = 3 + c_{n-1}$ car les opérations effectuées sont : la comparaison l.2 puis l.5 : une multiplication, une soustraction et enfin l'appel de `factoRecu(n-1)` qui constitue $c_{n-1}$ opérations élémentaires.

On montre alors facilement que $c_n = 3n + 1$, si bien que la complexité est d'ordre $n$, donc linéaire. C'est aussi le cas de `factoIter`.

**Le piège de la récursivité.** Voici un autre grand classique : la suite de Fibonacci.

```
1 def fiboRecu(n):
2     if n==0 or n==1:
3         return 1
4     else:
5         return fiboRecu(n-1) + fiboRecu(n-2)
```

**Exercice 3.** Taper et tester avec $n = 4$, $n = 25$ puis $n = 35$. Que remarquez-vous ?

**Question 1.** On note $c_n$ le nombre d'opérations élémentaires réalisées par `fiboRecu(n)`. On considère que l'opérateur logique `or` constitue également une opération élémentaire. Déterminer la relation de récurrence de $c_n$ pour la fonction `fiboRecu`.

**Question 2.** Montrer que $c_n \geq 2c_{n-2}$. En déduire que $c_n \geq q^n$ avec $q > 1$ un réel à préciser.

De cette relation de récurrence, on peut de même montrer que $c_n \leq 6 + 2c_{n-1}$ puis que $(c_n + 6) \leq 2(c_{n-1} + 6)$. On obtient alors que

$$c_n + 6 \leq 2^n(c_0 + 6) = 9 \times 2^n$$

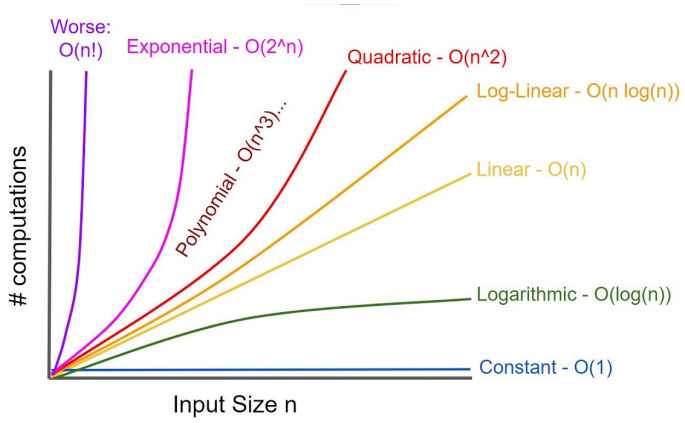
et donc on a $c_n \leq 3^n$ pour $n$ assez grand. La complexité de `fiboRecu` est ainsi exponentielle, cf la définition ci-dessous.

**Définition 9.2**

Si le coût $c_n$ vérifie à partir d'un certain rang

$$a^n \leq c_n \leq b^n$$

avec $1 < a \leq b$, on dit que le coût est *exponentiel*.




Un coût exponentiel est encore pire qu'un coût quadratique. Il faut l'éviter à n'importe quel prix. Une version itérative de `fiboRecu` sera beaucoup plus rapide :

```
def fiboIter(n):
    u,v = 1,1
    for k in range(2,n+1):    # si n ≤ 1, on ne rentre pas dans la boucle
        u,v = u+v,u
    return u
```

Cette version a une complexité linéaire : on peut sans problème calculer `fiboIter(10**5)`.

En général, si à l'étape `n` on appelle plusieurs fois les étapes `n-1` ou `n-2`, cela conduit à une complexité exponentielle.

## 4 Exponentiation rapide

On cherche à calculer $x^n$, avec $n \in \mathbb{N}^*$ et $x$ un nombre flottant non nul.

**Méthode naïve.** On calcule successivement $x^2, x^3$, etc. en se basant sur la définition mathématique $x^n = \underbrace{x \times x \times \cdots \times x}_{n \text{ fois}}$ :

```
def expoNaif(x,n):
    p = 1
    for blabla in range(n):    # on n'utilise pas la variable blabla
        p = p*x
    return p
```

**Question 3.** Quelle est la complexité de l'algorithme ci-dessus ?

**Méthode rapide.** On veut calculer $x^{37}$. Avec `expoNaif`, il faut 36 multiplications. Voici une méthode plus rapide :

$$\begin{aligned}
x^{37} &= x\cdot x^{18}\cdot x^{18} && \text{(2 mult., reste à calculer } x^{18}\text{)}\\
x^{18} &= x^9\cdot x^9 && \text{(1 mult., reste à calculer } x^9\text{)}\\
x^9 &= x\cdot x^4\cdot x^4 && \text{(2 mult., reste à calculer } x^4\text{)}\\
x^4 &= x^2\cdot x^2 && \text{(1 mult., reste à calculer } x^2\text{)}\\
x^2 &= x\cdot x && \text{(1 mult.)}
\end{aligned}$$

Ce qui fait seulement 7 multiplications ! Il s'agit d'une méthode par dichotomie : à chaque étape la puissance de $x$ à calculer est (à peu près) divisée par 2. On va écrire une fonction `expoRapide` : l'instruction `expoRapide(x,n)` va calculer $x^n$.

On rappelle que $n//2$ désigne le quotient de la division euclidienne de $n$ par 2. La méthode rapide repose sur le fait que

$$x^n = \begin{cases} (x^2)^{n//2} & \text{si } n \text{ est pair,}\\ x\cdot(x^2)^{n//2} & \text{si } n \text{ est impair.}\end{cases}$$

Ainsi, pour calculer $x^n$ avec `expoRapide(x,n)`, on se ramène au calcul de $x^{n//2}$ avec ..... À chaque étape, la puissance en exposant est remplacée par son quotient entier par 2. On continue jusqu'à ce que la puissance devienne 0, et dans ce cas le calcul est immédiat. L'algorithme (récursif) correspondant est le suivant :

```
def expoRapide(x,n):
    if n == 0:          # condition d'arrêt
        return 1
    ...
```

**Exercice 4.** Compléter la fonction ci-dessus.

On verra en DM ( !) que la complexité de cet algorithme est, comme la dichotomie, en $\log_2(n)$.

## 5 Exercices d'approfondissement

**Exercice 5** (*Somme récursive d'une liste*)**.** Écrire une fonction `sommeRecu` qui prend en argument une liste de flottants `L` et qui calcule la somme des éléments de cette liste de manière récursive.

**Exercice 6.** Déterminer ce que calcule la fonction suivante. Essayez d'abord sans tester la fonction, i.e. « à la main ».

```
def mystere(a):
    if a<10:
        return a
    return a%10 + mystere(a//10)
```

**Exercice 7** (*Méthode de Héron itérative*)**.** On considère un entier $a \in \mathbb{R}_+$ ainsi que la suite $(u_n)$ définie par :

$$u_0 = 1 \qquad \text{et} \qquad \forall n \in \mathbb{N} \quad u_{n+1} = \frac{1}{2}\left(u_n + \frac{a}{u_n}\right)$$

Écrire une fonction `heronIter` qui prend en argument un flottant $a \geq 0$ ainsi qu'un entier naturel $n$ et qui retourne le $n$-ième terme de la suite, de manière itérative. Vérifier numériquement que $u_n$ tend vers $\sqrt{a}$.

**Exercice 8** (*Méthode de Héron récursive*)**.** Écrire une version récursive `heronRecu` de la fonction `heronIter`. Faire attention à ce que le coût soit linéaire !

*Indication : pour calculer $u_n$, on pourra stocker $u_{n-1}$ dans une variable $x$ puis à la ligne suivante retourner un calcul sur $x$.*