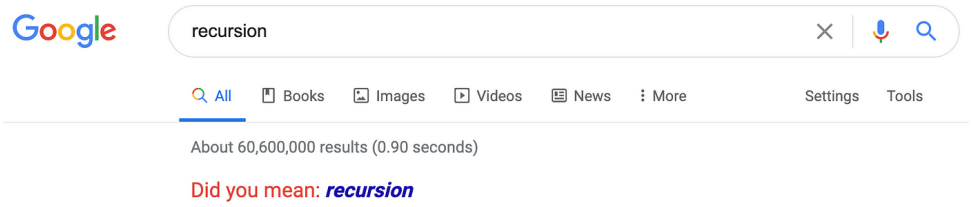
Chapitre 7
Récrusivité

Simon Dauguet
simon.dauguet@gmail.com

30 janvier 2025



1 Définition et premiers exemples

> Définition 1.1 (Fonction récursive) :
> Une fonction récursive est une fonction qui s'auto-appelera ou aura des appels de fonctions qui l'appellent elle-même.

La récursivité est, en quelque sorte, la version informatisé de suites définies par récurrence en mathématiques.

Exemple 1.1 :
Les factorielles en mathématiques sont souvent définie par récurrence :

$$\forall n \in \mathbb{N},\ n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n > 0 \end{cases}$$

On peut le traduire en Python :

```python
def factorielle(n:int) -> int :
    if n == 0 :
        return(1)
    else :
        return(n*factorielle(n-1))
```

Définition 1.2 (Fonction itérative) :
Une fonction itérative est une fonction qui n'est pas récursive.

Lors de l'appel d'une fonction récursive, il y a alors une pile d'exécution. Il est toujours possible de passer d'une fonction récursive à une fonction itérative en simulant itérativement la pile d'exécution de la fonction récursive.

Remarque :
Lors d'appels récursifs, la pile d'exécution grandie et pour une infinité d'appels, la pile débordera de l'espace mémoire. Pour palier ce problème, le langage Python instaure une limite pour le nombre de récursion (autour de 1000 selon vos machines) que l'on peut connaître grâce au `getter` suivant :

```
1 import sys
2 sys.getrecursionlimit()    # renvoie le nombre d'appels récursifs autorisés
```

Proposition 1.1 :
Une fonction récursive bien définie doit contenir :

- au moins un appel récursif (l'hérédité)
- au moins un critère pour terminer les appels (la terminaison)
- un `return` de valeurs (inutile pour une procédure)

Exemple 1.2 (Exemple détaillé avec pile d'appels) :
On souhaite calculer la somme des entiers compris entre `n1` et `n2`. On définit alors une fonction `S(n1:int, n2:int) ->` qui va vérifier $S(n_1, n_2) = 0$ si $n_1 > n_2$ et $S(n_1, n_2) = S(n_1, n_2 - 1) + n_2$ sinon. Ce qui donne :

```
1 def S(n1:int, n2:int) -> int :
2     if n1 > n2 :
3         return(0)
4     else :
5         return(S(n1,n2-1)+n2)
```

C'est donc une fonction récursive.

Si on dépile l'appel de cette fonction pour `S(1,4)`, on obtient :

S(1,4) = 4 + S(1,3)
S(1,3) = 3 + S(1,2)
S(1,2) = 2 + S(1,1)
S(1,1) = 1 + S(1,0)
S(1,0) = 0
S(1,1) = 1
S(1,2) = 3
S(1,3) = 6
S(1,4) = 10

2 Terminaison, Correction, Complexité des algorithmes récursifs

La terminaison, la correction et la complexité se font comme d'habitude. À la différence près que les démonstrations vont devoir utilisé des récurrences mathématiques.

Exemple 2.1 :
On considère la définition récursive des calculs des puissances de nombres :

```
1 def puissRec(x:float, n:int) -> float :
2     if n == 0 :
3         return(x)
4     else :
5         return(x*puissRec(x,n-1))
```

L'invariant de boucle va être la valeur de la variable **n**. On peut démontrer par récurrence facilement que la propriété : "Pour tout $x \in \mathbb{R}$, l'appel **puissRec(x,n)** s'arrête" est vraie. En effet, si $n = 0$, c'est clair. L'algorithme s'arrête et renvoie **x**.

Soit $n \in \mathbb{N}$. Supposons que $\forall x \in \mathbb{R}$, l'appel de **puissRec(x,n)** s'arrête. Soit $x \in \mathbb{R}$. On fait un appel de **puissRec(x,n+1)**. Alors cet appel renvoie **x*puissRec(x,n)**. Mais par hypothèse de récurrence **puissRec(x,n)** s'arrête. Donc il renvoie une valeur explicite. Alors **x*puissRec(x,n)** est aussi une valeur explicite. Et donc **puissRec(x,n+1)** s'arrête également.

On vient donc de montrer par récurrence que la propriété "$\forall x \in \mathbb{R}$, **puissRec(x,n)** s'arrête" est vraie pour tout $n \in \mathbb{N}$. Et donc, on vient bien de montrer la terminaison de cet fonction.

Pour la correction, on procède de la même manière. On montre par récurrence que $\forall n \in \mathbb{N}$, ($\forall x \in \mathbb{R}$, **puissRec(x,n)** renvoie la valeur de x^n).

Si $n = 0$, alors clairement $\forall x \in \mathbb{R}$, **puissRec(x,n)** renvoie $1 = x^0$.

Soit $n \in \mathbb{N}$. On suppose que $\forall x \in \mathbb{R}$, **puissRec(x,n)** renvoie x^n. Soit $x \in \mathbb{R}$. Alors **puissRec(x,n+1)** renvoie **x*puissRec(x,n)**. Or, par hypothèse de récurrence, **puissRec(x,n)** correspond à x^n. Donc **puissRec(x,n+1)** renvoie la valeur de $x \times x^n$, c'est-à-dire x^{n+1}.

Donc, par principe de récurrence, on vient de montrer que $\forall n \in \mathbb{N}$, ($\forall x \in \mathbb{R}$, **puissRec(x,n)** renvoie la valeur de x^n). Et donc la fonction **puissRec** est donc correcte.

La complexité se fait également par récurrence. On va calculer la complexité en fonction de la taille de n. Notons, pour tout $n \in \mathbb{N}$, $C(n)$ le nombre d'opération effectuées par l'appel de **puissRec(x,n)** pour tout $x \in \mathbb{R}$.

On note que **puissRec(x,0)** ne fait qu'un test et renvoie immédiatement la valeur de x. Donc **puissRec(x,0)** ne fait que deux opérations. Donc $C(0) = 2$.

Soit $n \in \mathbb{N}$ et $x \in \mathbb{R}$. Lors de l'appel de **puissRec(x,n+1)**, il y a une comparaison qui est faite ; puis un renvoie ; et un produit de **x** par la même fonction **puissRec(x,n)**. Donc on a $1 + 1 + 1 \times C(n)$ opérations. Autrement dit, $C(n+1) = 2 + C(n)$.

On vient donc de montrer que $\forall n \in \mathbb{N}$, $C(n+1) = 2 + C(n)$. Autrement dit, $(C(n))_{n\in\mathbb{N}}$ est une suite arithmétique de raison 2. Donc $\forall n \in \mathbb{N}$, $C(n) = C(0) + 2n = 2(n+1)$.

Et donc finalement, comme $C(n)$ est le nombre d'opération de **puissRec(x,n)**, la complexité de cette fonction est linéaire $O(n)$.

Exercice 1 :
On considère la méthode de Héron d'Alexandrie d'approximation de la racine carré d'un réel : Soit $a \geq 0$. On pose $u_0 = 1$ et $\forall n \in \mathbb{N}$, $u_{n+1} = \frac{1}{2}\left(u_n + \frac{a}{u_n}\right)$.

1. Écrire la fonction itérative `heronI(a:float, n:int) -> float` qui renvoie le n-ème terme de la suite de Héron approchant $\sqrt{a}$.
2. Écrire la fonction récursive `heronR(a:float, n:int) -> float` qui fait la même chose.
3. Faire la preuve de la terminaison de la fonction `heronR`.
4. Faire la preuve de la correction de `heronR`.
5. Étudier la complexité des deux fonctions `heronI` et `heronR`.

3 Méthode “diviser pour régner”

Les méthodes “diviser pour régner” sont des sous-catégories des méthodes récursives.

Définition 3.1 (Méthode “diviser pour régner”) :
La méthode “diviser pour régner” suit le schéma suivant :

- diviser : diviser le problème initial en un ou plusieurs sous-problèmes de plus petites tailles.
- régner : résoudre les problèmes de taille minimale (c'est la terminaison).
- assemblage : le solutions des sous-problèmes sont rassemblées pour construire la solution au problème initial.

Exemple 3.1 :
On souhaite dessiner l'ensemble tri-adique de Cantor :

Pour tracer l'ensemble de Cantor, il faut prendre un segment donné, puis le découper en trois morceaux et dessiner les deux morceaux extérieur (c'est la division).

On sait facilement tracer un segment entre deux points donné (c'est le règne).
Il ne reste plus qu'à assembler tout ça.

```
def line(a,b) :
    # règne : tracer le segment entre le point de coordonnées (a,0) et (b,0).
    plt.plot([a,b],[0,0],"k",lw=2)

def cantor(a,b,n) :
    # division : on fait des appels récursifs pour diviser le segment entre (a,0) et (b,0) en trois morceaux. Et on assemble.
    if n==1 :
        line(a,b)
    else :
        cantor(a,2/3*a+b/3,n-1)
        cantor(a/3+2*b/3,b,n-1)

def Cantor(n) :
    # On dessine le calque créé.
    cantor(0,1,n)
    plt.show()
```

On obtient alors les jolis dessins suivant :