Récursivité

1 — Récursivité simple

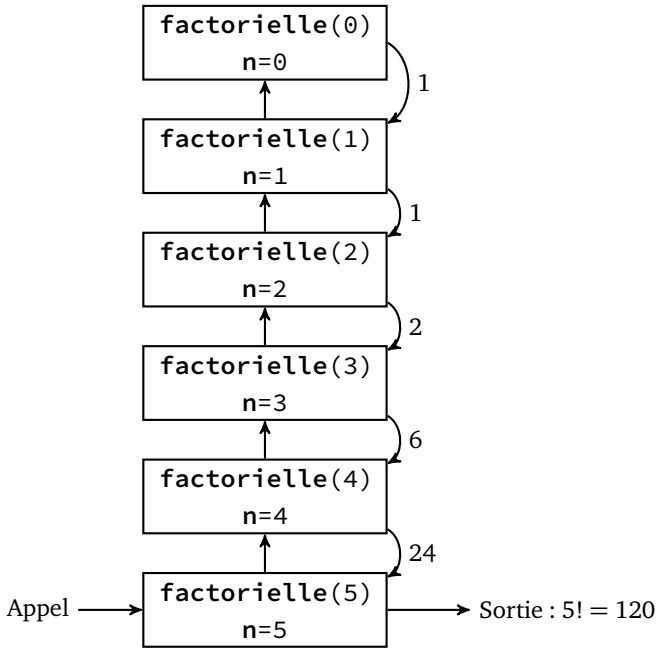
Une fonction est dite *récursive* lorsqu'elle s'appelle elle-même.

Exemple : Calcul de $n!$ Les relations $0! = 1$ et $\forall n \in \mathbb{N}^*, n! = n \times (n-1)!$ suggèrent de définir la fonction factorielle en fonction d'elle-même, c'est-à-dire récursivement, ce qui peut s'écrire de la façon suivante

```
def factorielle(n):
    if n==0:
        return(1)
    else:
        m = factorielle(n-1)
        return( n * m )
```

Suivons un exemple d'éxécution. Si on demande `factorielle(5)` alors Python va successivement :

- Charger et exécuter une copie de `factorielle`, avec `n=5`. Dans le corps de la fonction, il va devoir évaluer `factorielle(4)` ;
- Python va alors charger une nouvelle copie de la fonction. Mais cette copie utilise un nom de variable déjà utilisé. Pour éviter les ennuis, Python va mettre de côté la signification antérieur de `n` et de toutes les autres variables de `factorielle`. Dans la copie effectivement utilisée de `factorielle`, la variable `n` vaut 4.
- Cette copie doit maintenant évaluer `factorielle(3)`. Python va mettre de côté la copie précédente de `factorielle` et en créer une nouvelle dans laquelle `n` vaut 3. Il y a donc deux trois copies de `factorielle` en mémoire : deux sont mises de côté et une est effectivement utilisée.
  On recommencer ainsi avec `n` valant 2, puis 1 et enfin avec `n` valant 0.
- On a ainsi une pile de copies de la même fonction. Cette pile pourrait potentiellement grandir à l'infini, mais on a prévu une *condition d'arrêt*. En effet, lorsque `n` vaut 0, la fonction ne s'appelle pas elle-même : elle renvoie la valeur 1.
- À partir de là, Python va *dépiler* toutes les copies : la dernière (pour `n=0`) est effacé et renvoie 1 à l'avant-dernière. Celle-ci est ensuite effacée et renvoie 1×1 à la précédente, etc. Jusqu'à la dernière qui renvoie donc $5 \times 4 \times 3 \times 2 \times 1 \times 1$, c'est-à-dire $5!$.



Retenez que chaque appel récursif définit ses propres variables locales et met de côté les variables des autres appels.

La programmation récursive n'est jamais indispensable. Les algorithmes récursifs peuvent toujours être mis sous une forme itérative en utilisant intelligemment des boucles `for` ou `while`. Toutefois la programmation récursive s'impose d'elle-même dans une bon nombre de situations. De plus, elle permet parfois de minimiser de manière importante le nombre d'opérations.

Ex. 1 — Calcul de puissances

1. Écrire une fonction récursive **`puissance(x,n)`** qui calcule la valeur de x^n, avec $n \in \mathbb{N}$, en utilisant la formule suivante : $x^0 = 1$ et $\forall n \in \mathbb{N}^*,\ x^n = x \times x^{n-1}$.

Cet algorithme naïf n'est toutefois pas très performant. Par exemple, pour calculer 2^{20}, il nécessite 19 multiplications.
La remarque qui suit donne l'idée d'une meilleure méthode : $2^{20} = (2^{10})^2 = \big((2^5)^2\big)^2 = \big((2\times 2^4)^2\big)^2 = \Big(\big(2\times(2^2)^2\big)^2\Big)^2$. L'évaluation de la dernière expression écrite ne requiert plus que 5 multiplications. L'idée est d'utiliser l'élévation au carré dès que possible.

2. Écrire une autre fonction récursive **`puissance2(x,n)`** qui calcule la valeur de x^n, avec $n \in \mathbb{N}$, en utilisant l'algorithme de Lucas :

$$x^0 = 1 \qquad \forall n \in \mathbb{N}^*,\quad \begin{cases} x^n = (x^{n/2})^2 & \text{si } n \text{ est pair} \\ x^n = x \times (x^{(n-1)/2})^2 & \text{si } n \text{ est impair} \end{cases}$$

Ex. 2 — Calcul des coefficients binomiaux Écrire une fonction récursive **`binomial(n,p)`** qui calcule la valeur de $\binom{n}{p}$, en utilisant la petite formule :

$$\forall (n,p) \in \mathbb{N}^2,\quad 0 \leqslant p \leqslant n, \qquad p\binom{n}{p} = n\binom{n-1}{p-1}$$

On fera bien attention à la condition d'arrêt.

Ex. 3 — Algorithme d'Euclide L'algorithme d'Euclide permet de calculer le PGCD de deux nombres entiers. Il est basé sur la formule

$$\forall (a,b) \in \mathbb{N}^2, \qquad a \wedge b = b \wedge (a \mod b)$$

où $(a \mod b)$ désigne le reste de la division euclidienne de a par b.
Écrire une fonction récursive **`pgcdEuclide(a,b)`** qui renvoie le PGCD de deux entiers **`a`** et **`b`**. On réfléchira au préalable sur les conditions d'arrêt.
Vérification : $14\,546 \wedge 3\,571 = 1$ et $8\,742 \wedge 87 = 3$.

Ex. 4 — Recherche dichotomique dans une liste triée Soit **`L`** une liste de réels triée par ordre croissant et soit **`x`** une valeur donnée. La recherche par dichotomie de **`x`** dans la liste **`L`** est assez simple. On procède comme suit :
– On compare **`x`** à l'élément du milieu de la liste **`L[med]`** ;
– si **`x`** est égal à **`L[med]`**, alors on a terminé notre recherche ;
– si **`x < L[med]`**, alors on cherche l'élément **`x`** dans la première moitié de la liste
– si **`x > L[med]`**, alors on cherche l'élément **`x`** dans la seconde moitié de la liste ;
– puis on réitère ce procédé.
Écrire une fonction **`recherche_dicho(L,x)`** qui recherche l'élément **`x`** dans la liste **`L`** triée par ordre croissant selon cette méthode. La fonction renvoie **`True`** si **`x`** est dans la liste et **`False`** sinon .

★ **Ex. 5** — Anagrammes Écrire une fonction récursive qui renvoie la liste des permutations d'une chaîne de caractères (par exemple, pour `"abcde"` : `"eabcd"`, `"edabc"`, etc.).

★ **Ex. 6** — Expressions correctement parenthésées
On considère dans cet exercice les chaînes de caractère de longueur $2n$ (avec $n \in \mathbb{N}$), constituées exclusivement de parenthèses, comme par exemple `"(()())()"`.
On dit qu'une telle chaîne est correctement parenthésée si elle contient autant de parenthèses ouvrantes `"("` que fermantes `")"`, et, si de plus, lorsqu'on parcourt le mot de gauche à droite on trouve toujours plus de parenthèses ouvrantes que fermantes. Par exemple `"(()((("` et `"(()))("` ne sont pas bien parenthésées, alors que `"(()())()"` l'est.

1. Le nombre de telles chaînes est noté C_n et s'appelle le n–ième nombre de Catalan. La suite $(C_n)_{n\in\mathbb{N}}$ vérifie la relation de récurrence

$$C_0 = 1 \qquad \text{et} \qquad \forall n \in \mathbb{N}, \qquad C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i}$$

Écrire une fonction **`listeCatalan`** prenant en paramètre un entier **`n`** et renvoyant la liste des n premiers nombres de Catalan.
Vérification pour **`n`**`=7` : `[1, 1, 2, 5, 14, 42, 132, 429]`

★ 2. Écrire une fonction qui renvoie la liste de toutes les chaînes de cette sorte.
On pourra remarquer qu'elles s'écrivent `"("+ L(i) + ")"+ L(n-i-1)` où `L(i)` est une chaîne correctement parenthésée de longueur $2i$ avec i variant de 0 à $n-1$. Remarquez aussi que `L(0) = ""`.
Vérification : pour $n = 3$ on trouve `['()()()', '()(())', '(())()', '(()())', '((()))']`

2 — Récursivité double

Une fonction est dite *doublement récursive* lorsqu'elle fait deux appels récursifs.

Exemple : Suite de Fibonacci La suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ est définie par

$$F_0 = 0, \quad F_1 = 1 \qquad \forall n \in \mathbb{N}, \qquad F_{n+2} = F_{n+1} + F_n.$$

On peut calculer le terme de rang n de la façon suivante :

```
def fibonacci(n):
    if n==0 or n==1:
        return n
    else:
        return fibonacci(n-1)+fibonacci(n-2)
```

Ex. 7 — Suite de Fibonacci La suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ vérifie également les relations suivantes

$$\forall n \in \mathbb{N}, \qquad F_n = 2F_{\frac{n}{2}-1}F_{\frac{n}{2}} + F_{\frac{n}{2}}^2 \qquad \text{si } n \text{ est pair}$$
$$F_n = F_{\frac{n-1}{2}+1}^2 + F_{\frac{n-1}{2}}^2 \qquad \text{si } n \text{ est impair}$$

Proposer une fonction récursive **fibonacci2** permettant de calculer les termes de la suite de Fibonacci à partir de cette formule.
On pensera à utiliser des variables intermédiaires pour minimiser les appels à la fonction.
Vérfication : $F_{25} = 75\,025$

Ex. 8 — Calcul des coefficients binomiaux Écrire une fonction doublement récursive qui calcule la valeur de $\binom{n}{p}$, en utilisant la formule de Pascal.
Vérification : $\binom{13}{5} = 1\,287$