La récursivité

Langages et programmation

NSI - Tle
COURS

Contenu : Récursivité.
Capacités attendues : Écrire un programme récursif.
Analyser le fonctionnement d'un programme récursif.

I. Exemple introductif

Certains problèmes peuvent sembler difficile à résoudre d'emblée. Vous allez découvrir une manière d'écrire des algorithmes qui permet de résoudre élégamment certains problèmes. Voilà un outil puissant !

I.1. Exemple 1 :

Soit une fonction ***puissance*** à créer, qui prend deux arguments x, un réel, et n, un entier naturel, et qui renvoie x^n.
Une méthode directe est de partir de 1 puis d'écrire une boucle répétitive qui permet d'aboutir par multiplications successives à x^n en utilisant le fait que $x^n = x \cdot x^{n-1}$
Voici une possibilité :

```
def puissance(x: float, n: int) -> float:
    p = 1
    for i in range(n):
        p = x*p
    return p
```

Tester le code précédent dans *PythonTutor* pour vérifier que la fonction ***puissance*** renvoie bien x^n.

Mais si on faisait l'inverse ? On part de ce qui est compliqué x^n et on voit comment simplifier progressivement pour arriver à 1 !

Cela est possible, toujours en utilisant le fait que : $x^n \begin{cases} 1 & , \text{si } n = 0 \\ x \cdot x^{n-1}, & \text{sinon} \end{cases}$

Ainsi, on aurait l'idée d'écrire un programme comme celui ci-dessous :

```
def puissance (x : float, n : int) -> float:
    if n == 0 :
        return 1
    else :
        return x*puissance(x,n-1)
```

1. Réaliser une trace d'exécution de l'algorithme précédent avec x=2 et n=3. ***puissance(2,3)*** renvoie bien 2323 ?
2. Recopier le script précédent puis tester-le plusieurs fois, par exemple avec x=2 et n=10. Est-ce que ***puissance(2,10)*** renvoie bien 210210.

Aussi étonnant que cela puisse paraître de prime abord, cela fonctionne !
Nous avons ainsi créé une fonction ***puissance*** qui s'appelle elle-même : on appelle cela la **récursivité**.

II. Notion de récursivité

On qualifie de **recursive** toute fonction qui s'appelle elle-même.

II.1. Exemple 2

Outre l'exemple précédent, voici un deuxième exemple.
Dans une grande boite de Pétri contenant un milieu nutritif riche sont déposées 10 bactéries.
On suppose que chaque heure le nombre de bactéries est multiplié par 4. Voici une fonction récursive ***nb_bact*** qui renvoie le nombre de bactéries au bout de n jours, n étant un entier naturel saisi comme argument.

```
def nb_bact(n : int) -> int:
    if n == 0 :
        return 10
    else :
        return 4*nb_bact(n-1)
```

Cette fonction permet d'évaluer le nombre de bactéries au bout d'une journée (en supposant le milieu nutritif suffisant) :

```
>>> nb_bact(24)
2814749767106560
```

On peut y retrouver une suite : le nombre de bactéries au bout de n heures est donné par u_n.
Cette suite (u_n) est définie par la **relation de récurrence** suivante :

$$U_n \begin{cases} 10 & , \text{si } n = 0 \\ 4 \cdot U_{n-1}, & \text{si } n > 0 \end{cases}$$

Le nombre de bactéries obtenu au bout de 24 heures s'écrit dans le langage des suites comme égale à u_{24}.

Les premiers langages de programmation qui ont autorisé l'emploi de la récursivité sont LISP, développé à partir de 1958, et Algol 60 (à partir de 1960). Depuis, tous les langages de programmation généraux réalisent une implémentation de la récursivité.

II.2. Exemple 3

Repérer parmi les fonctions suivantes celles qui sont récursives :

fonction *f1* :

```
def f1(n : int) -> int:
    if n == 0 :
        return 0
    else :
        return n-1
```

fonction *f2* :

```
def f2(n : int) -> int:
    if n == 0 :
        return 0
    else :
        return f2(n-1)
```

fonction *f3* :

```
def f3(n : int) -> int:
    if n == 0 :
        return f3(n-1)
    else :
        return 0
```

fonction *f4* :

```
def f4(n : int) -> int:
    if n == 0 :
        return n-1
    else :
        return 0
```

Voici une fonction mystère nommée *myst* :

```
def myst(ma_liste : list) -> int:
    if ma_liste == [] :
        return 0
    else:
        ma_liste.pop(0)          # suppression du premier terme de la liste l
        return 1+myst(ma_liste)
```

1. Pourquoi cette fonction ***myst*** est une fonction récursive ?
2. Tester cette fonction avec quelques listes.
3. Quel est le rôle de cette fonction ***myst*** ?

III. Condition d'arrêt

III.1. Exemple 4

Exécuter le code suivant :

```
def mauvais_exemple() -> None :
    print("Mauvais exemple…")
    return mauvais_exemple()
```

Évidemment, comme prévu, ce programme ne s'arrête pas, car la fonction ***mauvais_exemple()*** est appelée à l'infini. L'exécution s'arrête d'elle-même (au bout de 1000 récurences). Nous sommes (volontairement) tombés dans un piège qui sera systématiquement présent lors d'une programmation récursive : le piège de la boucle infinie.

Mais attention : la récursivité ne DOIT PAS être associée à une auto-référence vertigineuse : c'est en algorithmique une méthode (parfois) très efficace, à condition de respecter une règle cruciale : l'existence d'un **CAS DE BASE**.
Ce "*cas de base*" sera aussi appelé "*condition d'arrêt*" ou encore "*condition de terminaison*", puisque la très grande majorité des algorithmes récursifs peuvent être perçus comme des escaliers qu'on dévale à toute vitesse, en déséquilibre jusqu'au sol qui assure notre arrêt.

Lorsque nous allons programmer une fonction récursive, nous allons donc commencer par la fin, c'est-à-dire par le moment où elle renvoie effectivement un résultat. C'est le cas de base. Pour arriver progressivement vers la situation finale, chaque appel récursif se fera en décrémentant un paramètre : cela assurera l'arrêt du programme.

III.2. Exemple 5

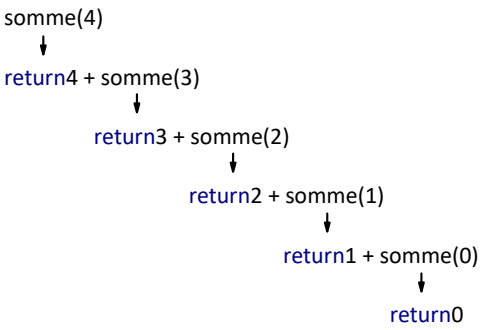
```
def mystere(n):
    if n == 0 :
        return 0
    else :
        return n + mystere(n-1)
```

Observer :

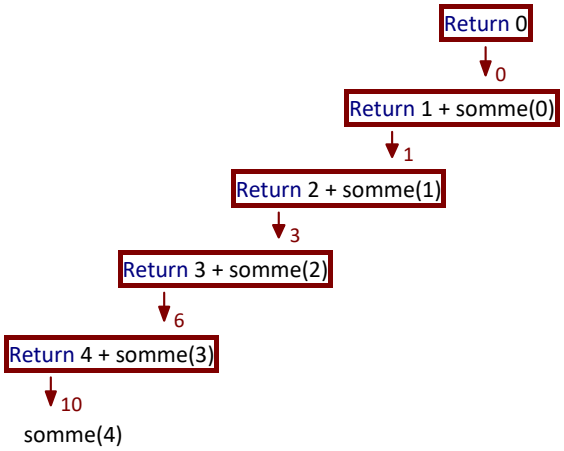
- le cas de base (si n vaut 0 on renvoie *vraiment* une valeur, en l'occurence 0)
- l'appel récursif
- la décrémentation du paramètre d'appel : **mystere(n-1)**

Cette fonction mystere(n) calcule la somme des entiers positifs inférieurs ou égaux à n.

S'il est assez aisé de dérouler l'exécution de la définition avec une boucle, comment représenter l'exécution de la fonction? On peut remplacer l'appel de la fonction par le return qui sera obtenu, et continuer ainsi jusqu'à arriver au cas 0.
Voici l'arbre d'appel de somme(4) :



Une fois arrivé à un cas de base, on remonte les valeurs obtenues au fur et à mesure.



La variable res de la version itérative correspond donc aux résultats successifs renvoyés par la version récursive. D'une certaine manière, la boucle calcule les résultats intermédiaires, en partant de 0. Cela revient à faire directement la remontée des valeurs, comme c'est le cas dans l'exemple ci-contre.

somme(0) = 0
somme(1) = 1 + somme(0) = 1
somme(2) = 2+somme(1) = 3
somme(3) = 3+somme(2) = 6
somme(4) = 4+somme(3) = 10

Attention ! L'existence d'une condition d'arrêt ne signifie pas que l'appel récursif s'arrête grâce à celle-ci.
Prenons l'exemple de l'exécution de **mystere(-1)** :

- **mystere(-1)** conduit par appel récursif à l'exécution de **mystere(-2)**
- **mystere(-2)** conduit par appel récursif à l'exécution de **mystere(-3)**
- **mystere(-3)** conduit par appel récursif à l'exécution de **mystere(-4)**
- ...

La condition d'arrêt n=0 n'est jamais atteinte et on obtient une suite infinie d'appels.
Ainsi, il est important d'ajouter une **précondition** pour imposer que n soit un entier naturel. D'où :

```
def mystere(n):
    assert type(n) == int and n >=0, "Vous devez saisir un entier naturel"
    if n == 0 :
        return 0
    else :
        return n + mystere(n-1)
```

À retenir !

Dans une fonction récursive, il faut s'assurer que la condition d'arrêt soit atteinte après un **nombre fini d'appels**.
Cette **condition d'arrêt** ne peut en aucun cas être un appel récursif.
Souvent, il est pertinent de choisir une condition d'arrêt correspondant à un "cas simple" pour lequel on connait la valeur à renvoyer.

III.3. Exemple 6

Revoici la fonction puissance vue à la fin du paragraphe I. :

```
def puissance(x : float, n : int) -> float:
    if n == 0 :
        return 1
    else :
        return x*puissance(x,n-1)
```

- La condition d'arrêt est : n=0.
- Cette condition correspond aussi au cas "simple" où $x_0 = 1$: on connaît le résultat à faire renvoyer par la fonction : 1.

III.4. Exemple 7

On veut réaliser un château de cartes géants qui prolonge le château de l'image ci-contre :

On note *n* le nombre d'étages du château et *nb_cartes(n)* le nombre de cartes nécessaires pour réaliser un château à *n* étages.
On admet que l'on peut connaître le nombre *nb_cartes(n)* de cartes nécessaires pour un château à *n* étages si on connaît déjà le nombre *nb_cartes(n-1)* en utilisant la relation suivante (appelée relation de récurrence en mathématiques) :

$$nb\_cartes(n) = nb\_cartes(n-1) + 2 + 3\cdot(n-1)$$

On veut à partir de ces informations construire une fonction récursive nommée ***nb_cartes*** qui renvoie finalement le nombre *nb_cartes(n)* si on donne en argument le nombre *n* d'étages voulus au château.

Voici un script incomplet pour cette fonction :

```
def nb_cartes(n: int) -> int:
    if ...........: # condition d'arrêt
        return ............
    else:           # cas général
        return ............
```

1. En s'aidant de la relation de récurrence liant *nb_cartes(n)* et *nb_cartes(n-1)*, compléter le retour du cas général.
2. Déterminer quel cas simple correspond à la condition d'arrêt puis compléter son renvoi.
3. Testez votre fonction obtenue.

Vous pouvez utiliser l'image ci-dessus pour connaître quelques valeurs à obtenir.

4. Rajouter une précondition pour assurer le bon fonctionnement de l'algorithme.

Dans l'exercice précédent, on est certain que la récursion prend fin car chaque nouvel appel se fait avec un paramètre n qui diminue.
Pour s'assurer de la **terminaison** d'un algorithme récursif, il suffit d'identifier une suite strictement décroissante d'entiers positifs ou nul.

III.5. Exemple 8

La **factorielle** d'un entier naturel n non nul, notée n!, est le produit de tous les nombres entiers compris entre 1 et n, c'est-à-dire :

$$\mathbf{n! = 1 \times 2 \times \ldots \times (n-1) \times n}$$

- 1! = 1
- 4! = 1 × 2 × 3 × 4 = 24
- 6! = 1 × 2 × 3 × 4 × 5 × 6 = 720

Il existe une relation "simple" entre n! et (n−1)!
En effet : $n! = 1 \times 2 \times \ldots \times (n-1) \times n = \underbrace{1 \times 2 \times \ldots \times (n-1)}_{(n-1)!} \times n = (n-1)! \times n$

1. Écrire une fonction récursive ***fact*** qui prend comme argument l'entier non nul n et qui renvoie n!
2. Quelle est la condition d'arrêt de la fonction récursive ?
3. Comment être certain que la situation de terminaison sera atteinte après un nombre fini d'appels récursifs ?
4. Rajouter une précondition.

IV. Pile d'exécution

Illustration sur l'exemple du paragraphe I

Nous avons vu plusieurs exemples de fonctions récursives. Le but est de comprendre au niveau de la gestion de la mémoire, comment peut bien fonctionner la récursivité.

Reprenons l'exemple initial sur la fonction récursive ***puissancec*** :

```
def puissance (x : float, n : int) -> float:
    if n == 0 :
        return 1
    else :
        return x*puissance(x,n-1)
```

Observer le déroulement de l'exécution de ce code étape par étape dans *PythonTutor*.

Il est possible de décrire ainsi la succession des étapes :

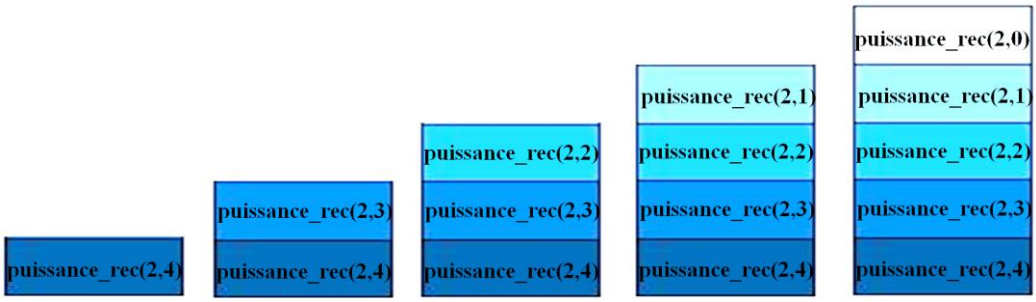
```
Appel à puissance_rec(2,4)
 2*puissance_rec(2,3) = ?
 Appel à puissance_rec(2,3)
  2*puissance_rec(2,2) = ?
  Appel à puissance_rec(2,2)
   2*puissance_rec(2,1) = ?
   Appel à puissance_rec(2,1)
    2*puissance_rec(2,0) = ?
    Appel à puissance_rec(2,0)
    Retour de la valeur 1
    2*1
   Retour de la valeur 2
   2*2
  Retour de la valeur 4
  2*4
 Retour de la valeur 8
 2*8
Retour de la valeur 16
```

On voit qu'il est nécessaire de mémoriser les paramètres et les résultats à retourner : on parle de **pile d'exécution**.
Une **pile d'exécution** permet de mémoriser des informations sur les fonctions en cours d'exécution dans un programme.
Le principe est le suivant :

- l'instruction située en haut de la pile d'exécution est en cours d'exécution,
- les instructions en dessous sont mises en pause dans l'attente de se retrouver au sommet de la pile d'exécution.

Voici une visualisation de ce qui se passe au niveau de la **pile d'exécution** lorsque l'on exécute le programme précédent :

Concrètement, ce qui est stocké à chaque étage de la pile, c'est l'adresse mémoire de l'instruction à exécuter. Pour simplifier, ici c'est l'instruction qui est écrite.



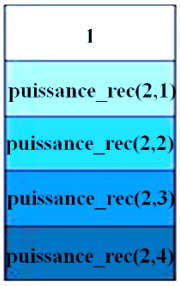
La première instruction ***puissance(2,4)*** est lancée.

Comme celle-ci fait appel à ***puissance(2,3)***, ***puissance(2,3)*** est la nouvelle instruction exécutée tandis que ***puissance(2,4)*** est mise en pause.

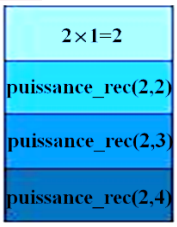
De même, celle-ci faisant appel à ***puissance(2,2)***, ***puissance(2,2)*** est la nouvelle instruction exécutée tandis que ***puissance(2,3)*** est mise en pause.

Ces appels successifs se reproduisent conduisant à un nouvel étage de la pile d'exécution jusqu'à ce que l'instruction à exécuter devienne ***puissance(2,0)*** : il y a eu un **empilement** de 5 espaces-mémoire.

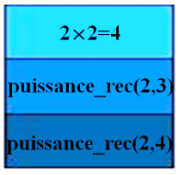
L'instruction ***puissance(2,0)*** renvoie 1 (et clôt l'exécution de ***puissance(2,0)***).



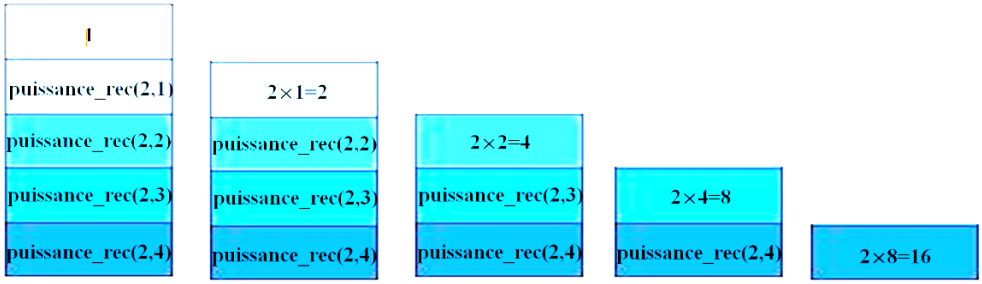
Comme l'instruction ***puissance(2,0)*** a été exécutée, elle est **dépilée** et l'instruction ***puissance(2,1)***, qui avait mise en attente, est désormais exécuté avec la valeur renvoyée par ***puissance(2,0)***.



L'instruction ***puissance(2,1)*** est désormais exécutée : elle renvoie la valeur 2 puis cette instruction est dépilée ; l'instruction suivante ***puissance(2,2)*** est exécutée.



On poursuit l'exécution des instructions successives en dépilant progressivement la pile d'exécution. On obtient ainsi la visualisation suivant du dépilement complet :



La pile d'exécution se ferme avec le renvoi de la valeur finale : 16.

V. Limitations

Reprenons l'exemple initial sur la fonction récursive ***puissance*** vue au *paragraphe I*.
En exécutant le code suivant, vous verrez apparaître un résultat :

```
>>> puissance(2,965)
31185004836479997057130823641200602594803925944304024085977300663081435810452563527889968210822432829520975731940507738187069343568649900949049559348200490942500088639860713695586526897568171674728958699133498812395793913361263599826388363569500689961048764169933688150661851487974125155123 2
```

Quelle puissance !
Par contre, en exécutant le code suivant, vous verrez apparaître, entre autres, un message d'erreur :

```
>>> puissance_rec(2,966)
RecursionError: maximum recursion depth exceeded in comparison
```

Le langage Python génère et gère automatiquement les espaces-mémoires de la partie dédiée de la mémoire physique de l'ordinateur : la pile d'exécution (ou pile de récursivité).
Comme tout système physique, sa capacité est limitée. Par défaut, l'implémentation de Pyhton limite la hauteur de la pile de récursivité à 1000.
Si l'exécution d'un appel récursif conduit à vouloir dépasser cette hauteur maximale, alors le message d'erreur ***RuntimeError: maximum recursion depth exceeded in comparison*** apparaît.

VI. Que préférer entre un code impératif et un code récursif ?

Tout algorithme récursif peut être transformé en un algorithme impératif.
La fonction récursive suivante qui permet de savoir si un caractère c est présent dans la chaîne de caractères ch :

```
def est_present(c : str, ch : str) -> bool:
    if ch == "" :
        return False                # cas d'arrêt négatif
    elif c == ch[0] :
        return True                 # cas d'arrêt positif
    else :
        return est_present(c,ch[1:]) #un appel récursif qui est la dernière chose à effectuer
```

peut facilement être réécrite en impératif :

```
def est_present_iter(c : str, ch : str) -> bool:
    bool = False
    i = 0
    while bool == False and i < len(ch) :
        if c == ch[i] :
            bool = True
        i = i + 1
    return bool
```

Comme tout algorithme récursif peut être transformé en un algorithme impératif, qu'est-il préférable d'écrire ?

Quelques avantages de la récursivité :

- La récursivité ajoute de la simplicité (de la **compacité**) lors de l'écriture de code, ce qui facilite le débogage,
- La récursivité est également préférée lors de la **résolution de problèmes très complexes** : une solution récursive décrit comment calculer la solution à partir d'un cas plus simple, au lieu de préciser chaque action à réaliser, on décrit ce qu'on veut obtenir, c'est ensuite au système de réaliser les actions nécessaires pour obtenir le résultat demandé.

La récursivité a ses propres limites :

- Les fonctions récursives requièrent généralement **plus d'espace de mémoire**.
- Les récursions peuvent excéder la taille de la pile de récursivité : il y a alors un **débordement de pile**
- Une fonction récursive peut être de plus grande compacité dans son écriture mais n'est pas forcément de plus petite **complexité** (en temps d'exécution ou en espace mémoire nécessaire).

Conclusion :

- si on recherche l'efficacité (une exécution rapide) et si le programme peut être écrit sans trop de difficultés en itératif, on préférera l'itératif.
- on préfère la récursivité surtout dans les situations où la solution itérative est difficile à obtenir, par exemple :
    - si les structures de données manipulées sont récursives (ex les arbres).
    - si le raisonnement lui-même est récursif. (ex le jeu des tours de Hanoï, voir TD)

VII. Récursivité multiple et croisée

VII.1. Récursivité multiple

Un algorithme récursif est dit multiple si l'un des cas qu'il distingue se résout avec plusieurs appels récursifs.

Considérons la suite dite de Fibonacci.
Fibonacci a publié en 1202 un recueil de problèmes pratiques, le *Liber abaci*. Le but était d'exposer les applications pour le commerce de l'utilisation des chiffres arabes et des algorithmes arithmétiques permettant de calculer avec ces chiffres arabes. Ce livre a conduit à l'utilisation des chiffres arabes en Occident plutôt que des chiffres romains.

Le problème le plus célèbre du *Liber abaci* est le suivant : Combien de couples de lapins aurons-nous à la fin de l'année si nous commençons avec un couple qui engendre chaque mois un autre couple qui procréé à son tour au bout de deux mois ? Cet énoncé sous-entend les conditions suivantes :

1. La maturité sexuelle du lapin est atteinte après un mois qui est aussi la durée de gestation.
2. Chaque portée comporte toujours un mâle et une femelle.
3. Les lapins ne meurent pas !!

Si on note **lapin(n)** le nombre de lapins au bout de n mois, on peut modéliser le problème "pratique" par une suite (lapin(n)). Cette suite (lapin(n)) est définie par la **relation de récurrence** suivante :

$$lapin(n) = \begin{cases} 1 & ,si\ n = 0 \\ 1 & ,si\ n = 1 \\ lapin(n-1) + lapin(n-2) & ,si\ n > 2 \end{cases}$$

Écrire une fonction récursive ***Fibo*** qui prend comme argument un entier naturel n et qui renvoie le nombre **lapin(n)**.

1. Tester votre fonction récursive en exécutant ***Fibo(10)***. Remarquez-vous quelque chose de particulier ?
2. Tester votre fonction récursive en exécutant ***Fibo(35)***. Remarquez-vous quelque chose de particulier ?
3. En terme de place mémoire, comparer l'augmentation nécessaire pour passer du calcul de ***Fibo(4)*** à celui de ***Fibo(5)***
4. En généralisant au passage de ***Fibo(n)*** à ***Fibo(n+1)***, déduire comment se comporte la compléxité en mémoire de cet algorithme.

L'exécution de cet algorithme récursif conduit ici à répéter des calculs, ce qui le rend vite inutilisable tel quel. Comme c'est le cas avec de nombreux algorithmes récursifs, il existe une stratégie qui cherche à modifier un tel algorithme pour éliminer les calculs redondants : on l'appelle la **programmation dynamique**. Elle consiste à :

1. Commencer par écrire un algorithme récursif pouvant faire des calculs redondants,
2. Stocker, dans un tableau, les résultats intermédiaires. On modifie alors l'algorithme récursif pour qu'il lise le résultat dans ce tableau s'il a déjà été calculé mais pour qu'il calcule le résultat puis le stocke dans le tableau sinon.
   On gagne ainsi en complexité en temps, tout en perdant en complexité en mémoire.
3. Ensuite, en analysant l'ordre des résultats appelés, il est même souvent possible d'écrire un algorithme itératif indépendant.

VII.2. Récursivité croisée

Pour l'instant, nous avons vu qu'une fonction peut s'appeler elle-même. Il est possible qu'une fonction appelle une deuxième fonction qui la rapelle en retour.

Deux algorithmes sont dits **mutuellement récursifs** si l'un fait appel à l'autre et l'autre à l'un. On parle aussi de **récursivité croisée**.

Cette définition peut être étendu à un plus grand nombre d'algorithmes.

Voici l'exemple classique : on construit deux fonctions ***pair*** et ***impair*** qui renvoie un booléen déterminant la parité du nombre n entré en argument, fonctions qui s'appellent l'une, l'autre au cours de leur exécution.
Voici le code en python de cette imbrication :

```
def pair(n):
    if n == 0:
        return True
    else:
        return impair(n-1)

def impair(n):
    if n == 0:
        return False
    else:
        return pair(n-1)
```

1. Exécuter à la main ***pair(2)*** en cherchant à comprendre en quoi il y a une récursivité croisée.
2. Visualiser la récursivité croisée en lançant étape par étape avec PythonTutorle programme ci-dessus en lançant la fonction ***pair(5)***.