```
def factiter1(n):
    f = 1
    for i in range(1, n+1):
        f = f * i
    return f
```

```
def factiter2(n):
    i = 1
    f = 1
    while i < n:
        i += 1
        f = f * i
    return f
```

```
def factrec(n):
    if n < 2:
        return 1
    else:
        return n*factrec(n-1)
```

```
def factrect(n, f=1):
    if n < 2:
        return f
    else:
        return factrect(n-1, n*f)
```

Les deux premières fonctions calculent $n!$ de manière itérative, les deux suivantes de manière récursive. La dernière est dite récursive terminale.

1. Définition

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

2. Schéma récursif

Les appels récursifs peuvent apparaitre dans différentes configurations. Voici deux schémas types en Python :

Cas d'une vraie fonction

```
def fonction(variables):
    if condition_arret:
        return resultat
    else:
        # lignes de code puis
        return fonction(variables)
```

Cas d'une procédure

```
def fonction(variables):
    if condition_arret:
        # dernières instructions
    else:
        # lignes de code
        # rappel(s) de la fonction
```

Les fonctions récursives vont s'exécuter « en boucle », se rappelant à chaque fois, jusqu'à ce que la **condition d'arrêt** soit vérifiée. On peut construire d'autres schémas récursifs. L'important est de **toujours s'assurer que les appels se terminent**.

3. Pile d'exécution et récursivité terminale

L'appel d'une fonction provoque l'écriture dans une pile appelée pile d'exécution :

- de l'adresse mémoire de l'instruction qui a appelé la fonction (afin d'y revenir une fois l'exécution de la fonction terminée)
- des valeurs des paramètres et des variables définies par la fonction.

Lors d'appels récursifs, la pile d'exécution reçoit les valeurs des variables à chaque niveau d'appels.

Exécuter trop d'appels récursifs d'une fonction peut faire déborder la pile d'exécution. Par défaut, les implémentations de Python permettent de l'ordre de 1000 ou 3000 appels récursifs. On peut consulter cette limite ou la modifier avec les fonctions getrecursionlimit() et setrecursionlimit() de la bibliothèque sys.

Récursivité terminale :

1er appel → 2e appel → 3e appel → … → dernier appel

Les données stockées dans la pile d'exécution peuvent être effacées après chaque exécution de la fonction.

Récursivité non terminale :

1er appel → 2e appel → 3e appel → … → dernier appel → … → 3e appel → 2e appel → 1er appel

L'exécution de la fonction se poursuit après le retour du rappel. La pile d'exécution doit conserver les données relatives à chaque appel.

Dans le cas d'une fonction récursive terminale (tail recursive), la dernière instruction est le simple appel récursif de la fonction (sans autre calcul). Une fonction récursive terminale est donc généralement plus performante qu'une fonction récursive non terminale pour plusieurs raisons :

- la pile conserve uniquement les données du dernier appel (au lieu d'empiler les données de tous les appels consécutifs)
- après le dernier appel de la fonction, l'exécution se terminée, alors que dans le cas précédent, on remonte tous les appels dans l'autre sens.

En fait, une fonction récursive terminale se comporte comme une boucle, et peut d'ailleurs être transformée de manière très simple en fonction itérative. Certains langages transforment les récursivités terminales en itérations lors de la compilation.

Petit exercice : convertir la fonction factrect en une fonction itérative.

Cas de Python

La récursivité terminale n'est pas optimisée en Python. L'interpréteur Python ne détecte pas qu'une fonction est récursive terminale et va conserver inutilement les données de chaque niveau d'appel dans la pile. C'est un choix de son créateur, Guido van Rossum, qui préfère remplacer les fonctions récursives terminales par des boucles. Aussi, une fonction Python récursive terminale étant souvent plus complexe, sera en général plus lente qu'une implémentation non récursive terminale.

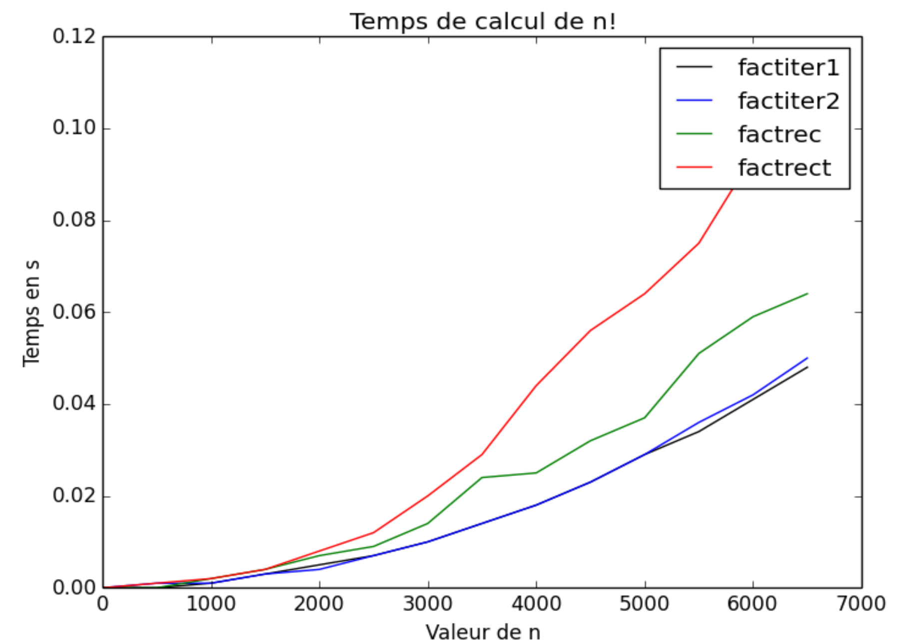
Dans d'autres langages, dits fonctionnels, n'ayant pas de structure de boucles, la récursivité terminale s'impose.

4. Comparaison des approches récursives et itératives

Alonzo Church, Stephen Kleene et Alan Turing ont créé chacun un modèle pour définir les fonctions calculables (qui peuvent être calculées à l'aide d'un algorithme). En 1937, Alan Turing a démontré que les fonctions calculables sont les mêmes dans les trois modèles.

Nous pouvons assimiler ces modèles à des langages de programmation, certains utilisant des fonctions récursives, d'autres pas. On démontre ainsi que les fonctions calculables sont les mêmes dans les différents langages proposés. On en déduit en particulier que tout algorithme récursif peut être remplacé par un algorithme itératif, et réciproquement.

La version récursive est en général un peu plus lente que la version itérative car la gestion des appels de fonction avec accès à la pile est plus lente que les boucles, mais la différence n'est pas toujours significative.



Certains algorithmes sont de nature récursive, lorsqu'un problème se décompose en sous-problèmes qui lui sont identiques. On les programme naturellement de manière récursive. La solution itérative peut être beaucoup plus complexe à programmer.

5. Récursivité mutuelle (ou croisée)

Il s'agit de deux fonctions (ou plus) s'appelant l'une l'autre. Vous réfléchirez à l'exemple suivant.

```
def even(n):
    if n == 0:
        return True
    else:
        return odd(n-1)
```

```
def odd(n):
    if n == 0:
        return False
    else:
        return even(n-1)
```

6. Preuves en récursivité

Les preuves de terminaison et de correction s'effectuent naturellement par récurrence.

<u>Exemple</u> : Preuve de correction et de terminaison de la fonction factrec

```
def factrec(n):
    if n < 2:
        return 1
    else:
        return n*factrec(n-1)
```

Terminaison : n diminue à chaque appel, et la fonction retourne 1 (et donc les appels se terminent) dès que $n < 2$. La fonction se termine.

Correction : Soit pour $n \in \mathbb{N}$, $\mathcal{P}(n)$: « factrec(n) renvoie $n!$ »
Initialisation : $\mathcal{P}(0)$ et $\mathcal{P}(1)$ est vraie
(la récurrence ne marche qu'à partir de $n = 2$)
Hérédité : supposons que $\mathcal{P}(n)$ est vraie pour $n \in \mathbb{N}$, avec $n \geq 1$.
Alors, factrec(n+1) renvoie $(n+1) \times$ factrec(n) $= (n+1) \times n! = (n+1)!$

7. <u>Complément</u> : autre écriture possible

```
def fact(n):
    return 1 if n < 2 else n * fact(n-1)
```