

# Récurtivité

## Définitions

Une fonction récursive comprend dans sa définition un ou plusieurs **appels vers elle-même**.

Mal écrite, une fonction récursive **peut ne jamais s'arrêter**.

Une fonction récursive doit par conséquent comprendre **une condition d'arrêt**, déterminée à partir du **cas de base**.

Une fonction qui n'est pas récursive est dite **itérative**.

## Exercice 1

La factorielle  $n!$  d'un nombre entier  $n \geq 1$  est définie par  $n! = 1 \times 2 \times 3 \times \dots \times n$ .

1. Compléter le tableau ci-dessous (calculatrice interdite !).

$n$	1	2	3	4	5	6	7
$n!$							

2. Quelle est la relation de récurrence vérifiée par la factorielle ?  
(En d'autres termes, exprimer  $n!$  en fonction de  $(n-1)!$ )

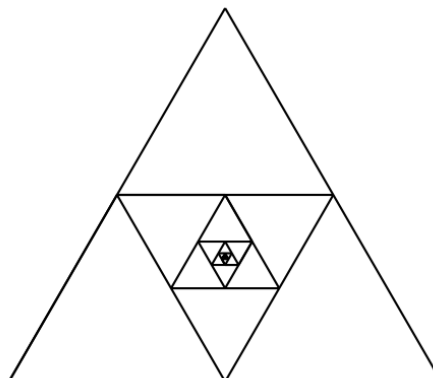
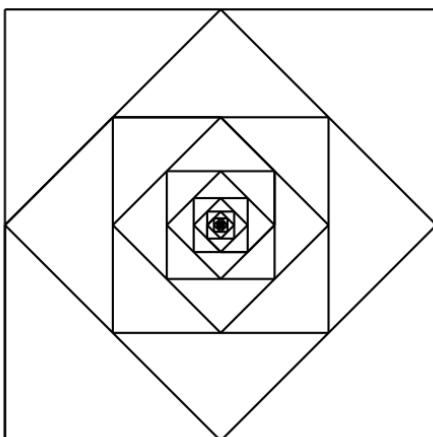
3. On considère la fonction ci-dessous.

```
1 def factorielle(n):  
2     return n*factorielle(n-1)
```

- (a) Expliquer pourquoi cette fonction ne renvoie pas la valeur attendue pour l'appel `factorielle(4)`.
- (b) Corriger la fonction pour qu'elle renvoie bien la valeur de  $n!$  pour une valeur entière du paramètre  $n$ .
- (c) Écrire un jeu de test permettant de vérifier les résultats (en se basant sur la première question).
- (d) Ajouter la précondition sur le type de  $n$ .
- (e) Combien d'appels rékursifs sont réalisés lors de l'appel `factorielle(4)` ?

## Exercice 2

Réaliser les figures suivantes à l'aide du module `Turtle` en utilisant des fonctions rékursives.



### Exercice 3

On considère la fonction récursive `fibonacci` ci-dessous, qui prend en argument un nombre entier.

```
1 def fibonacci(n):  
2     if n<=1: # condition d'arrêt  
3         return n  
4     return fibonacci(n-1)+fibonacci(n-2)
```

1. Quelle valeur renvoie l'appel `fibonacci(6)` ?
2. Combien d'appels récursifs ont été nécessaires ?
3. Prévoir la valeur renvoyée par l'appel `fibonacci(12)`.
4. Proposer une version itérative de la fonction `fibonacci`.

### Exercice 4 – (Palindrome)

Un palindrome est un mot pouvant se lire à l'endroit comme à l'envers, comme « été » ou « radar ».

On veut écrire une fonction `est_palindrome` prenant en entrée une chaîne de caractères `mot` et renvoyant un booléen indiquant si ce mot est un palindrome ou non.

1. Les cas de base sont les cas où `mot` est de longueur 0 ou 1. Que doit renvoyer la fonction dans ces cas ?
2. Dans tous les autres cas, `mot` est un palindrome si :
  - la première et la dernière lettre du mot sont identiques ;
  - le reste du mot (sans la première et la dernière lettre) est un palindrome.

Sur quelle valeur l'appel récursif se fait-il ?

3. Écrire la fonction `est_palindrome` incluant ces différents cas, en itératif et en récursif.

#### Aide

Pour prendre une partie d'une chaîne de caractères, on peut faire ce qu'on appelle du *slicing*, c'est-à-dire un découpage de la chaîne entre deux indices : pour une chaîne `chaine`, on peut choisir de considérer `chaine[i:j]` qui prendra les éléments de `i` à `j-1` de `chaine`.

### Exercice 5 – (Somme d'un tableau)

Une fonction récursive sur un paramètre de type `list` Python fonctionne de la même manière qu'avec un entier : il faut, à chaque nouvel appel, réduire le problème jusqu'au cas de base. C'est-à-dire ici, réduire le tableau initiale.

1. Écrire une fonction itérative (utilisant une boucle) permettant de calculer la somme des éléments d'un tableau de nombres `somme_it(tab)`. (Il est interdit d'utiliser la fonction Python `sum` !)

2. Pour quelle valeur du tableau est-il le plus facile de faire ce calcul, et que vaut-il dans ce cas ? (cas de base)
3. On peut faire du *slicing* sur les tableaux Python de la même manière que sur les chaînes de caractères.
  - (a) Comment prendre tous les éléments d'un tableau sauf le premier ?
  - (b) En déduire l'appel récursif de la fonction `somme_rec` qui sera fait pour faire le calcul de la somme d'un tableau de manière récursive.
4. Écrire la fonction récursive `somme_rec(tab)`.

#### Exercice 6 – (Le rendu de monnaie)

Le code suivant permet de rendre la monnaie avec l'algorithme glouton vu en classe de première. L'algorithme est cette fois-ci implémenté de manière récursive. Compléter le code :

```
1  def rendu_glouton(a_rendre, pieces):
2      """ rendu_glouton(int, list) -> list
3      a_rendre : la somme d'argent à rendre
4      pieces : un tableau de pièces possibles à rendre """
5      piece_max = pieces[0]
6
7      if a_rendre == 0:
8          return .....
9
10     if ..... >= piece_max:
11         return ..... + rendu_glouton(a_rendre - piece_max, pieces)
12
13     return rendu_glouton(a_rendre, .....)
```

14

```
15  valeurs_possibles = [100, 50, 20, 10, 5, 2, 1]
16
17  assert rendu_glouton(67, valeurs_possibles) == [50, 10, 5, 2]
18  assert rendu_glouton(291, valeurs_possibles) == [100, 100, 50, 20, 20, 1]
19  # si on ne dispose pas de billets de 100 :
20  assert rendu_glouton(291, valeurs_possibles[1:]) == [50, 50, 50, 50, 50, 20, 20, 1]
```