TP n°1 – Fonctions récursives

Notions utiles :

<u>Définition</u> : Une **fonction récursive** est une fonction qui contient au moins un appel à elle-même.

Par exemple : La fonction suivante calcule la factorielle d'un entier naturel n.

```
def factorielle(n) :
      #cas trivial 0 ! = 1
      if n == 0 :
            return 1
      #cas récursif ou hérédité
      return n*factorielle(n-1)
```

Pour bien implémenter une fonction récursive il faut s'assurer de sa terminaison. Pour cela on suit toujours les règles suivantes :

- La fonction doit contenir au moins un <u>cas trivial</u> ne comportant pas d'appel récursif.
- Les appels récursifs à la fonction <u>convergent toujours</u> vers le cas trivial.

Lors de l'exécution de la fonction chaque appel récursif met « en pause » l'exécution en cours, en attente d'obtenir le résultat qui est déterminé par l'appel suivant. Concrètement :

- Les appels sont tour à tour mis « en pause » jusqu'au dernier appel qui fournit un résultat (cas trivial). On appelle cela **le dépliage** ou la descente.
- Ce résultat est ensuite transmis à l'appel précédent qui l'utilise pour calculer son propre résultat et le transmettre à l'appel précédent, et ainsi de suite jusqu'au premier appel qui peut calculer le résultat final. On appelle cette phase **l'évaluation** ou la remontée.

Du point de vue de l'ordinateur l'appel d'une fonction récursive fait appel à **une pile d'exécution**. Une pile est une structure de type **LIFO** (Last In First Out). Les variables locales de chaque appel sont empilées successivement sur la pile. Lorsqu'on obtient un résultat exploitable au sommet de pile alors on peut commencer à dépiler.

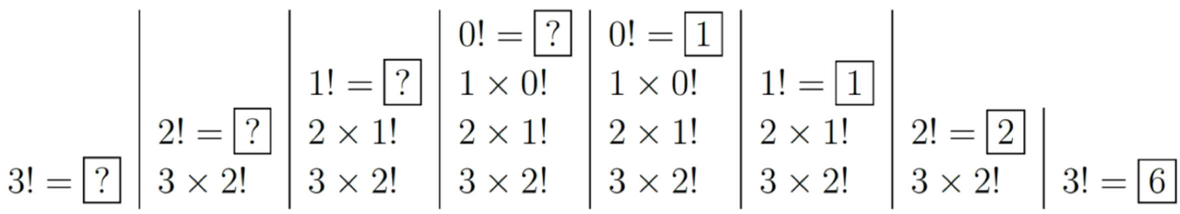


Figure 1 - Illustration du remplissage de la pile dans le cas de la fonction factorielle(3)

<u>**Attention !**</u> La pile d'exécution est une structure de taille limitée. Lorsque beaucoup d'appels sont nécessaires Python renvoie une erreur. Il faut donc être attentif lors de l'utilisation des fonctions récursives.

Exercice 1 : Adoptez des lapins

Vous faites l'acquisition d'un couple de bébés lapins. Au bout d'un mois ce couple devient adulte. Encore un mois plus tard il donne naissance à une couple de bébés lapins : vous possédez désormais 4 lapins. Par la suite tous les mois un couple d'adultes donne naissance à un couple de bébés qui mettent un mois à atteindre la maturité.

On obtient ainsi le schéma suivant :

Mois 0	**Mois 1**	**Mois 2**	**Mois 3**	**Mois 4**
mm	MM	MM mm	MM MM mm	MM MM MM mm mm

Tableau 1 - m : bébé lapin M : lapin adulte

1. À partir du tableau ci-dessus, conjecturer la relation de récurrence existant entre les nombres de couples de lapins au mois n $C(n) = c_n$, n+1 et n+2. Quel est le nom de cette célèbre suite ?
2. Écrire une fonction itérative `LapinsIt(n)` permettant de calculer le nombre de couples de lapins lors du mois `n` à partir d'une boucle `for`.
3. Écrire une fonction récursive `LapinsRec(n)` permettant de calculer le nombre de couples de lapins lors du mois `n`.

Vous pouvez vérifier la correction de vos fonctions à l'aide du tableau suivant :

c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}
1	1	2	3	5	8	13	21	34	55	89	144	233

4. Déterminer le temps de calcul du 35ème terme de la suite en utilisant les deux fonctions. Pour cela vous pouvez utiliser une structure utilisant le module `time` telle que celle-ci-dessous :

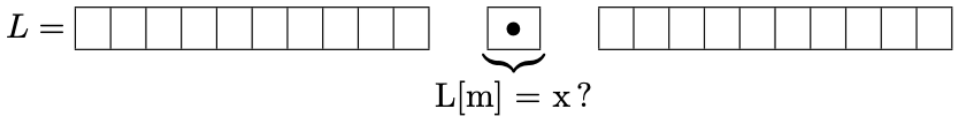
```
import time

# Définition des fonctions ici

debut_it = time.time()
# Appel à la fonction itérative ici.
fin_it = time.time()
duree_it = fin_it-debut_it

debut_rec = time.time()
# Appel à la fonction récursive ici.
fin_rec = time.time()
duree_rec = fin_rec-debut_rec

print(duree_it)
print(duree_rec)
```

5. Estimer la complexité de l'algorithme récursif en fonction de `n`. On montre que la complexité de l'algorithme récursif pour n > 2 est $A(n) \geq \left(\frac{3}{2}\right)^n$. Comment nomme-t-on ce type de complexité ?
6. Comparer les deux versions de l'algorithme en terme de simplicité et de complexité temporelle.

Exercice 2 : Diviser pour mieux régner

Nous disposons d'une liste d'entiers triés dans l'ordre croissant. Par une méthode dichotomique récursive on cherche à résoudre le problème de la recherche d'un élément x de cette liste L. On applique alors la stratégie du diviser pour régner.

- **Diviser** : On se place au milieu (ou presque) de la liste (notons m l'indice de cette position) et on découpe alors la liste en deux sous-listes.
- **Régner** : Si une sous-liste ne contient qu'un seul élément la réponse est obtenue en testant l'égalité de cet élément avec x, sinon la réponse pour une sous liste est obtenue de manière récursive.
- **Combiner** : On compare $L[m]$ et x : s'ils sont égaux on a trouvé, si $L[m]>x$ on cherche dans la sous-liste gauche, sinon on cherche dans la sous-liste droite.



1. Écrire une fonction récursive `rechDicho(L,x,i,j)` qui cherche l'élément `x` dans la liste triée `L` entre les indices `i` et `j`. La fonction retourne l'indice de `x` s'il est dans la liste et `False` sinon.
2. Comparer le temps d'exécution de votre fonction et d'une fonction qui parcourt simplement toute la liste dans le pire des cas.

Exercice 3 : Lire à l'envers

Un palindrome est un mot qui peut se lire dans les deux sens. Par exemple kayak, bob et laval sont les palindromes.

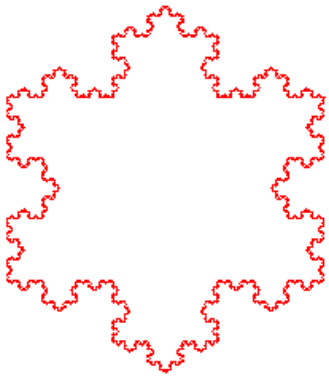
1. Écrire une fonction qui prend en paramètre une chaîne de caractère et qui retourne cette chaîne de caractères inversée.
2. En vous appuyant sur la fonction précédente écrire une fonction qui détermine si oui ou non un mot est un palindrome. Elle renverra True si le mot est un palindrome et False sinon.
3. Écrire une fonction récursive produisant le même résultat que la précédente mais cette fois-ci sans inverser l'ordre de la chaîne initiale.

Exercice 4 : Avez-vous la monnaie ?

Le système monétaire est donné sous la forme d'une liste `valeurs = [100,50,20,10,5,2,1]`. On suppose que chaque pièce ou billet est disponible sans limitation. On cherche à donner la liste des valeurs à rendre pour une somme donnée en argument. L'algorithme utilisé pour répondre à ce problème est de **type glouton**.

1. Écrire une fonction récursive `rendu_monnaie(a_rendre,rang)` prenant en argument la somme à rendre et le rang de la plus grande valeur de pièce ou billet disponible dans la liste des valeurs et qui retourne la liste des valeurs des pièces et billets à rendre. Par exemple, `rendu_monnaie(67,0)` renvoie `[50,10,5,2]` et `rendu_monnaie(102,1)` renvoie `[50,50,2]` (cas ou on a plus de billet de 100).

Exercice 5 : Le flocon de Van Koch



Le flocon de Von Koch, représenté à droite et imaginé en 1904 par le mathématicien suédois Helge Von Koch est l'une des premières courbes fractales à avoir été décrite. Les fractales permettent entre autres de modéliser le développement de certaines bactéries.

Pour construire ce flocon nous allons commencer par construire une courbe de Van Koch à l'aide du module turtle de Python. Dans ce module, la tortue est un outil de dessin pouvant tourner et avancer par segments successifs. Vous trouverez aisément l'aide de fonctionnement en ligne mais voici un rappel des fonctions les plus utiles.

Après avoir importé en module à l'aide de la commande **`import`** `turtle` **`as`** `tl` on peut utiliser :

- `tl.forward(cote)` pour faire avancer la tortue de `cote` pixels tout en traçant.
- `tl.left(angle)` pour faire tourner la tortue de `angle` degrés. Le sens dépend du paramétrage.
- `tl.setheading(angle)` pour déterminer l'orientation initiale `angle` en degrés.
- `tl.hideturtle()` pour cacher la tortue.
- `tl.speed(vitesse)` pour accélérer ou ralentir la tortue.
- `tl.color('couleur')` pour choisir la couleur du tracé.
- `tl.exitonclick()` pour que la fenêtre de dessin reste ouverte lorsque le tracé est terminé.

Une courbe de Van Koch de rang n=0 est un simple segment. Ensuite, à chaque incrémentation de n, tous les segments présents au rang précédent sont divisés en trois parts égales. Parmi ces trois parts, celle au centre est alors transformée en un triangle équilatéral dont on efface la base. La figure ci-dessous illustre ce principe :

1. Écrire une fonction récursive `courbeVonKoch(n,cote)` prenant en paramètre le rang `n` souhaité et la taille `cote` souhaitée pour le segment initial permettant de tracer la courbe de Von Koch de rang n. Attention tous les paramètres d'initialisation et de réglage sont à insérer hors de la fonction que l'on appellera ensuite dans l'éditeur.
2. Recopier et compléter la fonction suivante pour obtenir finalement le flocon de Von Koch de rang n.

```
def flocon(n,cote) :
    for i in range(…) :
            courbeVonKoch(n,cote)
            tl.left(…)
            i =
```