TP D'INFORMATIQUE 6
Récursivité

1 Exemple introductif

La programmation récursive se base sur le principe de l'**appel récursif**, c'est-à-dire l'appel d'une fonction dans sa propre définition. Similairement à une preuve par récurrence, une fonction récursive doit contenir au moins un cas de base sans appel récursif, et chaque appel récursif doit prendre un argument strictement inférieur à l'argument initial (pour un ordre dit **bien fondé**, comme l'ordre sur $\mathbb{N}$).
L'exemple suivant montre comment la définition récursive de la factorielle se traduit directement en fonction récursive :

```
def fact(n):
  if n==0:
    return 1                  #Cas de base
  else:
    return n*fact(n-1)        #Appel recursif
```

1. Détailler au papier le calcul de `fact(4)` en déroulant la définition récursive.
2. Recopier la fonction `fact` en Python et vérifier que l'appel `fact(4)` renvoie le résultat prédit.
3. Ajouter la ligne `print(n)` au début du corps de la fonction `fact` et exécuter à nouveau `fact(4)`. Observer que chaque appel, initial et récursifs, produit un affichage, et que la valeur de `n` change d'un appel à l'autre.
4. Que devrait-il se passer si `fact` est appelée sur un entier strictement négatif ? Tester.

2 Premières applications de la récursivité

1. Écrire une fonction récursive `pgcd` calculant le pgcd de deux entiers naturels a et b. On utilisera `a%b` le reste de la division euclidienne de a par b.
2. Recopier la procédure suivante :

   ```
   def afficher_ligne(n):
       print( '*' * n )
   ```

   Elle permet d'afficher une ligne formée de n caractères `*`, et on l'utilisera pour les questions suivantes.

   (a) Écrire une procédure récursive `tracer_triangle` prenant en argument un entier n et affichant un triangle formé des lignes de longueur $n, n-1, \ldots$
   Ainsi, `tracer_triangle(4)` devra afficher :

   ```
   ****
   ***
   **
   *
   ```

   (b) Écrire une procédure récursive `tracer_triangle2` similaire mais inversant le sens des lignes, de sorte à avoir la pointe en haut.

3. (a) Écrire une fonction `u` prenant en argument un flottant a et entier positif `n` et renvoyant le terme u_n de la suite définie par récurrence comme $u_0 = a$ et $u_{n+1} = \dfrac{1}{2}\left(u_n + \dfrac{a}{u_n}\right)$.

   (b) Tester sur `u(3,50)`, qui doit renvoyer `1.7320508075688772`. Si le temps de calcul est trop long, interrompre la console. Si chaque appel produit deux appels récursifs, le temps de calcul va croître en 2^n, on parle de **complexité exponentielle**. Le cas échéant, corriger ce comportement en n'utilisant qu'un seul appel récursif.

4. Écrire une fonction récursive `expo_rapide` prenant en argument deux entiers naturels k et n et renvoyant k^n selon la méthode de l'exponentiation rapide vue au TP précédent.
5. Écrire une fonction récursive `bin` prenant en argument un entier naturel et renvoyant son écriture binaire sous forme de liste de 0 et 1, le bit de poids faible étant en fin de liste. Ainsi, `bin(12)` doit renvoyer `[1,1,0,0]`. La fonction renverra la liste vide (`[]`) sur 0.

6. (a) Écrire une fonction `fibo` prenant en argument un entier positif `n` et renvoyant le n-ième terme de la suite de Fibonacci, définie par $F_0 = F_1 = 1$ et $F_{n+2} = F_n + F_{n+1}$.
   (b) La fonction `fibo` utilisant deux appels récursifs a une complexité exponentielle. Obtenir une version de complexité linéaire (on pourra utiliser une fonction auxiliaire récursive renvoyant deux termes consécutifs de la suite).

3 Recherche dichotomique

Dans cet exercice, on se propose d'écrire une version récursive de la recherche dichotomique dans un tableau trié. Pour éviter le coût de création d'un sous-tableau à chaque appel récursif, nous allons utiliser une fonction auxiliaire récursive prenant en argument les bornes entre lesquelles la recherche prend place.

1. Écrire une fonction récursive `recherche_dicho_bornes` prenant en argument une liste `L`, supposée triée, un élément `x`, un entier `a` et un entier `b` et testant si `x` est dans `L[a:b+1]` (ie entre les indices a et b inclus), en utilisant le principe de la recherche dichotomique.

   Ainsi, avec `l = [1, 3, 4, 6, 7, 10]` :

   `recherche_dicho_bornes(l, 4, 0, 1)` doit renvoyer `False`

   `recherche_dicho_bornes(l, 4, 0, 2)` doit renvoyer `True`
2. En déduire la fonction `recherche_dicho` prenant en argument une liste `L` triée et un élément `x`, et testant si `x` est dans `L`.

4 Tours de Hanoï

Le problème des tours de Hanoï se présente sous la forme de trois colonnes. Au départ, la première contient sept disques de tailles distinctes, empilés par taille décroissante (le plus grand en bas). Les deux autres colonnes sont vides. On peut déplacer un disque du sommet d'une colonne au sommet d'une autre colonne, mais seulement s'il ne recouvre pas un disque plus petit, de façon à conserver l'empilement par taille décroissante. L'objectif est de déplacer tous les disques sur la troisième colonne.

Écrire une procédure `hanoi` affichant les instructions pour résoudre le problème. On affichera `n -> n'` pour indiquer le déplacement du sommet de la colonne `n` au sommet de la colonne `n'`.

Par exemple, si le problème ne contenait que deux disques, la procédure devrait afficher :

```
1 -> 2
1 -> 3
2 -> 3
```

5 Flocon de Koch

L'objectif de cette partie est de tracer (une approximation finie de) la figure fractale du flocon de Koch. Nous allons utiliser pour le tracer la bibliothèque `turtle` qui permet de déplacer un curseur tout en dessinant sa trajectoire. Nous utiliserons les fonctions suivantes :

- `left`, prenant en argument un angle en degrés et faisant tourner le curseur sur lui-même de cet angle vers la gauche ;
- `right`, similaire à la fonction précédente mais vers la droite ;
- `forward`, faisant avancer le curseur de la distance donnée en argument ;
- `reset()`, permet d'effacer un tracer afin d'en recommencer un.

1. Charger la bibliothèque `turtle` avec la ligne `from turtle import *`
2. Tracer un carré de côté 100.
3. Chercher sur Wikipedia la définition récursive de la courbe de Koch et du flocon de Koch. En déduire une fonction `flocon` prenant en argument un entier n et une longueur l, et traçant l'approximation finie en n étapes du flocon de Koch partant d'un triangle équilatéral de longueur l. On pourra tester avec les valeurs $n = 4$ et $l = 200$.