Travaux pratiques 4

Récursivité (1)

Informatique tronc commun MPSI et PCSI

I Un exemple : la factorielle

La fonction factorielle peut être définie de manière itérative à l'aide de la relation $n! = \prod_{k=1}^{n} k$. Cela conduit au programme suivant :

```
def fact_it(n):
    p =1
    for k in range(1, n+1):
        p *= k
    return p
```

Néanmoins, la factorielle possède également une définition récursive :

$$0! = 1$$
$$n! = n \cdot (n-1)!$$

On constate que, pour connaître $n!$, il faut connaître $(n-1)!$, qui utilisera $(n-2)!$ ....

Définition 1 : Fonction récursive

On dit d'une fonction python qu'elle est récursive si, dans certains cas, elle s'appelle elle-même, avec de nouveaux paramètres (souvent plus petits).
En python, la création de fonctions récursives ne demande pas de syntaxe spécifique.

On peut alors définir la factorielle avec le code suivant :

```
def fact_rec(n):
    if n == 0:
        return 1          # condition d'arrêt
    return n*fact(n-1)    # appel récursif
```

On remarque qu'on ne fait pas appel tout le temps à la récursivité : pour $n = 0$ la fonction renvoie directement un résultat.

À retenir : terminaison

Une fonction récursive doit toujours comprendre au moins un cas terminal [a] qui ne demande pas d'appel récursif. Si elle est bien programmée, la fonction récursive doit toujours finir par aboutir à un tel cas, faute de quoi votre fonction part en récursivité infinie.

a. aussi appelé cas de sortie ou condition d'arrêt.

Dans le cas de la factorielle, comme elle fait appel à elle-même avec un paramètre diminué de 1, on aboutit forcément à 0 et la fonction donnera un résultat. En fait, cela n'est vrai que pour une variable (entière) **positive**, dans le cas où n est négatif, la fonction ne s'arrête jamais. En python, il y aura une limite dans le nombre d'appels récursifs [1].

1. Par défaut, limité à 1000 mais modifiable par la méthode `setrecursionlimit()` de la librarie `sys`.

On peut éviter cela soit en écrivant la condition dans la documentation et en espérant qu'elle sera respectée par les utilisateurs soit avec un test qui exclut les cas qui posent problème.

```
def fact(n):
    assert n>=0     # assertion : on vérifie que n est positif
    if n == 0:
        return 1
    else:
        return n*fact(n-1)
```

À retenir : `assert`

L'instruction `assert exp_bool` teste si l'expression booléenne `exp_bool` est vérifiée avant de poursuivre. Si l'expression est évaluée à `False`, la fonction est interrompue et un message d'erreur est renvoyé.

Enfin, il est parfois utile de représenter un **arbre des appels récursifs** (voir figure 1). Dans le cas de l'exécution de `fact_rec(4)`. On voit les différents appels qui apparaissent sous la forme d'une **pile**. C'est le dernier appel (la condition d'arrêt) qui permet de `dépiler`.

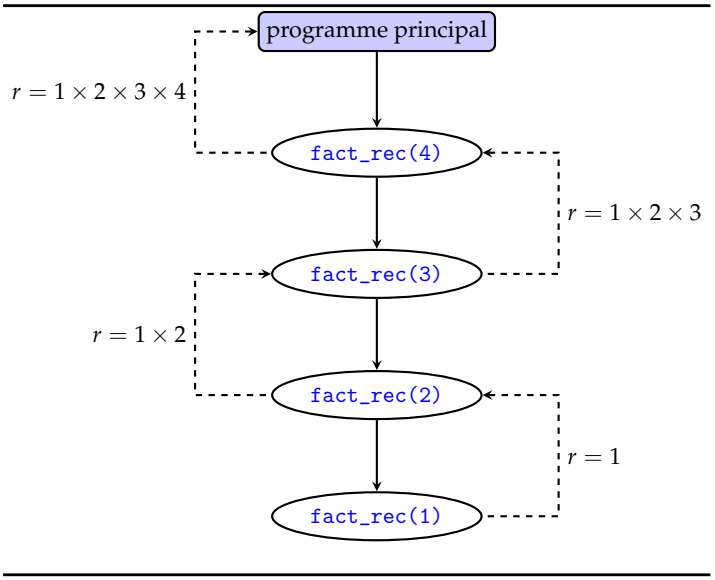


FIGURE 1 – Arbre d'appels récursifs

II Utilisation d'une relation de récurrence

Dans les exercices de cette section, on utilisera une relation de récurrence pour écrire une fonction récursive. On ne devra pas utiliser de boucles `while` ou `for`.

II.1 Récurrence simple

Remarque : dans vos programme, il ne sera pas nécessaire d'utiliser la méthode `assert`. On supposera que l'utilisateur choisit des "bonnes" valeurs de paramètres.

Exercice 1 - Produit
Écrire une fonction récursive `produit_entier(a, n)` qui renvoie le calcul $a \cdot n$ où n est un entier et a un flottant fournis en paramètres. Dans ce premier exemple, la relation de récurence est :

$$\forall n \geq 1, \quad a \cdot n = a + (n-1) \cdot a$$

Remarque : Cette fonction utilisera l'addition $+$ mais pas la multiplication $*$.
On pourra vérifier que `produit_entier(10, 9)` renvoie bien 90.

Exercice 2 - Somme de puissances
Écrire une fonction récursive `somme_puiss(n, p)` qui prend en argument deux entiers `n` et `p` et qui renvoie la somme

$$\forall n \geq 1, \quad \sum_{k=1}^{n} k^p$$

On pourra vérifier que `somme_puiss(10, 2)` renvoie `385`.

Exercice 3 - Suite de Heron
La suite de Héron est définie par $u_0 = 1$ puis $u_{n+1} = \frac{1}{2} \cdot \left(u_n + \frac{2}{u_n}\right)$. Écrire une fonction récursive `heron(n)` qui retourne le n-ième terme de la suite de Héron. On pourra vérifier que cette suite tend très rapidement vers $\sqrt{2}$.

Exercice 4 - Suite de Syracuse
La suite de Syracuse partant de l'entier positif a est définie par $u_0 = a$ puis

$$u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$$

Écrire une fonction récursive `syr_rec(n, a)` qui renvoie le terme d'indice n de la suite de Syracuse. On pourra vérifier que `print(syr_rec(6, 14))` renvoie 52.

Question difficile :

La conjecture de Syracuse affirme que pour tout entier $a > 0$, il existe un indice p tel que $u_p = 1$. Écrire une fonction récursive `syr_rec2(a)` qui détermine le plus petit entier p tel que $u_p = 1$. Autrement dit, le nombre d'itération nécessaire pour aller la première fois à 1 en partant de a.

On pourra vérifier que `print(syr_rec(14))` renvoie 17.

Exercice 5 - Suite récurrente générale
Écrire une fonction récursive `suite_rec(f, a, n)` qui prend en paramètres une fonction f, un flottant a et un entier n et qui retourne le n-ième terme de la suite définie par $u_0 = a$ puis $u_{n+1} = f(u_n)$.
Dans le cas où $f : x \mapsto \sin(x)$, on pourra vérifier que `suite_rec(f, 1, 10)` renvoie `0.46295789853781183`.

Exercice 6 - Modulo
Écrire une fonction récursive `modulo(n, d)` qui retourne le reste de la division euclidienne de n par d, tous deux positifs avec d non nul. Bien entendu, on n'utilisera pas le modulo (%) déjà implémentée en python. On pourra vérifier que `modulo(117, 7)` renvoie 5.

Exercice 7 - PGCD
Écrire une fonction récursive `pgcd(n, p)` qui retourne le PGCD [a] de n et p (avec $0 < p \leq n$) grâce à l'algorithme d'Euclide. On pourra vérifier que `pgcd(252, 105)` renvoie 21.

Rappel sur l'algorithme d'Euclide :

Le PGCD de deux nombres n'est pas changé si on remplace le plus grand d'entre eux par leur différence. Autrement dit, `pgcd(n, p)= pgcd(p, n-p)`. Si les entiers sont positifs, on peut réaliser `n//p` différences et `pgcd(n, p)= pgcd(p, n%p)`.

a. plus grand commun diviseur

II.2 Récurrence double

Bien entendu, on peut utiliser la récursivité même quand la relation de récurrence fait appel à plusieurs valeurs antérieures.

Exercice 8 - Nombres de Fibonacci
La suite de Fibonacci $(F_n)_{n\in\mathbb{N}^*}$ est définie par $F_0 = 0$, $F_1 = 1$ puis $F_{n+2} = F_{n+1} + F_n$ pour $n \geq 0$.

1. Écrire une fonction `fibo_rec(n)` qui renvoie la valeur F_n.
2. Vérifier que `fibo_rec(40)` vaut 165580141.
3. Représenter l'arbre d'appels récursifs pour `fibo_rec(4)`.
4. **Question à faire chez vous :** Prouver, par récurrence, que le calcul de F_n demande $F_{n+1} - 1$ additions.
5. **Question à faire chez vous :** En admettant ce résultat, montrer que la complexité de l'algorithme est en $K \cdot a^n$ avec $a = \frac{1+\sqrt{5}}{2}$ le *nombre d'or* qui est l'une des racines de l'équation $X^2 - X - 1 = 0$.

Exercice 9 - Coefficients binomiaux
On rappelle que $\binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1}$ pour $1 \leq p < n$.

1. Écrire une fonction `coef_binom_rec(n, p)` qui calcule de manière récursive $\binom{n}{p}$.
2. Calculer $\binom{30}{15} = 155117520$. Que pensez-vous de la complexité obtenue ?

Exercice 10 - Coefficients binomiaux bis
On rappelle que $\binom{n}{p} = \frac{n}{p}\binom{n-1}{p-1}$ pour $p \geq 1$. En utilisant cette relation de récurrence, écrire une fonction `binom_rec(n, p)` qui calcule de manière récursive $\binom{n}{p}$ qui effectue moins de calculs.

II.3 Travail sur une liste

Pour définir une fonction récursive sur une liste, on peut souvent trouver une relation de récurrence qui exprime le résultat pour une liste de taille n en fonction du dernier élément et de la liste des $n-1$ premiers éléments.
Par exemple la somme des termes d'une liste `[a0, a1, ..., ap]` est définie par

$$\sum_{k=0}^{p} a_k = a_p + \sum_{k=0}^{p-1} a_k$$

Cela donne la fonction

```
def somme(L):
    n = len(L)
    if n == 0:
        return 0
    else:
        return L[n-1] + somme(L[ : n-1])
```

Exercice 11 - Produit
Écrire une fonction récursive `produit(L)` qui calcule le produit des termes d'une liste. Ainsi, `produit ([4,3,7,9])` devra renvoyer 756.

Exercice 12 - Palindrome
Écrire une fonction récursive `test_palin(ch)` qui renvoie `True` si la chaîne de caractères fournie en paramètre est un palindrome et `False` dans le cas contraire. On vérifiera que `"kayak"` est un palindrome et que la phrase `"la mariée ira mal"` pourra être considérée comme un palindrome si on enlève l'accent et les espaces.

Exercice 13 - Croissance
Écrire une fonction récursive `test_croiss(L)` qui détermine si la liste fournie en paramètre est croissante.

Exercice 14 - Maximum
Écrire une fonction récursive `max_liste(L)` qui renvoie la valeur maximale des termes d'une liste fournie en paramètre.
Remarque : On n'utilisera pas la fonction `max` déjà implémentée en python.

II.4 Fonction auxiliaire

Dans les exemples ci-dessus, on copie une liste à chaque appel ; cela induit une complexité quadratique car on copie $n-1$ éléments puis $n-2$ jusqu'à 1, ce qui donne $\frac{n\cdot(n-1)}{2}$ copies.
Il est plus efficace d'utiliser des indices comme paramètres supplémentaires dans une fonction auxiliaire. Par exemple, la somme d'une liste peut être obtenue de la manière suivante :

```
def somme(L):
    def somme_aux(liste , ind):
        if ind >= len(liste ):
            return 0
        else:
            return liste[ind] + somme_aux(liste, ind+1)
    return somme_aux(L, 0)
```

Exercice 15 - Test de croissance avec fonction auxiliaire
Reprogrammez `test_croiss` et `max_liste` de cette manière.
Remarque : On n'utilisera pas la fonction `max` déjà implémentée en python.

Exercice 16 - Deux indices
Reprogrammez `test_palin` à l'aide d'une fonction auxiliaire récursive.

Exercice 17 - Fibonacci efficace
On revient sur la suite de Fibonacci (voir l'exercice 8). En écrivant une fonction auxiliaire qui utilise les deux dernières valeurs calculées en plus de l'indice, écrire une fonction qui calcule F_n de manière efficace.

III Un peu de dessin

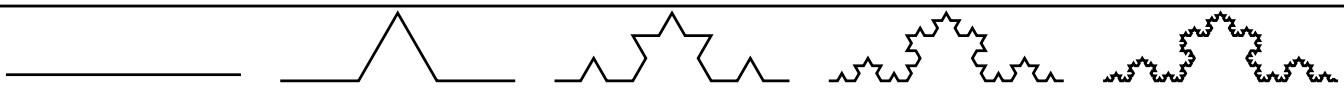
Exercice 18 - Triangle
On considère les programmes ci-dessous :

```
def triangle1(n):
    print(n*"X")
    if n>0:
        triangle1(n-1)
triangle1(4)
```

```
def triangle2(n):
    if n>0:
        triangle2(n-1)
    print(n*"X")
triangle2(4)
```

Prévoir ce qu'on obtient dans les deux cas.

Exercice 19 - Courbe de Koch
En figure 3, on observe la construction de la courbe de koch qui peut se définir par récurrence (voir figure 4). Au rang $n+1$, chaque segment AB de la courbe de rang n est divisée en parties égales et on ajoute le point E tel que le triangle CDE soit équilatéral.
Écrire un algorithme récursif qui génère le tracé de la courbe de Koch. Cet algorithme doit pouvoir aussi générer les courbes de la figure 5.



$n=0$ $n=1$ $n=2$ $n=3$ $n=4$

FIGURE 3 – À la n-ième itération, on constate qu'il y a 4^n segments.

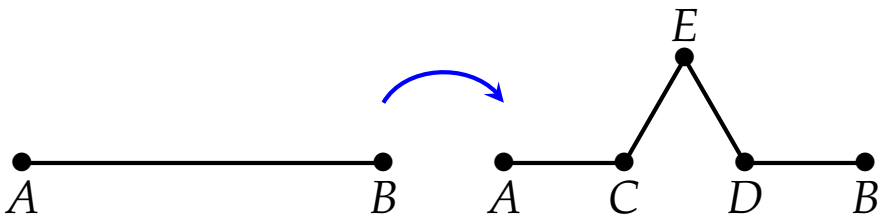
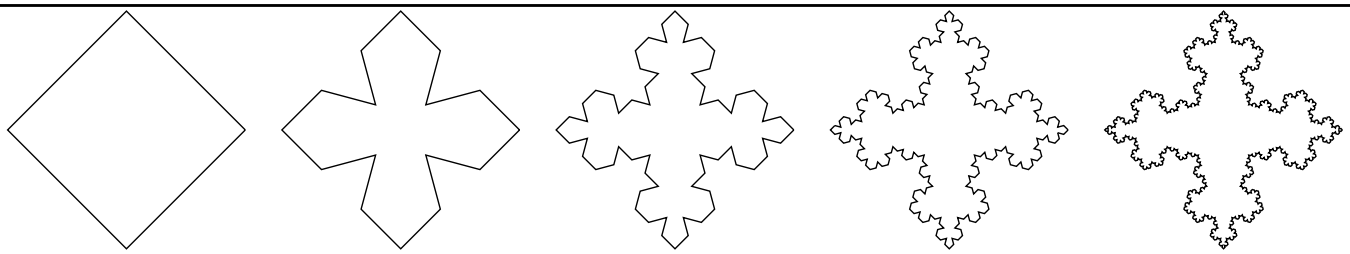


FIGURE 4 – Transformation de l'un des segments de la courbe.



$n=0$ $n=1$ $n=2$ $n=3$ $n=4$

FIGURE 5 – Transformation progressive d'un carré.

Les fonctions de `matplotlib` permettent le dessin mais on peut aussi utiliser le module `turtle`.
C'est un ensemble d'outils permettant de dessiner à l'aide d'instructions simples. Ses principales fonctions sont

- `reset()` : on efface tout et on recommence.
- `goto(x,y)` : aller à l'endroit de coordonnés x et y.
- `forward(distance)` : avancer d'une distance donnée en faisant un trait.
- `backward(distance)` : reculer. :
- `up()` : relever le crayon (pour pouvoir avancer sans dessiner).
- `down()` : abaisser le crayon (pour pouvoir recommencer à dessiner).
- `color(couleur)` : changer de couleur, `couleur` peut être une chaîne prédéfinie ('red', 'blue', 'green', etc)
- `left(angle)` : tourner à gauche d'un angle donné (exprimé en degrés).
- `right(angle)` : tourner à droite.
- `width(épaisseur)` : choisir l'épaisseur du tracé.
- `fill(1)` :remplir un contour fermé à l'aide de la couleur sélectionnée (on termine la construction par `fill(0)`).
- `write(texte)` : texte doit être une chaîne de caractères délimitée avec des " ou des '.

Il faut noter que le tracé peut être long.

IV Solutions

Solution de l'exercice 1 - Produit

```
def produit_entier(a, n):
    if n == 0:
        return 0
    else:
        return a + produit_entier(a, n-1)
```

Solution de l'exercice 2 - Somme de puissances

```
def somme_puiss(n, p):
    if n == 0:
        return 0
    else:
        return n**p + somme_puiss(n-1, p)
```

Solution de l'exercice 3 - Suite de Heron

```
from math import sqrt
def heron(n):
    if n==0:
        return 1
    else:
        return 0.5*(heron(n-1)+2/heron(n-1))
for i in range(5):
    print(abs(heron(i)-sqrt(2)))
```

Solution de l'exercice 4 - Suite de Syracuse

```python
# suite de syracuse recursive
def syr_rec(n,a):
    if n==0:
        return a
    else:
        if syr_rec(n-1, a)%2==0:
            return syr_rec(n-1, a)//2
        else:
            return 3*syr_rec(n-1, a)+1
# Cette solution a l'inconveniant de calculer deux fois syr_rec(n-1, a)
# une meilleures solution
def syr_rec(n,a):
    if n==0:
        return a
    else:
        u = syr_rec(n-1, a)
        if u % 2 == 0:
            return u//2
        else:
            return 3*u + 1

print(syr_rec(6, 14))
# nombre de fois pour retourner à  1
# le compteur commence à 0 et continue tant qu'on fait des appels recursifs
def syr_rec2(a):
    if a==1:
        return 0
    elif a%2==0:
        return 1 + syr_rec2(a//2)
    else:
        return 1 + syr_rec2(3*a+1)
print(syr_rec2(14))
```

Solution de l'exercice 5 - Suite récurrente générale

```python
def suite_rec(f, a, n):
    if n==0:
        return a
    else:
        return f(suite_rec(f, a, n-1))
from math import sin
print(suite_rec(sin, 1, 10))
```

Solution de l'exercice 6 - Modulo

```python
def modulo(n, d):
    if n < d:
        return n
    else:
        return modulo(n-d, d)
print(modulo(117, 7))
```

Solution de l'exercice 7 - PGCD

```
def pgcd(n, p):
    if p == 0:
        return n
    else:
        return pgcd(p, n%p)
print(pgcd(252, 105))
```

Solution de l'exercice 8 - Nombres de Fibonacci

1. Le programme est donné ci-dessous :

```
def fibo_rec(n):
    if n <= 1:
        return n
    else:
        return fibo_rec(n-1) + fibo_rec(n-2)
```

   Remarque : La fonction renvoie une valeur même pour n négatif.

2. Le calcul de `fibo_rec(40)` est assez long[2]. Cette algorithme est moins efficace que son implémentation itérative avec une boucle `for`.

3. L'arbre des appels a été représenté en figure 6.

4. Montrons par récurrence que le nombre d'additions C_n vérifie :

$$\forall n \geq 0, \quad C_n = F_{n+1} - 1$$

   - **Initialisation** : $C_0 = 0 = F_0 - 1$ et $C_1 = 0 = F_1 - 1$;
   - **Hérédité** : on suppose la propriété vraie aux rang n et $n+1$. On a alors :

$$\begin{aligned} C_{n+2} &= C_{n+1} + C_n + 1 \\ &= (F_{n+2} - 1) + (F_{n+1} - 1) + 1 \\ &= F_{n+3} - 1 \end{aligned}$$

   - **Conclusion** : la propriété est vraie pour tout $n \geqslant 0$.

5. **Remarque préliminaire :** Si l'implémentation itérative a une complexité temporelle en $O(n)$, la complexité de l'implémentation récursive est majorée[3] par $O\left(2^n\right)$.

   Pour $n \geqslant 2$, le nombre d'opérations élémentaires fait par l'algorithme vaut $C_n = C_{n-1} + 1 + C_{n-2} + 1$. On pose $U_n = C_n + 2$ car -2 est un point fixe de la récurrence précédente :

$$\forall n \geqslant 2, \quad U_n = U_{n-1} + U_{n-2}$$

   C'est une récurrence classique que l'on résoud par la méthode de l'équation caractéristique et on trouve :

$$U_n = K \cdot a^n + K' \cdot b^n \quad \text{où} \quad a = \frac{1+\sqrt{5}}{2} \approx 1,6 \quad \text{et} \quad b = \frac{1-\sqrt{5}}{2} \approx -0,6$$

   On en déduit $U_n = O\left(a^n\right)$ puis $C_n = O\left(a^n\right)$. La complexité temporelle est très "mauvaise" (complexité exponentielle).

2. voisin d'une minute sur une machine de 2017.
3. *Grosso modo*, on multiplie par deux le nombre d'additions à changement de niveau sur l'arbre binaire en figure **??**

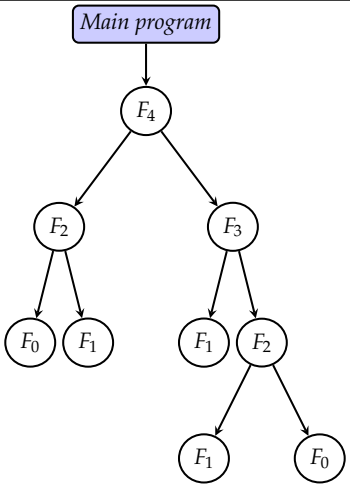


FIGURE 6 – Arbre d'appels récursifs

Solution de l'exercice 9 - Coefficients binomiaux

1. Le programme est donné ci-dessous.

```
def coef_binom_rec(n, p):
    if p==0 or p==n:
        return 1
    else :
        return coef_binom_rec(n-1, p) + coef_binom_rec(n-1, p-1)
print(coef_binom_rec(30,15))
```

2. Comme dans l'exercice 8, on prouve que le nombre d'additions utilisées pour calculer $\binom{n}{p}$ est $\binom{n}{p} - 1$, c'est un calcul lent.

Solution de l'exercice 10 - Coefficients binomiaux bis

```
def binom_rec(n, p):
    if p == 0:
        return 1
    else:
        return (n*binom_rec(n-1, p-1))//p
print(binom_rec(30,15))
```

On notera l'ordre des calculs et l'usage de la division entière pour obtenir un résultat entier. Le nombre d'opérations est de l'ordre de $2p$.

Solution de l'exercice 11 - Produit

```
def produit(L):
    n = len(L)
    if n == 0:
        return 1
    else:
        return L[-1] * produit(L[ : n-1])
print(produit(produit([4,3,7,9]))
```

Solution de l'exercice 12 - Palindrome

```
def test_palin(ch):
    n = len(ch)
    if n<=0:
        return True
    else:
        return (ch[0] == ch[n-1]) and test_palin(ch[1: n-1])
print(test_palin("kayak"))
print(test_palin("la mariée ira mal"))
print(test_palin("lamarieeiramal"))
```

Solution de l'exercice 13 - Croissance

On peut aussi retirer le premier terme.

```
def test_croiss(L):
    n  = len(L)
    if n<=1:
        return True
    else:
        return (L[1]>=L[0]) and test_croiss(L[1:])
print(test_croiss([1,12, 20, 14])) # renvoie False
print(test_croiss([1,12, 14, 20])) # renvoie True
```

Solution de l'exercice 14 - Maximum

```
def max_liste(L):
    n = len(L)
    if n==1:
        return L[-1]
    else:
        if L[-1]>=L[-2]:
            return max_liste([L[-1]]+L[:n-2])
        else:
            return max_liste([L[-2]]+L[:n-2])
print(max_liste([1, 10, 14, 4])) # renvoie 14
```

Le programme suppose que la liste n'est pas vide.

Solution de l'exercice 15 - Test de croissance avec fonction auxiliaire

```
def test_croiss(L):
    # la fonction auxiliaire vérifie que la sous-liste
    # au-dela de ind est croissante.
    def aux(liste, ind):
        if ind>=len(liste)-1:
            return True
        else:
            return aux(liste,ind+1) and (liste[ind+1]>=liste[ind])
    return aux(L, 0)
print(test_croiss([1,12, 20, 14]))

def max_liste(L) :
    def aux(liste, ind):
        if ind == len(liste)-1:
            return liste[ind]
        else :
            if liste[ind]>aux(liste,ind+1):
                return liste[ind]
            else:
                return aux(liste,ind+1)
    return aux(L, 0)
print(max_liste([1,12, 20, 14]))
```

Solution de l'exercice 16 - Deux indices
Ici, on aura besoin de deux indices qui se rejoigne vers le "centre" de la chaîne de caractères.

```
def test_palin(ch):
    def aux(i, j):
        if i >= j :
            return True
        else :
            return ch[i]==ch[j] and aux(i+1, j-1)
    return aux(0, len(ch)-1)
print(test_palin('kayak'))
```

Solution de l'exercice 17 - Fibonacci efficace

```
def fibo(n):
    def aux_fibo(n, f, ff):
        if n == 0:
            return f
        else:
            return aux_fibo(n-1, ff, f+ff)
    return aux_fibo(n, 0, 1)
```

Solution de l'exercice 18 - Triangle
On obtient dans les deux cas un triangle formé du caractère `"X"`. On constatera que la permutation entre l'instruction `if` et le `print` change la figure obtenue.

Solution de l'exercice 19 - Courbe de Koch
Il peut être utile de travailler avec des nombres complexes. On rappelle qu'on définit le complexe $z = 1 + 2i$ par `1+2*1j`. Le package `numpy` fournit alors tous les outils pour manipuler les complexes.

```
# courbe de koch recursive
import matplotlib.pyplot as plt
import numpy as np
def inserer(L):
    res = []
    for i in range(len(L)-1):
        zA = L[i]
        zB = L[i+1]
        a= np.abs(zB-zA)
        zC = zA+(zB-zA)/3
        zD = zA+2*(zB-zA)/3
        zE = zC+a/3*(np.cos(np.pi/3+np.angle(zB-zA))+1j*np.sin(np.pi/3+np.angle
            (zB-zA)))
        xE = np.real(zE)
        yE = np.imag(zE)
        res = res + [L[i]]+[zC, zE, zD]
    return res+[L[-1]]
def koch_rec(n, L):
    if n==0:
        return L
    else:
        return koch_rec(n-1, inserer(L))
L0 = [0*1j,3+0*1j]
L = koch_rec(4, L0)
Lx = [np.real(z) for z in L]
Ly = [np.imag(z) for z in L]
plt.figure()
plt.axis("equal")
plt.plot(Lx, Ly)
plt.show()
```

Avec le module `turtle`.

```
from turtle import *

def motif(c,n):
    if n==0 :
        forward(c)
    else :
        motif(c/3.0,n-1)
        left(60)
        motif(c/3.0,n-1)
        right(120)
        motif(c/3.0,n-1)
        left(60)
        motif(c/3.0,n-1)

motif(150, 4)
```