TP 6 : Récursivité
Correction

Exercice 1 1. On écrit :

```
def suite(n):
    if n==1: # condition d'arrêt
        return 1
    else : # appel ré cursif
        return 1+2*suite(n-1)
```

2. L'exécution de `suite(1000)` va appeler successivement `suite(999)`, `suite(998)`, `suite(997)`, `suite(996)` jusqu'à arriver à la condition d'arrêt. La structure (de pile) de mémorisation des résultats est limitée à environ 1000 appels récursifs en python. Si cette capacité est dépassée, comme ici, il est retourné : `RecursionError: maximum recursion depth exceeded in comparison`.

Exercice 2 1. On classe les différentes manières de construire une rangée de longueur n en deux parties disjointes :

- Celles dont la dernière brique est de longueur 2 : Il y en a `briques(n-2)`.
- Celles dont la dernière brique est de longueur 3 : Il y en a `briques(n-3)`.

Cela fournit la relation `briques(n)=briques(n-2)+briques(n-3)`.

2. On écrit la fonction récursive ci-après :

```
def briques(n):
    if n==1:
        return 0
    if n==2:
        return 1
    if n==3:
        return 1
    else :
        return briques(n-2)+briques(n-3)
```

3. L'éxécution de `briques(63)` retourne 20330163 au bout d'un certain temps... ce qui peut paraitre surprenant pour un aussi petit nombre (et à comparer à une version itérative). Pour déterminer `briques(63)`, il faut déterminer `briques(61)` et `briques(60)` qui demandent eux-mêmes de déterminer `briques(59)`, `briques(58)`, `briques(58)` (à nouveau), `briques(57)` et ainsi de suite jusqu'à ce que tous les appels successifs aboutissent à une condition d'arrêt. Le nombre d'appels peut devenir excessivement important et rendre un algorithme récursif moins efficace qu'une version itérative.

Exercice 3 1. En remarquant que la somme des chiffres de 841 est égale à la somme des chiffres de 84 plus le chiffre des unités de 841, on écrit

```python
def somme_chiffres(n) :
    assert n >=0
    if n//10==0: # Si n n'a qu'un seul chiffre dans son écriture décimale.
        return n
    else :
        return n%10 + somme_chiffres(n//10)
```

2. Une fonction récursive pouvant exécuter au maximum de l'ordre de 1000 appels récursifs, le plus petit entier qui posera le problème annoncé sera le plus petit entier dont l'écriture décimale possède 1001 chiffres (un appel pour le chiffre des dizaines, un appel pour le chiffres des centaines, ...), ie il s'agit de $n = 10^{1000}$.

3. On écrit :

```python
def racine_digitale (n) :
    assert n >=0
    if n//10==0: # Si n ne possède qu'un chiffre, on retourne n.
        return n
    else :
        return racine_digitale (somme_chiffres(n))
```

Exercice 4 On écrit :

```python
def triangle1 (n):
    if n==1:
        print('*')
    else :
        print('*'*n)
        triangle2 (n-1)

def triangle2 (n):
    if n==1:
        print('*')
    else :
        triangle1 (n-1)
        print('*'*n)
```

Exercice 5 a) On écrit :

```python
def hanoi(k,a,b,c):
    L=[]
    if k==1:                # S'il n'y a qu'un seul disque,
        L.append((a,c))     # on le déplace de la tour de départ à la tour finale.
    else :
        L=L+hanoi(k-1,a,c,b) # On déplace le haut de la tour vers la tour intermédiaire (phase 1).
        L.append((a,c))      # On déplace le gros disque vers la bonne tour (phase 2).
        L=L+hanoi(k-1,b,a,c) # On déplace la tour intermédiaire vers la bonne tour (phase 3).
    return L
```

b) On note $C(n)$ le nombre minimal de déplacements à réaliser dans le problème à n disques. Pour déplacer $n+1$ disques, il faut :

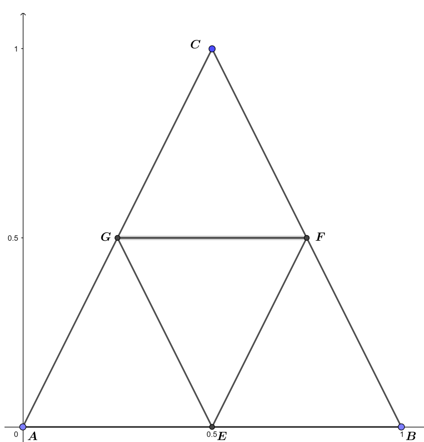
- Commencer par déplacer les n premiers sur une tour intermédiaire (phase 1) qui demande $C(n)$ déplacements,
- déplacer ensuite le gros disque vers la bonne tour (phase 2) qui demande 1 déplacement,
- replacer la tour intermédiaire sur le gros disque (phase 3) qui demande à nouveau $C(n)$ déplacements.

On obtient la relation $C(n+1) = 1+2C(n)$ avec la condition $C(1) = 1$ (un seul déplacement suffit avec un seul disque). La suite $(C(n))$ est la suite de l'exercice 1. L'exécution de `suite(32)` retourne `4294967295`. C'est le nombre de secondes qu'il faudra à l'étudiant pour résoudre le problème. Autrement dit, il lui faudra plus de 136 années...

Exercice 6 Pour créer toutes les listes à n éléments demandées, on ajoute 0 ou 1 à la fin de toutes les listes à $n-1$ éléments.

```python
def mot(n):
    if n==1:
        return [[0],[1]]
    else :
        L=mot(n-1)
        Rep=[]
        for m in L:
            Rep.append(m+[0])
            Rep.append(m+[1])
        return Rep
```

Exercice 7 On illustre la situation :



1. On note E le milieu de $[AB]$, F le milieu de $[BC]$ et G le milieu de $[AC]$. On a :

$$(x_E, y_E) = \left(\frac{x_A + x_B}{2}, \frac{y_A + y_B}{2}\right), \quad (x_F, y_F) = \left(\frac{x_B + x_C}{2}, \frac{y_B + y_C}{2}\right)$$
$$\text{et } (x_G, y_G) = \left(\frac{x_A + x_C}{2}, \frac{y_A + y_C}{2}\right).$$

2. On écrit :

```
import matplotlib.pyplot as pypl

def triangle (xA,yA,xB,yB,xC,yC,n):
    if n>0:
        pypl.plot([xA,xB,xC,xA],[yA,yB,yC,yA])   # On trace le triangle ABC
# Calcul des coordonnées des milieux des côtés du triangle père :
        xE = (xA+xB)/2 ; yE =(yA+yB)/2 # E est le milieu de [AB]
        xF = (xB+xC)/2 ; yF =(yB+yC)/2 # F est le milieu de [BC]
        xG = (xA+xC)/2 ; yG =(yA+yC)/2 # G est le milieu de [AC]
# Appel de la procédure pour les trois triangles fils :
        triangle (xA,yA,xE,yE,xG,yG,n-1) #Etape n-1 dans le triangle AEG
        triangle (xE,yE,xB,yB,xF,yF,n-1) #Etape n-1 dans le triangle EBF
        triangle (xG,yG,xF,yF,xC,yC,n-1) #Etape n-1 dans le triangle GFC
    pypl.show() # Uns fois les appels récursifs terminés, on affiche le résultat.
```

Exercice 8 1. Puisqu'il est demandé de retourner une nouvelle liste, on peut utiliser du slicing et la concaténation de listes pour répondre à la question posée.

```
def inserer(L,i,e):   # insérer dans une liste L l'élément e au rang i
    return L[:i]+[e]+L[i:]
```

2. On écrit :

```
def permutations(L): # générer toutes les permutations d'une même liste
    if len(L)==1:
        return [L]
    else :
        toutes=permutations(L[1:]) # On considère la liste privée de l'élément de rang 0.
        Res=[]
        for i in range(len(L)):
            for liste in toutes :
                Res.append(inserer(liste,i,L[0]))   # On insère l'élément de rang 0 de L à toutes les positions possibles de toutes les permutations de L[1:].
    return Res
```