Récursivité

1. Une fonction informatique a le droit de *s'appeler elle–même* ; on dit alors qu'elle est **récursive**.
2. Pour que la fonction récursive soit bien définie et rende une valeur, il y a deux points clés :
   - que l'on sache comment appliquer la fonction sur des « cas de base » ;
   - que le nombre d'appels récursifs ne soit pas infini.

> **Remarque**
> Ces deux points correspondent à l'initialisation et à l'hérédité dans une récurrence.

3. La récursivité peut masquer des coûts cachés.

1 Exemples

1.1 Factorielle

Deux définitions de la factorielle, en mathématiques, sont possibles :

1. Avec des pointillés : $n! = 1 \times 2 \times 3 \times \cdots \times n$
2. Par une relation de récurrence : $0! := 1$ et $(n+1)! := (n+1) \times n!$

Grâce aux boucles **for**, vous savez implémenter la première définition en python :

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

Mais on peut aussi utiliser la définition par récurrence, ce qui amène à programmer une fonction récursive :

```
def factorielle_recursif(n):
    if n==0:
        return 1
    else:
        return n*factorielle_recursif(n-1)
```

Le language Python autorise la récursivité : regardons, par exemple, comme il traite l'appel à `factorielle_recursif(5)` :

factorielle_recursif(5)= 5 × factorielle_recursif(4) ; *(on mémorise cette étape et on appelle alors la fonction factorielle_recursif)*

factorielle_recursif(4)= 4 × factorielle_recursif(3) ; *(on mémorise cette étape et on appelle alors la fonction factorielle_recursif)*

factorielle_recursif(3)= 3 × factorielle_recursif(2) ; *(on mémorise cette étape et on appelle alors la fonction factorielle_recursif)*

factorielle_recursif(2)= 2 × factorielle_recursif(1) ; *(on mémorise cette étape et on appelle alors la fonction factorielle_recursif)*

factorielle_recursif(1)= 1 × factorielle_recursif(0) ;

factorielle_recursif(0)=1

L'ordinateur a alors « empilé » les étapes, il va maintenant les « dépiler » en partant du bas et en remplaçant successivement :

factorielle_recursif(1)= 1 × 1=1
factorielle_recursif(2)= 2 × factorielle_recursif(1)=2 × 1=2
factorielle_recursif(3)= 3 × factorielle_recursif(2)=3 × 2 =6
factorielle_recursif(4)= 4 × factorielle_recursif(3)=4 × 6 = 24
factorielle_recursif(5)= 5 × factorielle_recursif(4)=5 × 24=120.

> **Méthode**
> La récursivité utilise le même type d'idée que la récurrence : si on connaît la valeur de f pour les cas de bases et que l'on sait calculer f(objet) en fonction de f(objet plus proche du cas de base), alors on sait calculer f de façon récursive.
> **Grâce à la récursivité Python gère lui même l'empilage et le dépilage des étapes, vous n'avez pas à vous inquiéter des « entrailles » de l'algorithme.**

1.2 Somme des inverses

Soit $n \in \mathbb{N}^*$, on cherche à calculer $\sum_{k=1}^{n} \frac{1}{k}$.

Voici un code **impératif** pour cette « somme des inverses » :

```
def somme_des_inverses_imperatif(n):
    S=0
    for k in range(1,n+1):
        S+=1/k
    return S
```

Attention

Pour écrire un code **récursif**, deux questions à se poser :

- Quel est/sont le/les cas de base triviaux ?
- Comment « démonter » un cas complexe pour le rendre plus simple ?

Une réponse possible ici :

- $\sum_{k=1}^{1} \frac{1}{k} = 1$
- $\forall n \geqslant 2 \quad \sum_{k=1}^{n} \frac{1}{k} = \left(\sum_{k=1}^{n-1} \frac{1}{k}\right) + \frac{1}{n}$

Exercice 1 Ecrire maintenant une fonction qui reçoit en argument un entier naturel n non nul, et retourne la valeur de $\sum_{k=1}^{n} \frac{1}{k}$. Cette fonction sera récursive et utilisera ce qui précède.

1.3 Somme des éléments d'une liste

Pour les listes, les cas de base sont la liste vide [] et les listes de longueur petite (1 ou 2). Pour les chaines de caractères, les cas de base sont la chaine vide ' ' et les chaines de longueur petite (1 ou 2).

Ainsi pour écrire récursivement une fonction s donnant la somme des éléments d'une liste (fonction déjà implémentée sous le nom de sum) on sépare cas de base / dé-construction :

1. s([])=0
2. s([a])=a
3. s([a1, a2, ..., an])=a1+s([a2, ...,an])

D'où le code :

```
def somme_elements_liste_recursive(l):
    if len(l)==0:
        return 0
    else:
        return l[0]+somme_elements_liste_recursive(l[1::])
```

Exercice 2 Proposez d'autres variantes récursives de cette fonction

1.4 Mise en application (20 minutes max)

Exercice 3 Ecrire pour chaque question, une fonction impérative d'une part, et une fonction récursive d'autre part, qui :

1. calcule la factorielle d'un entier passé en argument
2. reçoit en argument un entier naturel n non nul, et retourne la valeur de $\sum_{k=1}^{n} \frac{1}{k}$
3. calcule la somme des termes d'une liste de nombres passée en argument
4. reçoit un entier n et renvoie une liste contenant $0^2, 1^2, \cdots, n^2$
5. calcule le maximum d'une liste de flottants passée en argument
6. prend en argument la liste l et x, et retourne le nombre d'occurences de x dans l
7. prend en argument la chaine de caractères ch et le caractère c, et retourne le nombre d'occurences de c dans ch
8. décide si l'élément c appartient à la liste l
9. décide si la lettre (le caractère) c appartient à la chaine de caractère ch
10. calcule le pgcd de deux entiers a et b donnés en argument
11. prend en argument un entier naturel n non nul et retourne les deux entiers naturels p et q tels que $n = 2^p \times q$ avec q impair.

12. prend en argument une liste et renvoie la liste miroir
13. (plus difficile) prend en argument un entier n et retourne toutes les listes à n éléments formées de 0 et de 1. Le résultat sera une liste de ces listes. (Vous pourrez utiliser une boucle for)
14. calcule récursivement le coefficient binomial $\binom{n}{p}$, avec p et n entiers naturels
15. $\star$ calcule récursivement le coefficient binomial $\binom{n}{p}$ **en optimisant les calculs**

Exercice 4 Ecrire une fonction etoile_croissant_rec telle que l'appel etoile_croissant_rec (n) écrit : *
**
***
.....
*******...***
où la dernière ligne comporte n symboles *
Ecrire de même etoile_decroissant_rec .
La commande `'*'*k` est interdite bien sûr, on autorise seulement `chaine+'*'`.

1.5 Suites récurrentes : récursivité croisée, conservation des termes précédents

Exercice 5 Programmer en Python une fonction qui calcule le n eme terme de la/ des suites données. On proposera un programme impératif et un programme récursif :

1. $u_0 = 1, u_{n+1} = \sqrt{1+u_n}$
2. $u_0 = 1, u_1 = 1, u_{n+2} = u_{n+1} + u_n$
3. $u_0 = 3, v_0 = 5, u_{n+1} = \dfrac{u_n + v_n}{2}, v_{n+1} = \sqrt{u_n v_n}$
4. $u_0 = 1, u_n = \dfrac{1}{n}\sum_{k=0}^{n-1} \sin(u_k)$ (fonction sum de python autorisée)
5. $\star$ optimiser le code de la question précédente de façon à n'avoir aucun temps d'attente pour $n = 50$

Remarque
Dans certaines fonctions, on ne programme pas directement f récursivement, mais on créée une fonction auxiliaire récursive plus générale (parfois à l'intérieur de f), et on demande à f d'appeler cette fonction.
Analogie avec un principe que vous avez sans doute déjà rencontré en mathématiques : *lorsqu'une récurrence ne fonctionne pas, essayez de la renforcer.*
C'est fréquemment le cas pour optimiser un code qui sinon prendrait trop de temps à terminer : on a vu cela dans la dernière question de l'exercice précédent, et on le retrouve dans la partie suivante.

Exercice 6
1. Reprendre la suite définie par $u_0 = 1, u_1 = 1, u_{n+2} = u_{n+1} + u_n$ de l'exercice précédent : proposer une seconde fonction récursive qui calcule le n eme terme de la suite **sans calculs superflus**.
2. Comparer le temps calcul de u_{30} ou de u_{50} par les deux fonctions.
3. (5/2) : Tracer la courbe « temps de calcul en fonction de n » pour chacune des deux fonctions.
   Indication : regarder la documentation de la bibliothèque **time** sur internet.
4. Reprendre la question 2 en introduisant un dictionnaire.

2 TD : Recherche par dichotomie dans une liste triée

Exercice 7 On dispose d'une liste de nombres **triée par ordre croissant** et on se demande si le nombre x apparaît dans la liste.
Une méthode est de rechercher par dichotomie dans la liste : on compare x à l'élément a du milieu de la liste, si $x = a$ on a terminé, et sinon, selon que $x < a$ ou que $x > a$ on va rechercher x dans une des deux moitiés de la liste.

1. (PCSI) Ecrire une fonction Python **impérative** qui prend en argument une liste triée l et un élément x et teste si x apparait dans l.
2. Ecrire une fonction Python **récursive** qui prend en argument une liste triée l et un élément x et teste si x apparait dans l.
3. Dans la question précédente, votre code est-il optimisé en terme de cout temporel ? (NB : recopier une liste prend du temps...). Proposer un code plus efficace.

3 TD : Tours de Hanoï

Exposé du jeu Ce jeu des tours de Hanoï est constitué de trois tiges ; sur la première d'entre elles sont enfilés n disques de diamètres différents. Au début du jeu, ces disques sont tous positionnés sur la première tige du plus grand au plus petit. L'objectif est de déplacer tous ces disques sur la troisième tige, en respectant les règles suivantes :

1. un seul disque peut être déplacé à la fois
2. on ne peut jamais poser un disque sur un disque de diamètre inférieur.

Comment faire ?

Solution Raisonnons par récurrence : pour pouvoir déplacer le dernier disque, il est nécessaire de déplacer les $n-1$ disques qui le couvrent sur la tige centrale. Une fois ces déplacements effectués, nous pouvons déplacer le dernier disque sur la troisième tige. Il reste alors à déplacer les $n-1$ autres disques vers la troisième tige.

Exercice 8 Implémenter une fonction Python qui reçoit en entrée n le nombre de disques du jeu, et écrit en sortie une liste de déplacements permettant de terminer le jeu.

4 Coût temporel et spatial d'une fonction récursive

4.1 Pile d'appels récursifs

L'ordinateur « mémorise » l'ensemble des appels récursifs effectués au moyen d'une pile dans laquelle il stocke toutes les informations intermédiaires. Lorsqu'il arrive à un cas de base (où l'on sait calculer f sans appel récursif), l'ordinateur redescend toute la pile.

4.2 Pour aller plus loin : cout temporel : relation de récurrence

Le coût d'une fonction récursive obéit à une **relation de récurrence** qu'il s'agit de résoudre.
Ainsi :

1. pour la somme des éléments d'une liste : $C_{n+1} = O(1) + C_n$ donc $C_n = O(n)$. On traitera $O(1)$ comme un constante dans la résolution de la récurrence.
2. Pour la recherche par dichotomie dans une liste triée : $C_n = \mathrm{O}(1) + C_{\lfloor \frac{n}{2} \rfloor}$ On sait alors montrer que $C_n = \mathrm{O}(\ln n)$. Vous pouvez rendre cela (un peu) intuitif en prenant une puissance de 2 : $C_{2^m} = \mathrm{O}(1) + C_{2^{m-1}}$, de sorte que la suite de terme général $u_m = C_{2^m}$ est arithmétique donc $C_{2^m} \approx C^{\text{ste}} m$
Ainsi $C_n \approx C^{\text{ste}} \log_2(n) = \mathrm{O}(\ln n)$.

5 A retenir

1. Savoir expliquer ce qu'est une fonction récursive, sa condition de terminaison.
2. Connaître problèmes la récursivité permet de résoudre facilement.
3. Les deux pièges de la récursivité : les couts cachés liés aux appels récursifs multiples et à la copie de listes. La gestion de l'espace mémoire.

6 Exercices

Exercice 9 Exponentiation rapide
Soit la fonction :

```
def expo_rapide(x,n):
    p = 1 # p comme produit
    a = x
    e = n # e comme exposant
    while e != 0:
        if e % 2 == 1:
            p = p*a
            e = e - 1
        else:
            e = e // 2
            a = a * a
    return(p)
```

1. Rappelez son cahier des charges et le principe de fonctionnement
2. Donnez en une version récursive.

Exercice 10 Appartenance dans une liste imbriquée
On considère des listes contenant des entiers et d'autres listes, elle-mêmes formées de listes contenant des entiers et d'autres listes, elle-mêmes formées de listes contenant des entiers et d'autres listes, etc.
Ecrire une fonction `appartient_imbrique(l,a)` qui teste si l'entier a apparaît à un certain niveau de profondeur dans l.
Par exemple `appartient_imbrique([[[2],[3,4]],[12,[1,3]],3,[],2], 1)` renverra **True**.
Rappel : `isinstance(1, int)` renvoie True, `isinstance(1, list)` retourne False et `isinstance([1], list)` renvoie True.
⋆ Ecrire une fonction de même spécification, mais ne comportant ni récursivité ni appel à `copy` ou à `deepcopy`.

Exercice 11 Palindrome Ecrire une fonction qui teste si une liste est bien un palindrome

Exercice 12 Fractales
1. Googliser « sierpinski triangle » sur google images
2. Ecrivez alors une fonction Python qui génère les images que vous voyez dans les résultats. Vous vous familiariserez auparavant avec la bibliothèque turtle.

Exercice 13 Revoir l'ensemble des parties d'un ensemble (cf maths) avant de traiter cet exercice On se donne une liste d'éléments distincts représentant l'ensemble cotenant ces éléments.
Donner un fonction donnant la liste des parties de cet ensemble.

7 Solutions

Solution

Solution 1.2 Code recursif

```
def somme_des_inverses_recursif(n):
    if n==1:
        return 1
    else:
        return 1/n+ somme_des_inverses_recursif(n-1)
```

Solution

Solution 1.3 Par exemple :

```

def somme_elements_liste_recursive2(l):
    if l==[]:
        return 0
```

```
    else:
        return l[0]+somme_elements_liste_recursive2(l[1::])


def somme_elements_liste_recursive3(l):
    if l==[]:
        return 0
    else:
        return somme_elements_liste_recursive3(l[:-1])+l[-1]
```

Solution

Solution 1.4

```

def factorielle_imperatif(n):
    facto=1
    for k in range(1,n+1):
        facto=facto*k
    return facto

def factorielle_recursif(n):
    if n==0:
        return 1
    else:
        return n*factorielle_recursif(n-1)

def somme_des_inverses_rec(n):
    if n==1:
        return 1
    else:
        return 1/n+ somme_des_inverses_rec(n-1)


def somme_elements_liste(l):
    S=0
    for x in l:
        S+=x
    return S

def somme_elements_liste_rec(l):
    if len(l)==0:
        return 0
    else:
        return l[0]+somme_elements_liste_rec(l[1::])




def liste_des_carres(n):
    l=[]
    for k in range(n+1):
        l.append(k**2)
    return l


def liste_des_carres_rec(n):
    if n==0:
        return [0]
    else:
        l=liste_des_carres_rec(n-1)
        l.append(n**2)
        return l


def adjoindre(l,a):
    return l.append(a)




def maximum(l):
    M=l[0]
    for x in l:
        M=max(M,x)
    return M
```

```python


def maximum_rec(l):
    if len(l)==1:
        return l[0]
    else:
        return max(l[0], maximum_rec(l[1::]) )


def nombre_occurrences(x,l):
    compte=0
    for y in l:
        if y==x:
            compte+=1
    return compte


def nombre_occurrences_rec(x,l):
    if len(l)==0:
        return 0
    else:
        Nombre_occ_restantes=nombre_occurrences_rec(x, l[1::])
        if l[0]==x:
            return Nombre_occ_restantes+1
        else:
            return Nombre_occ_restantes


def pgcd_imperatif(a,b):
    a,b=abs(a), abs(b)
    if 0 in (a,b):
        return max(a,b)
    if  a>b:
        a,b=b,a #on a b>= a desormais
    while a>0:
        a,b=b%a,a
    return b


def pgcd(a,b):
    if a<0 or b<0:
        return pgcd(abs(a), abs(b))
    if a==0 or b==0 or a==b:
        return max(a,b)
    elif a>b:
        return pgcd(a-b,b)
    else:
        return pgcd(a,b-a)



def appartient(l,c):
    for x in l:
        if x==c:
            return True
    return False


def appartient_rec(l,c):
    if len(l)==0:
        return False
    else:
        return (l[0]==c) or appartient_rec(l[1::],c)



def miroir(l):
    ll=[]
    n=len(l)
    for i in range(n-1,-1,-1):
        ll.append(l[i])
    return ll


def miroir_rec(l):
    if len(l)<=1:
        l2=l
```

```
    else:
        l2=miroir_rec(l[1::])
        l2.append(l[0])
    return l2


def decomposition2pq_rec(n):
    if n%2==1:
        return (0,n)
    else :
        (p,q)= decomposition2pq_rec(n//2)
        return (p+1, q)



def binomial_rec(n,p):
    if n<p:
        return 0
    if n==p:
        return 1
    if p==0:
        return 1
    if n==0: ###auqeul cas p est <0
        return 0
    else:
        return binomial_rec(n-1,p)+binomial_rec(n-1,p-1)

# cette solution n'est pas du tout optimale
#si on veut ne pas de faire plusieurs fois certains calculs, il faut stocker les resultats
#une premiere solution est de construire entierement le triangle de pascal (NON FAIT ICI)
#une deuxieme est de ne construire que la partie du triangle qui nous est utile:
    dictionnaire dont les clefs
#sont les couples (n,p)


# Solution avec dictionnaire exterieur
dico={}
def binom(n,p):
    if (n,p) in dico:
        return dico[(n,p)]
    if n==p or p==0:
        return 1
    x=binom(n-1,p-1)
    y=binom(n-1,p)
    b=x+y
    dico[(n,p)]=b
    return b

#Dictionnaire en variable locale, fonction auxiliaire recursive
def binom_dico(a,b):
    dico={}
    def remplir_dico(n,p):
        if (n,p) not in dico:
            if n<p:
                dico[(n,p)]=0
            elif n==p:
                dico[(n,p)]=1
            elif p==0:
                dico[(n,p)]=1
            elif n==0: ###auqeul cas p<0
                dico[(n,p)]=0
            else:
                remplir_dico(n-1,p-1)
                remplir_dico(n-1,p)
                dico[(n,p)]=dico[(n-1,p-1)]+dico[(n-1,p)]
    remplir_dico(a,b)
    return dico[(a,b)]



def liste_0_1_rec(n):
    if n==0:
        return []
    if n==1:
        return [[0],[1]]
    else:
```

```
        L_n=[]
        L_n_moins_un=liste_0_1_rec(n-1)
        for l in L_n_moins_un:
            L_n.append(l+[0])
            L_n.append(l+[1])
        return L_n


##version recursive plus efficace

def decomposition2pq_rec_opti(n):
    def auxiliaire_decomp_rec(p,q):
        if q\%2==1:
            return p,q
        else:
            return auxiliaire_decomp_rec(p+1,q//2)
    return auxiliaire_decomp_rec(0,n)


def fact_opti(n):
    def aux_rec(accu,depart):
        if depart==1:
            return accu
        else:
            return aux_rec(accu*depart,depart-1)
    if n==0:
        return 1
    else:
        return aux_rec(1,n)

### avec un dictionnaire:
```

Solution

Solution 1.5

```

def suite1_rec(n):
    if n==0:
        return 1
    else:
        return sqrt(1+suite1_rec(n-1))




def suite2_rec(n):
    if n==0:
        return 1
    if n==1:
        return 1
    else:
        return suite2_rec(n-2)+suite2_rec(n-1)


def suite3_u_rec(n):
    if n==0:
        return 3
    else:
        return (suite3_u_rec(n-1)+suite3_v_rec(n-1))/2
def suite3_v_rec(n):
    if n==0:
        return 5
    else:
        return sqrt(suite3_u_rec(n-1)*suite3_v_rec(n-1))
#ou bien:
def suite3_uv_rec(n):
    if n==0:
        return 3,5
    else:
        a,b=suite3_uv_rec(n-1)
        return (a+b)/2, sqrt(a*b)

def suite4_rec(n):
    if n==0:
        return 1
```

```
    else:
        return sum([sin(suite4_rec(k)) for k in range(n)])/n
# l'execution de ce dernier code prend trop de temps ! En effet on recalcule  plein de fois les memes valeurs.
#Mieux vaut stocker les valeurs dans une liste et les appeler


def suite4_optimise(n):
    def aux_rec(liste_des_termes,nombre_iter_restant):  ## auxiliaire recursif: met a jour la liste des termes de la suite
        if nombre_iter_restant==0:
            return liste_des_termes
        else:
            nouveau_terme=sum([sin(a) for a in liste_des_termes])/len(liste_des_termes)
            return aux_rec(liste_des_termes+[nouveau_terme], nombre_iter_restant-1)
    return aux_rec([1],n)[-1]
```

Solution

Solution 1.5 Question 2 :

```

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

Question 4 :

```
#avec un dictionnaire
def fib_avec_dico(n,a,b):
    dico={0:a, 1:b}
    def remplir_dico(m):
        if m not in dico:
            remplir_dico(m-2)
            remplir_dico(m-1)
            dico[m]=dico[m-2]+dico[m-1]
    remplir_dico(n)
    return dico[n]
```

Solution

Solution 2

```

def recherche_dicho_imperatif(x, l):
    i, j = 0,len(l)
    while i < j:
        k = (i + j) // 2
        if l[k] == x:
            return True
        elif l[k] > x:
            j = k
        else:
            i = k + 1
    return False


# ce code n'est pas optimal en terme de cout !!!
def dicho_rec(x,l):
    if l==[]:
        return false
    else:
        m=len(l)//2
        a=l[m]
        if x==a:
            return True
        elif x<a:
            return dicho_rec(x,l[0:m]) # cout qui est cache: recopie les listes
        else:
            return dicho_rec(x,l[m+1::]) #cout cache: recopie les listes
```

```python


#un code meilleur: on travaille sur les indices du segment cible, mais on ne modifie pas la liste l

def recherche_dicho_rec(x,l):
    def aux_rec(i,j):
        if i>=j:
            return False
        m=(i+j)//2
        a=l[m]
        if x==a:
            return True
        elif x<a:
            return aux_rec(i,m)
        else:
            return aux_rec(m+1,j)

    return aux_rec(0, len(l))


#code meilleur et sans fonction auxiliaire
def recherche_dicho_rec_2(x,l,i=0, j=-1):
    if j==-1:
        j = len(l)
    if i>=j:
        return False
    m=(i+j)//2
    a=l[m]
    if x==a:
        return True
    elif x<a:
        return recherche_dicho_rec_2(x,l,i,m)
    else:
        return recherche_dicho_rec_2(x,l,m+1,j)
```

Solution

Solution 3

```python

def hanoi(depart, arrivee, n):
    if n==1:
        return [(1,depart, arrivee)]
    else:
        intermediaire=6-depart-arrivee
        deplacement_vers_intermediaire=hanoi(depart, intermediaire,n-1)
        deplacement_gros_disque=[(n,depart,arrivee)]
        deplacement_vers_arrivee=hanoi(intermediaire,arrivee,n-1)
        return
            deplacement_vers_intermediaire+deplacement_gros_disque+deplacement_vers_arrivee

def solution_hanoi(n):
    deplacement=hanoi(1,3,n)
    for x in deplacement:
        print('deplacer le disque num '+str(x[0])+' de la tige '+str(x[1])+' vers la tige
            '+str(x[2]))
```

Solution

Solution 6 Cdc : calcul de x^n avec une complexité en $O(\ln n)$.

```python

def expo_rapide_rec(x,n):
    if n==0:
        return 1
    if n==1:
        return x
    if n%2==0:
        return expo_rapide_rec(x**2,n//2)
    else:
        return x*expo_rapide_rec(x**2,n//2)
```

Solution

Solution 6 La récursivité rend cette fonction facile à programmer :

```
1
2  def appartient_imbrique(liste,a):
3      if liste==[]:
4          return False
5      x=liste[-1]
6      if x==a:
7          return True
8      elif isinstance(x,int):
9          return appartient_imbrique(liste[0:-1],a)
10     else:
11         return appartient_imbrique(x,a) or appartient_imbrique(liste[0:-1],a)
12
13
14 def appartient_imbrique_sans_rec(l,a):
15     tout_est_entier=False
16     Lold=l
17     while not(tout_est_entier):
18         tout_est_entier=True
19         n=len(Lold)
20         Lnew=[]
21         for i in range(n):
22             if Lold[i]==a:
23                 return True
24             elif isinstance(Lold[i], list):
25                 tout_est_entier=False
26                 Lnew+=Lold[i]
27         print(Lold,Lnew) #facultatif: permet de visualiser le travail du programme
28         Lold=Lnew
29     return False
```

Solution

Solution 6

```
1
2  from turtle import *
3
4  def tapis(s1, s2, s3,n,bit=True):
5      if n>0:
6          triangle(s1,s2,s3)
7          a,b,c=(s1+s2)/2, (s2+s3)/2, (s1+s3)/2 #calcul des milieu. np.array
8          tapis(a,b,s2,n-1, not(bit))
9          tapis(b,c,s3, n-1,not(bit))
10         tapis(a,c,s1, n-1,not(bit))
11
12
13 def triangle(a,b,c,bit=True): #tracer le triangle
14     up()
15     goto(a[0],a[1])
16     down()
17     pencolor('blue')
18     goto(b[0],b[1])
19     pencolor('red')
20     goto(c[0],c[1])
21     pencolor('green')
22     goto(a[0],a[1])
23     up()
```

Solution

Solution 6

```
1
2  def parties(l):
3      if l==[]:
4          return [[]]
5      else:
6          n=l[-1]
7          parties_sans_n=parties(l[:-1])
8          parties_avec_n=[p+[n]  for p in parties_sans_n]
9          return parties_sans_n+parties_avec_n
```