

PROGRAMMES RECURSIFS

Une procédure est récursive lorsqu'elle s'appelle elle même.

Il faut que la procédure ne s'appelle pas indéfiniment ,il faut donc un cas d'arrêt et être sûr qu'on y arrive , être sûr qu'on descend strictement. Le nombre de fois où la procédure s'appelle est limité avec python c'est 1000.

Il faut penser à une récurrence quand on fait un programme récursif :voir comment construire le niveau supérieur avec le niveau strictement inférieur .

Quand on utilise le niveau inférieur , quand on appelle le programme avec un n strictement inférieur, ,ou une variable strictement « inférieur » considérer que le programme sait faire et donne le résultat , ne plus penser à la récursivité,

Comme dans une récurrence il faut initialiser c'est-à-dire faire le ou les cas d'arrêt (Les cas $n=1$ ou $n=2$)

exemple 1: calcul de $n!$

```
def fact(n):  
    if n==1:  
        return (1)  
    if n>1 :  
        return(n*fact(n-1))
```

Détailler ce qui se passe avec $n=3$;

exemple 2 : suite de Fibonacci avec $u_0=u_1=1$

```
def f(n)  
    if n=0 or n=1 :  
        return(1)  
    if n >1 :  
        return(f(n-1)+f(n-2))
```

Un programme récursif peut être transformé en programme avec des boucles.

Les exemples 1 et surtout 2 sont de mauvais programmes récursifs car beaucoup plus lent en vitesse d'exécution et prenant plus de mémoires qu'une simple boucle ; de plus le n est limité à 1000, ce qui n'est pas le cas avec une boucle ;

l'intérêt d'un programme récursif est sa " simplicité " de programmation par rapport à un programme avec des boucles .

exemple 3 : la procédure renvoie l'écriture en base b ,la suite des chiffres en base b

```
def baserec(n,b):
```

```
    if n==0 :  
        s=""  
    if n==1:  
        s='1'  
  
    if n>1 :  
        r=n % b
```

```

q=n//b
s=baserec(q,b)
s=s+str(r)
return(s)

```

version liste :

Attention : avec `append` , `sort` , `remove` , `reverse` ,... pas d'affectation , pas de `return(s.append(x))` `s.append(x)` rajoute `x` à `s` mais `y=s.append(x)` renvoie pour `y` `none` (essayer `print(s.append(x))` ou `type`) et non pas : la liste `s + x`
on fera `s.append(x)` puis après `return(x)` ou `y=x`

```

def ba(n,b):

    if n==0 :
        s=[]

    if n==1:
        s=[1]

    if n>1 :
        r=n % b
        q=n//b
        s=ba(q,b)
        s.append(r) (surtout pas s=s;append(r))
    return(s)

```

Le même programme sans récursivité:

```

def f(n,b):
    m=n
    s=[]
    while m >0 :
        r=m%b
        m= m//b
        s.append(r)
    s.reverse()
    return(s)

```

EXERCICES SIMPLES

a) pgcd de a et b

si $a=b$ c'est a ; si $a>b$ c'est $\text{pgcd}(a-b,b)$; si $b>a$: $\text{pgcd}(b-a,a)$

b) $T_0=1$ $T_1=x$ et $T_n=2xT_{n-1}-T_{n-2}$ procédure $T(n,x)$

c) Pour multiplier deux entiers on peut utiliser que l'addition, multiplication et division par 2 : si x pair $xy = (x/2) (2y)$ et si x impair $xy = ((x-1)/2) (2y) + y$
Ecrire le programme qui entre x et y et sort xy par cette méthode

d) On entre une liste $u = [u[0], \dots, u[n-1]]$ on sort $S(u) = \sum_{0 \leq i < j \leq n-1} u_i u_j$

On utilisera la récurrence $S(u) = (u[1] + \dots + u[n-1])u[n] + S(u')$ où u' ne contient pas $u[n]$

Calculer S sans récursivité avec deux boucles for

e) Ecrire un programme récursif, puis non récursif pour savoir si une chaîne de caractères est un palindrome c'est à dire peut se lire dans les deux sens comme '121', 'abba'

f) Ecrire un programme récursif qui voit si deux listes sont des anagrammes comme rage [r,a,g,e] et [g,a,r,e] ; il s'agit de voir s'il y a les mêmes lettres le même nombre de fois ; on trouve une lettre et on recommence en l'éliminant de chaque côté ; ainsi par récursivité on n'a à traiter qu'une lettre.

`list.pop(i)` renvoie `list[i]` et supprime `L[i]`

on peut aussi essayer de travailler sur des chaînes de caractères ; niche et chien sont deux anagrammes

g) * Ecrire un programme récursif pour trouver le maximum d'une liste
par la récurrence $\max(L, a) = \max(\max(L), a)$

* Ecrire un programme qui cherche la (une) place, indice du maximum d'une liste L et qui le permute avec le dernier élément de la liste L.

***Ecrire un programme récursif de tri par sélection** : on cherche le maximum qu'on place à la fin et on recommence (récursivité) avec la liste sans le dernier terme.
Rappelons que `L[a:b]` donne la liste des éléments de `L[a]` à `L[b-1]`

h) programmer par récursivité x^n en utilisant que $x^{2n} = (x^n)^2$ et $x^{2n+1} = x(x^n)^2$

Voici deux exemples difficiles sans récursivité

exemple 4: tous les mots de lettres 0 ou 1 :

def suite01(n) :

 # renvoie la liste des mots de n lettres de 0 ou 1

 if n==1 :

 return(['0','1'])

 if n>1 :

 so=suite01(n-1)

 s1=[]

 for x in so :

 y=x+str(0)

 s1.append(y)

 for z in so:

 y=z+str(1)

 s1.append(y)

```
return(s1)
```

Version liste :

```
def suite01(n) :  
    # renvoie la liste de listes de n lettres de 0 ou 1  
    if n==1 :  
        return([[0],[1]])  
    if n>1 :  
        so=suite01(n-1)  
        s1=[]  
        s2=[]  
        for x in so :  
            y=x[:] # fait une copie si y=x lorsqu'on modifie y, x est aussi modifié  
            y.append(0)  
            s1.append(y)  
  
            for y in so :  
                z=y[:]  
                z.append(1)  
                s2.append(z)  
  
        s1.extend(s2)  
        return(s1)
```

exemple 5 :

pour insérer x dans L à la place i : L.insert(i,x)

```
permut:=proc(n)  
    # renvoie la liste des permutations 1,2,..., n  
def permut(n):  
    if n==1:  
        return([1])  
    if n==2:  
        return([1,2],[2,1])  
    if n>1 :  
        l1=permut(n-1)  
        s1=[]  
        for x in l1:  
            for i in range(n):  
                y=x[0:len(x)] # pour travailler sur une copie ; ou bien y=list(x)  
                y.insert(i,n)  
                s1.append(y)  
        return(s1)
```

EXERCICES

1/Ecrire un programme où on entre n et il sort une permutation au hasard de [1,2,...,n]

On formera toutes les permutations de longueur n ! et on en choisira une au hasard soit un numéro entre 1 et n ! `randint(1,n !)` ou `random.randint(1,n !)`

Puis répéter N fois l'opération et vérifier que la moyenne du nombre de points fixes est 1 (convergence lente, on fixera n et on ne refera pas réexécuter le programme doannat toutes les permutations)

On aura écrit un programme qui compte le nombre de points fixes d'une permutation p .

2/ Etant donné une liste u d'entiers le programme forme l'ensemble (set) de toutes les sommes possibles d'éléments de u en prenant au plus une fois chaque élément de la liste u ; si $u=[a,b,c]$ la sortie est $\{a,b,c,a+b,a+c,b+c,a+b+c\}$; on utilisera `union`

3/

Pour dessiner il faut utiliser `pyplot` du module `matplotlib`

```
import matplotlib.pyplot as plt
```

Pour dessiner des segments allant de (a,b) à (c,d) puis (e,f) (`plot` relie les points)

on écrit `plt.plot([a,c,e] , [b,d,f] , color= 'red')`

la liste 1 est pour les abscisses x et la liste 2 pour les ordonnées

puis `plt.axis('equal')` et `plt.show()`

On veut dessiner les branches d'un arbre à n étages .



$n=1$



$n=2$



$n=3$

A chaque fois la longueur des branches diminue $1/3$ ou de moitié ; la pente est $\pi/4$.

Ecrire une procédure récursive `arbre(a,b,k,n)` qui donne $r=[x,y]$ où x est la liste des abscisses et y celles des ordonnées en partant de (a,b) , les deux premières branches ont une longueur de $\sqrt{2}k$ (on va de (a,b) à $(a+k,b+k)$) et il y a n étages

Pour récupérer x on écrit `r[0]`

Pour faire le programme récursif il faut voir comment faire l'arbre n avec $n-1$: il y a les 2 branches basses puis 2 arbres $A1$ et $A2$ d'origine distincts c'est pourquoi on a généraliser le problème en rajoutant l'origine (a,b)



Comme `plot` relie les points il faudra revenir à l'origine (a,b) pour ne pas avoir de segment qui ne soit pas une branche ; pour $n=1$ bien revenir en (a,b)

On fera les abscisses de $A1$ (qui revient à son origine) puis les 2 branches basses sans revenir à l'origine $(a-k/3,b+k/3)$, (a,b) , $(a+k/3,b)$, puis les x de l'arbre l'arbre $A2$

$A1$ partant de $(a-k/3,b+k/3)$, avec $k/2$ et $n-1$ branches ; et l'arbre $A2$ partant de $(a+k/3,b+k/3)$

Si on préfère diviser par 2 on prend des $k/2$

Et de même avec les y

On ajoute 2 listes par `list1.extend(list2)`

Une fois le programme récursif fait on a [x,y]

On écrira `plt.plot(arbre(0,0,1,n)[0],arbre(0,0,1,n)[1])` , `plt.axis('equal')` et `plt.show()`

Solution de l'arbre

```
import matplotlib.pyplot as plt
```

```
def arbre(a,b,k,n):
```

```
    if n==1:
```

```
        x=[a-k,a,a+k,a]
```

```
        y=[b+k,b,b+k,b]
```

```
        return([x,y])
```

```
    if n>1:
```

```
        x0=[a-k/3,a,a+k/3]
```

```
        y0=[b+k/3,b,b+k/3]
```

```
        xa1=arbre(a-k/3,b+k/3,k/3,n-1)[0]
```

```
        xa1.extend(x0)
```

```
        xa2=arbre(a+k/3,b+k/3,k/3,n-1)[0]
```

```
        xa1.extend(xa2)
```

```
        ya1=arbre(a-k/3,b+k/3,k/3,n-1)[1]
```

```
        ya2=arbre(a+k/3,b+k/3,k/3,n-1)[1]
```

```
        ya1.extend(y0)
```

```
        ya1.extend(ya2)
```

```
        return([xa1,ya1])
```

```
plt.plot(arbre(0,0,1,4)[0],arbre(0,0,1,4)[1])
```

```
plt.show()
```