Informatique Python TD6

Exemples d'algorithmes récursifs

1. La récursivité

1.1. Présentation

On appelle fonction récursive une fonction qui comporte un appel à elle-même. Une fonction récursive doit respecter les conditions suivantes :

- Une fonction récursive contient un cas de base (exemple : `if n==0: return...`)
- Une fonction récursive doit modifier son état pour se ramener au cas de base au bout d'un nombre fini d'étapes.
- Une fonction récursive doit s'appeler elle-même.

Une fonction mathématique définie par une relation de récurrence (et une condition initiale), peut de façon naturelle être programmée de manière récursive.

Exemple : La factorielle d'un entier est définie selon : $n! = \begin{cases} 1 \; si \; n = 0 \\ n(n-1)! \; si \; n \geq 1 \end{cases}$

Le script suivant insère des instructions `print` qui permettent de bien comprendre l'exécution d'une fonction récursive :

```
 1 def fact(n):
 2     if n==0:
 3         return 1
 4     else:
 5         print('-'*(n-1)+'-> appel de fact('+str(n-1)+')')
 6         y=n*fact(n-1)
 7         print('='*(n-1)+'=> reponse de fact('+str(n-1)+')')
 8     return y
 9
10 print(fact(4))
```

Son exécution donne :

```
----> appel de fact(3)
---> appel de fact(2)
--> appel de fact(1)
-> appel de fact(0)
=> reponse de fact(0)
==> reponse de fact(1)
===> reponse de fact(2)
====> reponse de fact(3)
24
```

La machine applique la règle de la récurrence $n! = n(n-1)! \; si \; n \geq 1$ (ligne 6) tant que l'entier est différent de 0, ce qui introduit des calculs intermédiaires jusqu'à aboutir au cas de base $n = 0$. Les calculs en suspens sont alors achevés dans l'ordre inverse jusqu'à obtenir le résultat final. Les différents calculs sont stockés dans une pile d'appel, le dernier calcul appelé est le premier exécuté.

1.2. Complexité

On note K_n la complexité de la fonction `fact(n)` en nombre d'opérations effectuées. Alors :

$$K_n = K_{n-1} + 1 \text{ avec } K_0 = 0$$

soit $K_n = n$

La complexité de la fonction est donc linéaire en $O(n)$ (K_n < constante * n).

2. Suite de Fibonacci

2.1. Présentation

La suite de Fibonacci est définie comme suit : $u_n = \begin{cases} 1 \; si \; n = 0 \\ 1 \; si \; n = 1 \\ u_{n-1} + u_{n-2} \; \; sinon \end{cases}$

2.2. Codage

Récursif

```
def fibo(n):
    if n==0 or n==1:
        return 1
    else:
        return fibo(n-1)+fibo(n-2)
```

Itératif

```
def fibo_it(n):
    u,v=1,1
    for i in range(n-1):
        u,v=v,u+v
    return v
```

Voici les résultats pour quelques valeurs de *n* :

```
fibo(10)= 89 en :  6.508827209472656e-05  sec
fibo_it(10)= 89 en :  4.0531158447265625e-06  sec
fibo(20)= 10946 en :  0.0029120445251464844  sec
fibo_it(20)= 10946 en :  3.814697265625e-06  sec
fibo(30)= 1346269 en :  0.3621852397918701 sec
fibo_it(30)= 1346269 en :  5.0067901611328125e-06  sec
fibo(40)= 165580141 en :  41.65290021896362 sec
fibo_it(40)= 165580141 en :  5.9604644775390625e-06  sec
```

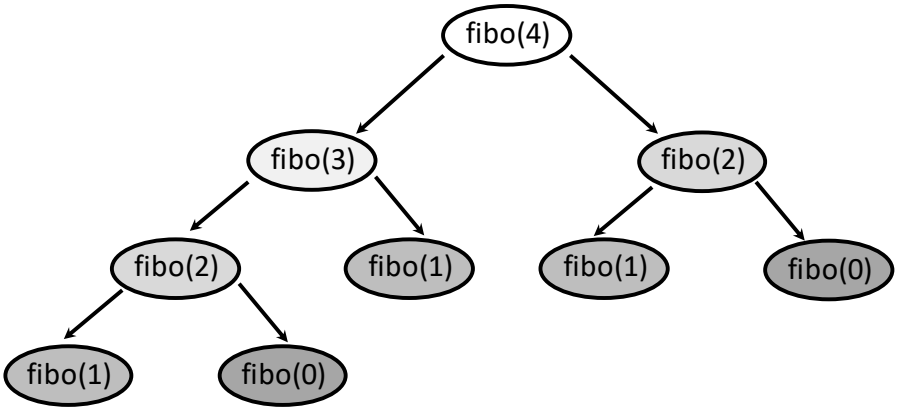
À partir du script :

```
import time
for n in range(10,50,10): #de 10 inclus à 50 exclu avec un pas de 10
    t0=time.time()
    f_rec=fibo(n)
    t1=time.time()
    print("fibo("+str(n)+")=",f_rec,"en : ",(t1-t0)," sec")

    t0=time.time()
    f_it=fibo_it(n)
    t1=time.time()
    print("fibo_it("+str(n)+")=",f_it,"en : ",(t1-t0)," sec")
```

Évidemment les durées obtenues dépendent du processus mais les résultats sont clairs et montrent un problème avec la fonction récursive.

La fonction récursive est une traduction de la relation de récurrence de la suite en partant du « haut », néanmoins elle n'est pas efficace en pratique. Pour s'en convaincre, il suffit de représenter sous forme d'arbre binaire les calculs nécessaires à l'obtention de `fibo(n)`. On donne l'exemple de `n = 4` :



Hormis les feuilles (`fibo(0)` et `fibo(1)` égales à 1 d'après la définition), le calcul de chaque nœud demande l'addition de deux sous-arbres. Certains sous-arbres identiques apparaissent rapidement à plusieurs reprises ce qui provoque un gaspillage de ressources considérable avec l'augmentation du rang du terme à calculer.
L'application de la relation de récurrence en partant du « bas » correspond au chemin tout à gauche dans cet arbre. Celui-ci passant par un nombre restreint de nœuds il y aura une grande économie de calculs.
L'exécution d'une fonction récursive se représente naturellement sous forme d'arbre comme le montre la commande `fibo(4)`. Le nombre moyen de branches issues de chaque nœud est appelé facteur de branchement de la fonction. Le facteur de branchement de `fact` est égal à 1 alors que celui de `fibo` est égal à $\lambda = \left(1+\sqrt{5}\right)/2$ voir section (2.3)

2.3. Complexité (hors programme réservé aux curieux)

On note K_n la complexité de la fonction `fibo(n)` en nombre d'additions effectuées.
Alors :

$$K_n = K_{n-1} + K_{n-2} + 1 \text{ avec } K_0 = 0 \text{ et } K_1 = 0$$

soit $\left(K_n + 1\right) = \left(K_{n-1} + 1\right) + \left(K_{n-2} + 1\right)$ ou $K'_n = K'_{n-1} + K'_{n-2}$ en posant $K'_n = K_n + 1$
Il apparaît alors une suite récurrente linéaire d'ordre 2 dont la résolution donne :

$$K'_n = A\left(\frac{1+\sqrt{5}}{2}\right)^n + B\left(\frac{1-\sqrt{5}}{2}\right)^n \text{ avec A et B dans R}$$

La complexité de la fonction est donc exponentielle en $O\left(\lambda^n\right)$ (avec $\lambda = \left(1+\sqrt{5}\right)/2$, le nombre d'or), alors qu'elle est linéaire pour la version itérative. Dans ce cas l'algorithme itératif est donc beaucoup plus efficace que l'algorithme récursif

3. Conclusion

La mise en œuvre des algorithmes récursifs nécessite le plus souvent une pile (complexité en espace). C'est la difficulté d'implanter cette pile ou d'éviter son emploi qui a fait dire pendant longtemps que les programmes récursifs étaient moins efficaces que les programmes itératifs, mais la situation a changé. En fait, le débat sur le choix entre codage récursif ou itératif est aussi vieux que l'informatique et les progrès de la compilation des langages de programmation réduit encore la différence d'efficacité.

La présentation récursive permet d'exhiber simplement des algorithmes beaucoup plus astucieux (et donc plus efficaces) comme pour les tours de Hanoï. Les algorithmes récursifs sont souvent élégants, concis et rapides, de plus ils se prêtent bien à la récurrence.

Malheureusement, les algorithmes récursifs présentent une complexité en espace (du fait de la pile d'exécution). Par défaut, Python limite le nombre d'appels récursifs à 1000. Cette limite peut néanmoins être modifiée selon :

```
import sys
sys.setrecursionlimit(10000)
```

Enfin, la récursivité doit être utilisée avec attention pour ne pas faire "exploser" la pile d'appel comme pour l'exemple de la suite de Fibonacci.

4. Application

Exercice 1 : méthodes d'exponentiation codées récursivement

a) Pour un réel positif a fixé, a^n peut se définir, comme une fonction de l'entier n, par récurrence selon :

$$a^0 = 1 \ \ et \ \ a^n = a \cdot a^{n-1} \ \ si \ n > 1$$

Écrire une fonction récursive `puissance(a,n)` qui permet de calculer a^n.
Donner également une version itérative `puissance_it(a,n)` de cette fonction.

b) Pour élever un nombre a à la puissance n, il existe un algorithme bien plus performant que la méthode naïve précédente ; il s'agit de la méthode dite d'*exponentiation rapide*. Étant donnés un réel positif a et un entier n, il suffit de remarquer que :

$$a^n = \begin{cases} \left(a^{n/2}\right)^2 & si \ n \ est \ pair \\ a \cdot \left(a^{(n-1)/2}\right)^2 & si \ n \ est \ impair \end{cases} \quad \text{et } a^0 = 1$$

Cet algorithme met en application le principe diviser pour régner.
Écrire une fonction récursive `expo_rapid(a,n)` basée sur cette méthode.
Comparer les temps de calcul de ces deux méthodes avec :
a=1.000 000 01 et n=100 000 000. Que peut-on en déduire ?

La formule de Pascal est la plus célèbre des formules des coefficients binomiaux mais elle n'est pas la plus adaptée pour le calcul de p parmi n par un programme informatique.

Exercice 2 : calcul des coefficients binomiaux par la formule du pion

a) Rappeler la formule du pion puis coder une fonction itérative bino_it(n,p) qui calcule et renvoie p parmi n par application de la formule du pion itérée. On se ramènera au cas où p<=n/2 par application de la formule de symétrie des coefficients binomiaux.
b) Écrire une fonction récursive bino_rec(n,p) qui calcule et renvoie p parmi n par application de la formule du pion.

Exercice 3 : calcul des coefficients binomiaux par la formule de Pascal

a) Écrire une fonction récursive binom_rec(n,p) qui calcule et renvoie p parmi n par application de la relation de Pascal.
b) (facultatif, compliqué) Rappeler la relation de Pascal puis coder une fonction itérative binom_it(n,p) qui calcule de proche en proche les coeff. binomiaux par la formule de Pascal, les stocke dans une liste de listes et renvoie p parmi n.