Informatique

5

Fonctions récursives

Résumé

Principe : la fonction s'auto appelle d'après le principe : Traiter – Diviser – Régner – Combiner

Exemple

```python
def fact_classique(n):
    Res = 1
    for i in range(1,n+1):
        Res = Res * i
    return Res
```

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

Avantages	Inconvénients
Programmation claire et simple à comprendre de de problèmes complexes définis facilement étape par étape	Complexité très sensible à la programmation évoluant rapidement en puissance ex a^n, a nombre d'auto-appels récursifs à chaque étape de complexité $O(1)$
	Limite de la taille de la pile d'exécution (essayer 987! et 988! en récursif) Un algorithme récursif qui traite les n termes d'une liste n'est pas optimal…
	Preuve et terminaisons plus complexes à démontrer
	Création de nouvelles variables locales à chaque exécution (mémoire)
	Les variables étant locales, pour récupérer des infos sur l'exécution, il faut déclarer des variables globales ou les mettre en argument (cf compteurs)

Exemples simples d'analyse d'algorithmes

Décompte d'auto-appels par transmission de la fonction

```python
def fact(n):
    if n==0:
        return 1,1
    else:
        Res,C = fact(n-1)
        return Res*n,C+1

print(fact(5))
```

Décompte d'auto-appels par variable globale

```python
def fact(n):
    global C
    C+=1
    if n==0:
        return 1
    else:
        return fact(n-1)*n
C = 0
print(fact(5),C)
```

Utilisation de fonctions pour un suivi avancé

Initialisation du compteur

```python
def init_compteur():
    global Compteur,Compteur_Max
    Compteur = Compteur_Max = 0
```

Affichage du compteur

```python
def affiche_compteur():
    print("Compteur: ",Compteur,", Max: ",Compteur_Max)
```

Incrément du compteur et enregistrement de la plus longue pile d'exécution

Pour le nombre d'exécutions, n'utiliser que `incremente_compteur(1)` en début de fonction

Pour la plus longue pile d'exécution, ajouter `incremente_compteur(-1)` en sortie de fonction – A la fin, le compteur vaut alors 0 (ne pas utiliser de return en cours de fonction, il en faut un seul à la fin et l'incrément de -1 juste avant)

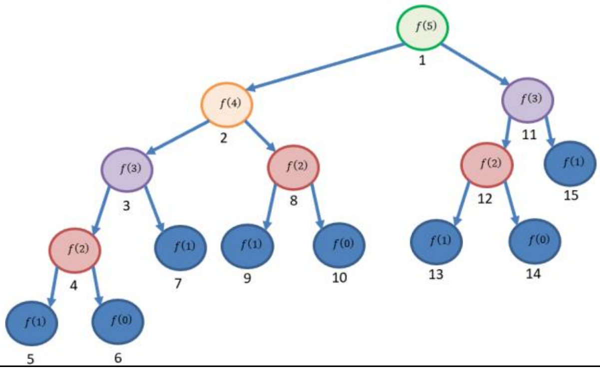
```python
def incremente_compteur(i):
    global Compteur,Compteur_Max
    Compteur += i
    if Compteur > Compteur_Max:
        Compteur_Max = Compteur
```

Amélioration des temps d'exécution : Exemple Fibonacci pour $f(5)$

$$\forall n \in \mathbb{N}, f_n = \begin{cases} 0 & si\ n = 0 \\ 1 & si\ n = 1 \\ f_{n-1} + f_{n-2} & sinon \end{cases}$$

Code normal
15 appels

```python
def fibonacci_Rec(n):
    if n==0:
        Res = 0
    elif n==1:
        Res = 1
    else:
        Res = fibonacci_Rec(n-1)+fibonacci_Rec(n-2)
    return Res
```



$$O(\varphi^n)$$

$$\varphi = \frac{1+\sqrt{5}}{2}$$

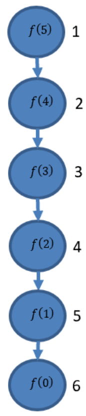
Vectorisation
6 appels
(n+1)

On propage $\begin{bmatrix} f_n \\ f_{n+1} \end{bmatrix}$, ce qui permet de ne pas appeler 2 fois la fonction à chaque étape

```python
def fibonacci_Rec_Vect(n):
    if n==0:
        Res = [0,1]
    else:
        Res = fibonacci_Rec_Vect(n-1)
        Res = [Res[1] , Res[0] + Res[1]]
    return Res

def fibo_V(n):
    Res = fibonacci_Rec_Vect(n)
    return(Res[0])
```

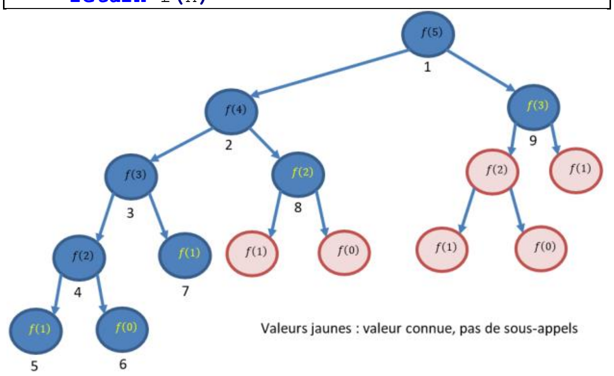
$$O(n)$$



Mémoïsation
9 appels
(2n-1)

Stockage des valeurs calculées à chaque n dans une table
Evite de recalculer des valeurs déjà déterminée à une étape précédente, et diminue ainsi les sous appels associés

```python
def fibonacci_Mem(n):
    def f(n):
        if tab_val[n] is None:
            tab_val[n] = f(n - 1) + f(n - 2)
        return tab_val[n]
    tab_val = [0, 1] + [None] * (n - 1)
    return f(n)
```



Valeurs jaunes : valeur connue, pas de sous-appels

$$O(n)$$

Complexité

Principe de détermination	Exemple *(Cas d'un auto-appel au rang n-1)*
Soit $O(n^\alpha)$ la complexité à l'étape n Soit $C(n)$ la complexité globale à l'ordre n Poser (?) une fonction $f(n) = g(n) * C\big(h(n)\big)$	$f(n) = f(n-1) + O(n^\alpha)$
Ecrire les calculs réalisés à chaque appel	$\begin{cases} f(n) - f(n-1) = O(n^\alpha) \\ f(n-1) - f(n-2) = O((n-1)^\alpha) \\ \dots \\ f(1) - f(0) = O(1) \end{cases}$
Sommer les égalités (sommes télescopiques)	$[f(n) - f(n-1)] + \dots + [f(1) - f(0)]$ $= O(n^\alpha) + O((n-1)^\alpha) + \dots + O(1^\alpha)$
Déterminer mathématiquement $f(n)$ puis $C(n)$ Traiter les éventuels cas particuliers	$f(n) = f(0) + O\left(\sum_{k=1}^{n} k^\alpha\right) = O\left(\sum_{k=1}^{n} k^\alpha\right)$ $f(n) = C(n) = O(n^{\alpha+1})$

Résultats généraux

1	Auto-appel 1 fois au rang n-1 $C(n) = C(n-1) + O(n^\alpha)$		$C(n) = O(n^{\alpha+1})$
2	Auto-appel γ>1 fois au rang n-1 - γ cst $C(n) = \gamma C(n-1) + O(n^\alpha)$		$C(n) = O(\gamma^n)$
31	Auto-appel 1 fois au rang n/2 $C(n) = C\left(\frac{n}{2}\right) + O(n^\alpha)$	$\alpha = 0$	$C(n) = O(\ln n)$
32		$\alpha \geq 1$	$C(n) = O(n^\alpha)$
41	Auto-appel γ>1 fois au rang n/γ - γ cst $C(n) = \gamma C\left(\frac{n}{\gamma}\right) + O(n^\alpha)$	$\alpha = 0$	$C(n) = O(n)$
42		$\alpha = 1$	$C(n) = O(n\ln(n))$
43		$\alpha \geq 2$	$C(n) = O(n^\alpha)$
5	Auto-appel aux rangs n-1 et n-2 $C(n) = aC(n-1) + bC(n-2) + O(1)$ ⚠ $f(n-1) + 2 * f(n-2) \leftrightarrow a = b = 1$ $f(n-1) + f(n-2) + f(n-2)$ $\leftrightarrow a = 1; b = 2$		$\left\{ r_1 = \frac{a + \sqrt{a^2+4b}}{2} ; r_2 = \frac{a - \sqrt{a^2+4b}}{2} \right.$ $C(n) = O({r_1}^n + {r_2}^n) = \begin{cases} O({r_1}^n) \text{ si } \lvert r_1\rvert > \lvert r_2\rvert \\ O({r_2}^n) \text{ si } \lvert r_2\rvert > \lvert r_1\rvert \end{cases}$

Cas particuliers à connaître

$C(n) = C(n-1) + O(1)$ Suite arithmétique		$C(n) = O(n)$
$C(n) = \gamma C(n-1) + O(1)$ Suite arithmético géométrique	$\gamma = 1$	$C(n) = O(n)$
	$\gamma \neq 1$	$C(n) = O(\gamma^n)$

Dangers

```python
def rec(n):
    if n==0: return 1
    else:
        Un_m1 = rec(n-1)
        if Un_m1 < 1:
            return Un_m1 + 1
        else:
            return Un_m1 / 2
```

```python
def rec(n):
    if n==0: return 1
    else:
        if rec(n-1) < 1:
            return = rec(n-1) + 1
        else:
            return = rec(n-1) / 2
```

$O(n)$	Pour $n = 100$, avec un temps de calcul d'environ $10^{-6}\ s$ quand $n = 1$, temps de $100 * 10^{-6} = 10^{-4}\ s$	$O(2^n)$	Pour $n = 100$, avec un temps de calcul d'environ $10^{-6}\ s$ quand $n = 1$, temps de $2^{100} * 10^{-6} = 4.10^{16}\ années$

Remarquez que ce cas s'apparente à de la « mémoïsation locale » : On stocke la valeur pour la réutiliser plusieurs fois sans refaire d'appels récursifs inutiles