Informatique

5

Fonctions récursives

Cours

Fonctions récursives

1.I. La récursivité par l'exemple

1.I.1 Factoriel n

L'exemple classique que l'on prend pour illustrer la récursivité est le calcul de $(n!)_{n\in\mathbb{N}}$

En mathématiques, cette suite est définie par récurrence : $\begin{cases} 0! = 1 \\ \forall n \in \mathbb{N}^*, n! = n * (n-1)! \end{cases}$

On peut définir une fonction classique suivante :

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

Il est possible de programmer ce calcul par récursivité :

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

La fonction, après avoir été mise en mémoire, est capable de s'appeler autant de fois que nécessaire afin d'arriver au résultat direct qu'elle connaît, puis de remonter au résultat attendu. On parle de fonction récursive.

Le principe d'exécution est le suivant, par exemple pour $4!$:

Je demande $4!$

- 4 est différent de 0 - Je cherche $4 * 3!$ – je dois calculer $3!$
  - 3 est différent de 0 - Je cherche $3 * 2!$ – je dois calculer $2!$
    - 2 est différent de 0 - Je cherche $2 * 1!$ – je dois calculer $1!$
      - 1 est différent de 0 - Je cherche $1 * 0!$ – je dois calculer $0!$
        - 0 est égal à 0 - Je retourne 1
      - Je calcule $1 * 0! = 1 * 1 = 1$, je retourne 1
    - Je calcule $2 * 1! = 2 * 1 = 2$, je retourne 2
  - Je calcule $3 * 2! = 3 * 2 = 6$, je retourne 6
- Je calcule $4 * 3! = 4 * 6 = 24$, je retourne 24

1.I.2 Fibonacci

D'une manière générale, les suites définies par récurrence se prêtent bien à l'utilisation de la récursivité. Voici l'exemple de la suite de Fibonacci :

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

$$[0,1,1,2,3,5,8,13,21 \dots]$$

Voici la manière de la programmer :

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

On remarquera que l'on peut appeler plusieurs fois la même fonction en récursivité. On verra toutefois que cette manière de programmer est très inefficace.

1.II. La récursivité

1.II.1 Principe

La récursivité est la base de la stratégie dite « diviser pour régner » :

- Traiter les cas de base
- Diviser : le problème à traiter est divisé en sous problèmes plus simples
- Régner : on applique récursivement l'algorithme à chaque sous problème
- Combiner : on trouve la solution du problème initial en combinant les différents résultats intermédiaires

1.II.2 Terminaison d'un algorithme récursif

Comme lorsque l'on utilise une condition « while », il n'est pas forcément évident de savoir si un algorithme récursif a toujours une fin.

On peut démontrer qu'un algorithme récursif se termine ou non, par récurrence. Prenons un exemple d'algorithme qui ne se finit pas :

```
def sans_fin(n):
    if n==1:
        return 0
    else:
        return sans_fin(n/10)
```

Cet algorithme a-t-il une fin quel que soit n ?

- Rang 1 : $n = 1$
- Rang 2 : il faut que $(n/10)$ soit égal à 1, soit $n = 10^{2-1}$
- Rang i : il faut que $n = 10^{i-1}$

On voit bien que cet algorithme ne peut fonctionner que si n est une puissance de 10. L'exécution de cet algorithme pour un nombre qui n'est pas puissance de 10 va conduire à une boucle infinie.

Il est aussi possible d'exhiber une variable dont la valeur diminue au cours de la résolution. Dans cet algorithme, à tout moment on appelle l'algorithme de $n/10$, cette valeur tend donc vers 0. Dès qu'elle passe en dessous de 1, on sait que l'algorithme ne va pas s'arrêter :

```
def sans_fin(n):
    print(n)
    if n==1:
        return 0
    else:
        return sans_fin(n/10)
```

```
>>> sans_fin(1000)
1000
100.0
10.0
1.0
>>> sans_fin(3)
3
0.3
0.03
0.003
0.0003000000000000003
3.0000000000000004e-05
3.0000000000000005e-06
3.0000000000000004e-07
3.0000000000000004e-08
3.0000000000000004e-09
3.0000000000000005e-10
3.0000000000000006e-11
3.0000000000000005e-12
3.0000000000000003e-13
3.0000000000000005e-14
3.0000000000000006e-15
3.0000000000000004e-16
...
```

1.II.3 Exécution des algorithmes récursifs

Il faut bien comprendre ce qui est exécuté par l'ordinateur lorsque l'on fait appel à des fonctions récursives pour comprendre ensuite comment utiliser au mieux la récursivité. Voyons deux aspects : la pile d'exécution et les variables locales

1.II.3.a Pile d'exécution (stack)

1.II.3.a.i Principe

A chaque fois qu'une fonction est appelée, cet appel est stocké dans ce que l'on appelle la pile d'exécution.

Lorsqu'une fonction Fonction_1 qui appelle une seconde fonction Fonction_2 qui appelle une 3° fonction Fonction_3 est exécutée, voici schématiquement ce qu'il se passe dans la pile d'exécution :

Appel fonction 1	Appel fonction 2	Appel fonction 3
Fonction_1 - Var_Loc_1	Fonction_2 - Var_Loc_2 Fonction_1 - Val_Loc_1	Fonction_3 - Var_Loc_3 Fonction_2 - Var_Loc_2 Fonction_1 - Val_Loc_1

A chaque étape, la pile stocke les fonctions en cours dans l'ordre d'exécution ainsi que les variables locales associées. A partir du moment où la dernière fonction Fonction_3 est appelée, elle exécute son script puis la fonction Fonction_2 peut terminer son travail, puis la fonction Fonction_1 et c'est terminé.

On peut simplement voir cette pile d'exécution en intégrant une erreur dans la fonction 3. La console nous renvoie un « Traceback »

Exemple :

```
def Fonction_1(n):
    return Fonction_2(n)

def Fonction_2(n):
    return Fonction_3(n)

def Fonction_3(n):
    return (1/n)
```

La console nous renvoie :

```
>>> Fonction_1(0)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "C:\Users\DDP\Dropbox\Privé\Professionnel\Cours\CPGE\Chapitres\Info IPT\IPT 2\TP\TP2 - Récursivité\TP2 - Récursivité.py", line 39, in Fonction_1
    return Fonction_2(n)
  File "C:\Users\DDP\Dropbox\Privé\Professionnel\Cours\CPGE\Chapitres\Info IPT\IPT 2\TP\TP2 - Récursivité\TP2 - Récursivité.py", line 42, in Fonction_2
    return Fonction_3(n)
  File "C:\Users\DDP\Dropbox\Privé\Professionnel\Cours\CPGE\Chapitres\Info IPT\IPT 2\TP\TP2 - Récursivité\TP2 - Récursivité.py", line 45, in Fonction_3
    return (1/n)
ZeroDivisionError: division by zero
```

On voit clairement qu'après appelle de Fonction_1 a été appelée Fonction_2, qui a appelé Fonction_3, lieu de l'erreur.

1.II.3.a.ii Exemple

Introduisons une erreur dans la fonction factoriel n ainsi que le stockage d'une variable locale a:

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

Au moment de combiner les résultats, on a programmé une division par 0. Exécutons cet algorithme :

```
>>> fact_recursif(5)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "C:\Users\DDP\Dropbox\Privé\Professionnel\Cours\CPGE\Chapitres\Info IPT\IPT 2\TP\TP2 - Récursivité\TP2 - Récursivité.py", line 15, in fact_recursif
    a = n*fact_recursif(n-1)
  File "C:\Users\DDP\Dropbox\Privé\Professionnel\Cours\CPGE\Chapitres\Info IPT\IPT 2\TP\TP2 - Récursivité\TP2 - Récursivité.py", line 15, in fact_recursif
    a = n*fact_recursif(n-1)
  File "C:\Users\DDP\Dropbox\Privé\Professionnel\Cours\CPGE\Chapitres\Info IPT\IPT 2\TP\TP2 - Récursivité\TP2 - Récursivité.py", line 15, in fact_recursif
    a = n*fact_recursif(n-1)
  File "C:\Users\DDP\Dropbox\Privé\Professionnel\Cours\CPGE\Chapitres\Info IPT\IPT 2\TP\TP2 - Récursivité\TP2 - Récursivité.py", line 15, in fact_recursif
    a = n*fact_recursif(n-1)
  File "C:\Users\DDP\Dropbox\Privé\Professionnel\Cours\CPGE\Chapitres\Info IPT\IPT 2\TP\TP2 - Récursivité\TP2 - Récursivité.py", line 15, in fact_recursif
    a = n*fact_recursif(n-1)
  File "C:\Users\DDP\Dropbox\Privé\Professionnel\Cours\CPGE\Chapitres\Info IPT\IPT 2\TP\TP2 - Récursivité\TP2 - Récursivité.py", line 13, in fact_recursif
    return 1/0
ZeroDivisionError: division by zero
```

On voit qu'à chaque appel de la fonction, on a stocké dans la pile d'exécutions une valeur et une fonction en attente.

On comprendra que le nombre d'appel est limité par un problème de mémoire dans l'ordinateur utilisé. Dès qu'un programme essaie d'accéder à une zone mémoire interdite par le système d'exploitation, ce programme sera crashé.

Ainsi, essayons d'exécuter la fonction factorielle pour 988 (en corrigeant l'erreur introduite précédemment) :

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

fact_recursif(988)
```

```
RecursionError: maximum recursion depth exceeded in comparison
```

Pour éviter un crash, Python limite le nombre d'appels imbriqués à 987.

Cette limitation n'est pas contraignante pour la plupart des applications car on traitera généralement des problèmes ou l'auto-appel se fera au rang n/2 par exemple (ce qui limiterait n à environ 2^{987}). Cette erreur est donc souvent plutôt liée à une erreur de programmation.

Remarque : on comprendra que les fonctions récursives qui traitent une liste en passant par chacun de leurs éléments, ne pourront traiter des listes de taille supérieure à la limite de la pile d'exécution…

1.II.3.b Variables locales

A chaque exécution d'une fonction, des variables locales sont créés. Ces variables, lors de chaque appel, peuvent avoir le même nom, mais ne pointerons pas vers la même adresse mémoire. Ainsi, dans l'exemple suivant :

```
def Fonction_1(n):
    a = Fonction_2(n)
    return a

def Fonction_2(n):
    a = Fonction_3(n)
    return a

def Fonction_3(n):
    a = 1/n
    return a
```

La pile se comporte ainsi :

Appel fonction 1	Appel fonction 2	Appel fonction 3
Fonction_1 - a_1	Fonction_2 - a_2 Fonction_1 - a_1	Fonction_3 - a_3 Fonction_2 - a_2 Fonction_1 - a_1

L'exécution des sous fonctions implique le stockage de variables à chaque étape en plus de la liste des fonctions en cours d'exécution.

A chaque appel d'une sous fonction, un nouvel « espace » de variables **locales** est donc créé.

ATTENTION : vous créez régulièrement des codes récursifs utilisant un compteur… Ce compteur est différent à chaque appel de la fonction !!! Ce n'est donc pas la bonne méthode. Il est possible de propager un compteur en le mettant en argument de la fonction ou en le créant en variable globale, mais ce n'est souvent pas ce qui est attendu !

1.II.3.c Complexité

Nous démontrerons un peu plus tard dans ce cours les résultats suivants.

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

Ce code est d'une complexité à chaque étape en $O(1)$.

Soit le code suivant, exécutant « presque » la même chose, et donnant le même résultat :

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

Ce code est en $O(3^n)$… Essayez fact_recursif(987) avec les deux versions, vous verrez…

La manière de coder peut grandement influencer la complexité en récursif !

1.II.3.d Suivi d'exécution récursive

Deux objectifs :

- Compter le nombre d'auto-appels récursifs d'une fonction
- Déterminer la taille maximale de la pile d'exécution (limitée à 987)

1.II.3.d.i Exemples simples

Voici des exemples simples pouvant être utilisés dans des cas particuliers :

Décompte d'auto-appels par transmission de la fonction	Décompte d'auto-appels par variable globale

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

1.II.3.d.ii Fonctions plus évoluées

Soient les trois fonctions suivantes :

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

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

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

(Lors de l'utilisation avec i=-1, ne pas utiliser de return au milieu de la fonction récursive, il en faut un seul à la fin et placer l'incrément de -1 juste avant)

Il suffit d'ajouter à la fonction à suivre aux bons endroits :

```
incremente_compteur(1)
```

On a alors le **nombre total d'appels de la fonction** :

```
def fibonacci(n):
    incremente_compteur(1)
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return fibonacci(n-1)+fibonacci(n-2)

init_compteur()
print(fibonacci(5))
affiche_compteur()
```

```
5
Compteur:  15  , Max:  15
```

La fonction a été exécutée 15 fois.

Remarque : il est normal, pour le moment, que Compteur_Max ne serve pas ! (cf page suivante)

A ce stade, on peut se demander pourquoi on a créé un compteur max dans la fonction init_compteur. Voici pourquoi. En décrémentant le compteur à chaque sortie de fonction, on peut savoir **le nombre maximum d'appels imbriqués** :

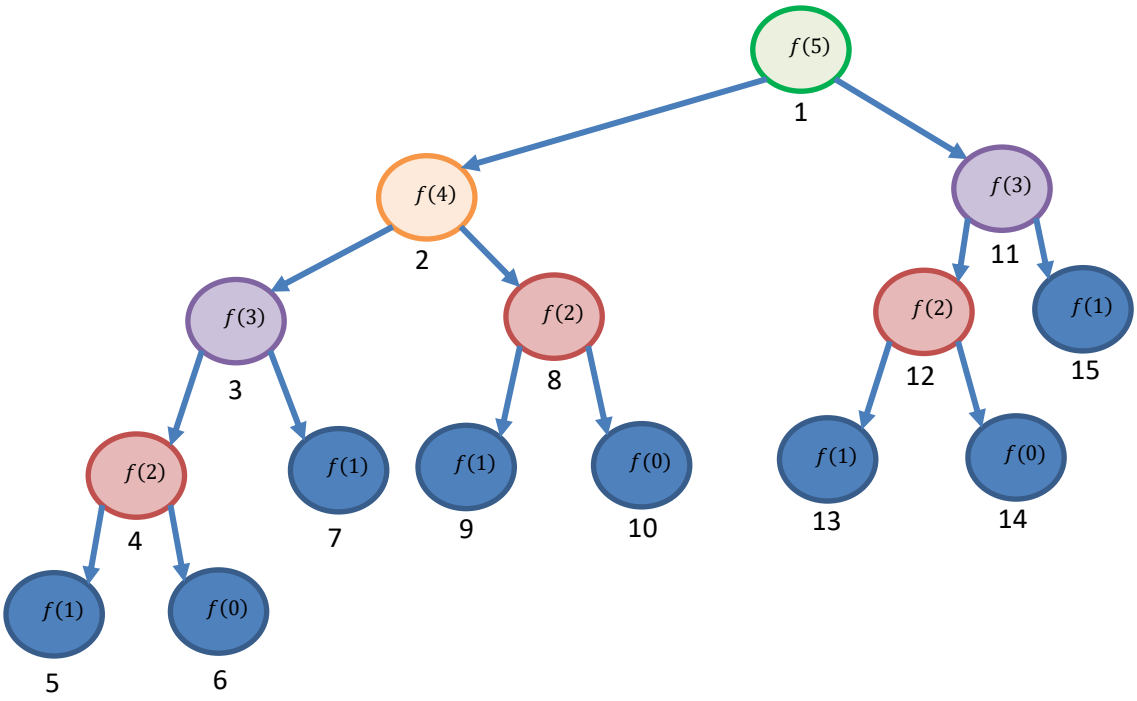
```
def fibonacci(n):
    incremente_compteur(1)
    if n==0:
        Res = 0
    elif n==1:
        Res = 1
    else:
        Res = fibonacci(n-1)+fibonacci(n-2)
    incremente_compteur(-1)
    return Res

init_compteur()
print(fibonacci(5))
affiche_compteur()
```

```
5
Compteur:  0  , Max:  5
```

Le compteur, autant incrémenté que décrémenté, est bien revenu à 0. Sa valeur maximale 5 correspond au nombre de fois où la fonction a été exécutée de manière imbriquée (nombre d'ouvertures max sans nouvelles fermetures).

Illustrons ce qu'il se passe lors de l'exécution de cet algorithme récursif :



On remarquera que l'on recalcule plusieurs fois les mêmes termes. Pour l'appel de $f(5)$, on calcule deux fois $f(3)$ et trois fois $f(2)$ par exemple. Ce n'est pas optimisé.

Remarque : L'illustration ci-dessus serait différente si l'on avait écrit `Res = fibonacci(n-2)+fibonacci(n-1)`

1.II.4 Récursif vs itératif

	Avantages	Inconvénients
Algorithme récursif	Simple à comprendre	Capacité de la pile d'exécution Complexité souvent plus importante et sensible à la programmation
Algorithme itératif	« Pas de limite de taille » Incomparables à cette du récursif	Moins lisible

Reprenons l'exemple de l'algorithme de Fibonacci :

Type	Code	Résultat Compteur

Récursif	`def fibonacci_Rec(n):`
	`  incremente_compteur(1)`
	`  if n==0:`
	`  Res = 0`
	`  elif n==1:`
	`  Res = 1`
	`  else:`
	`  Res = fibonacci_Rec(n-1)+fibonacci_Rec(n-2)`
	`  return Res`
	
	`init_compteur()`
	`print(fibonacci_Rec(5))`
	`affiche_compteur()`
		15

Itératif	`def fibonacci_It(n):`
	`  Res = [0,1]`
	`  for i in range(n):`
	`  incremente_compteur(1)`
	`  Res = [Res[1],Res[0]+Res[1]]`
	`  return Res[0]`
	
	`init_compteur()`
	`print(fibonacci_It(5))`
	`affiche_compteur()`
		5

On remarque qu'il est clairement plus intéressant pour cet algorithme de Fibonacci d'utiliser l'algorithme itératif.

1.II.5 Amélioration des performances de l'algorithme récursif

1.II.5.a Vectorisation

Il est toutefois possible d'améliorer la complexité de l'algorithme récursif en diminuant le nombre d'appels de la fonction récursive à l'aide du stockage dans une liste (vectorisation) comme décrit ci-dessous :

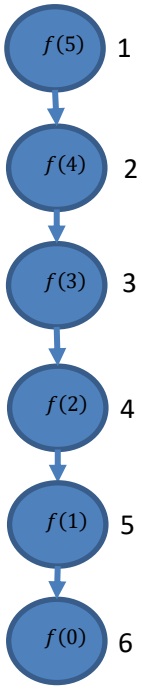
$$\begin{cases} \begin{bmatrix} f_0 \\ f_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ \forall n \in \mathbb{N}^*, \begin{bmatrix} f_n \\ f_{n+1} \end{bmatrix} = \begin{bmatrix} f_n \\ f_n + f_{n-1} \end{bmatrix} \end{cases}$$

A chaque nouvelle itération, on a stocké les deux termes précédents.

Type	Code	Résultat Compteur

Récursif
Vectorisé
	`def fibonacci_Rec_Vect(n):`
	`  incremente_compteur(1)`
	`  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])`
	
	`init_compteur()`
	`print(fibo_V(5))`
	`affiche_compteur()`
		6
		(n+1)

On remarque qu'il n'y a plus qu'un appel récursif à chaque étape au lieu de deux, soit une complexité en temps en $O(n)$.



$O(n)$

1.II.5.b Mémoïsation

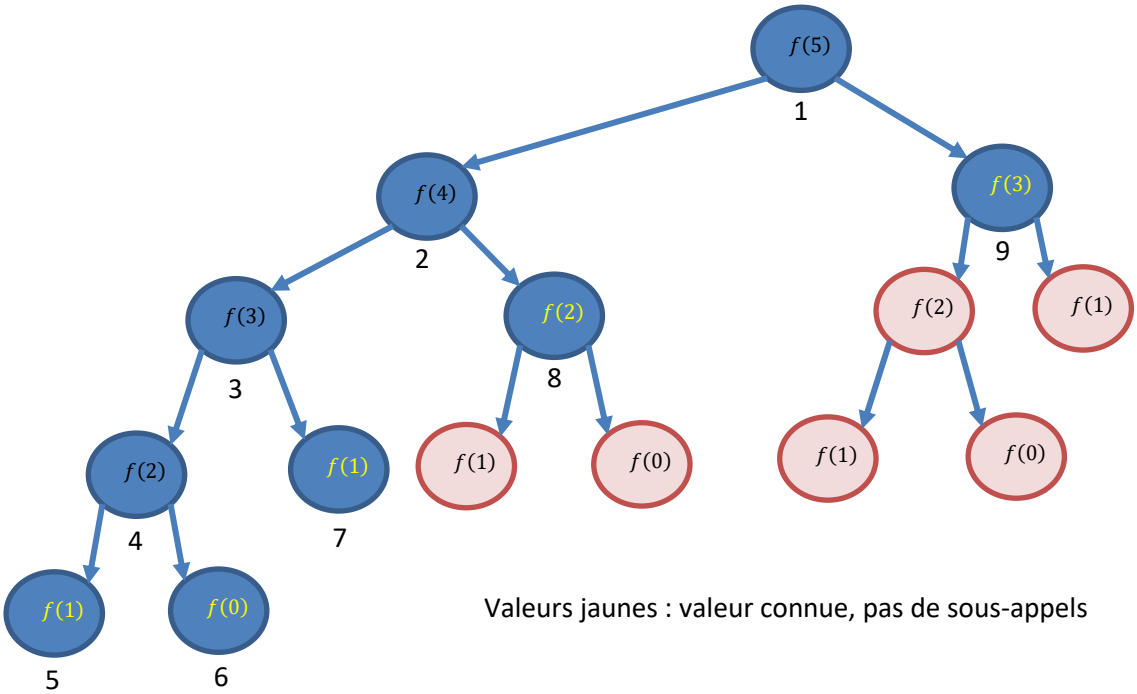
Il est possible d'éviter des appels récursifs inutiles en mémorisant les valeurs des étapes intermédiaires au fur et à mesure qu'elles sont calculées.

Prenons l'exemple de l'algorithme de Fibonacci :

Type	Code	Résultat Compteur

Récursif Mémorisé	`def fibonacci_Mem(n):`
	`  def f(n):`
	`  incremente_compteur(1)`
	`  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)`
	
	`init_compteur()`
	`print(fibonacci_Mem(5))`
	`affiche_compteur()`
	
	Remarque : `f(n)` est définie dans `fibonacci_Mem(n)` afin que tab_val soit locale dans cette fonction mais bien vue de f, devant exister à chaque appel de f
		9
		(2n-1)

Illustration dans le cas de $n = 5$ avec la ligne « `tab_val[n] = f(n - 1) + f(n - 2)` » car l'appel de $n - 2$ d'abord change l'illustration :



$O(n)$

Valeurs jaunes : valeur connue, pas de sous-appels

Connaissant les deux premières valeurs $f(0)$ et $f(1)$, seules 4 nouvelles valeurs sont calculées. En termes d'appels et donc de complexité, il y aura pour cet exemple 9 appels récursifs de f. On remarquera aisément sur la figure que quel que soit n, le nombre d'appels de f sera de $2n - 1$, soit une complexité en temps en $O(n)$.

1.III. Complexité d'une fonction récursive

Soit $C(n)$ la complexité d'une fonction récursive au rang n.

Supposons que la fonction récursive exécute un nombre de calcul de complexité $O(n^\alpha)$ à chaque appel, à chaque étape.

En général, on fait quelques calculs (sommes, différences, produits, divisions, ce qui donne une complexité en $O(1)$. Il n'est toutefois pas impossible de trouver des boucles qui dépendent de n…

Traitons différents exemples souvent rencontrés.

1.III.1 Auto-appel une fois au rang n-1

1.III.1.a Cas général

$$C(n) = C(n-1) + O(n^\alpha)$$

$$\begin{cases} C(n) - C(n-1) = O(n^\alpha) \\ C(n-1) - C(n-2) = O((n-1)^\alpha) \\ \dots \\ C(1) - C(0) = O(1) \end{cases}$$

Sommons ces égalités :

$$[C(n) - C(n-1)] + [C(n-1) - C(n-2)] + \cdots + [C(1) - C(0)] = O(n^\alpha) + O((n-1)^\alpha) + \cdots + O(1^\alpha)$$

$$C(n) - C(0) = O(n^\alpha) + O((n-1)^\alpha) + O(1^\alpha) = O\left(\sum_{k=1}^{n} k^\alpha\right)$$

$$C(n) = C(0) + O\left(\sum_{k=1}^{n} k^\alpha\right) = O\left(\sum_{k=1}^{n} k^\alpha\right)$$

Or :

$$\sum_{k=1}^{n} k^\alpha = n^\alpha \sum_{k=1}^{n} \frac{k^\alpha}{n^\alpha} = \frac{n^{\alpha+1}}{n}\left[\sum_{k=1}^{n}\left(\frac{k}{n}\right)^\alpha\right] = n^{\alpha+1}\left[\frac{1}{n}\sum_{k=1}^{n}\left(\frac{k}{n}\right)^\alpha\right]$$

$$\lim_{n\to+\infty}\left[\frac{1}{n}\sum_{k=1}^{n}\left(\frac{k}{n}\right)^\alpha\right] = \int_0^1 t^\alpha dt = \frac{1}{\alpha+1} \quad (Somme\ de\ Riemann)$$

Donc :

$$\boldsymbol{C(n) = O\left(n^{\alpha+1}\right)}$$

Exemple : Fonction factorielle

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

Pour cette fonction, à l'étape n, la complexité vaut $O(1) = O(n^0)$, soit $\alpha = 0 : C(n) = O(n)$

1.III.1.b Cas particulier

$$C(n) = C(n-1) + O(\mathbf{1})$$

1.III.1.b.i Via une suite

$$C(n) = C(n-1) + O(1)$$

$$C(n+1) = C(n) + O(1)$$

C'est une suite arithmétique :

$$n \geq 1, C(n) = C(0) + nO(1) = O(n)$$

1.III.1.b.ii Par sommes télescopiques

Prenons l'exemple de la fonction Factoriel :

$$\begin{cases} C(n) - C(n-1) = O(1) \\ C(n-1) - C(n-2) = O(1) \\ \dots \\ C(1) - C(0) = O(1) \end{cases}$$

On obtient simplement :

$$C(n) - C(0) = n * O(1) = O(n)$$

Soit :

$$C(n) = O(n)$$

1.III.2 Auto-appel γ>1 fois au rang n-1

1.III.2.a Cas général

On a :

$$C(n) = \gamma C(n-1) + O(n^\alpha) \quad ; \quad \gamma \text{ constant}$$

Posons :

$$f(n) = \frac{C(n)}{\gamma^n}$$

On a alors :

$$f(n) = \frac{C(n)}{\gamma^n} = \frac{\gamma C(n-1)}{\gamma^n} + O\left(\frac{n^\alpha}{\gamma^n}\right) = \frac{C(n-1)}{\gamma^{n-1}} + O\left(\frac{n^\alpha}{\gamma^n}\right) = f(n-1) + O\left(\frac{n^\alpha}{\gamma^n}\right)$$

On a donc :

$$\begin{cases} f(n) - f(n-1) = O\left(\frac{n^\alpha}{\gamma^n}\right) \\ f(n-1) - f(n-2) = O\left(\frac{(n-1)^\alpha}{\gamma^{n-1}}\right) \\ \dots \\ f(1) - f(0) = O\left(\frac{1^\alpha}{\gamma^1}\right) \end{cases}$$

En sommant les égalités comme précédemment, il vient :

$$f(n) - f(0) = O\left(\sum_{k=1}^{n} \frac{k^\alpha}{\gamma^k}\right)$$

Soit la suite u_n telle que :

$$u_n = \frac{n^\alpha}{\gamma^n}$$

Alors :

$$f(n) = f(0) + O\left(\sum_{i=1}^{n} u_i\right)$$

On a :

$$\frac{u_{n+1}}{u_n} = \frac{\frac{(n+1)^\alpha}{\gamma^{n+1}}}{\frac{n^\alpha}{\gamma^n}} = \frac{1}{\gamma}\left(\frac{n+1}{n}\right)^\alpha \underset{n\to\infty}{\to} \frac{1}{\gamma} < 1$$

Ainsi, d'après le critère de d'Alembert, c'est une série convergente. C'est ici que la condition $\gamma > 1$ intervient. Si $\gamma = 1$, la suite u_{n+1} est constante, la somme $f(n)$ diverge. Ce cas a été traité au paragraphe précédent.

On a finalement :

$$O\left(\sum_{i=1}^{n} u_i\right) = O(1)$$

Soit :

$$f(n) = O(1)$$

$$\frac{C(n)}{\gamma^n} = O(1)$$

$$\boldsymbol{C(n) = O(\gamma^n)}$$

Attention : il est très couteux de rappeler plusieurs fois la même fonction récursive à une étape ! On ne le fera que si c'est strictement nécessaire !!!

1.III.2.b Cas particuliers

Dans le cas simple d'un appel γ fois à l'ordre $n-1$ avec une complexité des calculs à l'ordre n en $O(1)$, on a :

$$C(n) = \gamma C(n-1) + O(1)$$

1.III.2.b.i Via une suite

On peut se ramener à une suite arithmético géométrique de la forme :

$$C(n+1) = \gamma C(n) + b$$

La solution est obtenue simplement (que vous avez vu en maths), est :

Si $\gamma = 1$:

$$C(n) = C(0) + nb = O(n)$$

Si $\gamma \neq 1$:

$$\begin{cases} r = \dfrac{b}{1-a} \\ C(n) = \gamma^n (C(0) - r) + r \end{cases} = O(\gamma^n)$$

On retrouve évidemment les résultats généraux précédents.

Cette démarche sera simple à mettre en place sans les astuces du cas général si on vous demande de calculer une complexité dans ce cas.

1.III.2.b.ii Par récurrence

$$\begin{cases} C(n) = \gamma C(n-1) + O(1) \\ C(n-1) = \gamma C(n-2) + O(1) \\ \dots \\ C(1) = \gamma C(0) + O(1) \end{cases}$$

On obtient simplement :

$$C(n) = \gamma C(n-1) + O(1) = \gamma[\gamma C(n-2) + O(1)] + O(1)$$

$$= \gamma[\gamma[\gamma C(n-2) + O(1)] + O(1)] + O(1)$$

$$= \gamma^n C(0) + \gamma^{n-1} O(1) + \cdots + \gamma^2 O(1) + \gamma\, O(1) + O(1)$$

Soit :

$$C(n) = O(\gamma^n + \gamma^{n-1} + \cdots + \gamma^2 + \gamma + 1) = O(\gamma^n)$$

1.III.2.c A retenir

Soient les deux codes suivants et leur complexité associée :

$$u_0 = 1, n \geq 1, u_{n+1} = \begin{cases} u_n + 1 \; si \; u_n < 1 \\ \frac{u_n}{2} \; sinon \end{cases}$$

Code Python	Complexité temporelle

`def rec(n):`
`  if n==0:`
`  return 1`
`  else:`
`  Un_m1 = rec(n-1)`
`  if Un_m1 < 1:`
`  Un = Un_m1 + 1`
`  else:`
`  Un = Un_m1 / 2`
`  return Un`
	$a_n = 0$
	$a_{n+1} = a_n + c \quad ; \quad c \epsilon \mathbb{N}$
	Suite arithmétique
	$\Rightarrow O(n)$
	
	Pour $n = 100$, avec un temps de calcul d'environ $10^{-6}\ s$ quand $n = 1$, temps de :
	$100 * 10^{-6} = 10^{-4}\ s$

`def rec(n):`
`  if n==0:`
`  return 1`
`  else:`
`  if rec(n-1) < 1:`
`  Un = rec(n-1) + 1`
`  else:`
`  Un = rec(n-1) / 2`
`  return Un`
	$a_n = 0$
	$a_{n+1} = 2a_n + c \quad ; \quad c \epsilon \mathbb{N}$
	Suite arithmético géométrique
	$\Rightarrow O(2^n)$
	
	Pour $n = 100$, avec un temps de calcul d'environ $10^{-6}\ s$ quand $n = 1$, temps de :
	$2^{100} * 10^{-6} = 4.10^{16}\ années$

Il faut veiller à ne pas appeler inutilement plusieurs fois une même fonction récursive… Lorsque l'on ne crée pas de variable intermédiaire dans le second code, la complexité passe de $O(n)$ à $O(2^n)$…

1.III.3 Auto-appel 1 fois au rang n/2

Un exemple d'algorithme qui s'appelle une fois au rang n/2 est un algorithme de dichotomie qui se divise en 2 à chaque appel.

Partons donc d'une fonction s'appelant 1 fois au rand $\frac{n}{2}$ telle que la complexité à un rang n vaut $O(n^\alpha)$ avec $\alpha \geq 0$.

On a :

$$C(n) = C\left(\frac{n}{2}\right) + O(n^\alpha)$$

Posons :

$$f(n) = C(2^n)$$

On a alors :

$$f(n) = C(2^n) = C\left(\frac{2^n}{2}\right) + O\left(2^{n\alpha}\right) = C(2^{n-1}) + O(2^{n\alpha}) = f(n-1) + O((2^\alpha)^n)$$

$$f(n) - f(n-1) = O((2^\alpha)^n)$$

On a donc :

$$\begin{cases} f(n) - f(n-1) = O((2^\alpha)^n) \\ f(n-1) - f(n-2) = O((2^\alpha)^{n-1}) \\ \dots \\ f(1) - f(0) = O((2^\alpha)^1) \end{cases}$$

En sommant les égalités comme précédemment, il vient :

$$f(n) - f(0) = O\left(\sum_{k=1}^{n} (2^\alpha)^k\right) \Rightarrow f(n) = O\left(\sum_{k=1}^{n} (2^\alpha)^k\right)$$

1.III.3.a $\alpha = 0$

$$f(n) = O\left(\sum_{k=1}^{n} 1\right) = O(n)$$

$$C(2^n) = O(n) \Rightarrow C(n) = O(\log_2 n) = \boldsymbol{O(\ln n)}$$

1.III.3.b $\alpha \geq 1$

$$f(n) = O\left(\sum_{k=1}^{n} (2^\alpha)^k\right)$$

$\sum_{k=1}^{n}(2^\alpha)^k$ est une somme partielle d'une série géométrique $\left(\sum_{k=0}^{n} q^k = \frac{1-q^{n+1}}{1-q}\ avec\ q \neq 1\right)$:

$$\sum_{k=1}^{n} (2^\alpha)^k = \sum_{k=0}^{n-1} (2^\alpha)^{k+1} = 2^\alpha \sum_{k=0}^{n-1} (2^\alpha)^k = 2^\alpha \frac{1-(2^\alpha)^n}{1-2^\alpha} \underset{n\to\infty}{\sim} 2^\alpha \frac{(2^\alpha)^n}{2^\alpha} = (2^\alpha)^n$$

$$f(n) = O((2^\alpha)^n)$$

$$C(2^n) = O((2^n)^\alpha) \Rightarrow C(n) = O((n)^\alpha) = \boldsymbol{O(n^\alpha)}$$

1.III.4 Auto-appel γ>1 fois au rang n/γ

Prenons un exemple pour comprendre ce que veut dire ce titre. Certains algorithmes sont basés sur la stratégie dite « Diviser pour régner ». Ainsi, par exemple, on divise le problème en 2, on appelle donc deux fois la même fonction récursive sur deux moitiés de problème. Cet appel va à nouveau diviser par deux le travail, et ainsi de suite… On a donc un auto-appel 2 fois au rang n/2 et ainsi de suite.

Partons donc d'une fonction s'appelant γ fois au rand $\frac{n}{\gamma}$ telle que la complexité à un rang n vaut $O(n^\alpha)$.

On a :

$$C(n) = \gamma C\left(\frac{n}{\gamma}\right) + O(n^\alpha) \quad ; \quad \gamma \text{ constant}$$

Posons :

$$f(n) = \frac{C(\gamma^n)}{\gamma^n}$$

On a alors :

$$f(n) = \frac{C(\gamma^n)}{\gamma^n} = \frac{\gamma C\left(\frac{\gamma^n}{\gamma}\right)}{\gamma^n} + \frac{O((\gamma^n)^\alpha)}{\gamma^n} = \frac{C(\gamma^{n-1})}{\gamma^{n-1}} + O((\gamma^n)^{\alpha-1}) = f(n-1) + O((\gamma^{\alpha-1})^n)$$

$$f(n) - f(n-1) = O((\gamma^{\alpha-1})^n)$$

On a donc :

$$\begin{cases} f(n) - f(n-1) = O((\gamma^{\alpha-1})^n) \\ f(n-1) - f(n-2) = O((\gamma^{\alpha-1})^{n-1}) \\ \dots \\ f(1) - f(0) = O(\gamma^{\alpha-1}) \end{cases}$$

En sommant les égalités comme précédemment, il vient :

$$f(n) - f(0) = O\left(\sum_{k=1}^{n} (\gamma^{\alpha-1})^k\right)$$

Soit la suite u_n telle que :

$$u_n = (\gamma^{\alpha-1})^n$$

Alors :

$$f(n) = f(0) + O\left(\sum_{i=1}^{n} u_i\right)$$

On a :

$$\frac{u_{n+1}}{u_n} = \frac{(\gamma^{\alpha-1})^{n+1}}{(\gamma^{\alpha-1})^n} = \gamma^{\alpha-1}$$

1.III.4.a $\alpha = 0$

$$\frac{u_{n+1}}{u_n} = \gamma^{0-1} = \frac{1}{\gamma} < 1$$

Ainsi, d'après le critère de d'Alembert, c'est une série convergente.

On a finalement :

$$O\left(\sum_{i=1}^{n} u_i\right) = O(1)$$

Soit :

$$f(n) = f(0) + O(1)$$

$$\frac{C(\gamma^n)}{\gamma^n} = f(0) + O(1)$$

$$C(\gamma^n) = f(0)\gamma^n + O(\gamma^n)$$

$$C(n) = nf(0) + O(n)$$

$$\boldsymbol{C(n) = O(n)}$$

1.III.4.b $\alpha = 1$

$$\frac{u_{n+1}}{u_n} = \gamma^{1-1} = 1 \Leftrightarrow u_{n+1} = u_n$$

$$O\left(\sum_{i=1}^{n} u_i\right) = O(nu_1) = O(n)$$

$$f(n) = f(0) + O(n) = O(n)$$

$$\frac{C(\gamma^n)}{\gamma^n} = O(n)$$

$$C(\gamma^n) = O(n\gamma^n)$$

Posons : $p = \gamma^n \Leftrightarrow n = \log_\gamma(p)$

En remplaçant dans la formule précédente, il vient :

$$C(p) = O\big(p \log_\gamma(p)\big) = O\left(p \frac{\ln(p)}{\ln(\gamma)}\right) = O(p \ln(p))$$

Ou encore, en revenant à une notation avec n :

$$\boldsymbol{C(n) = O(n \ln(n))}$$

1.III.4.c $\alpha \geq 2$

$$f(n) = f(0) + O\left(\sum_{k=1}^{n} (\gamma^{\alpha-1})^k\right)$$

$\sum_{k=1}^{n} (\gamma^{\alpha-1})^k$ est une somme partielle d'une série géométrique $\left(\sum_{k=0}^{n} q^k = \frac{1-q^{n+1}}{1-q} \; avec \; q \neq 1\right)$:

$$\sum_{k=1}^{n} (\gamma^{\alpha-1})^k = \sum_{k=0}^{n-1} (\gamma^{\alpha-1})^{(k+1)} = \gamma^{\alpha-1} \sum_{k=0}^{n-1} (\gamma^{\alpha-1})^k$$

$$= \gamma^{\alpha-1} \frac{1-(\gamma^{\alpha-1})^n}{1-\gamma^{\alpha-1}} \underset{n\to\infty}{\sim} \gamma^{\alpha-1} \frac{(\gamma^{\alpha-1})^n}{\gamma^{\alpha-1}} = (\gamma^{\alpha-1})^n$$

Avec $\gamma > 1$

Ainsi :

$$f(n) = f(0) + O((\gamma^{\alpha-1})^n) = O((\gamma^{\alpha-1})^n)$$

Soit :

$$\frac{C(\gamma^n)}{\gamma^n} = O(\gamma^{\alpha n - n})$$

$$C(\gamma^n) = \gamma^n O(\gamma^{\alpha n - n}) = O(\gamma^{\alpha n})$$

Posons : $p = \gamma^n \Leftrightarrow n = \log_\gamma(p)$

En remplaçant dans la formule précédente, il vient :

$$C(p) = O\left(\gamma^{\alpha \log_\gamma(p)}\right) = O\left(\gamma^{\alpha \log_\gamma(p)}\right) = O\left(\left(\gamma^{\log_\gamma(p)}\right)^\alpha\right) = O((p)^\alpha)$$

Ou encore, en revenant à une notation avec n :

$$\boldsymbol{C(n) = O(n^\alpha)}$$

1.III.5 Auto-appel aux rangs n-1 et n-2

1.III.5.a Contexte

Dans certains algorithmes, on appelle la fonction $a > 0$ fois à l'ordre $n-1$ et $b > 0$ fois à l'ordre $n-2$. Supposons une complexité des calculs à l'ordre n en $O(1)$.

Exemple de Fibonacci :

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

1.III.5.b Complexité

$$C(n) = aC(n-1) + bC(n-2) + O(1)$$

On peut se ramener à une suite arithmético géométrique de la forme :

$$C(n) - aC(n-1) - bC(n-2) = k$$

En introduisant une suite $u_n = C(n) + c$ et en remplaçant dans l'équation précédente $C(n)$ par $u_n - c$, on obtient :

$$u_n - c - a(u_{n-1} - c) - b(u_{n-2} - c) = k$$

$$u_n - c - au_{n-1} + ac - bu_{n-2} + bc = k$$

$$u_n - au_{n-1} - bu_{n-2} + c(a + b - 1) = k$$

$$u_n - au_{n-1} - bu_{n-2} = k + c(1 - a - b)$$

Il suffit alors de choisir c tel que $k + c(1 - a - b) = 0$:

$$c = \frac{k}{a + b - 1}$$

On obtient alors la suite récurrente linéaire d'ordre 2 suivante :

$$u_n - au_{n-1} - bu_{n-2} = 0$$

Le polynôme caractéristique associé vaut :

$$r^2 - ar - b = 0 \quad ; \quad \Delta = a^2 + 4b > 0 \quad ; \quad \begin{cases} r_1 = \dfrac{a + \sqrt{a^2 + 4b}}{2} \\ r_2 = \dfrac{a - \sqrt{a^2 + 4b}}{2} \end{cases}$$

On a donc : $u_n = \lambda {r_1}^n + \mu {r_2}^n$

On détermine alors λ et μ avec les valeurs de $u_0 = C(0) + c$ et $u_1 = C(1) + c$, et on obtient :

$$C(n) = u_n - c = \lambda {r_1}^n + \mu {r_2}^n - c$$

On aura donc : $\boldsymbol{C(n) = O({r_1}^n + {r_2}^n) = \begin{cases} O({r_1}^n) \; si \; |r_1| > |r_2| \\ O({r_2}^n) \; si \; |r_2| > |r_1| \end{cases}}$ puis identifier lequel de r_1 ou r_2 l'emporte sur l'autre.

Dans le cas généralement rencontré où $a = b = 1$:

$$\begin{cases} r_1 = \dfrac{1+\sqrt{5}}{2} = \varphi \approx 1{,}6 \\ r_2 = \dfrac{1-\sqrt{5}}{2} \end{cases}$$

Avec φ le « nombre d'or »

$$\boldsymbol{C(n) = O(\varphi^n)}$$

1.III.5.c Remarques

1.III.5.c.i Exemple 1

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

On a bien toujours $a = b = 1$! Il n'y a bien qu'un auto-appel aux deux rangs $n-1$ et $n-2$.

Attention : si $f(n)$ est une fonction récursive, écrire $f(n-1) + 2 * f(n-2)$ correspond à 1 seul appel à chacun des rangs $n-1$ et $n-2$, soit aussi $a = b = 1$. On aurait $b = 2$ si on écrivait $f(n-1) + f(n-2) + f(n-2)$

1.III.5.c.ii Exemple 2

Supposons que l'on programme ces fonctions en auto appelant bien 2 fois à l'un des rangs $n-1$ ou $n-2$:

$\forall n \in \mathbb{N}, f_n = \begin{cases} 0 & si \; n = 0 \\ 1 & si \; n = 1 \\ \boldsymbol{f_{n-1} + f_{n-1}} + f_{n-2} & sinon \end{cases}$	$\forall n \in \mathbb{N}, f_n = \begin{cases} 0 & si \; n = 0 \\ 1 & si \; n = 1 \\ f_{n-1} + \boldsymbol{f_{n-2} + f_{n-2}} & sinon \end{cases}$
$\begin{cases} a = 2 \\ b = 1 \end{cases}$	$\begin{cases} a = 1 \\ b = 2 \end{cases}$
$\begin{cases} r_1 = \dfrac{2+\sqrt{4+4}}{2} \approx 2{,}4 \\ r_2 = \dfrac{2-\sqrt{4+4}}{2} \approx -0{,}4 \end{cases}$	$\begin{cases} r_1 = \dfrac{1+\sqrt{1+8}}{2} = 2 \\ r_2 = \dfrac{1-\sqrt{1+8}}{2} = -1 \end{cases}$
$C(n) = O(2{,}4^n)$	$C(n) = O(2^n)$

On retrouve la sévérité d'un double auto-appel à une étape précédente qui crée un arbre d'auto-appels comme dans le cas d'un auto-appel γ fois au rang $n-1$… Il faudra évidemment stocker la valeur de l'auto-appel réalisé plusieurs fois puis simplement utiliser le résultat, on sera alors en $O(\varphi^n)$.