III

Révisions 3 : Complément sur la récursivité et application aux tris fusion et rapide

FIGURE III.1 – Un exemple de peinture "récursive"

PLAN DU CHAPITRE

1 Récursivité terminale

1.1 Définition

Définition 1-1: RÉCURSIVITÉ TERMINALE

Une fonction récursive est dite **terminale** si elle renvoie directement la valeur obtenue par son appel récursif. Ainsi, dans une fonction récursive terminale, la valeur renvoyée par l'appel récursif est directement la valeur obtenue par celui-ci sans autre calcul, et donc **sans remontée de calcul dans la pile d'exécution**.

AVANTAGES :

- Absence de toute remontée dans une pile d'exécution $\Longrightarrow$ moindre complexité spatiale, en particulier si le nombre de récursions est important.
- Certains langages compilateurs détectent les structures récursives terminales et optimisent l'occupation mémoire $\Longrightarrow$ aucun "surcoût machine" par rapport à la version itérative (ce peut être le cas de Python qui permet aussi de réaliser des programmes compilés).
- L'écriture d'une structure récursive terminale est souvent plus claire et très facile à interpréter car **proche d'une structure itérative**.
- Les fonctions récursives terminales sont très facilement transformables en leurs homologues itératives : **c'est la dérécursivation. (cf 2)**

1.2 Structure générale d'un algorithme récursif terminal

Les structures de fonctions suivantes **ne sont pas récursives terminales** :

Listing III.1 –

```
Définition f(p)
    si condition(P)      #Condition sur les paramètres P pour entrer dans la récursion
    alors
        Traitement_1(P) #Traitement 1 des paramètres d'appel P
        f(g(P))         #Appel à fonction récursive f avec paramètres P modifiés par la fonction g
        Traitement_2(P) #Traitement 2 des paramètres d'appel initiaux
    sinon
        B(P)            #Traitement du cas de base
```

car l'appel récursif n'est pas le dernier traitement du cas récursif engagé. Il faudra en effet une remontée complète de la pile pour évaluer `Traitement_2(P)`.

Listing III.2 –

```
Définition f(p)
    si condition(P)      #Condition sur les paramètres P pour entrer dans la récursion
    alors
        Traitement_1(P) #Traitement 1 des paramètres d'appel
        P*f(g(P))       #Appel à la fonction récursive f avec paramètres P modifiés par la fonction g et multip

    sinon
        B(P)            #Traitement du cas de base
```

car l'appel récursif est impliqué dans un calcul avec les paramètres (multiplication). Il faudra en effet une remontée complète de la pile pour évaluer la valeur invoquée initialement.

La structure d'une fonction récursive terminale est la suivante :

Listing III.3 –

```
Définition f(p)
    si condition(P)   #Condition sur les paramètres P pour entrer dans la récursion
    alors
        Traitement(P)   #Traitement des paramètres d'appel
```

```
5          f(g(P))          #Appel à la fonction récursive f avec les paramètres g(P) modifiés par la fonction g
6      sinon
7          B(P)             #Traitement du cas de base
```

Dans cette fonction, l'appel récursif n'est pas inclu dans un traitement, mais est la dernière étape du cas récursif et renvoie directement la valeur invoquée sans calcul. La fonction `f(p)` est donc récursive terminale.

1.3 Premier exemple : factorielle récursive terminale

Rappelons la version récursive classique de la factorielle :

Listing III.4 – Fonction factorielle en récursif

```
1 def fact(n):
2     if n==1: #cas de base!!!
3         return 1
4     else:
5         return n*(fact(n-1)) # Appel récursif non terminal
```

Dans cette fonction, l'appel récursif comporte un traitement, la multiplication par n, qui nécessitera un stockage temporaire dans la pile d'exécution.

IDÉE : on souhaite bâtir une **version récursive terminale de la factorielle**.

La solution consiste à stocker dans une variable supplémentaire (qui viendra donc enrichir la liste des paramètres de l'appel, et donc n n'est plus seul!!) le résultat du calcul $\times n$. On bâtit pour cela la fonction `facRTerm(m,n)` suivante :

$$facRTerm(m,n) = \begin{cases} m & \text{si } n = 1 \\ facRTerm(m \times n, n-1) & \text{sinon} \end{cases}$$

Décomposons l'appel $facRterm(m,n)$:

$$\begin{aligned} facRTerm(m,n) &= facRterm(m \times n, n-1) \\ &= facRterm(m \times n \times (n-1), n-2) \\ &\dots \\ &= facRterm(m \times \underbrace{n \times (n-1) \dots \times 2}_{=n!}, 1) \\ &= m \times n! \end{aligned}$$

Par conséquent, on calcule la factorielle de n en appelant `facRTerm(1,n)` $= n!$.

Exercice de cours: **(1.3) - n° 1.** STRUCTURE DES APPELS

1. *Ecrire les appels récursifs successifs lorsque l'on calcule par exemple* `facRterm(1,4)`.
2. *Conclure.*

Pour éviter d'invoquer un appel de fonction contenant le paramètre supplémentaire $m = 1$, il suffit d'imbriquer la fonction `FacRterm` dans une fonction à appeler ne comportant que n comme argument :

Listing III.5 – Fonction factorielle récursive terminale

```
1 def fact(n):
2     def facRterm(m,n):
3         if n=1:
4             return m
5         else:
6             return facRterm(m*n,n-1)
7     return facRterm(1,n)
```

1.4 Quelques exemples

a - Suite de Fibonacci récursive terminale

On rappelle la définition de la suite de Fibonacci de premiers termes $(0, 1)$:

$$u_0 = 0, u_1 = 1, \qquad u_{n+2} = u_{n+1} + u_n$$

et le script de la fonction récursive non terminale correspondante :

Listing III.6 –

```
def fiborec(n):
    #algorithme récursif
    if n==1 or n==2 :
        return 1
    return fiborec(n-1)+fiborec(n-2)
```

Exercice de cours: **(1.4) - n° 2.** SUITE DE FIBONACCI

1. *Bâtir une fonction récursive terminale* `fiboRterm(a,b,n)` *de calcul de la suite de Fibonacci de premiers termes* (a, b) *et qui renvoie le* $n^{ième}$ *terme de la suite si l'on appelle* `fiboRterm(a,b,n)`.
2. *Détailler les appels successifs à la fonction récursive terminale lors du calcul de* `fiboRterm(a,b,n)`*, par exemple en analysant l'appel* `fiboRterm(0,1,4)`.
3. *Comparer le fonctionnement de cette fonction avec celui de la version récursive non terminale. On pourra établir l'arbre des appels de la fonction récursive non terminale afin de conclure, par exemple en analysant là-encore l'appel* `fiborec(4)`

b - PGCD récursif terminal

On rappelle l'algorithme d'Euclide de calcul du PGCD :

- on calcule le reste de $a \,//\, b$ que l'on stocke dans r.
- tant que $a\%b \neq 0$ faire : $\begin{cases} r = a\%b \\ a = b \\ b = r \end{cases}$
- On renvoie b.

Le script récursif terminal est donc :

Listing III.7 – PGCD récursif terminal

```
def pgcdRterm(a,b):
    if a%b==0:
        return b
    else:
        return pgcdRterm(b,a%b)
```

On constate que l'algorithme d'Euclide est "par essence" **récursif terminal** puisque l'appel récursif, d'une part n'est pas engagé dans un calcul stocké dans la pile d'exécution, mais se veut ici direct, et d'autre part l'appel récusif est la dernière évaluation dans l'algorithme.

c - Recensement dans une liste en version récursive terminale

Dans le chapitre `Révisions 2: principe de la récursivité`, nous avions traité le recensement des "0" dans une liste en binaire. On rappelle ici le script proposé en corrigé :

Listing III.8 –

```
def nombreZerosRec(t,i):
    if i>len(t)-1 or t[i]==1:
        return 0
    else:
    return 1+nombreZerosRec(t,i+1)
```

Exercice de cours: **(1.4) - n° 3.** VERSION RÉCURSIVE TERMINALE

Proposer une version **récursive terminale** *de cette fonction.*

1.5 En résumé

Pour élaborer une fonction récursive terminale, il faut :

- au moins un paramètre de la fonction convergent vers le ou les cas de base.
- un paramètre permettant le stockage des opérations normalement en attente dans une pile d'exécution d'une fonction récursive non terminale.
- et évidemment comme toujours la présence d'un ou plusieurs cas de base permettant l'arrêt de la récursivité.

2 Dérécursivation

2.1 Principe et intérêt

- On peut toujours théoriquement transformer une fonction itérative en fonction récursive et vice-versa. Ce n'est parfois pas une bonne idée lorsque la complexité de l'algorithme explose en version récursive [1].
- Toute fonction **récursive non nécessairement terminale** peut être transformée en une fonction itérative, mais parfois au détriment de la complexité.
- Toute fonction **récursive terminale** est très facilement transformable en son équivalent itératif, et ce en raison du caractère quasi-itératif de la récursivité terminale. Pour ainsi dire, une fonction récursive terminale est simplement une **itération sans commande de boucle**.

Définition 2-1: RÉRÉCURSIVATION

Le procédé de transformation d'une fonction récursive en son équivalent itératif porte le nom de **dérécursivation**.

2.2 Exemples de mise en oeuvre de la dérécursivation

a - Factorielle

Listing III.9 –

```
def facRterm(m,n):
        if n=1:
            return m
        else:
            return facRterm(m*n,n-1)
```

Listing III.10 –

```
def faciter(n):
    m=n
    while n!=1:
        m=m*(n-1)
        n=n-1
    return m
```

1. c'était par exemple le cas du calcul récursif non terminal de la suite de Fibonacci.

b - Suite de Fibonacci

Listing III.11 –

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

Listing III.12 –

```
def fiboiter(a,b,n):
    u0=a
    u1=b
    p=0
    while p!=n:
        u0,u1=u1,u0+u1
        p=p+1
    return u0
```

c - PGCD

Listing III.13 –

```
def pgcdRterm(a,b):
    if a%b==0:
        return b
    else:
        return pgcdRterm(b,a%b)
```

Listing III.14 –

```
def pgcd(a,b):
    while a%b!=0:
        a,b=b,a%b
    return b
```

3 Tris récursifs

3.1 Tri fusion («Merge Sort»)

Ce tri fait partie de la catégorie des tris de type `diviser pour régner`.

- PRINCIPE :
  - la liste initiale à trier est coupée en deux sous-listes que l'on trie récursivement
  - les deux sous-listes triées sont ensuite **fusionnées** en faisant appel à une fonction auxiliaire `fusion`.
  - l'algorithme de fusion consiste à comparer les éléments de tête des deux sous-listes et à sélectionner le plus petit d'entre-eux afin de l'ajouter à une liste auxilaire. Dès qu'une liste est épuisée, on concatène la liste auxilaire avec les éléments restants de la liste non épuisée.

  On peut représenter le fonctionnement de l'algorithme sous forme arborescente, par exemple sur la liste initiale $[3,4,6,2,5,1,8]$; l'algorithme procède d'abord à la division récursive de la liste initiale en sous-listes :

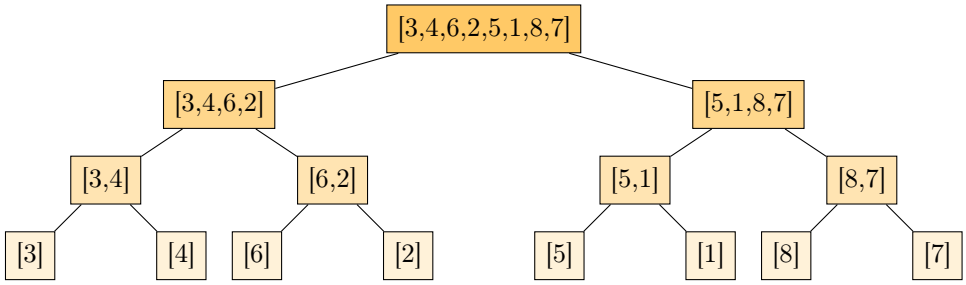

FIGURE III.2 – Arborescence du tri fusion : division récursive en sous-listes

puis effectue la fusion ordonnée des sous-listes :

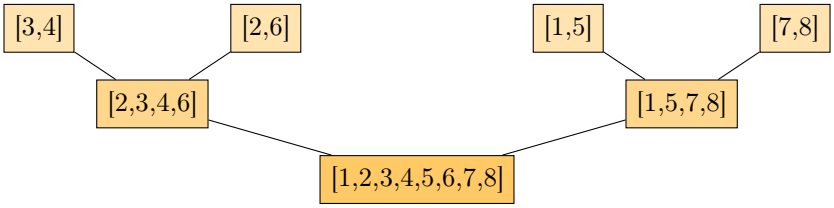

FIGURE III.3 – Arborescence du tri fusion : fusion ordonnée des sous-listes

- IMPLÉMENTATION :

On commence par implémenter la fonction `fusion(L1,L2)` qui réalise la fusion des deux sous-listes `L1` et `L2` :

Listing III.15 –

```
def fusion(L1,L2):
    auxil=[]
    i=0
    j=0
    while i<len(L1) and j<len(L2): #tant qu'aucune liste n'est épuisé on poursuit
        if L1[i]<L2[j]:
            auxil.append(L1[i])
            i=i+1
        else:
            auxil.append(L2[j])
            j=j+1
# une des deux listes est désormais vide; on ajoute donc les éléments de l'autre liste à auxil
    if i=len(L1):  #on teste si la liste L1 est épuisé
        auxil+=L2[j:]
    else:
        auxil+=L1[i:]
    return auxil
```

On écrit ensuite la fonction permettant de trier les sous-listes récursivement :

Listing III.16 –

```
def trifusion(L):
    n=len(L)
  if n<=1: #cas de base
      return L
  m=n//2 #on créé l'indice médian qui coupe la liste en 2
  return fusion(trifusion(L[0:m]),trifusion(L[m:n]))
```

- COMPLEXITÉ :

Choisissons pour simplifier une liste initiale dont le nombre d'éléments est une puissance de 2, soit $N = 2^p$.

Appelons $T(N)$ le nombre de comparaisons effectuées par la fonction `Tri_fusion` et $F(N)$ le nombre de comparaisons effectuées par `fusion` pour trier un tableau de longueur totale N.
On a la récurrence suivante :

$$T(N) = T(N/2) + T(N/2) + F(N) = 2T(N/2) + F(N)$$

puisque les deux appels récursifs se font sur deux listes de même dimension.

► **Meilleur des cas :**
Dans le meilleur des cas, `fusion` n'examine que les éléments d'une seule des deux sous listes et donc $F(N) = N/2$.

Finalement :

$$T(N) = 2T(N/2) + N/2$$

$$T(2^p) = 2T(2^{p-1}) + 2^{p-1}$$

soit en divisant par 2^p :

$$\frac{T(2^p)}{2^p} = \frac{T(2^{p-1})}{2^{p-1}} + \frac{1}{2}$$

En posant $u_p = \dfrac{T(2^p)}{2^p}$, on peut définir la suite $u_p = u_{p-1} + \dfrac{1}{2}$, suite arithmétique de raison $\dfrac{1}{2}$. Le terme général de cette suite est :

$$u_p = \underbrace{u_0}_{=1} + \frac{p}{2}$$

soit :

$$T(2^p) \simeq \frac{p}{2} \times 2^p$$

et comme $N = 2^p$ on a finalement : $T(N) \simeq \dfrac{p}{2}N = \dfrac{N}{2} \times \log_2 N$

Ainsi la complexité dans le meilleur des cas est $\boxed{C(N) = \mathcal{O}(N \log N)}$.

NB : si N n'est pas une puissance de deux, le raisonnement reste le même mais par encadrement de N ($2^p < N < 2^{p+1}$).

- **Pire des cas :**
  Cette fois, tous les éléments des deux listes sont examinés, soit : $F(N) = N - 1$.

  La démarche est identique et le résultat également !

3.2 Tri rapide («Quick Sort»)

Le tri rapide s'appuie comme le tri fusion sur le paradigme *diviser pour régner*. La liste à trier est encore une fois scindée en deux sous-listes ; la liste "d'en bas" qui contiendra les éléments plus petits que ceux de la liste "d'en haut". L'élément `p` situé à **l'indice de séparation** des deux sous-listes est choisi arbitrairement et est appelé **pivot**. Les éléments plus petits que `p` sont placés dans la sous-liste "d'en bas" et les éléments plus grands dans celle "du haut".

- Principe
  - On choisit au hasard dans la liste un élément **pivot** `p`.
  - On compare les éléments des deux sous listes au pivot `p` afin de les classer dans la bonne sous-liste.
  - On itère le processus en réalisant récursivement le tri de chaque sous-liste

- Implémentation :
  Pour simplifier la procédure, nous choisirons le pivot en tête de la sous-liste à trier, soit la position gauche (indicée g plus bas).

  On écrit d'abord la procédure d'échange permettant la permutation de deux éléments $L[i]$ et $L[j]$ de la liste :

  Listing III.17 –

```
def echange(L,i,j):
    L[i],L[j]=L[j],L[i]
```

On écrit ensuite la fonction de partition permettant d'organiser la sous-liste comprise entre les indices g et d par rapport au pivot :

Listing III.18 –

```
def partition(L,g,d):
    assert g<d #on vérifie que les indices sont bien organisés
    ......
```

On prend comme pivot l'élément à gauche de la sous liste L :

Listing III.19 – choix pivot

```
....
    pivot=L[g]
....
```

On parcourt ensuite la liste par une boucle inconditionnelle afin de placer les éléments par rapport au pivot (appel à `echange(L,i,j)` si nécessaire) :

Listing III.20 – parcours de la liste

```
....
    m=g #on initialise un indice courant à l'indice de gauche de la liste
    for i in range(g+1,d):
        if L[i]<pivot:
            m=m+1 #on incrémente l'indice courant de la position à laquelle il faudra replacer le pivot
            echange(L,i,m) #on permute les éléments d'indice i et m
    if m!=g: # si le pivot a bougé, on va devoir le remettre à sa place m connue
        echange(L,g,m)
    return m # on renvoie finalement la position du pivot
```

A titre d'exemple, faisons tourner ce code manuellement sur la liste `L` ci-dessous. On notera m l'indice courant du pivot, i l'indice de la boucle avec le point noir (•) indiquant sa position dans la liste en traitement :
NB : chaque ligne du tableau ci-dessous indique la situation de la liste une fois l'itération de rang i effectuée.

$$L = [7^g_m, 6, 3, 8, 4, 1, 9^d]$$

$i = g+1$	7^g	6_m•	3	8	4	1	9^d
$i = g+2$	7^g	6	3_m•	8	4	1	9^d
$i = g+3$	7^g	6	3_m	8•	4	1	9^d
$i = g+4$	7^g	6	3	4_m	8•	1	9^d
$i = g+5$	7^g	6	3	4	1_m	8•	9^d

– Fin boucle –
– Puis replacement du pivot à sa position définitive :

1^g	6	3	4	7_m	8	9^d

On finit par la partie récursive du tri qui consiste à trier la liste `L` contenue entre les indices g et d :

Listing III.21 –

```
def Tri_rapide_rec(L,g,d):
    if g>=d-1: #teste si la liste contient un seul élément dans ce cas ne rien faire
```

```
3           return
4       m=partition(L,g,d) #on coupe la liste en deux, on l'organise, et on renvoie la position m définitive d
5   #on procède ensuite au tri récursif des sous-listes à partir de cette césure de liste
6       Tri_rapide_rec(L,g,m)
7       Tri_rapide_rec(L,m+1,d)
```

L'appel au tri pour une liste `L` doit se faire en appelant la totalité de la liste :

```
Tri_rapide_rec(L,0,len(L))
```

- Complexité :

La fonction `partition` réalise la comparaison avec le pivot des éléments de $g+1$ à d exclu (donc d-1) ; ce qui correspond donc à $d-g-1$ comparaisons. Si N est le nombre d'éléments dans la liste, alors `partition` fait N-1 comparaisons lors de son premier appel. Puis les 2 appels récursifs dont la complexité va dépendre du "cas" :

▶ Pire des cas :
le pire des cas correspond à un pivot qui reste toujours en position extrême tout le temps donc l'une des sous-listes est vide et l'autre comporte $N-1$ éléments. On a alors pour les appels récursifs $N-1$ comparaisons donc :

$$T(N) = N - 1 + T(N-1)$$

soit en définissant la suite récurrente $u_{N+1} = u_N + N$

de terme général $u_N = \underbrace{u_0}_{=0} + 1 + 2 + ... + (N-1) = \sum_{k=1}^{N-1} k$

D'où la complexité : $C(N) = N\dfrac{N-1}{2} \longrightarrow C(N) \sim \mathcal{O}(N^2)$

▶ Meilleur des cas :
Le meilleur des cas correspond à un pivot exactement au milieu de la liste à chaque récurrence. Dans ces conditions les appels récursifs trient chacun $N/2$ éléments :

$$T(N) = N - 1 + 2T(N/2)$$

Cette récurrence est la même que celle développée plus haut pour le tri fusion. Elle conduit à une complexité en :

$$\boxed{C(N) = \mathcal{O}(N \log N)}$$