Numérique et Sciences Informatiques
Chapitre III - Récursivité - Diviser pour régner

Classe de terminale

I. Récursivité

I.1. Généralités

Un **programme récursif** est un programme organisé de manière à se rappeler lui-même, c'est-à-dire qu'au cours de son déroulement, il demandera sa propre exécution.
Une **fonction récursive** est donc une fonction qui s'appelle elle-même.
Les éléments caractéristiques d'une fonction récursive sont les suivants :

- il faut au moins une situation qui ne consiste pas en un appel récursif. On appelle cette situation le **cas d'arrêt**.
- chaque appel récursif doit se faire avec des données qui permettent de se rapprocher d'une **situation de terminaison**.

L'algorithme correspondant à une fonction récursive est donc le suivant :

```
def ma_fonction(param):
    if cas_d_arrêt:
        return valeur
    else:
        return ma_fonction(autre_param_rapprochant_du_cas_d_arrêt)
```

A chaque fois que la fonction s'appelle, les variables utilisées dans la fonction précédente doivent être stockées en mémoire. Par conséquent, une fonction récursive consomme beaucoup plus de mémoire qu'une fonction non récursive. *Python* fixe la limitation des appels récursifs à 3000 (on peut le changer).

Exemple

On appelle fonction factorielle de paramètre n la fonction qui retourne le nombre entier $1 \times 2 \times 3 \times ... \times (n-1) \times n$. On note ce nombre $n!$. Ainsi on a : $n! = 1 \times 2 \times 3 \times ... \times (n-1) \times n$.
On peut implémenter la fonction factorielle par un paradigme impératif.

```
n = input("Donnez un nombre entier.")
resultat = 1
for i in range(int(n)):
    resultat *= (i + 1)
print(resultat)
```

L'exécution du programme donnera dans la console :

```
Donnez un nombre entier.
5
120
```

Exemple - suite

On peut aussi implémenter la fonction factorielle par un paradigme fonctionnel.

```
def factorielle(n):
    """
    retourne n!
    :param n: (int) un entier
    :returne: (int) un entier
    :CU: n>=0
    """
    resultat = 1
    for i in range(n):
        resultat *= (i + 1)
    return resultat
```

On pourra alors utiliser la fonction :

```
>>> factoriel(5)
120
```

Enfin, on peut implémenter la fonction factorielle avec une fonction récursive.

```
def factorielle_recursif(n):
    """
    retourne n!
    :param n: (int) un entier
    :returne: (int) un entier
    :CU: n>=0
    """
    if n<=1:
        return 1
    else:
        return n * factorielle_recursif(n-1)
```

On pourra alors utiliser la fonction :

```
>>> factorielle_recursif(5)
120
```

Que se passe-t-il à l'exécution de ce programme ?
Lors de l'appelle récursif, le programme empile des données dans une Pile.
Etat de la Pile :

n	5	4	3	2	1
$n \leq 1$ (empile si faux)	Faux	Faux	Faux	Faux	Vrai

Pile
$2 \times factorielle\_recursif(1)$
$3 \times factorielle\_recursif(2)$
$4 \times factorielle\_recursif(3)$
$5 \times factorielle\_recursif(4)$

Dès que la condition d'arrêt est effective, le programme va pouvoir dépiler la Pile.

Pile	Résultat
$factorielle\_recursif(1)$	1
$2 \times factorielle\_recursif(1)$	2
$3 \times factorielle\_recursif(2)$	6
$4 \times factorielle\_recursif(3)$	24
$5 \times factorielle\_recursif(4)$	120

On peut visualiser le fonctionnement de cet appel récursif dans Python Tutor.

I.2. Récursivité terminale

Un algorithme récursif simple est **terminal** lorsque l'appel récursif est le dernier calcul effectué pour obtenir le résultat. Il n'y a pas de "calcul en attente". L'avantage est qu'il n'y a rien à mémoriser dans la pile.
Pour rendre terminal un algorithme récursif, on utilise un accumulateur placé en paramètre. Voici la fonction factorielle en récursivité terminale.

```
1 def fact_term(n, acc = 1):
2     if n <= 1:
3         return acc
4     else:
5         return fact_term(n-1, acc * n)
```

Exemple

Avec le même exemple du calcul de `fact_term(5)`, on a :

étape n°	1	2	3	4	5
n	5	4	3	2	1
$n \leq 1$	Faux	Faux	Faux	Faux	Vrai
acc	1	5	20	60	120
renvoi	ce que renvoie fact_term(4, 5)	ce que renvoie fact_term(3, 20)	ce que renvoie fact_term(2, 60)	ce que renvoie fact_term(1, 120)	120

On peut visualiser le fonctionnement de cet appel récursif dans Python Tutor

I.3. Récursivité multiple

Un algorithme récursif est **multiple** si l'un des cas qu'il distingue se résout avec plusieurs appels récursifs.

Exemple

On appelle coefficient binomial de p parmi n le nombre noté $\binom{n}{p}$ et défini ainsi :

- Si $n = p$ ou si $p = 0$ alors $\binom{n}{p} = 1$.
- $\forall n > p > 0, \binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1}$

On peut implémenter le calcul d'un coefficient binomial de façon récursive :

```
1 def coef_binom(n,k):
2     if n==k or k==0:
3         return 1
4     else:
5         return coef_binom(n-1,k)+coef_binom(n-1,k-1)
```

La fonction `coef_binom()` est récursive multiple puisqu'elle s'appelle deux fois.
On peut visualiser le fonctionnement de cet appel récursif dans Python Tutor

I.4. Structures récursives

Une structure de données récursive est une structure de données qui s'obtient à partir de l'une au moins de ses sous-structures.
Une Liste chaînée est une structure de données récursive.

On rappelle qu'une Liste chaînée d'éléments d'un ensemble E, est :

- soit la Liste vide
- soit un couple (x, l) où $x \in E$ et l est une Liste d'éléments de E.
  x est appelé la **tête** et l le **reste**.

On peut donc construire les Listes chaînées de façon récursives :
Pour construire une Liste chaînée, on peut écrire la fonction récursive :

```
1  import les_listes
2
3  def creation_liste_recursif(*args):
4      if len(args) == 0:
5          return les_listes.List()
6      else:
7          mon_tableau = list(args) # création d'un tableau à partir des arguments
8          prems = mon_tableau.pop(0) # suppression du premier élément
9          # *mon_tableau récupère les éléments de mon_tableau et les place en paramètres
10         return les_listes.List(prems, creation_liste_recursif(*mon_tableau))
```

```
1  >>> creation_liste_recursif(1, 2, 3)
2  (1.(2.(3.())))
```

Les Arbres, que nous verrons dans le prochain chapitre, sont aussi un exemple de structure de données récursive.

II. Diviser pour régner

II.1. Généralités

La méthode « **pour régner** » est une méthode algorithmique qui consiste à :

- découper un problème initial en sous-problèmes (Diviser)
- résoudre chaque sous-problème (Régner)
- calculer une solution au problème initial à partir des solutions des sous-problèmes (Combiner)

Le tri-fusion est un exemple d'algorithme utilisant la méthode "diviser pour régner". Cet algorithme utilise aussi le paradigme de la récursivité multiple.

```
def tri_fusion(tableau, debut = 0, fin = None):
    """
    tri le tableau mis en paramètre entre les indice debut et fin
    :param tableau:(list) un tableau
    :param debut:(int) un entier, par défaut 0
    :param fin:(int ou NoneType) un entier, par défaut None
    :CU: debut >= 0, fin >= 0
    """
    if fin == None:
        fin = len(tableau) - 1
    if debut < fin:
        #Diviser le problème
        milieu = (debut + fin) // 2
        #Résoudre chaque sous-problème
        tri_fusion(tableau, debut, milieu)
        tri_fusion(tableau, milieu + 1, fin)
        #combine les solutions des sous-problèmes
        fusion(tableau, debut, milieu, fin)

def fusion(tableau, debut, pivot, fin):
    """
    Replace les éléments du tableau compris entre les indices debut et fin dans l'ordre croissant.
    :param tableau:(list) un tableau
    :param debut:(int) un entier
    :param pivot:(int) un entier
    :param fin:(int ou NoneType) un entier
    :CU: debut >= 0, fin >= 0, pivot >= 0, debut < fin
    """
    gauche = []
    droite = []
    for indice in range(debut, pivot + 1):
        gauche.append(tableau[indice])
    for indice in range(pivot + 1, fin + 1):
        droite.append(tableau[indice])
    i = 0
    j = 0
    for k in range(debut, fin + 1):
        if i == len(gauche) and j < len(droite):
            tableau[k] = droite[j]
            j += 1
        elif j == len(droite) and i < len(gauche):
            tableau[k] = gauche[i]
            i += 1
        elif gauche[i] <= droite[j]:
            tableau[k] = gauche[i]
            i += 1
        else:
            tableau[k] = droite[j]
            j += 1
```

Complexité en temps du tri fusion

Appelons $t(n)$ le nombre d'opérations pout trier une liste de taille n.
D'une part, on divise le problème en deux sous-problèmes de taille égale à $t\left(\frac{n}{2}\right)$.
D'autre part, la fusion des deux sous-listes prend un temps n puisqu'on compare les n valeurs de la liste.

On peut donc écrire l'équation suivante correspondant à la première étape :

$$t(n) = 2t\left(\frac{n}{2}\right) + n$$

qu'on peut écrire aussi

$$t(n) = 2^1 t\left(\frac{n}{2^1}\right) + 1n$$

A la seconde étape, on a :

$$t\left(\frac{n}{2}\right) = 2t\left(\frac{n}{4}\right) + \frac{n}{2}$$

Et donc :

$$t(n) = 2 \times \left(2t\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n = 4t\left(\frac{n}{4}\right) + n + n = 4t\left(\frac{n}{4}\right) + 2n = 2^2 t\left(\frac{n}{2^2}\right) + 2n$$

De même, avec une étape supplémentaire, on aurait :

$$t(n) = 2^3 t\left(\frac{n}{2^3}\right) + 3n$$

Donc au bout de k étapes dans le tri fusion d'une liste de taille n, on obtient par récurrence :

$$t(n) = 2^k t(\frac{n}{2^k}) + kn$$

De plus, $t(1) = 1$ puisqu'il nous suffit d'une comparaison pour trier un tableau de longueur 1.
Donc si le nombre d'étapes k correspond au cas d'arrêt de la récursivité, on doit avoir :

$$\frac{n}{2^k} = 1 \Leftrightarrow n = 2^k \Leftrightarrow log_2(n) = k$$

On obtient alors :
$t(n) = nt(1) + log_2(n) \times n$
$t(n) = n \times 1 + log_2(n) \times n$
$t(n) = n\,(1 + log_2(n))$

Or $1 + log_2(n) = \mathcal{O}(log_2(n))$
D'où $t(n) = \mathcal{O}(nlog_2(n))$

II.2. Comparaison des tris connus

Nom	Complexité	Coût en opération pour un tableau de $100\,000\,000 = 10^8$ valeurs (assez fréquent dans les grandes entreprises type banque)	Durée pour un ordinateur cadencé à 3GHz nécessitant 10 cycles pour une comparaison
Tri par sélection	$\mathcal{O}(n^2)$	$\left(10^8\right)^2 = 10^{16}$	$\frac{10^{16}}{\frac{3\times10^9}{10}} = \frac{10^{16}}{3.10^8} = \frac{10^8}{3} \approx 3,3.10^7 s \approx 381 j$
Tri par insertion	$\mathcal{O}(n^2)$	$\left(10^8\right)^2 = 10^{16}$	$\frac{10^{16}}{\frac{3\times10^9}{10}} = \frac{10^{16}}{3.10^8} = \frac{10^8}{3} \approx 3,3.10^7 s \approx 381 j$
Tri Fusion	$\mathcal{O}\left(nlog_2(n)\right)$	$\left(10^8\right) \times log_2\left(10^8\right) \approx 2,7.10^9$	$\frac{2,7.10^9}{\frac{3\times10^9}{10}} = \frac{2,7.10^9}{3.10^8} = 9s$