La récursivité et le langage C

I. Ferrané et M.C. Lagasquie

18 octobre 2017

Table des matières

1 Les prérequis **1**

2 Définitions **1**
2.1 Les exemples de base . . . 2
2.1.1 Pour la récursivité terminale : la fonction Factorielle . . . 2
2.1.2 Pour la récursivité non terminale : la fonction de Fibonacci . . . 2
2.1.3 Pour la récursivité croisée : les fonctions Pair et Impair . . . 3

3 Exercices basiques **4**
3.1 La série harmonique . . . 4
3.2 Calcul financier . . . 4
3.3 La fonction Multiplication . . . 4
3.4 La fonction d'Ackermann . . . 5
3.5 Les combinaisons . . . 5
3.6 Nombre pair ou impair de 1 dans un tableau . . . 5

4 Autres exercices **5**
4.1 La fonction de pavage . . . 5
4.2 Le jeu de Hanoi . . . 5
4.3 Le tri rapide . . . 5

5 Conclusion sur la récursivité **7**

Bibliographie **7**

A Corrigés des exercices basiques **8**
A.1 La série harmonique . . . 8
A.2 Calcul financier . . . 8
A.3 La fonction Multiplication . . . 8
A.4 La fonction d'Ackermann . . . 9
A.5 Les combinaisons . . . 10
A.6 Nombre pair ou impair de 1 dans un tableau . . . 10

B Corrigés des autres exercices **10**
B.1 La fonction Puissance . . . 10
B.2 Fonctions de codage et décodage . . . 11
B.2.1 Version transformable terminale (sans utiliser de variable *accumulateur*) . . . 11
B.2.2 Version terminale (en utilisant une variable *accumulateur*) . . . 12
B.3 La fonction de pavage . . . 13
B.4 Le jeu de Hanoi . . . 14
B.5 Le tri rapide . . . 17
B.6 Nb. de façons d'exprimer un nombre sous la forme d'une somme de nombres . . . 18
B.7 Exercice final . . . 19

1 Les prérequis

Ce document fait suite aux documents “Bases sur le langage C” et “Compléments sur le langage C” étudiés en première partie de cours.
Il vous faut en particulier connaître la notion de *fonction* (sous-programme).
Il a été bâti en s'appuyant essentiellement sur les ouvrages suivants : [Dri90, Del92, Got93, RS94, Kal90, HV93].

2 Définitions

Définition 1 *Une fonction récursive est une fonction qui s'appelle elle-même.*

Il existe différents types de récursivité suivant le type de l'appel récursif (voir [HV93]).

Définition 2 *Une fonction récursive est dite* primitive *si cette fonction ne figure pas dans les arguments de l'appel récursif.*

Cela veut dire que l'*appel récursif* ne peut pas être de la forme $f(a_1, \ldots, a_n)$ avec un des $a_i = f(a'_1, \ldots, a'_n)$.

Définition 3 *Une fonction primitive est dite* terminale *si, dans l'appel récursif, elle n'apparaît pas dans les arguments d'une autre fonction.*

Il s'agit donc d'une fonction récursive qui ne laisse aucun calcul en suspens. L'*instruction contenant l'appel récursif* sera donc de la forme $f(a_1, \ldots, a_n)$ avec tous les a_i étant des expressions ne contenant aucun appel récursif à la fonction f.
Il est bien connu (voir [Kal90]) que de telles fonctions sont traduisibles de manière automatique sous forme itérative.
Or, il existe des fonctions qui, bien que non récursives terminales, peuvent s'écrire sous forme terminale et bénéficier ainsi des algorithmes de traduction automatique en version itérative.

Définition 4 *Soit une fonction f récursive sous la forme suivante :*

$$\begin{aligned} f(x_1, \ldots, x_n) &= g(x_1, \ldots, x_n) \text{ si } t(x_1, \ldots, x_n), \\ f(x_1, \ldots, x_n) &= h(x_1, \ldots, x_n) \oplus f(k_1(x_1, \ldots, x_n), \ldots, k_n(x_1, \ldots, x_n)) \text{ sinon.} \end{aligned}$$

et respectant les propriétés suivantes :
- *$x_1, \ldots, x_n$ sont les paramètres de f,*
- *$t(x_1, \ldots, x_n)$ est une expression booléenne,*
- *chaque $k_i(x_1, \ldots x_n)$ est du même type que le paramètre x_i correspondant,*
- *aucun des $k_i(x_1, \ldots x_n)$ ne contient d'appel récursif à la fonction f,*
- *les expressions $g(x_1, \ldots, x_n)$ et $h(x_1, \ldots, x_n)$ renvoient une valeur du même type que celle renvoyée par f,*
- *ni $g(x_1, \ldots, x_n)$, ni $h(x_1, \ldots, x_n)$ ne contiennent d'appel récursif à la fonction f,*
- *la loi $\oplus$ doit être associative.*

Une telle fonction sera dite transformable terminale.

On a alors la propriété suivante :

Propriété 1 *(voir [Kal90]) La fonction f transformable terminale et donnée sous la forme suivante :*

$$\begin{aligned} f(x_1, \ldots, x_n) &= g(x_1, \ldots, x_n) \text{ si } t(x_1, \ldots, x_n), \\ f(x_1, \ldots, x_n) &= h(x_1, \ldots, x_n) \oplus f(k_1(x_1, \ldots, x_n), \ldots, k_n(x_1, \ldots, x_n)) \text{ sinon.} \end{aligned}$$

peut s'écrire sous la forme terminale suivante :

$$\begin{aligned} f(x_1, \ldots, x_n, x_{n+1}) &= x_{n+1} \text{ si } t(x_1, \ldots, x_n), \\ f(x_1, \ldots, x_n, x_{n+1}) &= f(k_1(x_1, \ldots, x_n), \ldots, k_n(x_1, \ldots, x_n), x_{n+1} \oplus h(x_1, \ldots, x_n)) \text{ sinon.} \end{aligned}$$

*avec le paramètre supplémentaire x_{n+1} initialisé à $g(x_1, \ldots, x_n)$ lors de l'appel de la fonction (ce paramètre sert d'*accumulateur *pour conserver l'état du calcul courant et éviter d'avoir à “remonter” les appels récursifs pour finir le calcul).*

Attention à la terminologie. Dans la littérature (voir [HV93]), on trouve une autre définition pour la "terminalité". Sachez toutefois que ces 2 notions se rejoignent et que c'est pour cela que nous avons choisi d'appliquer ici la définition donnée ci-dessus.

2.1 Les exemples de base

2.1.1 Pour la récursivité terminale : la fonction FACTORIELLE

Pour $n \geq 0$:

$$\begin{aligned} Fact(n) &= 1 \text{ si } n = 0, \\ Fact(n) &= n \times Fact(n-1) \text{ sinon.} \end{aligned}$$

Cette fonction est primitive mais non terminale. Son code en langage C est :

Solution :

```
int Fact(int n)
{
if (n == 0) return 1;
else return (n × Fact(n − 1));
}
```

Par contre, on constate qu'il est très facile de transformer cette fonction récursive en fonction itérative :

Solution :

```
int Fact(int n)
{
int res = 1;
while (n! = 0)
        {
        res = res × n;
        n − −;
        }
return (res);
}
```

Il est aussi facile de voir que cette fonction est transformable terminale (puisque $\times$ est une loi associative). Donc en appliquant la propriété 1 page précédente, on obtient la version terminale de cette fonction :

$$\begin{aligned} Fact(n, r) &= r \text{ si } n = 0, \\ Fact(n, r) &= Fact(n-1, r \times n) \text{ sinon.} \end{aligned}$$

avec $r = 1$ lors de l'appel de $Fact$.

Solution :

```
int Fact(int n, int r)
{
if (n == 0) return r;
else return (Fact(n − 1, n × r));
}
```

avec l'appel Fact(n, 1).

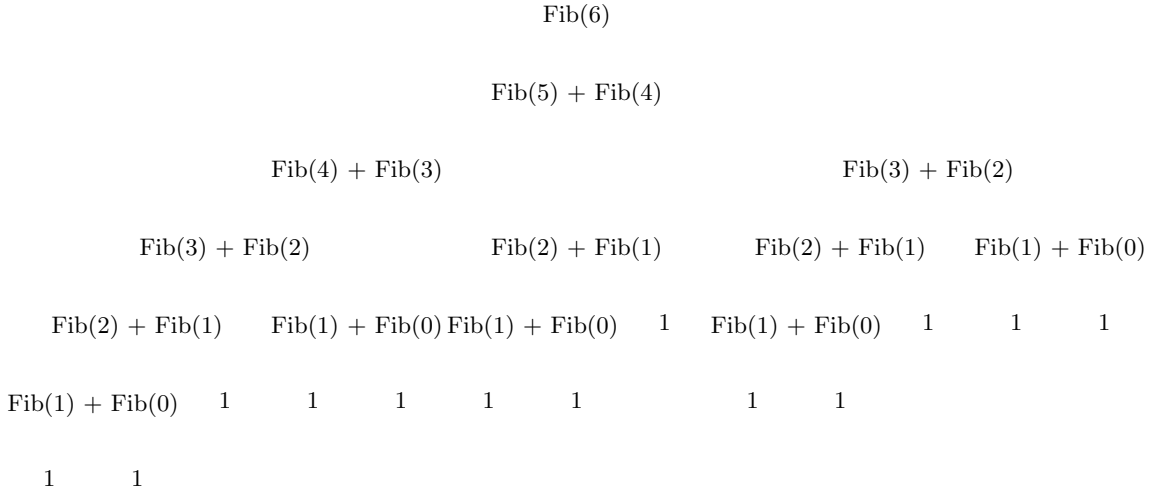
2.1.2 Pour la récursivité non terminale : la fonction de FIBONACCI

Pour $n > 0$:

$$\begin{aligned} Fib(n) &= 1 \text{ si } n = 1, 0 \\ Fib(n) &= Fib(n-1) + Fib(n-2) \text{ sinon} \end{aligned}$$

Solution :
int Fib(int n)
{
if $((n == 1)||(n == 0))$ return 1 ;
else return $(\text{Fib}(n-1) + \text{Fib}(n-2))$;
}

Remarque : si on fait l'arbre d'appel de cette fonction, par exemple pour $n = 6$, on constate rapidement que l'on est amené à refaire plusieurs fois les mêmes calculs :



Récursivité ne veut donc pas dire efficacité (on peut avoir des fonctions récursives très efficaces et d'autres pas du tout !). Par contre, l'écriture d'un programme récursif est souvent très simple, car très proche de la formulation mathématique du problème à résoudre.
Ici aussi, on constate que cette fonction n'est pas terminale. Par contre, on voit aussi qu'elle est non transformable terminale donc la propriété 1 page 1 ne s'applique pas.
Et pourtant, en cherchant un peu et en exploitant l'idée de l'accumulateur, on peut en trouver une version terminale (mais le processus de traduction n'est pas automatique) :

Solution :
int fibrecterm (int n, int rp, int rap)
{
if $((n == 1)||(n == 0))$
    return rp ;
else
    return fibrecterm$(n-1,\ rp+rap,\ rp)$;
}
avec l'appel fibrecterm(n,1,1).

2.1.3 Pour la récursivité croisée : les fonctions Pair et Impair

Les fonctions permettant de savoir si un entier $n \geq 0$ est pair ou impair ne sont a priori pas récursives :

$$\begin{aligned} \text{Pair}(n) &= 1 \text{ si } n \text{ est pair}, 0 \text{ sinon}, \\ \text{Impair}(n) &= 1 \text{ si } n \text{ est impair}, 0 \text{ sinon}. \end{aligned}$$

Si on veut écrire ces fonctions sous la forme récursive, on a plusieurs possibilités dont l'une qui "croise la récursivité" (il s'agit de fonctions qui ne sont pas récursives mais qui s'appellent mutuellement) :
Pour $n \geq 0$:

$$\begin{aligned} \text{Pair}(n) &= 1 \text{ si } n = 0, \text{Impair}(n-1) \text{ sinon,} \\ \text{Impair}(n) &= 0 \text{ si } n = 0, \text{Pair}(n-1) \text{ sinon.} \end{aligned}$$

Solution :

```
int pair (int n)
{
if (n == 0) return 1;
else return (impair(n - 1));
}
```

```
int impair (int n)
{
if (n == 0) return 0;
else return (pair(n - 1));
}
```

3 Exercices basiques

Pour chaque exercice, dire si la fonction proposée est primitive, terminale, transformable terminale. Et si la réponse à la dernière question est "oui", donner la version terminale correspondante.

3.1 La série harmonique

Énoncé 1 *Écrire la fonction récursive calculant la série harmonique issue d'un entier $n > 0$:*

$$S(n) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}$$

3.2 Calcul financier

Énoncé 2 *Étudier la fonction récursive calculant la somme "capital + intérêts" accumulée après n années de placement au taux annuel t, sachant que la somme placée initialement est c et que tous les ans, au 1^{er} janvier, on augmente notre capital en placement de la somme ac.*

3.3 La fonction Multiplication

Énoncé 3 *Écrire une fonction récursive qui calcule $a \times b$ pour a et b entiers ≥ 0. Utiliser la méthode suivante :*

$$\begin{aligned} a \times b &= 0 \textit{ si } a \textit{ ou } b = 0, \\ a \times b &= a + (a \times (b-1)) \textit{ si } b \geq 1. \end{aligned}$$

Énoncé 4 *Écrire une fonction récursive qui calcule $a \times b$ pour a et b entiers ≥ 0, en utilisant la méthode optimisée suivante (fonction de la parité de b) :*

$$\begin{aligned} a \times b &= 0 \textit{ si } a \textit{ ou } b = 0, \\ a \times b &= (a \times 2) \times (\frac{b}{2}) \textit{ si } b > 1 \textit{ et pair}, \\ a \times b &= a + ((a \times 2) \times (\frac{b-1}{2})) \textit{ si } b \geq 1 \textit{ et impair.} \end{aligned}$$

Énoncé 5 *Étudier une fonction récursive qui calcule $a \times b$ pour a et b entiers ≥ 0, en optimisant la version donnée dans l'énoncé 3 mais sans tester la parité de b :*

$$\begin{aligned} a \times b &= 0 \text{ si } a \text{ ou } b = 0, \\ a \times b &= a \text{ si } b = 1, \\ a \times b &= (2 \times a) + (a \times (b-2)) \text{ si } b > 1. \end{aligned}$$

3.4 La fonction d'Ackermann

Énoncé 6 *Écrire une fonction récursive qui calcule la fonction d'Ackermann pour n et m deux entiers ≥ 0 :*

$$\begin{aligned} Ack(0,n) &= n+1, \\ Ack(m,0) &= Ack(m-1,1), \\ Ack(m,n) &= Ack(m-1, Ack(m,n-1)) \end{aligned}$$

3.5 Les combinaisons

Énoncé 7 *Écrire la fonction récursive permettant de calculer la combinaison C_n^p pour n et p deux entiers ≥ 0 (avec $n \geq p$) :*

$$\begin{aligned} C_n^p &= 1 \text{ si } n = p \text{ ou } p = 0 \\ C_n^p &= C_{n-1}^p + C_{n-1}^{p-1} \end{aligned}$$

3.6 Nombre pair ou impair de 1 dans un tableau

Énoncé 8 *Écrire les fonctions récursives croisées permettant de savoir si un tableau de bits t de dimension N contient un nombre pair (resp. impair) de 1. Le tableau sera codé sous la forme d'un tableau d'entiers. Hypothèse : on part en appelant* nbpair_de_1 *en premier.*

4 Autres exercices

4.1 La fonction de pavage

Énoncé 9 *Étudier une fonction récursive qui calcule le nombre de possibilités pour paver un couloir $2 \times N$ avec des dalles 1×2.*

4.2 Le jeu de Hanoi

Énoncé 10 *Implémenter la procédure récursive permettant de résoudre le jeu des tours de Hanoï.*
Ce jeu est un casse-tête composé de 3 piques numérotées 1, 2 et 3 et de N disques perforés de taille tous différents.
Le but de ce jeu est de passer d'une configuration initiale (tous les disques sont sur la pique 1 rangés par ordre décroissant) à une configuration finale (tous les disques sont sur la pique 3 rangés par ordre décroissant). Voir sur la figure 1, un exemple avec 3 disques.
Attention : Les déplacements des disques se font en respectant la règle suivante (voir figure 2) :
à chaque étape, sur chaque pique, il n'y a jamais un disque recouvert par un disque plus grand !

4.3 Le tri rapide

Énoncé Écrire la fonction récursive réalisant le tri rapide d'un tableau d'entiers non triés.
La méthode de tri rapide se définit de la manière suivante :

- on choisit un pivot (en général le premier élément de la zone à trier),
- on partitionne la zone à trier en 3 partitions (à gauche celle contenant les éléments du tableau qui sont plus petits que le pivot, puis au milieu celle ne contenant que le pivot, et enfin à droite celle contenant les éléments du tableau qui sont plus grands que le pivot),

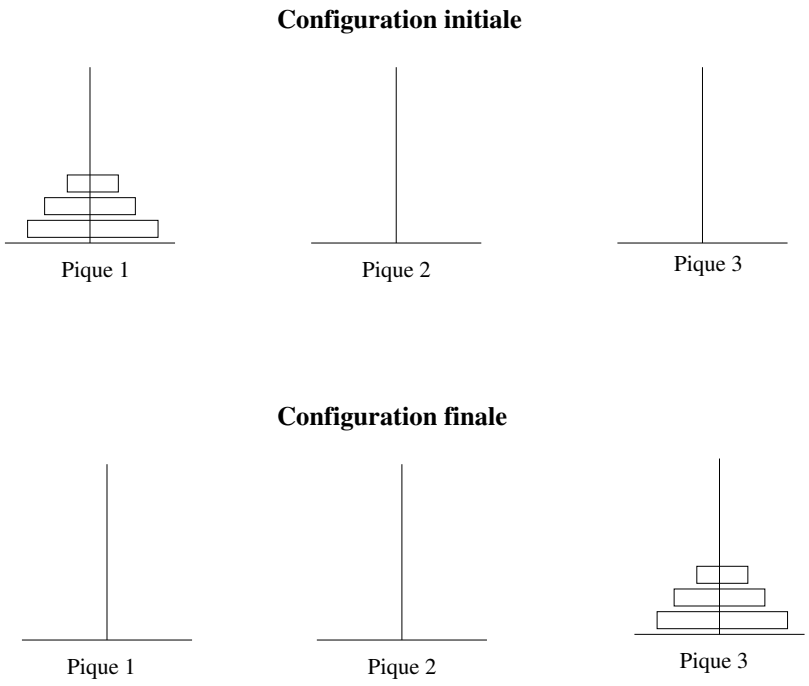


FIGURE 1 – Exemple du jeu de Hanoï avec 3 disques (configurations initiale et finale)

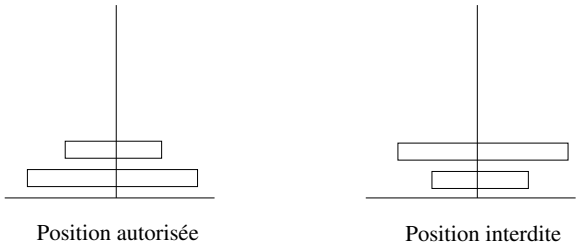


FIGURE 2 – Configurations autorisées et interdites lors du déplacement des disques

— puis on relance le processus sur les partitions non encore triées.

Sur un exemple, cela donne la chose suivante (pour chaque étape, les pivots sont en gras et les sous-tableaux restant à traiter sont encadrés) :

101	213	20	123	79	47	195
79	47	20	101	123	213	195
20	47	79	101	123	213	195
20	47	79	101	**123**	213	195
20	47	79	101	123	**213**	195
20	47	79	101	123	195	213

Quelques éléments de complexité

Prenons quelques hypothèses :

— nous devons trier n éléments (avec $n = 10^6$),

— nous avons à notre disposition une machine qui exécute 10^6 instructions à la seconde.

Étudions maintenant les différentes méthodes :

— avec une méthode de tri simple, il faut n^2 instructions pour résoudre le problème, donc 300 heures !

— avec la méthode du tri rapide, il faut $n \times \log n$ instructions pour résoudre le problème, donc 6 secondes !

5 Conclusion sur la récursivité

Une fonction récursive est une fonction qui s'appelle elle-même.

Il existe plusieurs types de récursivité (attention à la terminologie !) :

— primitive ou non,

— terminale ou non,

— transformable-terminale ou non,

— croisée.

Il existe des processus de traduction systématique d'une fonction récursive terminale vers une fonction itérative (avec une boucle).

D'autre part, il faut savoir qu'une fonction récursive n'est pas toujours efficace, même si elle est très souvent simple à écrire.

Références

[Del92] Delannoy (Claude). – *Exercices en langage C.* – Eyrolles, 1992. ISBN : 2-212-08251-7.

[Dri90] Drix (Philippe). – *Langage C norme ANSI : vers une approche orientée objet.* – Masson, 1990. ISBN : 2-225-81912-2.

[Got93] Gottfried (Byron S.). – *Programmation en C : cours et problèmes.* – Ediscience international, Mc Graw Hill, 1993. ISBN : 2-7042-1230-9.

[HV93] Hardin (Thérèse Accart) et Viguié (Véronique Donzeau-Gouge). – *Concepts et outils de programmation. Le style fonctionnel, le style impératif avec CAML et Ada.* – InterEditions, 1993.

[Kal90] Kaldewaij (Anne). – *Programming : The Derivation of Algorithms.* – Prentice Hall, 1990.

[RS94] Rigaud (Jean-Marie) et Sayah (Amal). – *Programmation en langage C.* – Cépaduès, 1994.