

Chapitre 6

Récurtivité

Un **sous-programme** est dit **récurtif** lorsqu'il fait appel à lui-même ou à des sous-programmes qui lui font références. On parle de **récurtivité directe** lorsque le sous-programme s'appelle lui-même et de **récurtivité croisée** lorsque plusieurs sous-programmes s'appellent mutuellement.

6.1 Réversivité directe

Commençons par la réversivité directe, à propos d'exemples divers. La théorisation de cette notion ne concerne pas un cours d'initiation à la programmation tel que celui-ci.

6.1.1 Exemple de la fonction factorielle

Introduction.- Considérons l'exemple, devenu classique, de la *factorielle*. À un entier naturel n donné on associe l'entier, noté $n!$, défini par :

$$n! = n.(n - 1).(n - 2)....2.1,$$

et en particulier $0! = 1$.

Un programme itératif.- Nous savons programmer cette fonction sans faire appel à la réversivité. En général il intervient alors au moins une boucle, ou itération. On parle alors de **programme itératif**, par opposition à un **programme réversif**, dans lequel intervient de la réversivité. Le programme itératif repose sur le fait que: $n! = \prod_{k=1}^n k$. On a donc le programme suivant :

```
/* fact.c */
#include <stdio.h>

int fact(int n)
{
    int F, k;
    F = 1;
    for (k = 2; k <= n; k++) F = F*k;
    return F;
}

void main(void)
{
    int n;
    printf("n = ");
    scanf("%d",&n);
    printf("%d! = %d.\n", n, fact(n));
}
```

Un programme récursif- Le programme récursif repose sur la définition récursive suivante de la fonction factorielle :

$$0! = 1,$$
$$(n + 1)! = (n + 1).n!.$$

Ceci donne lieu au programme suivant :

```
/* Programme fact2.c */

#include <stdio.h>

int fact(int n)
{
    if (n == 0) return 1;
    else return n*fact(n-1) ;
};

void main(void)
{
    int n;
    printf("n = ");
    scanf("%d",&n);
    printf("%d ! = %d.\n", n, fact(n));
}
```

Commentaire- On remarque que le corps de la fonction fait appel à la fonction elle-même. Nous n'avions jamais dit que cela était permis jusqu'à maintenant. Cela est permis dans certains langages de programmation tel que le langage C. Bien entendu cela complique l'implémentation d'un tel langage mais facilite la tâche du programmeur.

6.1.2 Autres exemples

Exercice 1.- Écrire deux fonctions C, l'une utilisant un algorithme itératif, l'autre un algorithme récursif, permettant de calculer, l'entier naturel n étant donné en entrée, la somme des n premiers entiers naturels non nuls.

Exercice 2.- Écrire deux fonctions C, l'une utilisant un algorithme itératif, l'autre un algorithme récursif, permettant de calculer, l'entier naturel n étant donné en entrée, la somme des n premiers entiers naturels non nuls à la puissance cinq.

Exercice 3.- Écrire deux fonctions C, l'une utilisant un algorithme itératif, l'autre un algorithme récursif, permettant de calculer, l'entier naturel n étant donné en entrée, la somme $1.2 + 2.3 + 3.4 + \dots + n.(n + 1)$.

Exercice 4.- Écrire une fonction C, utilisant un algorithme récursif, permettant de calculer la puissance x^n , avec x réel et n entier naturel.

Exercice 5.- Écrire une fonction C, utilisant un algorithme récursif, permettant de calculer la puissance x^k , avec x réel non nul et k entier relatif.

Exercice 6.- Écrire une fonction C, utilisant un algorithme récursif, permettant de calculer le n -ième nombre de Fibonacci.

Exercice 7.- Écrire une fonction C, utilisant un algorithme récursif, permettant de calculer le coefficient binomial C_n^p , où n et p sont des entiers naturels.

Exercice 8.- Écrire une fonction récursive permettant de calculer x^n pour x réel et n entier naturel en utilisant la définition récursive suivante :

$$\begin{aligned} x^n &= 1 \text{ pour } n = 0, \\ x^n &= (x^{\frac{n}{2}})^2 \text{ pour } n \text{ pair,} \\ x^n &= x^{n-1} \cdot x \text{ pour } n \text{ impair.} \end{aligned}$$

Exercice 9.- (**Fonction d'Ackermann**)

Écrire une fonction C permettant de calculer l'application A définie sur N^2 de la façon suivante :

$$\begin{aligned} A(0, n) &= n + 1, \\ A(m, 0) &= A(m - 1, 1) \text{ pour } m \text{ non nul,} \\ A(m, n) &= A(m - 1, A(m, n - 1)) \text{ pour } m \text{ et } n \text{ non nuls.} \end{aligned}$$

6.1.3 Le problème de l'arrêt

Introduction.- Il ne faut pas croire que tout programme syntaxiquement correct avec des appels récursifs permet de définir une application. Considérons, par exemple, la pseudo-définition suivante :

$$\text{IDIOT}(n) = \text{IDIOT}(n+1) + 1.$$

Il est évident qu'un programme associé à cette pseudo-définition va boucler quelle que soit la valeur de n .

Une pseudo-définition par récurrence étant donnée, il faut donc démontrer qu'elle définit bien une application. Ceci revient à démontrer que, quelles que soient les valeurs des arguments, le programme s'arrêtera. Ce n'est pas du tout facile pour des programmes assez compliqués.

Exercice.- Montrer que la fonction d'Ackermann définie ci-dessus est bien une application (définie sur N^2 en entier).

Prospective.- Une étude plus approfondie du problème de la justification des définitions par récurrence fait l'objet d'un cours plus avancé, comme nous l'avons déjà dit.

6.1.4 Inversion d'une phrase

Le problème.- On introduit une phrase au clavier se terminant par un point (et ne contenant pas d'autre point que celui-ci). Le programme doit écrire cette phrase à l'écran dans l'ordre inverse.

Une session d'utilisation de ce programme sera, par exemple :

```
Écrire une phrase :
Ceci est un exemple de phrase.
Cette phrase dans l'ordre inverse est :
.esarhp ed elpmexe nu tse iceC
```

Un algorithme itératif.- Il suffit de considérer une phrase comme un tableau de caractères et d'énumérer les éléments de ce tableau dans l'ordre inverse.

Un algorithme récursif.- On lit le premier caractère de cette phrase. Si c'est un point on l'affiche et c'est terminé. Sinon on fait appel à notre procédure pour la fin de la phrase (qui est un mot plus court) puis on affiche ce caractère et ce sera alors terminé.

Un programme récursif.- Cet algorithme conduit au programme suivant pour un compilateur C pour MS-DOS :

```
/* Programme inverse.c */

#include <stdio.h>
#include <conio.h> /* non portable */

void inverse(void)
{
    char C;
    C = getche();
    if (C != '.') inverse();
    else
        printf("\nCette phrase dans l'ordre inverse est :\n");
    putchar(C);
}

void main(void)
{
    printf("Ecrire une phrase :\n");
    inverse();
}
```

6.1.5 Le tri-fusion

Introduction.- Nous avons déjà vu l'intérêt du tri de tableaux et un certain nombre d'algorithmes naturels pour résoudre ce problème. Nous allons voir ici une autre méthode de tri, le **tri-fusion** (*merge sort* en anglais).

Principe du tri-fusion

L'idée de départ est simple : on sépare le tableau en deux parties égales (à un élément près si le nombre d'éléments n'est pas pair), on trie la première partie, on trie la deuxième partie, puis on fusionne ces deux parties triées, c'est-à-dire qu'on prend le plus petit élément du premier sous-tableau, le plus petit élément du deuxième sous-tableau et on place le plus petit des deux dans le nouveau tableau et on recommence ainsi de suite. Bien entendu pour trier les deux parties on fait de même, ce qui conduit à un algorithme récursif. Pour qu'il n'y ait pas de problème d'arrêt il suffit de remarquer qu'un tableau à un seul élément est nécessairement trié.

Première partie de l'algorithme

Ceci nous conduit, par exemple, à la fonction suivante, faisant appel à une fonction de fusion, qui trie la partie de tableau comprise entre les indices a et b inclus [on ne le fait pas seulement entre 1 et n à cause de l'appel récursif] :

```
/* Fusion.c */

#include <stdio.h>

const int max = 100;
void tri_fusion (int TA[ ], int a, int b )
{
    int c;
    if (a != b)
    {
        c = (a + b)/2;
        tri_fusion(TA, a, c);
        tri_fusion(TA, c+1, b);
        fusion(TA, a, c, b);
    }
}
```

Principe de la fusion

Reste à écrire la fonction de fusion. Cette fonction a pour arguments un tableau TA, d'éléments numérotés de 1 à n , et des entiers a, b, c tels que $1 \leq a \leq c < b \leq n$, les parties du tableau entre a et c d'une part, $c + 1$ et b d'autre part, étant triées. Cette fonction doit changer la valeur du tableau pour que la partie comprise entre a et b soit triée.

L'idée est de considérer ces deux parties triées comme des tas de cartes triées. On prend la carte du dessus dans chacun des tas, on les compare, on met celle de valeur la plus petite dans un autre tas, on prend une nouvelle carte (s'il en reste) dans le tas auquel elle appartenait, et on continue ainsi de suite.

Deuxième partie de l'algorithme.- Ceci nous conduit, par exemple, à la fonction suivante:

```
void fusion(int TA[ ], int a, int c, int b)
{
    int TB[max];      /* tas auxiliaire */
    int i, j, k, l;   /* les indices respectifs de la premiere
                       partie, de la deuxieme partie et du tas auxiliaire */
    i = a;
    j = c+1;
    k = a;
    while((i <= c) && (j <= b))
    {
        if (TA[i] < TA[j])
        {
            TB[k] = TA[i];
            i++;
        }
        else
        {
            TB[k] = TA[j];
            j++;
        }
        k++;
    };
    /* Si le deuxieme tas n'est pas vide */
    if (j <= b)
        for (l = j; l <= b; l++) TB[l] = TA[l];
    /* Si le premier tas n'est pas vide */
    if (i <= c)
        for (l = i; l <= c; l++)
        {
            TB[k] = TA[l];
            k++;
        }
    /* On copie le tableau auxiliaire dans TA */
    for (k = a; k <= b; k++) TA[k] = TB[k];
}
```

Un programme complet

Ceci nous conduit au programme complet suivant nous permettant de tester nos fonctions :

```
/* Fusion.c */
#include <stdio.h>
const int max = 100;

void fusion(int TA[ ], int a, int c, int b);

void tri_fusion (int TA[ ], int a, int b );

void main(void)
{
    int TA[max];
    int n, i;
    /* Nombre effectif d'elements */
    printf("Entrer le nombre d'elements du tableau : ");
    scanf("%d", &n);
    /* Saisie du tableau */
    for (i = 1; i <= n; i++)
    {
        printf("element numero %d : ", i);
        scanf("%d", &(TA[i]));
    }
    /* Tri */
    tri_fusion(TA, 1, n);
    /* Affichage */
    printf("Les elements sont, dans l'ordre croissant : ");
    for (i = 1; i <= n; i++) printf(" %d", TA[i]);
    putchar('\n');
}
```

6.1.6 Le problème des tours de Hanoï

Le problème

On dispose de trois plots et de 64 disques tous de rayons différents, percés en leur centre de façon à passer à travers les plots.

Au départ les 64 disques sont sur le premier plot, rangés par taille, le plus grand tout en bas. Le but est de déplacer ces disques pour les amener sur le troisième plot en suivant les règles suivantes :

- on ne peut déplacer qu'un disque à la fois ;
- à chaque instant et sur chaque plot, un disque ne peut être placé qu'au-dessus d'un disque de rayon plus grand.

Un algorithme récursif

L'algorithme.- Commençons par généraliser le problème au cas de n disques, le problème initial revenant à prendre n égal à 64.

Pour $n = 1$ c'est facile, il n'y a même pas à utiliser le plot du milieu : on déplace directement le disque du premier au troisième plot.

Supposons que l'on sache résoudre le problème pour une valeur n et montrons qu'on peut alors le résoudre pour la valeur $n + 1$. En échangeant les rôles des plots 2 et 3, on peut déplacer n disques du plot 1 vers le plot 2, en utilisant le plot 3 comme plot auxiliaire. Déplaçons alors le disque restant (le plus grand) du plot 1 vers le plot 3. En échangeant le rôle des plots 1 et 2, on peut finalement déplacer les n disques du plot 2 vers le plot 3.

Précisions sur le cahier des charges.- Écrivons un programme qui, étant donnée l'entrée n , entier naturel non nul, nous donne la liste des déplacements à effectuer pour résoudre le problème, sous la forme :

déplacer un disque du plot 1 vers le plot 2,

le disque en question étant le plus haut sur le plot.

Nous verrons tout à l'heure pourquoi nous gardons le problème sous sa forme généralisée plutôt que dans le cas particulier $n = 64$.

La fonction fondamentale.- Nous avons vraiment intérêt à introduire la fonction `DEPLACE(N, P, Q, R)` qui consiste à déplacer N disques du plot P vers le plot R , en se servant du plot Q comme plot intermédiaire.

Implémentation

Un programme.- Ceci nous conduit au programme suivant :

```
/* hanoi.c */
#include <stdio.h>

void deplace(int n, int P, int Q, int R)
{
    if (n == 1)
    {
        printf("Déplacer un disque du plot %d", P);
        printf(" vers le plot %d\n", R);
    }
    else
    {
        deplace(n-1, P, R, Q);
        deplace(1, P, Q, R);
        deplace(n-1, Q, P, R);
    }
}

void main(void)
{
    int n;
    printf("Entrer le nombre de disques : ");
    scanf("%d", &n);
    deplace(n, 1, 2, 3);
}
```

Un exemple de session.- Pour $n = 3$ on obtient :

```
Entrer le nombre de disques : 3
Déplacer un disque du plot 1 vers le plot 3
Déplacer un disque du plot 1 vers le plot 2
Déplacer un disque du plot 3 vers le plot 2
Déplacer un disque du plot 1 vers le plot 3
Déplacer un disque du plot 2 vers le plot 1
Déplacer un disque du plot 2 vers le plot 3
Déplacer un disque du plot 1 vers le plot 3
```

Complexité

Évaluation de la complexité.- Soit $d(n)$ le nombre de déplacements de disques nécessaire pour résoudre le problème des tours de Hanoï à n tours, suivant notre algorithme. On a évidemment $d(1) = 1$ et $d(n + 1) = 2.d(n) + 1$. Il n'est pas difficile de démontrer par récurrence que :

$$d(n) = 2^n - 1.$$

Exercice.- 1°) En supposant que l'on déplace un disque par seconde, quel est le temps nécessaire pour résoudre le problème initial?

2°) Même question si on considère que le déplacement d'un disque est une opération élémentaire sur le plus gros ordinateur actuel.

3°) En supposant qu'on puisse écrire 40 lignes sur une page, combien faut-il de pages pour imprimer le résultat? Quelle sera la hauteur du listing, en supposant que 200 feuilles ont une épaisseur d'un centimètre?

6.2 Récursivité croisée

Introduction.- La possibilité de déclarer une fonction sans la définir prend tout son intérêt à propos de la récursivité croisée. En effet une fonction ne peut être utilisée qu'à condition qu'elle ait été définie (ou déclarée) or, dans le cas de la récursivité croisée, on ne pourrait pas s'en sortir juste à l'aide des définitions.

Un exemple.- Considérons les trois suites d'entiers naturels $(u_n)_{n \in \mathbb{N}}$, $(v_n)_{n \in \mathbb{N}}$ et $(w_n)_{n \in \mathbb{N}}$ définies par récurrence simultanée de la façon suivante :

$$\begin{aligned} u_0 &= 1, v_0 = 2, w_0 = 3, \\ u_{n+1} &= 2.u_n + 3.v_n + w_n, \\ v_{n+1} &= u_n + v_n + 2.w_n, \\ w_{n+1} &= u_n + 4.v_n + w_n. \end{aligned}$$

Nous voudrions un programme qui demande une valeur de l'entier naturel n et affiche alors les valeurs de u_n , de v_n et de w_n .

Exercice.- Écrire un programme C itératif qui effectue ce qui est demandé.

Un programme récursif.- Un programme récursif est, dans ce cas, plus naturel qu'un programme itératif. On a, par exemple :

```
/* Programme suite.c */
#include <stdio.h>

int W(int n)
{
    if (n == 0) return 3;
    else return(U(n-1) + 4*V(n-1) + W(n-1));
}

int U(int n)
{
    if (n == 0) return 1;
    else return(2*U(n-1) + 3*V(n-1) + W(n-1));
}

int V(int n)
{
    if (n == 0) return 2;
    else return(U(n-1) + V(n-1) + 2*W(n-1));
}

void main(void)
{
    int n;
    printf("Entrer un entier : n = ");
    scanf("%d",&n);
    printf("\nu(n) = %d, v(n) = %d, w(n) = %d\n",
           U(n), V(n), W(n));
}
```

Historique

La récursivité n'était pas disponible dans les premiers langages (évolués). Bien entendu elle peut être émulée dans tout langage. Le premier langage dans lequel la récursivité est émulée régulièrement est le langage IPL de Newell, Shaw et Simon.

Le premier langage à fournir un mécanisme automatique pour la récursivité est le langage LISP (fin des années 1950). Algol 60 et ses successeurs (Pascal, C, Modula-2 et ADA) permettent tous la récursivité, comme beaucoup de langages modernes.

McCARTHY, John & al., **Lisp 1.5 Programmer Manual**, M.I.T. Press, 1962.

Newell, A. (Ed.), **Information Processing Language-V Manual**, Prentice-Hall, 1961.