1

RÉCURSIVITÉ

1.1 INTRODUCTION

L'itérativité en algorithmique est un concept qui permet la mise en œuvre d'un bloc d'instructions qui doit s'exécuter d'une manière itérée.

Dans certains cas, la définition du problème à résoudre se décrit par récurrence, tels que les fonctions récursives en mathématiques. Par conséquent, la récursivité en algorithmique permet de résoudre certains problèmes d'une manière très simple et directe, alors que la résolution itérative, nécessite beaucoup plus de travail et de structures de données intermédiaires.

Dans ce chapitre nous introduisons le principe de la récursivité, le passage réciproque récursive-itérative, et une comparaison entre les deux versions. Des figures illustratives ont été intégrées à chaque étape.

Ce chapitre comporte quatre sections qui décrivent respectivement, la construction d'un algorithme récursif, l'environnement d'une fonction récursive, le fonctionnement de la récursivité, le passage réciproque récursivité-itérativité et quelques exemples d'application.

1.2 CONSTRUCTION D'UN ALGORITHME RÉCURSIF

L'algorithme récursif peut être défini comme suit :

Définition 1.2.1 (algorithme récursif) *Un algorithme récursif est un algorithme qui s'appelle lui même.*

Concevoir un algorithme récursif est un peu comme définir une suite par récurrence en mathématiques, il faudra :

1. Un ou plusieurs cas de base, dans lequel (lesquels) l'algorithme ne fait pas appel à lui même, sinon l'algorithme ne peut pas s'arrêter.
2. Un ou plusieurs cas inductif(s), dans lequel (lesquels) l'algorithme fait appel à lui même.

Chaque appel récursif doit, en principe, se rapprocher d'un cas de base, de façon à permettre la terminaison du programme.

Exemple 1.2.2 (calcul de la factorielle) *On rappelle que la factorielle d'un nombre n se définit comme suit : $n! = 1 * 2 * 3... * n$.*

La solution itérative qui calcule la factorielle de n s'exprime via la fonction « fact » comme suit :

La version itérative :

```
int fact(int n)
{
   if (n==0) return 1;
   else {
   int p=1;
   for (int i=1; i<n; i++)
     p=p*i ;
   return p;
 }
}
```

La version récursive :

L'écriture de n! sous forme d'une suite par récurrence, notée « fact » se donne comme suit :

$$\begin{cases} fact_0 = 1 \\ fact_n = n * fact_{(n-1)} \end{cases}$$

Ce qui se traduit, d'une manière directe, par la fonction récursive nommée ***int fact(int)*** *en C++ ci-dessous :*

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

L'arbre d'exécution de la fonction « fat » avec le paramètre effectif d'entrée « 3 », se donne comme suit :

Comme nous pouvons le constater via la figure 20, le parcours d'un arbre d'exécution suit un ordre postfixe pour calculer la valeur de « fact(3) ».

1.3 ENVIRONNEMENT D'UNE FONCTION RÉCURSIVE

1.3.1 *Environnement local*

Chaque appel récursif dispose de ses propres variables locales. Elles sont déclarées à chaque appel (voir l'exemple ci-dessous1.3.1), ceci est également vrai pour les paramètres locaux de la fonction.

Les variables de portée locale sont des variables propres à chaque appel récursif de la fonction.

Exemple 1.3.1 (fonction *cmb* **avec variables locales)** *Soit la fonction récursive cmb ci-dessous :*

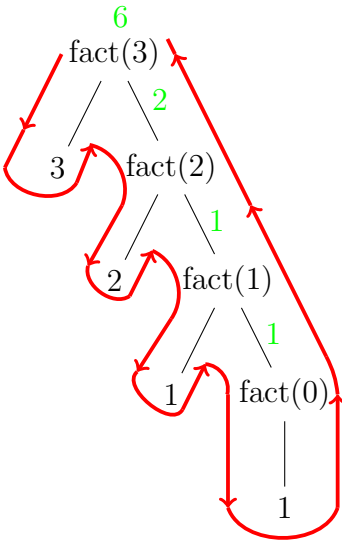


FIGURE 20 – Arbre d'exécution de $fact(3)$

```
int cmb(int n, int p)
{
  int a,b;
  if ((p==0)||(p==n)) return 1;
  else
        {
          a=cmb(n-1,p-1);
          b=cmb(n-1,p);
          return a+b;
        }
}
```

Les variables a et b sont de portée locale, elles sont donc propres à chaque appel. En d'autres termes, un nouvel espace mémoire est réservé aux deux variables à chaque appel récursif de la fonction. Ce qui donne l'arbre d'exécution illustré dans la figure 21.

1.3.2 *Environnement global*

Dans le cas où la variable est partagée par tous les appels récursifs d'une fonction, on parle de variables globales. Ces variables doivent être déclarées à l'extérieur de la fonction.

Exemple 1.3.2 (fonction *cmb* avec variables globales) *Soit la fonction cmb citée dans l'exemple 1.3.1 avec les variables a et b déclarées globales, ce qui donne le code suivant :*

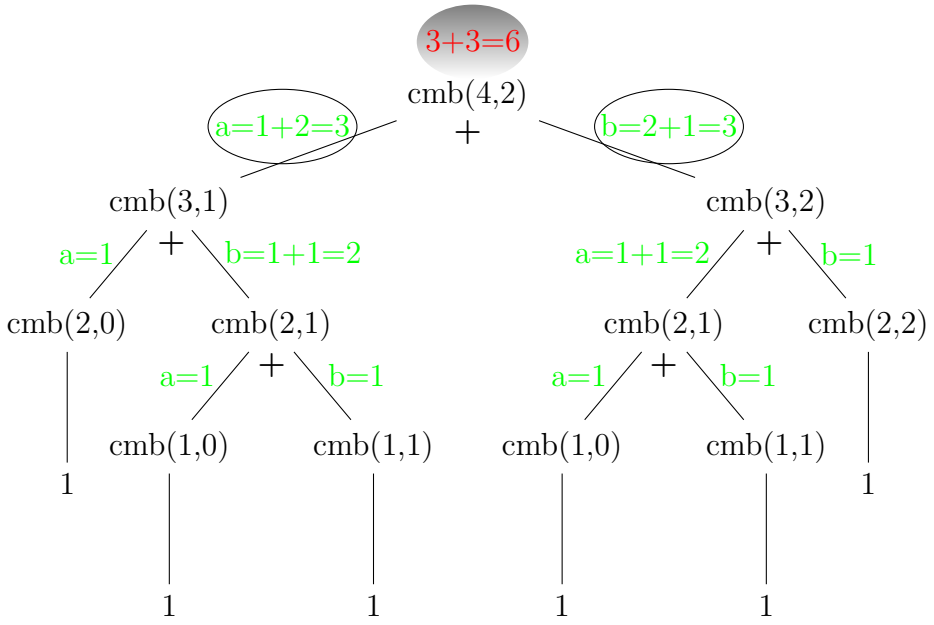


FIGURE 21 – Arbre d'exécution de $cmb(4,2)$ avec a et b variables locales

```
int a,b;
int cmb(int n, int p)
{
  if ((p==0)||(p==n)) return 1;
  else
        {
          a=cmb(n-1,p-1);
          b=cmb(n-1,p);
          return a+b;
        }
}
```

L'arbre d'exécution de $cmb(4,2)$ *est illustré dans la figure 22.*
Les valeurs successives affectées aux variables a et b se donnent comme suit :

$a = 1 \to 1 \to 3 \to 1 \to 2$

$b = 1 \to 2 \to 1 \to 1 \to 3$

ce qui donne en fin de compte la valeur de « cmb(4,2) » qui est égale à $a + b = 5$.

Nous constatons que la valeur de $cmb(4,2) = 5$ *avec a et b globales, alors que* $cmb(4,2) = 6$ *avec a et b locales (voir l'exemple 1.3.1).*

1.4 FONCTIONNEMENT DE LA RÉCURSIVITÉ

L'un des inconvénients majeurs de la récursivité est sa consommation en espace mémoire. Ceci est du à l'utilisation d'une pile d'exécution 1.4.1, une structure dont la mémoire réservée augmente à chaque nouvel appel récursif de la fonction.

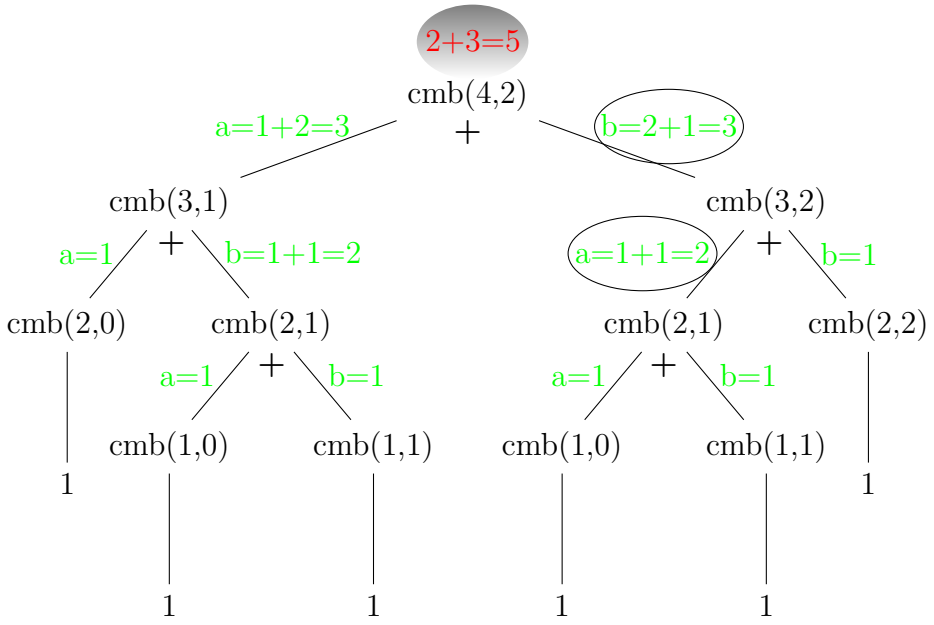


FIGURE 22 – Arbre d'exécution de $cmb(4,2)$ avec les variables a et b globales

Définition 1.4.1 (pile d'exécution) *La pile d'exécution du programme en cours est un emplacement mémoire destiner à mémoriser les paramètres, les variables locales ainsi que les adresses de retour des fonctions en cours d'exécution.*

Lors de l'exécution d'un programme, cette pile permet de :

1. mémoriser le contexte appelant lors de chaque appel de la fonction (adresses des variables, adresse de la prochaine instruction à exécuter en sortant de la fonction, etc.). A ce niveau, le nouveau contexte est empilé dans la pile d'exécution.
2. au retour, ou en atteignant le cas de base, le contexte est dépilé.

En d'autres termes, tant que le cas de base n'est pas atteint la taille de la pile d'exécution augmente.

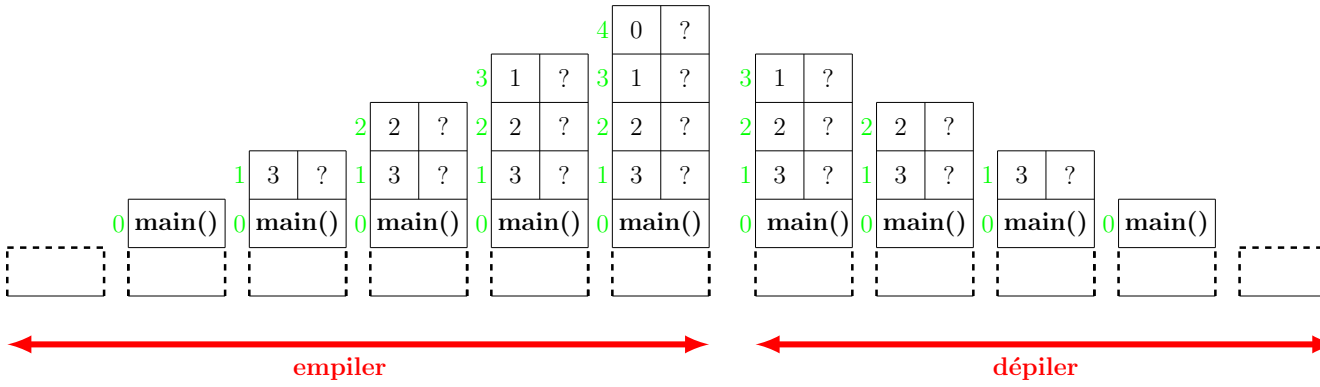
Exemple 1.4.2 (pile d'exécution de *fact(3)*) *Soit la fonction récursive « fact » définie précédemment. On donne ci-dessous la table d'exécution 6 de la fonction $fact(3)$, suivie de la pile d'exécution illustrée dans la figure 23.*

La table 6 [GREFFIER, 2007] est la table d'exécution s'apprêtant à l'exécution de « fact(3) ». Comme nous pouvons le constater à travers l'arbre d'exécution (voir la figure 20), nous avons quatre appels de la fonction « fact » : fact(3), fact(2), fact(1) puis fact(0). A chaque appel, le contexte correspondant est empilé, jusqu'à ce qu'on atteint le cas de base ($n = 0$), pour dépiler les contextes dans l'ordre inverse de l'empilement : fact(0), fact(1), fact(2) puis fact(3). L'opération de multiplication est réalisée à chaque fois que le contexte est dépilé.

TABLE 6 – Pile d'exécution de $fact(3)$

Num d'appel	Num Retour	Contexte	Action
1 (empiler)		n=3	n*fact(2)
2 (empiler)		n=2	n*fact(1)
3 (empiler)		n=1	n*fact(0)
4 (empiler)		n=0	return(1)
	(dépiler)→ 4	n=0	return(1)
	(dépiler)→ 3	n=1	return(1*1)
	(dépiler)→ 2	n=1	return(2*1)
	(dépiler)→ 1	n=1	return(3*2)
	(dépiler)	Pile Vide	Retour au point appelant

Les actions « empiler » 23a et « dépiler » 23b réalisées respectivement via l'appel récursif et le cas de base sont illustrées dans la figure 23.



(a) La pile d'exécution au cours d'empilement

(b) La pile d'exécution au cours du dépilement

FIGURE 23 – Pile d'exécution au cours d'exécution de $fact(3)$

1.5 LE PASSAGE ALGORITHME RÉCURSIF-ALGORITHME ITÉRATIF

L'inconvénient majeur des algorithmes récursifs est le problème d'encombrement mémoire. Par conséquent, on essaie de diminuer la taille de la pile d'exécution en utilisant des algorithmes itératifs.

Pour écrire un algorithme équivalent, on doit gérer dans l'algorithme la pile des sauvegardes (ne garder que celles utiles). En d'autres termes, « *Transformer un algorithme récursif en une version itérative équivalente dans le but de diminuer la taille de la pile d'exécution.* »

Malheureusement, il n'existe pas de règle générale pour passer d'une version récursive vers une version itérative et vice versa. Dans la littérature, nous ne trouvons que des cas particuliers de fonctions récursives, dont le passage suit une forme bien définie.

Nous introduisons dans cette section les cas suivants :

— un seul appel récursif terminal,
— un seul appel récursif non terminal,

Lorsque la fonction récursive comporte plus d'un appel récursif (deux ou plus), l'écriture de la version itérative devint très difficile à réaliser.

1.5.1 *Un seul appel récursif terminal*

Définition 1.5.1 (fonction récursive terminale) *Un algorithme est dit récursif terminal si aucun traitement n'est effectué à la remontée d'un appel récursif (sauf en cas de retour de la valeur).*

Généralement, une fonction récursive terminale suit la structure ci-dessous :

```
void fRec1(T x){
      if (c(x)) A0(x);
      else {
              A1(x);
              fRec1(F1(x));
             }
      }
```

La version itérative de $fRec1$ se donne directement via la fonction $fIter1$ comme suit :

```
void fIter1(T x)
{
      while(!c(x)){
       A1(x);
      }
      A0(x);
}
```

Exemple 3.5.2 (transformation de la fonction « enumerer ») *Soit la fonction récursive enumerer :*

```
void enumerer(int x)
{
      if (n<=0) cout << "fin";
      else {
              cout << n;;
              enumerer(n-1);
             }
}
```

La version itérative équivalente se donne comme suit :

```
void enumerer(int x){
      while(n>0) {
          cout << n;
          n=n-1;
        }
        cout << "fin";
      }
```

1.5.2 *Un seul appel récursif non-terminal*

Définition 1.5.3 (algorithme récursif non terminal) *Un algorithme est dit récursif non terminal si un traitement est effectué à la remontée d'un appel récursif.*

Généralement, une fonction récursive non-terminale suit la structure suivante :

```
T1 fRec2(T2 x){
      if (c(x)) return F(x);
      else return (u(G(x), fRec2(H(x))));
}
```

La version itérative et équivalente à $fRec2$ se donne par la fonction $fIter2$ comme suit :

```
T1 fIter2(T2 x){
     ... p;
     if (c(x)) return F(x);
      else {
                p=G(x); x=H(x);
                while(!c(x)) {
                    p=u(p,G(x));
                    x=H(x)
              }
           return u(p,F(x));
        }
}
```

Exemple 1.5.4 (transformtion de la fonction « fact ») *Soit la fonction récursive fact :*

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

La version itérative équivalente :

```
int fonct(int n){
      if (n==0) return 1;
      else {
                int p=n; n=n-1;
                while(n!=0) {
                    p=n*p;
                    n=n-1;
                }
            return p*n;
        }
    }
```

1.6 EXEMPLES D'APPLICATION

1.6.1 *Calcul du Plus Grand Commun Diviseur (PGCD)*

Soit la définition suivante :

```
PGCD(p, q) = si p = 0 alors q
             sinon si q = 0 alors p
                   sinon si q = p alors PGCD(p-q, q)
                         sinon PGCD(p, q-p)
```

Avant de travailler sur la solution algorithmique du calcul du *PGCD*, nous commençons par donner l'écriture mathématique correspondante :

$$PGCD : \mathbb{N}^2 \longrightarrow \mathbb{N}$$

$$(p,q) \longmapsto PGCD(p,q) = \begin{cases} q & si \quad p = 0 \\ p & si \quad q = 0 \\ PGCD(p-q,q) & si \quad q \leq p \\ PGCD(p,q-p) & si \quad q > p \end{cases}$$

L'ensemble de départ $\mathbb{N}^2$ exprime le nombre des paramètres d'entrée qui est de 2 et leur type qui est le *int* (ou *unsigned int*). L'ensemble d'arrivée $\mathbb{N}$ exprime le type de retour de la fonction *PGCD*. Ce qui donne le prototype : *int PGCD(int, int)* (ou l'entête *int PGCD(int p, int q)*).

Concernant le corps de la fonction, nous réalisons une traduction brut de la partie

$$PGCD(p,q) = \begin{cases} q & si \quad p = 0 \\ p & si \quad q = 0 \\ PGCD(p-q,q) & si \quad q \leq p \\ PGCD(p,q-p) & si \quad q > p \end{cases}$$

dans le langage algorithmique (en C++). Ce qui donne la fonction récursive *PGCD* suivante :

```
int PGCD(int p, int q)
{
        if  (p == 0) return q;
        else if (q == 0) return p;
                else if (q <= p)  return PGCD(p-q, q);
                        else return PGCD(p, q-p);
}
```

1.6.2 *La suite de Fibonacci*

La suite de *Fibonacci* se donne comme suit :

$$\begin{cases} Fibo_1 = Fibo_2 = 1 \\ Fibo_{n+2} = Fibo_{n+1} + Fibo_n \end{cases}$$

En passant par l'écriture de la suite sous forme d'une fonction mathématique nommée *Fibo* :

$$Fibo : \mathbb{N}^* \longrightarrow \mathbb{N}^*$$
$$n \longmapsto Fibo(n) = \begin{cases} 1 & \text{si n=1,2} \\ Fibo(n-1) + Fibo(n-2) & sinon \end{cases}$$

Les ensembles de départ et d'arrivée donne respectivement, des informations sur les paramètres d'entrées et le type de retour. Le prototype de la fonction correspond à *int Fibo(int)* (ou l'en-tête *int Fibo(int n)*).

La fonction récursive se donne en C++ comme suit :

```
int Fibo(int n) {
 if  (n==1 || n==2) return 1;
 else return Fibo(n-2)+Fibo(n-1);
}
```