Chapitre 7 : La Récursivité

7.1 Définitions :

7.1.1 Objet récursif :

Un objet est dit récursif s'il est utilisé directement ou indirectement dans sa définition.

Exemple :
En définissant une expression arithmétique <expr> ou un identificateur <idf> ou une constante <cste> nous donnons une définition récursive comme suit :
Si θ est un opérateur on aura : <expr> → <expr> θ <expr> / <idf> / <cste>.

7.1.2 Programmation récursive :

La programmation récursive est une technique de programmation qui remplace les instructions de boucle (while, for, etc.) par des appels de fonctions.

7.1.3 Action paramétrée récursive :

Une action paramétrée P est dite récursive si son exécution provoque ou entraîne un ou plusieurs appels à P. Ces appels sont dits récursifs.

7.1.4 Algorithme récursif :

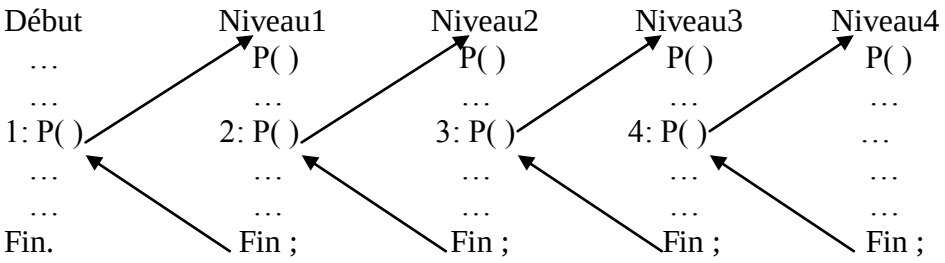
Un algorithme récursif est un algorithme qui contient une ou plusieurs actions paramétrées récursives.

7.1.5 Auto-imbrication :

L'auto-imbrication est le fait qu'une action paramétrée P peut s'appeler elle même avant que sa première exécution ne soit terminée, la seconde exécution peut de nouveau faire appel à P et ainsi de suite.

Exemple :
Soit l'action paramétrée suivante :

```
Action P( )
 Début
   P( );
 Fin ;
```



Chaque appel à P se fait à un niveau donné. L'exécution de P au niveau i se termine avant l'exécution de P au niveau i-1.
Remarque : Le nombre de niveaux doit être fini quelque soit les valeurs des paramètres d'appel, autrement l'algorithme va boucler indéfiniment.

7.2 Principes de construction d'algorithmes récursifs :

Exemple :

Le calcul de la valeur factorielle d'un nombre donné n (n>=0) peut se faire de deux manières différentes :

1ère méthode :

n ! = 1*2*3*…*n-1*n sachant que 0 !=1

On obtient alors l'algorithme itératif (classique) donné par la fonction suivante :

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

2ème méthode :

0!=1 ; 1!=1 ; 2!=1*2 ; 3!=1*2*3 ; 4!=1*2*3*4 ; 5!=1*2*3*4*5 …
=2 =6 =24 =120 …

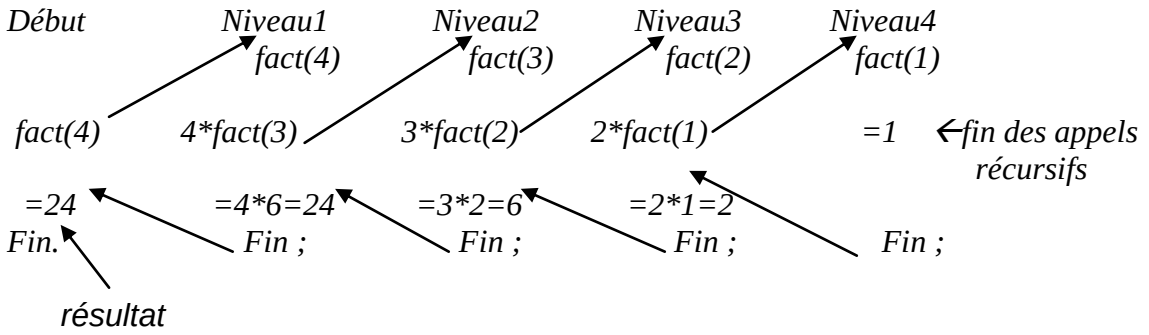
Nous remarquons que:
0!=1 ; 1!= 0!*1 ; 2!=1!*2 ; 3!= 2!*3 ; 4!=3!*4 ; 5!=4!*5

De façon générale pour un n donné n>0 on a : n! = (n-1) !*n
On constate donc la définition de n! est récursive, puisqu'elle se réfère à elle même quand elle applique (n-1)!

Le calcul de la valeur factorielle d'un nombre est défini par :
a) Si n=0 ou n=1 alors n !=1
b) Si n>0 alors n != (n-1) !*n

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

Déroulement de fact (4)



Remarques :

a) Quand n=0, la valeur de n ! est donnée directement. 0 est appelé ***valeur de base***.

b) Pour un n≠0 donné, la valeur de n ! est définie en fonction d'une valeur plus petite que n et plus voisine de la valeur de base 0.

c) La variable n, est testée à chaque fois pour savoir s'il faut exécuter
Si n=0 alors n !=1
ou Si n>0 alors n != (n-1) !*n
n est appelé ***variable de commande***

Les principes de construction d'actions paramétrées récursives sont donc :

- Le nombre d'appels récursifs (niveaux) doit être fini. Il faut donc que les paramètres contiennent une ou plusieurs variables de commande qui sont testées à chaque niveau pour savoir si on doit continuer ou non les appels récursifs.
- Déterminer le ou les cas particuliers qui sont exécutées directement sans appels récursifs. Dans ces cas, les variables de commandes sont égales aux valeurs de base (n=0).
- Décomposer le problème initial en sous problèmes de même nature, telle que des décompositions successives aboutissent toujours à l'un des cas particuliers.
- Le principe de la récursivité est que les appels récursifs doivent être uniquement sur des données plus petites n != n* (n-1) !
Plus petit que n !

7.3 Schémas généraux d'actions récursives :

1er schéma :

```
Action P( )
  Début
    Si < condition> alors  P( )
                    sinon Q ;
  Fin ;
```

2ème schéma :

```
Action P( )
  Début
     Tant que  < condition> faire  P( )
     Q ;
  Fin ;
```

Où Q permet la résolution directe du problème (ne contient pas d'appel récursif).

7.4 Récursivité directe et récursivité indirecte :

Une action qui fait appel à elle-même explicitement dans sa définition est dite récursive directement ou récursivité simple.

Si une action A fait référence (ou appel) à une action B qui elle fait appel directement ou indirectement à A, on parle alors de récursivité indirecte ou récursivité croisée.

```
Action A( )
 Début
   Si < condition> alors  B( )
                  sinon QA ;
 Fin ;
```

```
Action B( )
  Début
   Si < condition> alors  A( )
                  sinon QB ;
  Fin ;
```

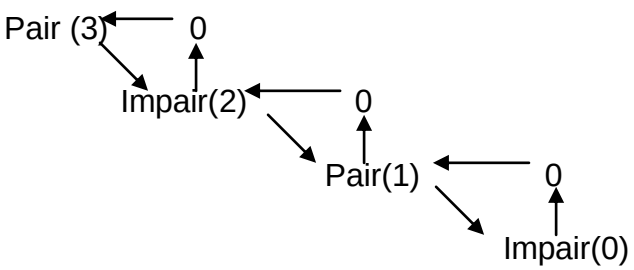
Exemple :

- Récursivité directe : <expr> → <expr> θ <expr>
- Récursivité indirecte :
  <expr> → <terme> θ <terme>
  <terme> → <expr>

- Pour construire une définition récursive de la parité, remarquons tout d'abord que 0 est un entier pair, et qu'il n'est pas impair. Ensuite remarquons que un entier n est pair (resp. impair) ssi l'entier n-1 est impair (resp. pair).

```
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);
}
```



Résumé :

```
Action récursive (paramètres)
  <Déclaration des variables locales> ;
Début
  Si (Test d'arrêt) alors < instructions du point d'arrêt >
  Sinon
      instructions ;
      récursive(paramètres changés) /* appels récursifs */
      instructions ;
   Fsi ;
Fin ;
```

7.5 Différents types de récursivité :

a) **Récursivité simple :** une fonction simplement récursive, c'est une fonction qui s'appelle elle-même une seule fois, comme c'était le cas pour la fonction factorielle.

b) **Récursivité multiples :** une fonction peut exécuter plusieurs appels récursifs – typiquement deux parfois plus.

Exemple :
```
void afficheMotRec (chaine TabMot[ ], int n, int  i)
{ if (i<n)  { afficheMotRec(TabMot, n, i+1) ;
              printf("%s", TabMot[i]);
              afficheMotRec(TabMot, n, i+1);
            }
}
```

c) **Récursivité à droite :**

Si l'exécution d'un appel récursif n'est jamais suivie par l'exécution d'une autre instruction, cet appel est dit récursif à droite ou encore appelée ***récursivité terminale***. L'exécution d'un tel appel termine l'exécution de l'action et ***ne nécessite pas une pile***.
Une fonction récursive ***non terminale nécessite une pile***.

Exemple :
```
void afficheMotRec1 (chaine TabMot[ ], int n, int  i)
{ if (i<n)  { printf("%s", TabMot[i]);
              afficheMotRec(TabMot, n, i+1);
            }
} /* afficheMotRec1 est une fonction récursive terminale */

void afficheMotRec2 (chaine TabMot[ ], int n, int  i)
{ if (i<n)  { afficheMotRec(TabMot, n, i+1) ;
              printf("%s", TabMot[i]);
            }
} /* afficheMotRec2 est fonction récursive non terminale */
```

7.6 Fonctionnement de la récursivité

Un programme ne peut s'exécuter que s'il est chargé en mémoire centrale, chaque instruction du programme se trouve à une adresse donnée de la mémoire.
Lorsqu'un programme fait appel à une fonction, le système sauvegarde l'adresse de retour (adresse de l'instruction qui suit l'appel), ainsi que les valeurs des variables locales.
Quand une fonction ***f*** appelle une fonction ***g,*** on doit sauvegarder l'adresse de retour de ***f*** (paramètres et variables locales) avant l'appel de ***g***, ce contexte doit être récupéré après le retour de ***g***.

S'il y a plusieurs appels imbriqués, le système gère ***une pile*** pour sauvegarder (empiler) les différents contextes des différents appels récursifs.

Les paramètres de l'appel récursif changent. A chaque appel les variables locales sont stockées dans une pile. Ensuite les paramètres ainsi que les variables locales sont désempilées au fur et à mesure qu'on remonte les niveaux.
Lors de l'i^{ème} appel sont empilés :

- Les valeurs des paramètres au niveau i
- Les valeurs des variables locales du niveau i
- L'adresse de retour au niveau i

A la fin des appels récursifs retour du niveau i+1 au niveau i :

- Retour au programme principal si la pile est vide
- Dépiler l'adresse de retour
- Dépiler le contexte du niveau i (les valeurs des variables du niveau i
- Exécuter l'instruction suivant le dernier appel

7.6 Elimination de la récursivité :

La récursivité ***simplifie la structure d'un programme*** mais la plupart du temps, le gain en simplicité vaut une baisse relative des performances d'exécution.
La récursivité est souvent couteuse en temps et en espace mémoire car elle nécessite l'emploi de techniques spéciales de compilation, à savoir ***le concept de pile***.

Ces techniques sont généralement plus ***coûteuses en temps d'exécution*** que celles fondées sur l'itération. Aussi certains langages de programmation n'admettent pas la récursivité (exemple : Fortran). Ainsi il arrive que l'on souhaite éliminer la récursivité.

A cet effet il est intéressant de noter que l'on peut montrer que, si le langage de programmation utilisé le permet, il est toujours possible de transformer une action itérative en une action récursive ; cependant, la réciproque n'est pas vraie.

Les problèmes qu'il faut résoudre en utilisant la récursivité sont les problèmes ***typiquement récursifs*** et non itératifs, c'est-à-dire, soit des problèmes qui ne peuvent pas être résolus de façon itérative, soit des problèmes pour lesquels une ***formulation*** récursive est particulièrement ***simple et naturelle.***

D'une manière générale, on évitera donc d'utiliser la récursivité lorsqu'on peut la remplacer par une définition itérative, à moins de bénéficier d'un gain considérable en simplicité.

Exemple :

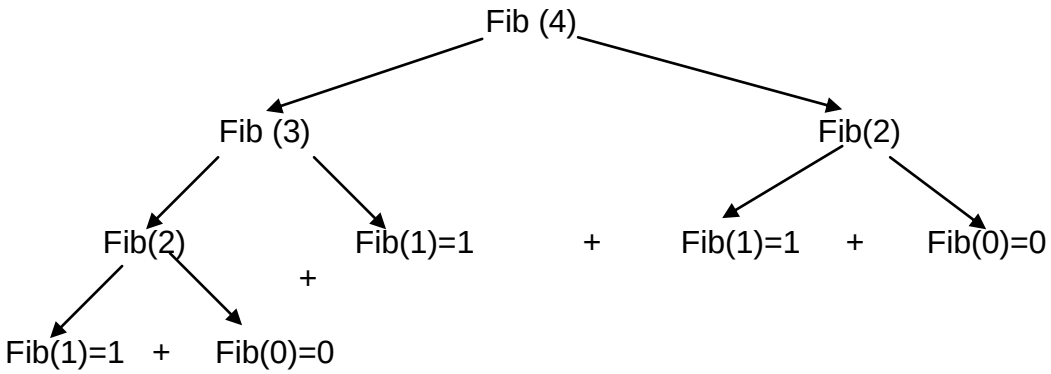
Les nombres de Fibonnaci : $F_0=0$, $F_1=1$

$$F_n=F_{n-1} + F_{n-2} \quad n>=2.$$

La fonction récursive permettant d'obtenir ces nombres est :

```
int  Fib(int n)
{ if (n= = 0) return 0;
  else if  (n= =1)  return 1;
        else  return Fib(n-1)+Fib(n-2);
}
```

L'exécution de cette fonction récursive pour n=4 nous donne l'arbre suivant :



Fib(4)=3 (9 appels pour arriver au résultat)

Les valeurs successive de cette suite : 0, 1, 1, 2, 3, 5, 8, 12, 21, 34, 55,…

Voici maintenant la fonction itérative équivalente à la fonction récursive Fib.

```
int  Fib( int n)
{  int x, y, z, i ;
   x=1 ;  y=1 ; z=1 ; /* x=Fib(0)  et y= Fib(1) */
   for( i=2 ; i<=n ; i++)
     {  z=x+y ;
        x=y;
        y=z;
     }
   return z ;
}
```

Pour n=4 : x=1
y=1
i=2 : z=2 x=1 y=2
i=3 : z=3 x=2 y=3
i=4 : z=5 x=3 y=5
Résultat z=5

Le problème se situe au nombre d'appels à la fonction, nous constatons que pour la solution récursive le nombre d'appels est un nombre ***exponentiel*** (c'est une mauvaise solution très coûteuse) alors que la solution itérative ne coute que ***n appels***.

Cette version itérative peut à son tour se convertir en une nouvelle version récursive :

```
int  Fib(int x, int y, int n)
{ if (n= = 0 ||  n= =1)  return y;   else  return Fib(y, x+y, n-1);
}
```

n=4 : x=1, y=1
Fib(1,1,4) → Fib(1,2,3) → Fib(2,3,2) → Fib(3,5,1) = 5 donc Fib(4)=5

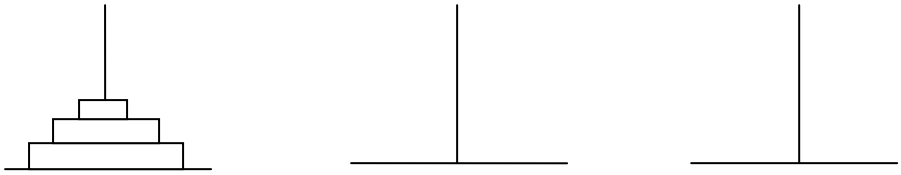
On a que 4 appels, le temps d'exécution est devenu ***linéaire***.

Comme on peut le constater, l'élimination de la récursivité est parfois très simple, elle revient à écrire une boucle, à condition d'avoir bien fait attention à l'exécution. Mais parfois elle est extrêmement difficile à mettre en œuvre.

Exemple : les tours de Hanoï

Le problème des tours de Hanoï consiste à déplacer N disques de diamètres différents d'une tour de départ à une tour d'arrivée en passant par une tour intermédiaire et ceci en un minimum de coups, tout en respectant les règles suivantes :

- on ne peut déplacer plus d'un disque à la fois,
- on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.



Tour A | Tour B (intermédiaire) | Tour C (résultat)

7.5.1 Elimination de la récursivité terminale

Un algorithme est dit récursif terminal (ou récursif à droite) s'il ne contient aucun traitement après un appel récursif.
- Dans ce cas le contexte de la fonction n'est pas empilé.
- L'appel récursif sera remplacé par une boucle while.

Cas1 :

```
f(x) /* récursive*/
{ if (condition(x))  {A ; f(g(x));}
}
```

```
f(x) /* itrérative*/
{ while(condition(x)) { A ; x=g(x) ;}
}
```

Cas2 :

```
f(x) /* récursive*/
{ if (condition(x))  {A ; f(g(x));}
  else  B ;
}
```

```
f(x) /* itrérative*/
{ while(condition(x)) { A ; x=g(x) ;}
  B ;
}
```

7.5.2 Elimination de la récursivité non terminale

a) cas d'un seul appel récursif:

Ici pour pouvoir dérécursiver, il va falloir sauvegarder le contexte de l'appel récursif.

Cas1 :

```
f(x) /* récursive*/
{ if (condition(x))  {A ; f(g(x));}   → nécessite une pile
  B ;
}
```

```
f(x) /* itrérative*/
{ pile p=initpile();
  while(condition(x)) { A ; empiler(&p,x); x=g(x) ;}
  while(!pilevide(p)) {desempiler(&p,&x) ;B ;}
}
```

Cas2 :

```
f(x) /* récursive*/
{ if (condition(x))
      {A1 ; f(g(x)); A2 ;}
  else B ;
}
```

```
f(x) /* itrérative*/
{ pile p=initpile();
  while(condition(x))
       { A1 ; empiler(&p,x); x=g(x) ;}
  B;
  while(!pilevide(p)) {desempiler(&p,&x) ;A2 ;}
}
```

b) cas de deux appels récursifs:

Le 2ème appel est récursif à droite (terminal)

```
f(x) /* récursive*/
{ if (condition(x))  {A ; f(g(x)); f(h(x)) ;}
 }
```

Si on élimine le 2ème appel :

```
while(condition(x)) {A ; f(g(x)) ; x=h(x) ;}
```

Le schéma itératif équivalent à f est :

```
f(x) /* itérative*/
{ pile p=initpile();
  while(condition(x))
   { while(condition(x) { A(x) ; empiler(&p,x); x=g(x) ;}
     desempiler(&p,&x) ;x=h(x) ;
   }
}
```

ALGORITHME Q(U)
si *C*(*U*) alors *D*(*U*);Q(a(*U*));*F*(*U*)
sinon *T*(*U*)

ALGORITHME Q'(U)
empiler(nouvel_appel, U)
tant que pile non vide faire
dépiler(état, V)
si état = nouvel_appel alors $U \leftarrow V$
si C(U) alors D(U)
empiler(fin, U)
empiler(nouvel_appel, a(U))
sinon T(U)
si état = fin alors $U \leftarrow V$
F(U)

Exercices :

1) Déroulez les fonctions calcul1 et calcul2, que constatez-vous ?

```
float calcul1 (int n)
{ if (n==0) return (2) ;
  else return(1/2(calcul1(n-1)+2)) ;
}
```

C'est une fonction qui se termine.

```
float calcul2 (int n)
{ if (n==0) return (2) ;
  else return(1/2(calcul2(n-2)+2)) ;
}
```

calcul2 ne se termine pas si ***n est impair***
calcul2 se termine si ***n est pair***

$\forall$ n le résultat de calcul1 est égal à 2
$\forall$ n pair le résultat de calcul2 est égal à 2

2) Ecrire une fonction itérative puis récursive qui calcul la somme des n premiers nombres.

- ```
  int sommeIter (int n)
  {  int i, s=0 ;
     for(i=1 ;i<=n ; i++)
      s=s+i ;
    return s ;
  }
  ```

- ```
  int sommeIter (int n)
  {  int i, s=0 ;
     for(i=n ;i>0 ; i--)
      s=s+i ;
    return s ;
  }
  ```

- ```
  int sommeRec(int n)
  {  if (n==0) return 0 ;
     else  return(n+somme(n-1)) ;
  }
  ```

Remarque: Le concept de récursivité est spécialement mis en valeur dans les définitions mathématiques. Les mathématiques utilisent plutôt le mot récurrence.

3) Le plus grand commun diviseur (pgcd):
Le pgcd de deux entiers A et B est le plus grand entier qui divise à la fois A et B.

```
int pgcdRec(int A, int B)
{
  if (B==0) return A ;
  else  return( pgcd(B, A%B);
}
```

```
int pgcdIter(int A, int B)
{ int reste;
  while (B !=0)
    { reste=A%B ;
      A=B ;  B=reste;
    }
 return(A);
}
```
```

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#define max 50
typedef char chaine[max];
```

int palind(chaine t,int i,int j)

```
{ printf("palindrom i= %d  j= %d \n",i,j);
  if(i>=j) return(1);
  else if (t[i]==t[j]) return(palind(t,i+1,j-1));
       else return(0);
}
main()
{ chaine ch; int res, l;
  printf("donnez un mot\n");
  scanf("%s",ch); l=strlen(ch)-1;
  res=palind(ch,0,l);
  if (res==0) printf("%s n'est pas un mot palindrome\n",ch);
  else printf("%s est un mot palindrome\n",ch);
  getch();
}
#include<stdio.h>
#include<conio.h>
#include<string.h>
#define max 50
typedef char chaine[max];
```

void Inverser(chaine t,int i,int j)

```
{char x;
  if (i<j) { x=t[i];t[i]=t[j],t[j]=x;Inverser(t,i+1,j-1);}
}

main()
{ chaine ch; int l;
  printf("donnez un mot\n");
  scanf("%s",ch); l=strlen(ch)-1;
  Inverser(ch,0,l);
  printf("\n apres inversion : %s",ch);
  getch();
}
```