7. La récursivité

En programmation, nombreux sont les problèmes qu'on résout en répétant plusieurs fois des séquences d'instructions.

Certains langages sont munis de structures de contrôles répétitivesfs. C'est le cas notamment pour la langage C, qui dispose des boucles `pour` (`for`) et `tant que` (`while`).

Mais certains problèmes se résolvent simplement en résolvant un sous problème de même nature, mais plus simple…Cette méthode de résolution s'appelle la *récursivité*.

7.1.Définition

Une fonction récursive est une fonction qui s'appelle elle-même. Chaque appel à la fonction est indépendant des autres, avec ses propres variables.
L'exemple le plus classique d'emploi de la récursivité est l'écriture de la fonction factorielle. Pour rappel, la factorielle d'un nombre *n* est définie comme *n* fois la factorielle du nombre *n*-1, et la factorielle de 1 est 1.

```
Alrotithme Fact
Variables
  n : entier
Fonction factorielle (n : entier) :entier
  Variable f :entier
  Début
      Si n == 1 Alors
          f = 1
      Sinon
          f = factorielle (n-1) * n
      Fin Si
    return f
Fin Fonction
Début
    n ← 13
    Ecrire (n,' ! = ‘, factorielle (n))
Fin
```

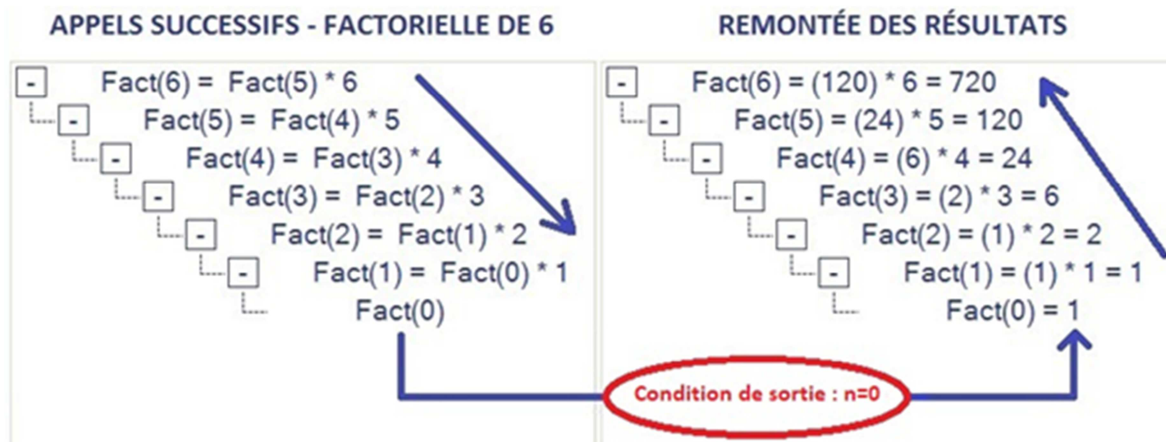
```c
#include <stdio.h>

/* Fonction qui calcule n! */
int factorielle(int n)
{
  if (n == 0)
    return 1;
  else
    return n * factorielle(n - 1);
}

int main(void)
{
  int n=13;
  printf("%d! = %d\n", n, factorielle(n));
  return 0;
}
```

7.2. Pile d'exécution

- Lorsqu'on fait des appels de fonctions, la pile d'appels de fonctions permet de mémoriser l'endroit du programme où la fonction a été appelée, avec les paramètres et la valeur de retour. Lorsqu'une fonction a terminé son exécution, elle est dépilée.

- Lors de l'exécution d'une fonction récursive, on appelle la même fonction plusieurs fois, ce qui produit plusieurs empilages :

Exemple



N.B :

- L'écriture itérative de la fonction factorielle est très simple et dans ce cas, on préfère l'utiliser plutôt que la version récursive.
- Un usage irréfléchi de fonctions récursives peut poser des problèmes d'exécution sur certaines implémentations. En effet, si la récursivité est profonde i.e. les appels récursifs sont nombreux, les fonctions sont appelées et ne retournent pas immédiatement leur résultat donc les appels s'accumulent. Ces appels sont souvent stockés dans une zone mémoire appelée "pile" et qui est de taille limitée. Si cette taille est dépassée, on obtient une erreur à l'exécution ("une explosion de la pile"). Par exemple, si on exécute la fonction ci-dessus avec une valeur de n très grande, on peut obtenir une erreur à l'exécution : ainsi, le code ci-dessous

```
#include <stdio.h>
/* Retourne le produit n*x où n>=0 */
int produit(int n, int x)
{
  if (n > 0)
    return produit(n - 1, x) + x;
  else
    return 0;
}
int main(void)
{
  int n = 500000, x = 5;

  printf("%d * %d = %d\n", n, x, produit(n, x));
  return 0;
}
```

S'exécute de la manière suivante :

```
$ gcc -W -Wall -std=c99 -pedantic -o x recursifPile.c
$ ./x
Erreur de segmentation
$
```

En fait, le code récursif utilisé peut-être dérécursifié de la manière suivante :

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

/* Retourne le produit n*x où n>=0 */
int produit_bis(int n, int x)
{
  int i = 0;
  int temp = 0;

  for (i = 0; i < n; i++)
    temp = temp + x;

  return temp;
}

int main(void)
{
  /* on suppose que la valeur 500000 tient dans un i
  int n = 500000, x = 5;

  printf("%d * %d = %d\n", n, x, produit_bis(n, x));
  return 0;
}
```

qui affiche le message suivant : `500000 * 5 = 2500000`

7.3. Règles de conception d'un algorithme récursif

Règle 1 :

Tout algorithme récursif doit distinguer plusieurs cas dont l'un au moins ne doit pas contenir d'appels récursifs, sinon il y a risque de cercle vicieux et de calcul infini. Les cas non récursifs d'un algorithme récursifs sont appelés *cas de bases*. Les conditions que doivent satisfaire les données dans ces cas de bases sont appelées *conditions de terminaison.*

Règle 2 :

Tout appel récursif doit se faire avec des données plus proches de données satisfaisant les conditions de terminaison.

7.4.Type de récursivité

7.4.1. Récursivité simple ou linéaire

Un algorithme récursif est *simple* ou *linéaire* si chaque cas qu'il distingue se résout en au plus un appel récursif. Ainsi l'algorithme de calcul de n! est récursif simple.

7.4.2. Récursivité multiple

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

Exemple :

```
Fonction combinaison (N : entier, P : entier) : entier
Début
Si (P=0 ou N=0) Alors
 Return 1
Sinon
  Return (combinaison (N-1, P) + combinaison (N-1, P-1))
FinSi
Fin Fonction
```

7.4.3. Récursivité croisée ou mutuelle

Deux algorithmes sont *mutuellement récursifs* si l'un fait appel à l'autre et l'autre à l'un.

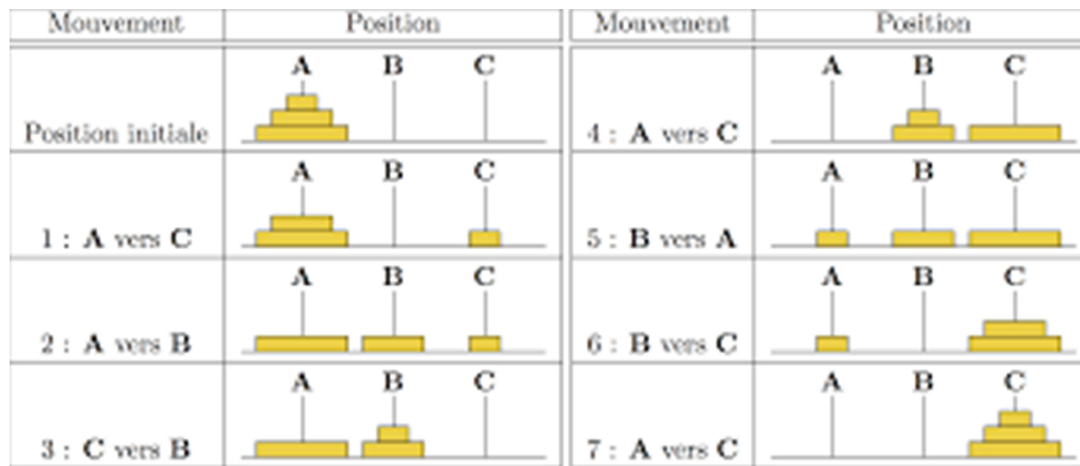
Parité d'un entier P(n) = prédicat de test de parité de l'entier n. I(n) = prédicat de test d'imparité de l'entier n. Solution mutuellement récursive

```
Fonction P (n : entier) : Booléen
Début
  Si n= 0 alors
     P(n) = vrai
   Sinon
     P(n) = I(n−1)
   fin si
Fin Fonction
```

```
Fonction I(n :entier):Booléen
Début
  Si n= 0 alors
     I(n) = faux
  Sinon
    I(n) = P(n−1)
  fin si
Fin Fonction
```

Exemple : tours de Hanoi

C'est un jeu de réflexion qui consiste à déplacer un nombre donné de disque d'une tour "départ" à une tour "arrivée", à l'aide d'une troisième tour "intermédiaire" et ceci en respectant deux règles :
- Ne déplacer qu'un seul disque à la fois.
- Ne pas placer un disque sur un autre plus petit.



__Solution__

```
#include <stdio.h>
void hanoi (int n, char A, char B, char C)
{
   if(n==1)
   printf("deplacer %c vers %c\n",A,C);
   else
   {
     hanoi(n-1,A,C,B);
     printf("deplacer %c vers %c\n",A,C);
     hanoi(n-1,B,A,C);
   }
}

main()
{
  char a='a',b='b',c='c';
  int n;
   while(1)
  {
   printf("Entrer n : ");
   scanf("%d",&n);
   printf("\n");
   hanoi(n,a,b,c);
  }
}
```