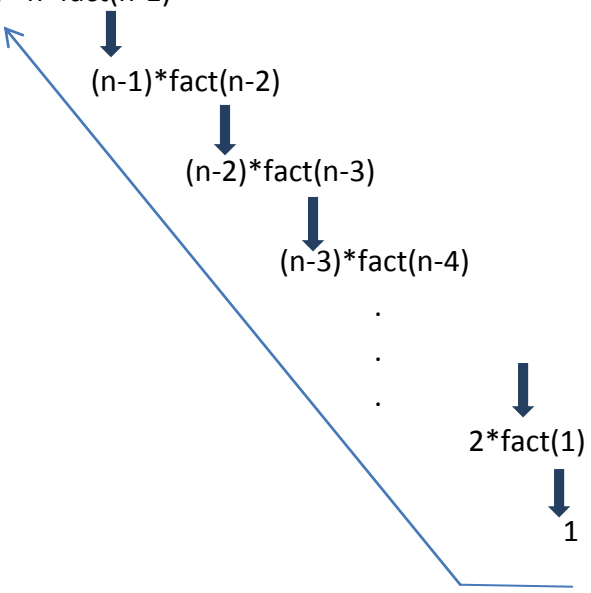
LA RECURSIVITE

Définitions

- Il s'agit de sous-programmes qui s'appellent eux-mêmes, on entend également le terme « auto-appel ».
- La récursivité est une manière simple et élégante de résoudre certains problèmes algorithmiques.
- Il y a différents types de récursivités :
    a. Récursivité simple.
    b. Récursivité double.
    c. Récursivité croisée.
    d. Récursivité terminale.

Etude de la récursivité

- **La récursivité simple** appelée simplement récursivité est la plus utilisée parmi les récursivités citées ci-dessus.
  Etudiant celle-ci à l'aide d'un exemple trivial : La factorielle.
  Fact(n) = n !
  Cette fonction peut être étudiée de deux manières : itérative et récursive.
    - Itérative
      Fact(n)= n(n-1)(n-2)(n-3) … 3*2*1
      Exemple fact(6)= 6*5*4*3*2*1
    - Récursive
      Fact(n)= n*fact(n-1)



Puis on commence par le bas c'est à dire 1 pour avoir :

Fact(n)= 1*2*3* . . . (n-2)(n-1)n

Comme on peut le remarquer, il faut avoir une condition de sortie « way out » pour s'arrêter et faire le chemin inverse pour avoir le résultat.

Pour bien assimiler le processus, écrivons les deux solutions de la factorielle : itérative et récursive.

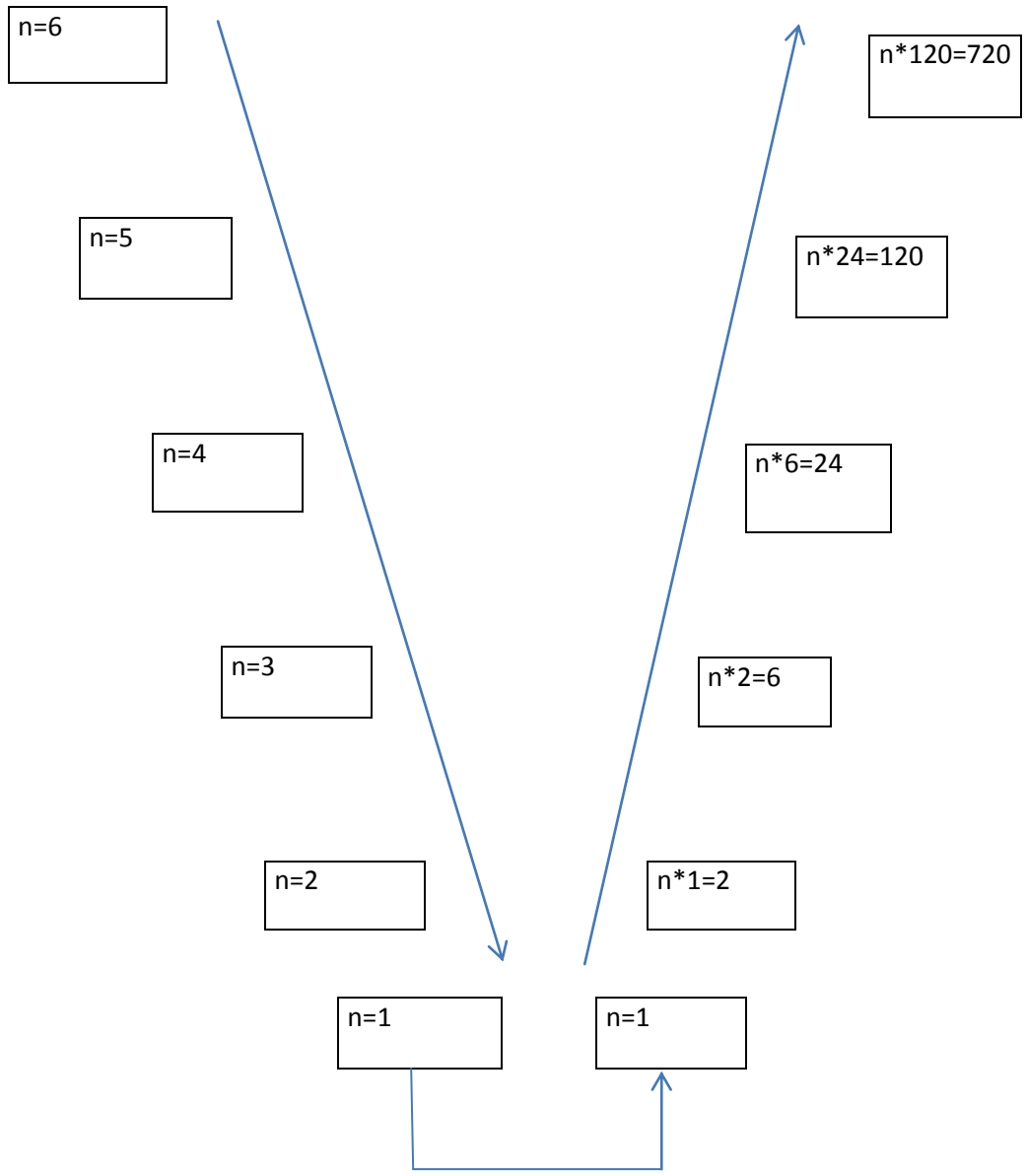
```
/* Factorielle itérative*/

Int   fact(Int  n)

{

        Int i ;

        Int res= 1 ;

        For (i= 1 ; i<= n ; i= i+1)

            Res= res*i ;

        Return res ;

}


/* Factorielle récursive*/

Int   fact(Int n) ;

{

        If (n == 1)

           Return 1 ;

        Else

            Return   n*fact(n-1) ;

}
```

Dans cette deuxième forme d'écriture de la factorielle, on remarque que le 2^{ième} « return » est en fait l'appel récursif et en soustrait 1 à chaque appel jusqu'à ce que n soit égal à 1 qui est notre condition de sortie « wait out », mais le travail de la fonction n'est pas terminé :

Lorsque la fonction rencontre le « wait out », elle remonte dans tous les appels précédents pour calculer la factorielle avec les valeurs précédemment trouvées !

Voyons cela grâce à ce petit schéma :

Fact(6) = 120



- **La récursivité double**, dans ce cas on aura deux appels récursif dans le corps du sous-programme. On verra cela quand on étudera la structure de données appelée arbre binaire.

- **La récursivité croisée**, c'est qu'on ait deux sous-programmes qui s'appellent l'un l'autre. Par exemple si ces deux sous-programmes sont appelés A et B, on aura :
  A appelle B et B appelle A jusqu'à atteindre le « way out ».
  Ce genre de récursivité est rarement usité.

- **Récursivité terminale**. Les sous-programmes récursifs sont très gourmants en mémoires, on étudiera ce cas dans le point qui suit. Pour cela, il existe la récursivité terminale pour régler un certain nombre de problèmes.
  Cette forme de récursivité n'est pas forcément connue par tout le monde. On peut dire que la récursivité terminale utilise seulement la phase de descente, sans la remontée et ceci ne devient possible sauf si la dernière expression « return » renvoie directement la valeur obtenue par l'appel récursif.
  Ecrivons maintenant la fonction factorielle, en utilisant la récursivité terminale :

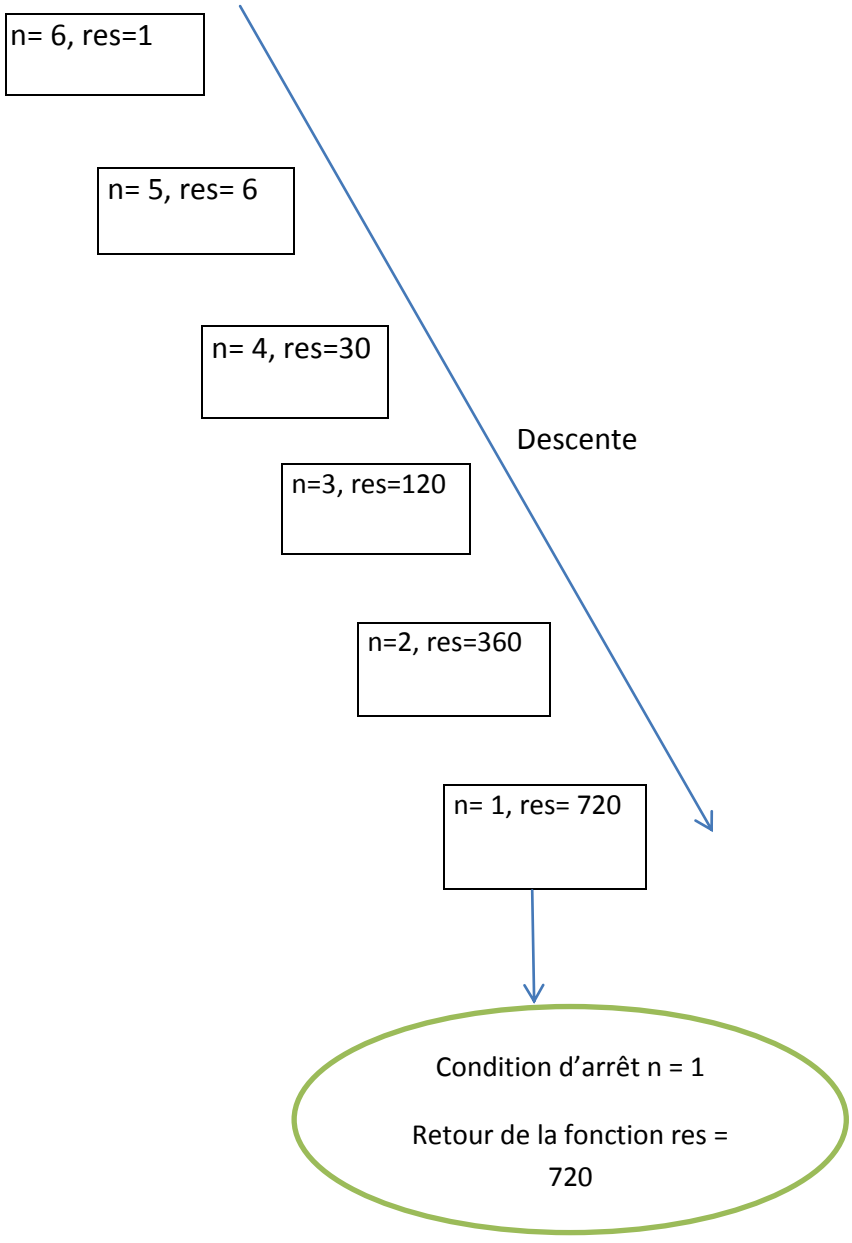
```
Int  fact_term(int n, int res )
{
        If (n == 1)
          Return res;
        Else
            Return   fact_term(n-1, n*res);
}
```

NB: Bien sur, au moment de l'appel res est égal à 1.

On remarque aussi qu'on a un paramètre supplémentaire, ceci est un passage obligatoire pour avoir une récursivité terminale !

Illustrons ceci grâce à ce shéma :

Fact(-) = 720



Le schéma nous montre qu'il y ait bel et bien une descente mais pas de remontée.

Dangers et précautions

La récursivité est un moyen puissant pour résoudre certains problèmes de façon élégante, mais ce moyen reste dangereux pour deux raisons essentielles :

- **Dépassement de capacité** : La cause quand on manipule de très grands nombres. Ce phénomène se produit lorsqu'on essaie de stocker une valeur que le type avec lequel elle a été déclarée n'est pas capable de la recevoir !

  Pour cela il faut surtout éviter le type « Int » ; préférez-lui un type comme « long » et même « double » pour assurer un minimum de viabilité de votre programme.

  Donc la fonction fonctionnelle peut s'écrire comme suit :

```
Unsigned long  fact(unsigned int n)
{
        If (n == 1)
           Return 1;
        Else
            Return  n*fact(n-1);
}
```

  On Remarque l'utilisation de "unsigned long" et "unsigned int" car l'argument n est positif de même pour le résultat.

  Le deuxième problème est :

- **Débordement de pile** : stack overflow.
  Celle-ci est la cause la plus souvent rencontrée dans le « plantage » de programmes avec des sous-programmes récursifs.
  C'est quoi une pile ?
  Le système utilise une structure de données appelée « pile » pour gérer les appels et retours des sous-programmes, la capacité de cette pile est fixée lors de la compilation du programme.
  Souvent avec de très grands nombres, cette pile déborde et le programme plante! : on dit qu'il y ait un « stack overflow » ou simplement « overflow ».
  A titre indicatif, dans l'exemple de la fonctionnelle, il nous faut environ 135000 appels récursif pour exploser la pile.

Et le langage « **Ocaml** » est un langage très adapté à la récursivité du moment que c'est un langage fonctionnel de même que le langage « **Haskell** ».

Avec Ocaml la fonction factorielle s'écrit comme suit :

```
Let rec  fact(n) =
    If n=1  then 1
    Else   n*fact(n-1)
```

Alors qu'avec haskell, on aura:

```
Fact 1 = 1

Fact n = n*fact(n-1)
```

En langage java, la factorielle s'écrit comme suit:

```
static  long  fact(Int n) {
    if (n == 1)
        return 1;
    else
        return  n*fact(n-1);
```

On remarque que pratiquement tous les langages se ressemblent plus au moins, c'est pour cela qu'il faut surtout maitriser l'algorithmique.