Rappel : Récursivité

Pour plus de détails consulter le cours de L1
http://dept-info.labri.fr/ENSEIGNEMENT/algoprog/support.html

Récursivité

Définition :
Lorsqu'un algorithme contient un appel à lui-même, on dit qu'il est récursif.
Lorsque deux algorithmes s'appellent l'un l'autre on dit que la récursivité est croisée

Complexité
Un algorithme récursif nécessite de conserver les contextes récursifs des appels. La récursivité peut donc conduire à une complexité mémoire plus grande qu'un algorithme itératif.

Calculer a^n : faire n fois a*a*a…

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

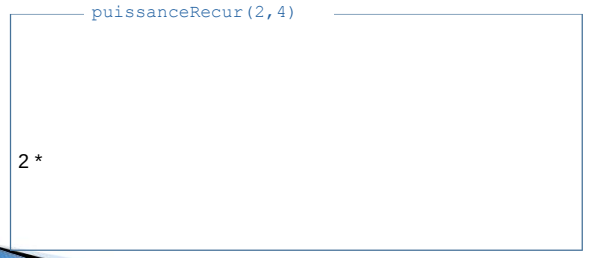

Calculer a^n : a* a^{n-1}

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

Calculer a^n : $a * a^{n-1}$

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

Calculer `puissanceRecur(2,4)`

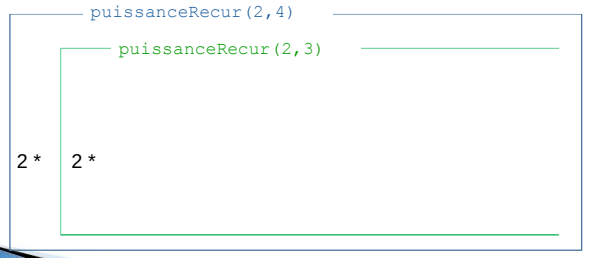


6

Calculer a^n : $a * a^{n-1}$

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

Calculer `puissanceRecur(2,4)`

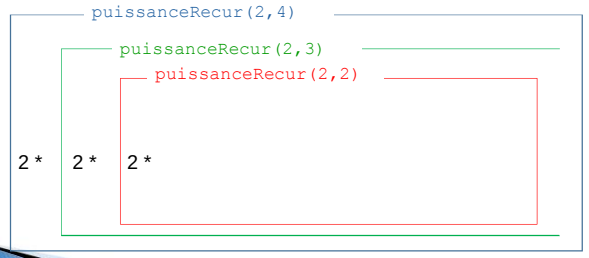


7

Calculer a^n : $a * a^{n-1}$

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

Calculer `puissanceRecur(2,4)`

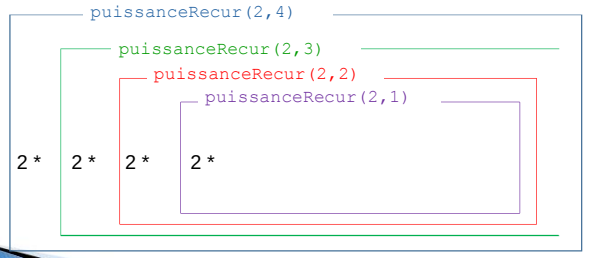


8

Calculer a^n : $a * a^{n-1}$

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

Calculer `puissanceRecur(2,4)`

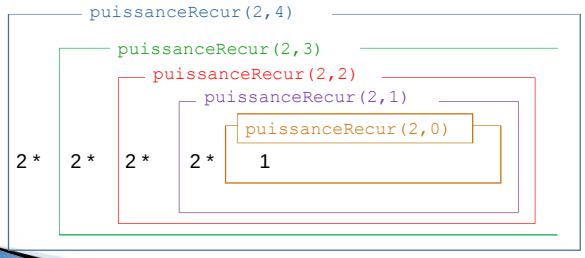


9

Calculer a^n : $a * a^{n-1}$

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

Calculer `puissanceRecur(2,4)`

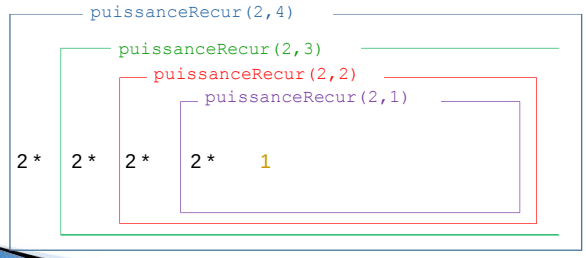


10

Calculer a^n : $a * a^{n-1}$

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

Calculer `puissanceRecur(2,4)`

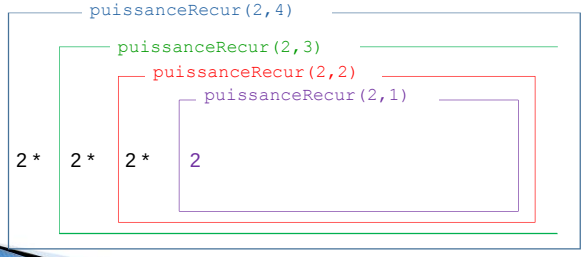


11

Calculer a^n : $a * a^{n-1}$

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

Calculer `puissanceRecur(2,4)`

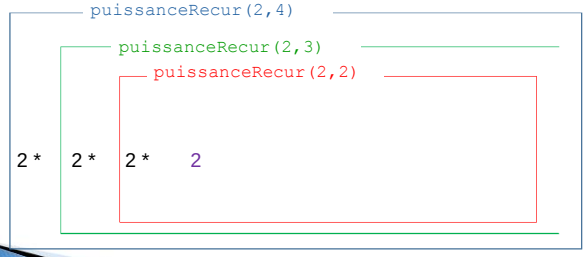


12

Calculer a^n : $a * a^{n-1}$

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

Calculer `puissanceRecur(2,4)`

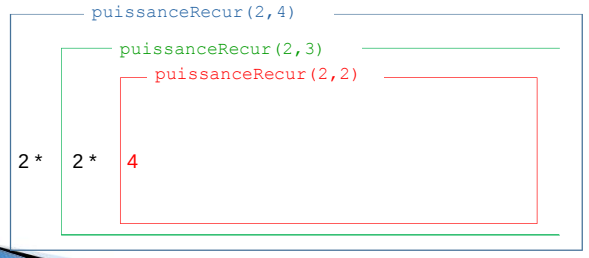


13

Calculer a^n : $a * a^{n-1}$

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

Calculer `puissanceRecur(2,4)`

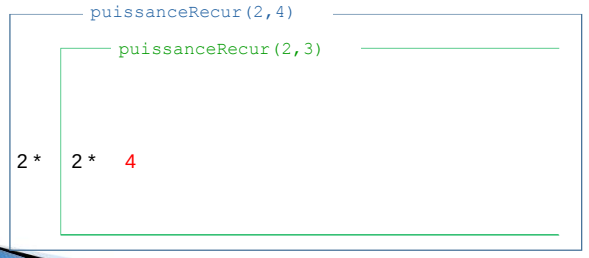


14

Calculer a^n : $a * a^{n-1}$

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

Calculer `puissanceRecur(2,4)`

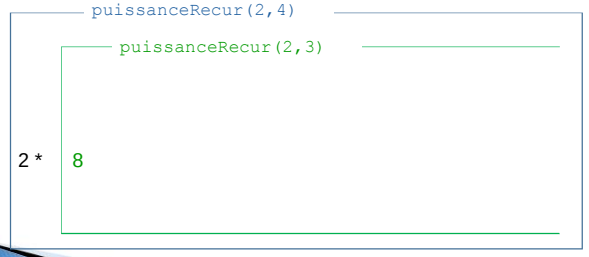


15

Calculer a^n : $a * a^{n-1}$

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

Calculer `puissanceRecur(2,4)`

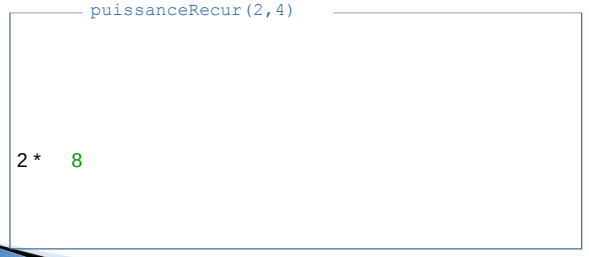


16

Calculer a^n : $a * a^{n-1}$

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

Calculer `puissanceRecur(2,4)`

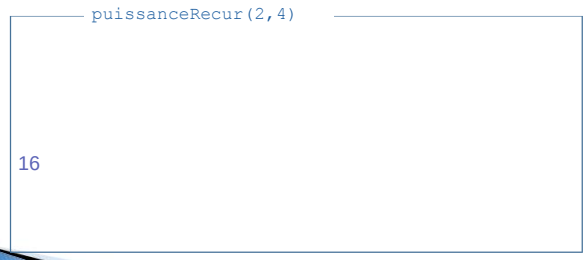


17

Calculer a^n : $a^* a^{n-1}$

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

Calculer `puissanceRecur(2,4)`



Calculer a^n : $a^* a^{n-1}$

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

`puissanceRecur(2,4)` ⟶ 16

Récursivité

- La récursivité consiste à remplacer une boucle par un appel à la fonction elle-même.

Chaque fois que l'on désire programmer une fonction récursive, on doit répondre aux questions suivantes :

- Comment le problème au rang n se déduit-il de la solution à un (des) rang(s) inférieur(s) ?
- Quelle est la condition d'arrêt de la récursivité ?

Calculer a^n : $a^* a^{n-1}$

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1); /*condition d'arret */
  return (a*puissanceRecur(a,n-1));
}
```

Calculer `puissanceRecur(2,4)`

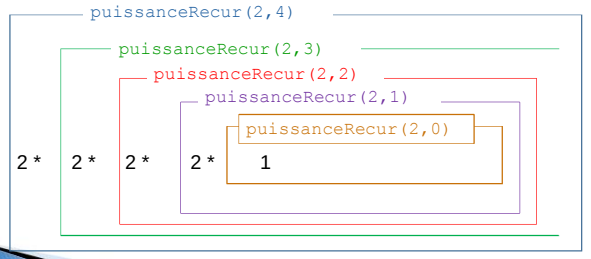
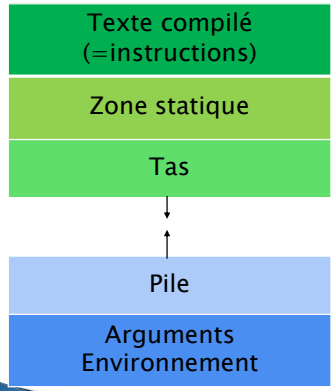


Schéma conceptuel de la mémoire



- La zone statique et le texte sont fixés avant le lancement
- Le arguments sont initialisés au lancement du programme
- La pile et le tas évoluent pendant l'exécution

Complexité : nombre d'appels

Soit :

$u_1 = 1$

$u_n = u_{n-1} + n$

Fonction itérative :

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

Fonction récursive

```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```

Complexité en temps : *n pour les 2 versions.*

Complexité mémoire : *1 pour la version itérative MAIS n pour la version récursive.*

Complexité : nombre d'appels

Somme des entiers :

```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```

suiteR(5)

Complexité : nombre d'appels

Somme des entiers :

```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```

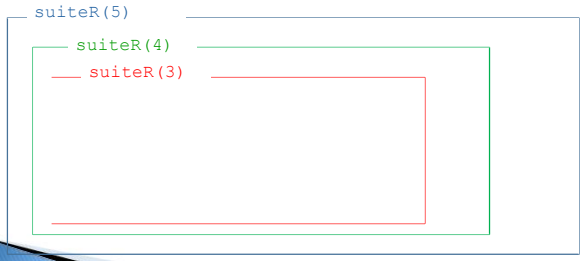
suiteR(5)

suiteR(4)

Complexité : nombre d'appels

Somme des entiers :

```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```

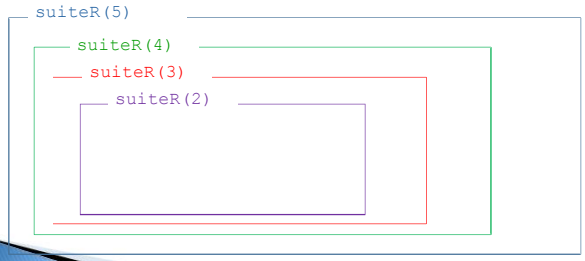


26

Complexité : nombre d'appels

Somme des entiers :

```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```

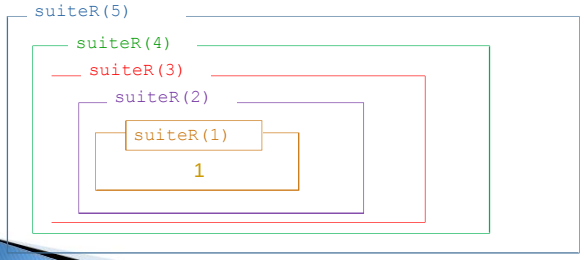


27

Complexité : nombre d'appels

Somme des entiers :

```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```

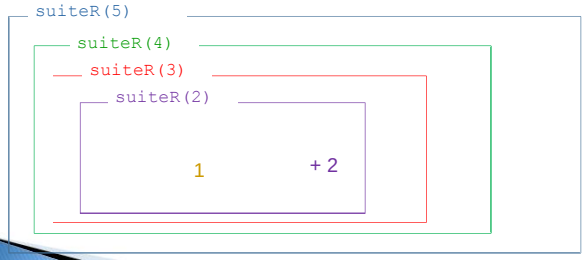


28

Complexité : nombre d'appels

Somme des entiers :

```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```

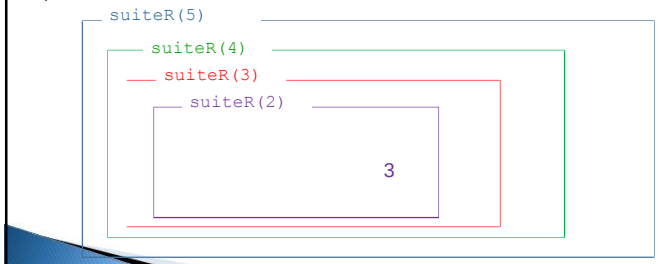


29

Complexité : nombre d'appels

Somme des entiers :

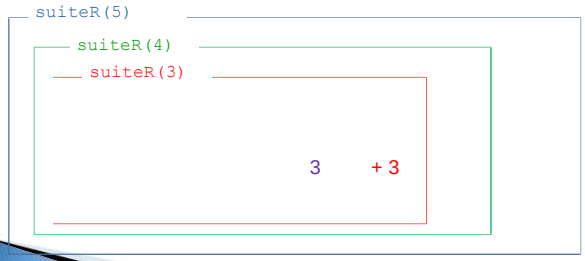
```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```



Complexité : nombre d'appels

Somme des entiers :

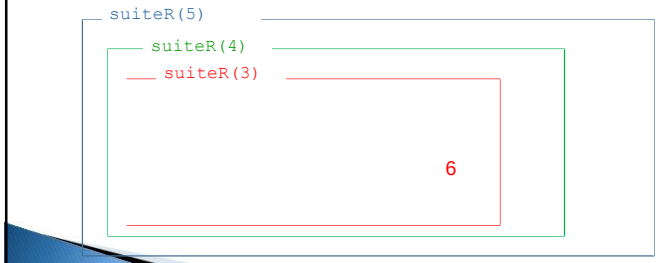
```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```



Complexité : nombre d'appels

Somme des entiers :

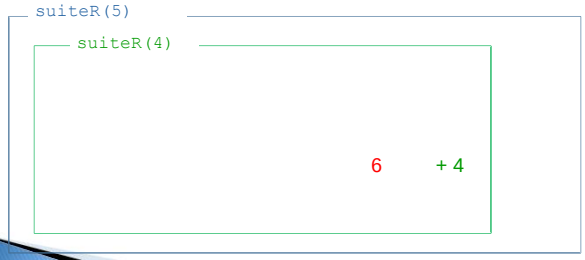
```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```



Complexité : nombre d'appels

Somme des entiers :

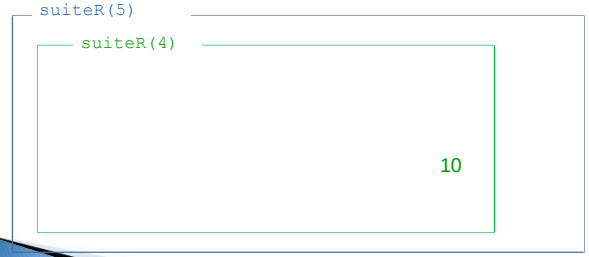
```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```



Complexité : nombre d'appels

Somme des entiers :

```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```

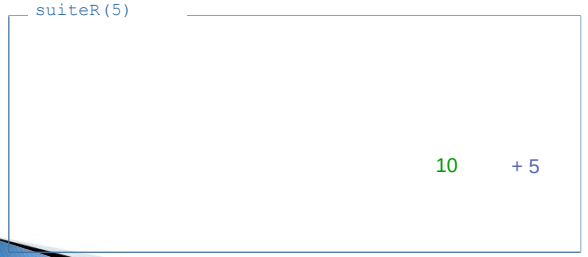


34

Complexité : nombre d'appels

Somme des entiers :

```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```

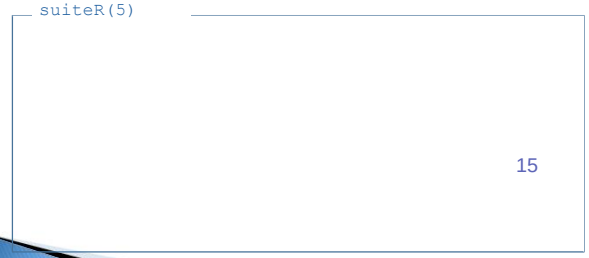


35

Complexité : nombre d'appels

Somme des entiers :

```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```

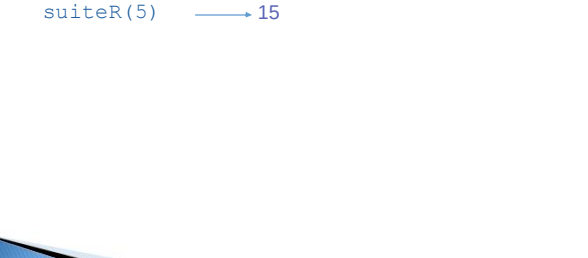


36

Complexité : nombre d'appels

Somme des entiers :

```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```



37

Complexité : Fibonacci

Soit :

$u_0=1$, $u_1=1$

$u_n= u_{n-1}+ u_{n-2}$ *pour* $n>2$

```
int fibo(int n){
  if(n<2)
    return(1);
      return(fibo(n-1)+fibo(n-2));
}
```

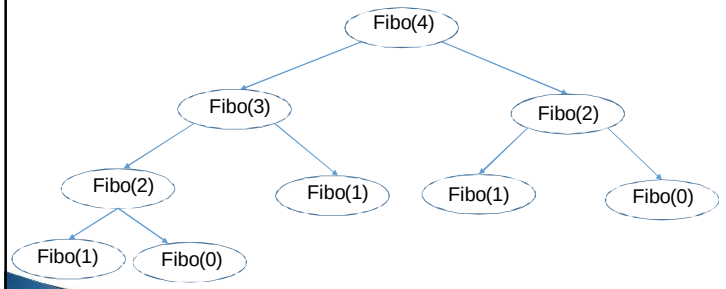
⚠ Complexité en $\Omega(\Phi^n)$ avec $\Phi \cong 1{,}61$ donc complexité exponentielle. Il existe d'autres algorithmes en O(n) et même en $O(\log_2(n))$ pour faire le même calcul.

Complexité : Fibonacci

Soit :

$u_0=1$, $u_1=1$

$u_n= u_{n-1}+ u_{n-2}$ *pour* $n>2$



Exemple : Rechercher un élément dans un tableau itératif

```
int recherche(int t[], int n, int x){
  int i = 0;
  while(i<n){
     if (t[i]==x)
       return(i);
     i++;
    }
 return(-1);
}
```


Exemple : Rechercher un élément dans un tableau récursif

```
int recherche(int t[], int n, int x){
  if (n==0)
   return (-1);
  if (t[n-1]==x)
    return(n-1);
 return(recherche(t, n-1,x);
```

Récursivité Terminale

Définition 1.4 :
Un algorithme récursif présente une récursivité terminale si et seulement si la valeur retournée par cet algorithme est une valeur fixe, ou une valeur calculée par cet algorithme.

Autrement dit si aucun traitement n'est effectué à la remontée d'un appel récursif (sauf le retour d'une valeur).

L'algorithme *facRecur* ne présente pas de récursivité terminale.

Récursivité Terminale

L'algorithme `facRecur` ne présente pas de récursivité terminale.

```
fonction facRecur(val n:entier):entier;
  debut
    si n==0 alors
      retourner(1)
    sinon
      retourner(n*facRec(n-1))
    finsi
  fin
finfonction
```

Récursivité Terminale

Exemple

```
fonction facRecurTerm(val n:entier;
                      val res:entier):entier;
  debut
    si n==0 alors
      retourner(res)
    sinon
      retourner(facRecurTerm(n-1,n*res))
    finsi
  fin
finfonction
```

L'algorithme `facRecurTerm` présente une récursivité terminale.
La factorielle se calcule par l'appel `facRecurTerm(n,1)`

Récursivité Terminale

Exemple : calcul de la fonction factorielle

```
fonction facIter(val n:entier):entier;
  debut
    var i,p:entier;
    p=1:
    pour i=2 à n faire
      p=p*i;
    finpour
    retourner(p)
  fin
Finfonction
```

La fonction **`factIter`** est meilleure en temps et mémoire.

Récursivité Terminale

Intérêt : des compilateurs détectent cette propriété et optimisent le stockage de l'environnement de la fonction.

Ainsi `facRecurTerm` aura une complexité identique à `facIter`.

ATTENTION.

Dans le cas d'un algorithme présentant deux appels récursifs, rendre la récursivité terminale ne permet pas obligatoirement au compilateur d'obtenir une complexité inférieure.