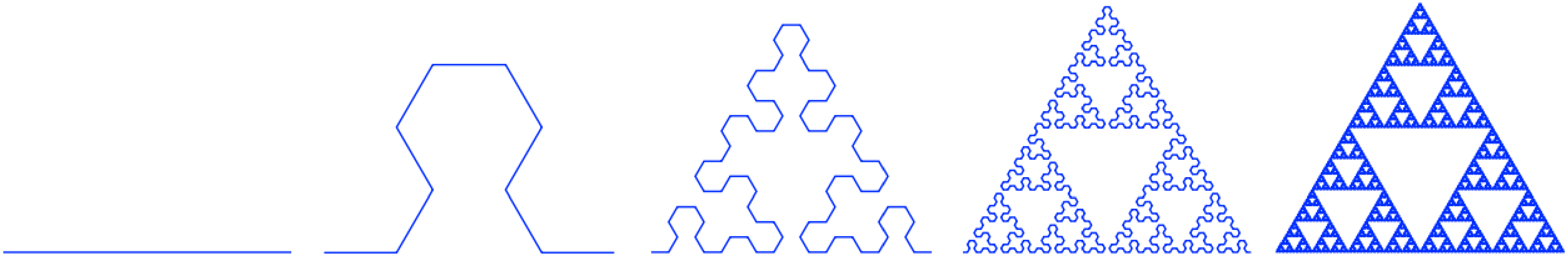
Récursivité (1/3)

Une construction est récursive si elle se définit à partir d'elle-même.

<u>Exemple</u> : le dessin de la Vache qui rit

<u>Autre exemple</u> : le triangle de Sierpinski



Récursivité (2/3)

En informatique, un programme est dit récursif s'il s'appelle lui même. Il s'agit donc forcément d'une fonction.

<u>Exemple</u> : la factorielle, n! = 1 x 2 x ... x n donc n! = n x (n-1)!

```
// cette fonction renvoie n! (n est supposé supérieur ou égal à 1)
fonction avec retour entier factorielle(entier n)
début
     retourne n*factorielle(n-1);
fin
```

L'appel récursif est traité comme n'importe quel appel de fonction.

Récursivité (3/3)

```
factorielle(3)
début
     retourne 3*factorielle(2);
fin
```

```
factorielle(2)
début
     retourne 2*factorielle(1);
fin
```

```
factorielle(1)
début
     retourne 1*factorielle(0);
fin
```

```
factorielle(0)
début
     retourne 0*factorielle(-1);
fin
```

Condition d'arrêt (1/2)

Puisqu'une fonction récursive s'appelle elle-même, il est impératif qu'on prévoit une condition d'arrêt à la récursion, sinon le programme ne s'arrête jamais!

On doit toujours tester en premier la condition d'arrêt, et ensuite, si la condition n'est pas vérifiée, lancer un appel récursif.

*Exemple de la factorielle : **si n ≠ 1**, n! = n x (n-1)!, **sinon n! = 1**.*

```
// cette fonction renvoie n! (n est supposé supérieur ou égal à 1)
fonction avec retour entier factorielle(entier n)
début
     si (n = 1) alors
          retourne 1;
     sinon
          retourne n*factorielle(n-1);
     finsi
fin
```

Condition d'arrêt (2/2)

6

```
factorielle(3)
début
     si (3 = 1) alors retourne 1;
     sinon retourne 3*factorielle(2);
     finsi
fin
```

2

```
factorielle(2)
début
     si (2 = 1) alors retourne 1;
     sinon retourne 2*factorielle(1);
     finsi
fin
```

1

```
factorielle(1)
début
     si (1 = 1) alors retourne 1;
     sinon retourne 1*factorielle(0);
     finsi
fin
```

Pile d'exécution (1/4)

La récursivité fonctionne car chaque appel de fonction est différent.

L'appel d'une fonction se fait dans un contexte d'exécution propre, qui contient :

- l'adresse mémoire de l'instruction qui a appelé la fonction
- les valeurs des paramètres et des variables définies par la fonction

Exemple : exécution du programme Toto

```
programme Toto
     entier i;
début
     i <- 2;
     écrire factorielle(2);
     écrire "bonjour";
     i <- factorielle(i);
fin
```

Pile d'exécution (2/4)

<u>programme Toto</u>

```
444    entier i;
445    début
446        i <- 2;
447        écrire 2;
448        écrire "bonjour";
449        i <- 2;
450    fin
```

<u>factorielle(2)</u> : n = 2, retour #449

```
463    si (n = 1) alors
464        retourne 1;
465    sinon
466        retourne n*1;
467    finsi
```

<u>factorielle(1)</u> : n = 1, retour #466

```
765    si (n = 1) alors
766        retourne 1;
767    sinon
768        retourne n*factorielle(n-1);
769    finsi
```

Pile d'exécution (3/4)

Prévoir à l'avance le nombre d'appels d'une fonction récursive pouvant être en cours simultanément en mémoire est impossible. La récursivité suppose donc une allocation dynamique de la mémoire (à l'exécution).

Quand il n'y a pas de récursivité, on peut réserver à la compilation les zones mémoire nécessaires à chaque appel de fonction.

Les langages de programmation permettent pour la plupart la programmation récursive, mais ce n'est pas le cas de certains langages anciens (COBOL, FORTRAN, BASIC, ...).

En programmation fonctionnelle (LISP, CAML, …) ou en programmation logique (PROLOG), les programmes sont toujours récursifs.

Pile d'exécution (4/4)

<u>Attention</u> : exécuter trop d'appels de fonction fera déborder la pile d'exécution!

```
public static void testPile(int nbAppels){
     System.out.println("appel numéro " + nbAppels);
     testPile(nbAppels + 1);
}
...
testPile(1);
```

```
appel numéro 1
appel numéro 2
...
appel numéro 5613
Exception in thread "main" java.lang.StackOverflowError
    at sun.nio.cs.SingleByte.withResult(SingleByte.java:44)
    at sun.nio.cs.SingleByte.access$000(SingleByte.java:38)
    at sun.nio.cs.SingleByte$Encoder.encodeArrayLoop(SingleByte.java:187)
```

Récursif versus itératif (1/4)

Il est souvent possible d'écrire un même algorithme en itératif et en récursif.

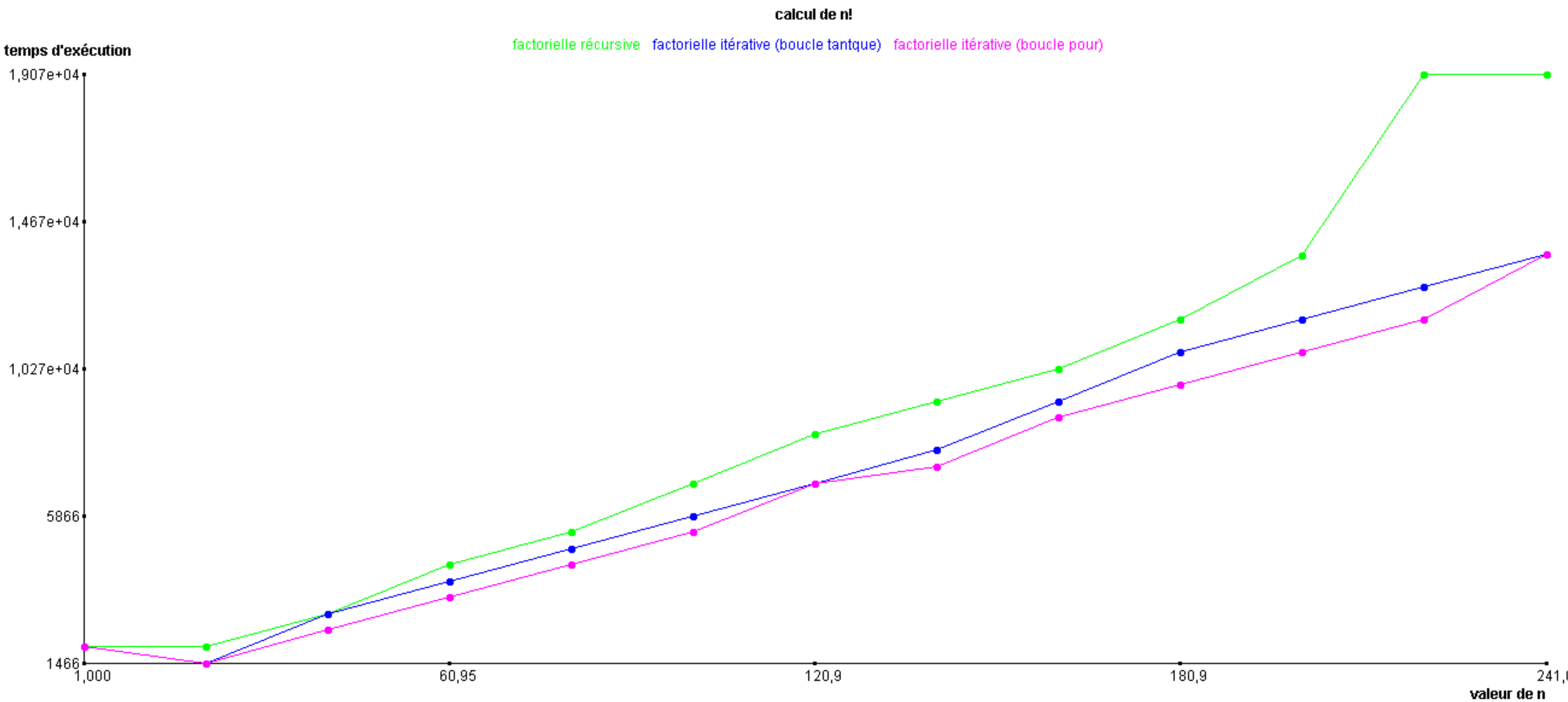
Exemple :

```
fonction avec retour entier factorielleBis(entier i)
     entier résultat;
début
     résultat <- i;
     tantque (i > 1) faire
          i <- i - 1;
          résultat <- résultat * i;
     fintantque
     retourne résultat;
fin
```

L'exécution d'une version récursive d'un algorithme est généralement un peu moins rapide que celle de la version itérative, même si le nombre d'instructions est le même (à cause de la gestion des appels de fonction).

Récursif versus itératif (2/4)

Comparaison expérimentale du calcul de la factorielle en itératif et en récursif.

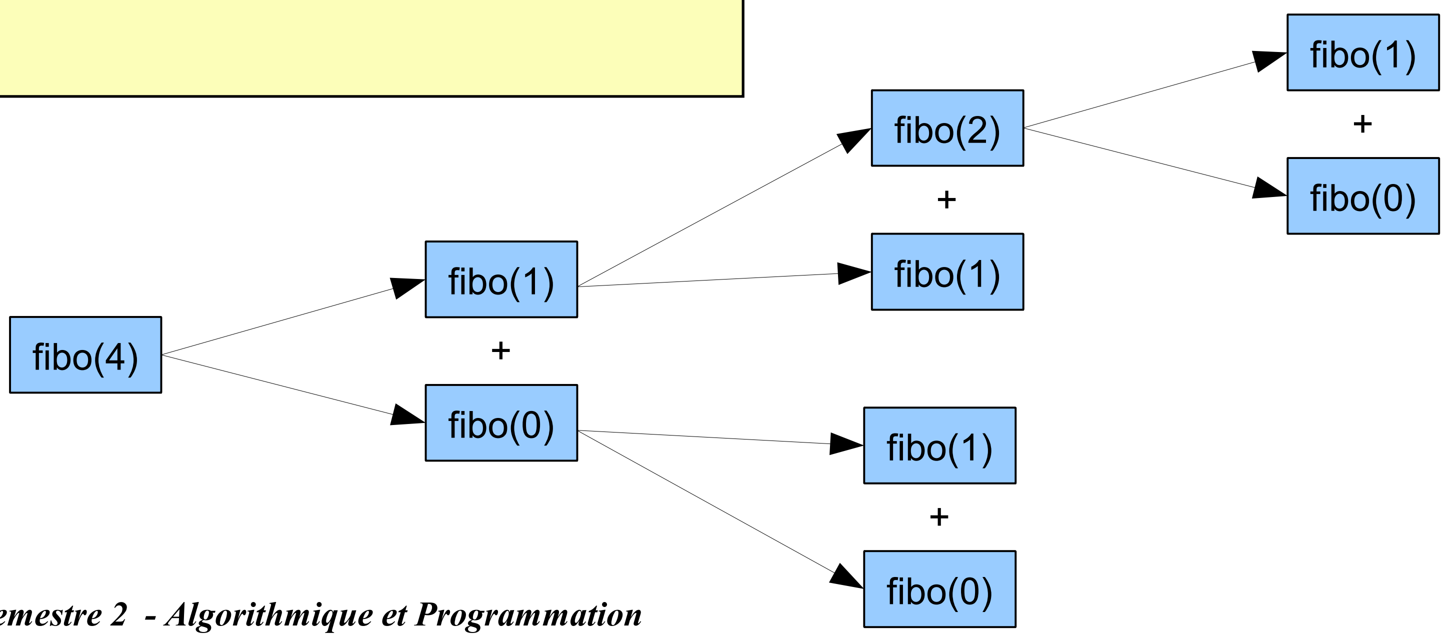


Récursif versus itératif (3/4)

Un algorithme récursif mal écrit peut conduire à exécuter bien plus d'instructions que la version itérative.

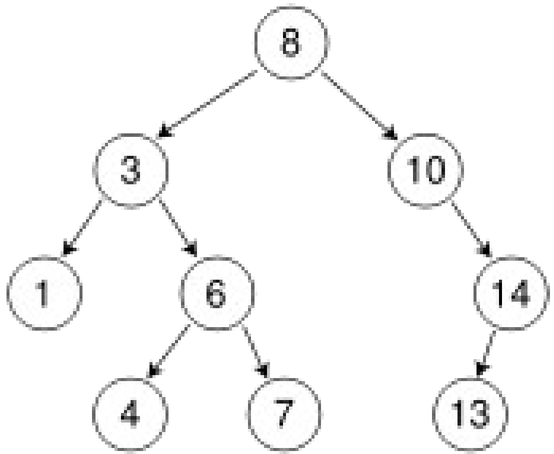
<u>Exemple</u> : la suite de Fibonacci

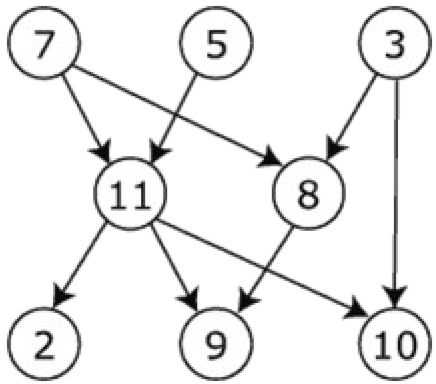
```
fonction avec retour entier fibo(entier n)
début
     si ((n=1) ou (n=0)) alors
          retourne 1;
     sinon
          retourne fibo(n-1)+fibo(n-2);
     finsi
fin
```



Récursif versus itératif (4/4)

Sur des structures de données naturellement récursives, il est bien plus facile d'écrire des algorithmes récursifs qu'itératifs.





Certains algorithmes sont extrêmement difficiles à écrire en itératif.

Récursivité imbriquée

La récursivité imbriquée consiste à faire un appel récursif à l'intérieur d'un autre appel récursif.

<u>Exemple</u> : la suite d'Ackerman

$$A(m,n) = n+1 \text{ si } m = 0,$$

$$A(m,n) = A(m-1,1) \text{ si } n=0 \text{ et } m > 0$$

$$A(m,n) = A(m-1, A(m,n-1)) \text{ sinon}$$

```
fonction avec retour entier ackerman(entier m, entier n)
début
     si (m = 0) alors
          retourne n+1;
     sinon
          si ((m>0) et (n=0)) alors
               retourne ackerman(m-1,1);
          sinon
               retourne ackerman(m-1,ackerman(m,n-1));
          finsi
     finsi
fin
```

Récursivité croisée

La récursivité croisée consiste à écrire des fonctions qui s'appellent l'une l'autre.

Exemple :

```
// cette fonction renvoie vrai si l'entier est pair, faux sinon
// on suppose que l'entier est positif ou nul
fonction avec retour booléen estPair(entier n)
début
     si (m = 0) alors
          retourne VRAI;
     sinon
          retourne estImpair(n-1);
     finsi
fin
```

```
// cette fonction renvoie vrai si l'entier est impair, faux sinon
// on suppose que l'entier est positif ou nul
fonction avec retour booléen estImpair(entier n)
début
     si (m = 0) alors
          retourne FAUX;
     sinon
          retourne estPair(n-1);
     finsi
fin
```

Ecrire un algorithme récursif

<u>Problème</u> : écrire un algorithme récursif réalisant un certain traitement T sur des données D.

1- décomposer le traitement T en sous traitements de même nature mais sur des données plus petites

2- trouver la condition d'arrêt

3- tester éventuellement sur un exemple

4- écrire l'algorithme

Dichotomie récursive (1/2)

<u>Exemple</u> : écrire une fonction récursive qui recherche par dichotomie un élément dans un tableau d'entiers. La fonction renvoie l'indice de l'élément s'il existe et -1 sinon.

1- Décomposition du traitement : rechercher un élément dans le tableau va conduire, si on ne trouve pas l'élément au milieu, à relancer la recherche sur une moitié du tableau, puis sur un quart, etc.

A chaque appel récursif, il faut donc savoir entre quels indices i et j on cherche l'élément.

```
fonction avec retour entier dicho(entier[] t, entier n, entier i, entier j)
...
```

2- Condition d'arrêt : la recherche s'arrête quand on trouve l'élément ou quand il n'y a plus de case où chercher (i>j).

Dichotomie récursive (2/2)

4- Ecriture de l'algorithme :

```
// on suppose t trié par ordre croissant
// on cherche l'élément n dans t entre i et j inclus
fonction avec retour entier dicho(entier[] t, entier n, entier i, entier j)
début
     si (i>j ou t[(i+j)/2]=n) alors
          si (i>j) alors
               retourne -1;
          sinon
               retourne (i+j)/2;
          finsi
     sinon
          si (t(i+j)/2] > n) alors
               retourne dicho(t,n,i,(i+j)/2 - 1);
          sinon
               retourne dicho(t,n,(i+j)/2 + 1,j);
          finsi
     finsi
fin
```

<u>Remarque</u> : pour chercher n dans tout le tableau t, il faut appeler **`dicho(t,n,0,t.longueur-1)`**

Inversion récursive (1/2)

<u>Exemple</u> : écrire une fonction récursive qui inverse l'ordre des éléments dans un tableau d'entiers.

1- Décomposition du traitement : on échange les éléments situés aux extrémités du tableau, et on inverse l'ordre des éléments situés entre ces deux extrémités.

0	7	2	5	3

A chaque appel récursif, il faut donc savoir entre quels indices i et t.longueur-1-i on travaille.

```
fonction sans retour inverse(entier[] t, entier i)
...
```

2- Condition d'arrêt : l'inversion doit s'arrêter quand l'indice i est supérieur ou égal à t.longueur/2.

Inversion récursive (2/2)

4- Ecriture de l'algorithme :

```
// on inverse les éléments d'indices compris entre i et t.longueur-1-i
fonction sans retour inverse(entier[] t, entier i)
     entier temp;
début
     si (i<t.longueur/2) alors
          temp <- t[i];
          t[i] <- t[t.longueur-1-i];
          t[t.longueur-1-i] <- temp;
          inverse(t,i+1);
     finsi
fin
```

Remarque : pour inverser tout le tableau t, il faut appeler **`inverse(t,0)`**

Récursivité terminale et non terminale (1/4)

Une fonction récursive est dite terminale si aucun traitement n'est effectué à la remontée d'un appel récursif (sauf le retour d'une valeur).

Une fonction récursive est dite non terminale si le résultat de l'appel récursif est utilisé pour réaliser un traitement (en plus du retour d'une valeur).

<u>Exemple de non terminalité</u> : forme récursive non terminale de la factorielle, les calculs se font à la remontée.

```
fonction avec retour entier factorielleNT(entier n)
début
     si (n = 1) alors
          retourne 1;
     sinon
          retourne n*factorielleNT(n-1);
     finsi
fin
```

Récursivité terminale et non terminale (2/4)

6

```
factorielleNT(3)
début
     si (3 = 1) alors retourne 1;
     sinon retourne 3*factorielleNT(2);
     finsi
fin
```

2

```
factorielleNT(2)
début
     si (2 = 1) alors retourne 1;
     sinon retourne 2*factorielleNT(1);
     finsi
fin
```

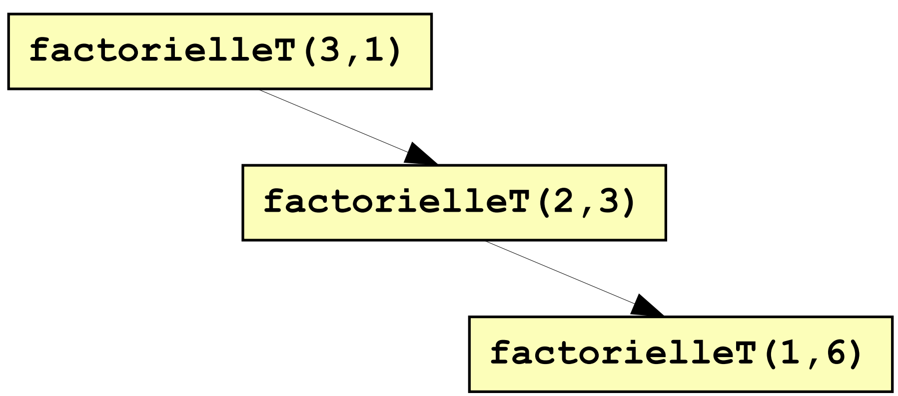
1

```
factorielleNT(1)
début
     si (1 = 1) alors retourne 1;
     sinon retourne 1*factorielleNT(0);
     finsi
fin
```

Récursivité terminale et non terminale (3/4)

<u>Exemple de terminalité</u> : forme récursive terminale de la factorielle, les calculs se font à la descente.

```
// la fonction doit être appelée en mettant resultat à 1
fonction avec retour entier factorielleT(entier n, entier resultat)
début
     si (n = 1) alors
          retourne resultat;
     sinon
          retourne factorielleT(n-1, n * resultat);
     finsi
fin
```



Intérêt de la récursivité terminale

Une fonction récursive terminale est en théorie plus efficace (mais souvent moins facile à écrire) que son équivalent non terminale : il n'y a qu'une phase de descente et pas de phase de remontée.

En récursivité terminale, les appels récursifs n'ont pas besoin d'êtres empilés dans la pile d'exécution car l'appel suivant remplace simplement l'appel précédent dans le contexte d'exécution.

Certains langages utilisent cette propriété pour exécuter les récursions terminales aussi efficacement que les itérations (ce n'est pas le cas de Java).

Il est possible de transformer de façon simple une fonction récursive terminale en une fonction itérative : c'est la dérécursivation.

Dérécursivation (1/3)

Une fonction récursive terminale a pour forme générale :

```
fonction avec retour T recursive(P)
début
     I0
     si (C) alors
          I1
     sinon
          I2
          recursive(f(P));
     finsi
fin
```

T est le type de retour

P est la liste des paramètres

C est la condition d'arrêt

I0 le bloc d'instructions exécuté dans tous les cas

I1 le bloc d'instructions exécuté si C est vraie

I2 et le bloc d'instructions exécuté si C est fausse

f la fonction de tranformation des paramètres

La fonction itérative correspondante est :

```
fonction avec retour T iterative(P)
début
     I0
     tantque (non C) faire
          I2
          P <- f(P);
          I0;
     fintantque
     I1
fin
```

Dérécursivation (2/3)

__Exemple__ : dérécursivation de la factorielle terminale

```
// cette fonction doit être appelée avec a=1
fonction avec retour entier factorielleRecurTerm(entier n, entier a)
début
     si (n <= 1) alors
          retourne a;
     sinon
          retourne factorielle(n-1,n*a);
     finsi
fin
```

```
fonction avec retour entier factorielleIter(entier n, entier a)
début
     tantque (n > 1) faire
          a <- n*a;
          n <- n-1;
     fintantque
     retourne a;
fin
```

Dérécursivation (3/3)

Une fonction récursive non terminale a pour forme générale :

```
fonction avec retour T recursive(P)
début
     I0
     si (C) alors
          I1
     sinon
          I2
          recursive(f(P));
          I3
     finsi
fin
```

T est le type de retour

P est la liste des paramètres

C est la condition d'arrêt

I0 le bloc d'instructions exécuté dans tous les cas

I1 le bloc d'instructions exécuté si C est vraie

I2 et I3 les blocs d'instructions exécutés si C est fausse

f la fonction de tranformation des paramètres

La fonction itérative correspondante doit gérer la sauvegarde des contextes d'exécution (valeurs des paramètres de la fonction.

La fonction itérative correspondante est donc moins efficace qu'une fonction écrite directement en itératif.