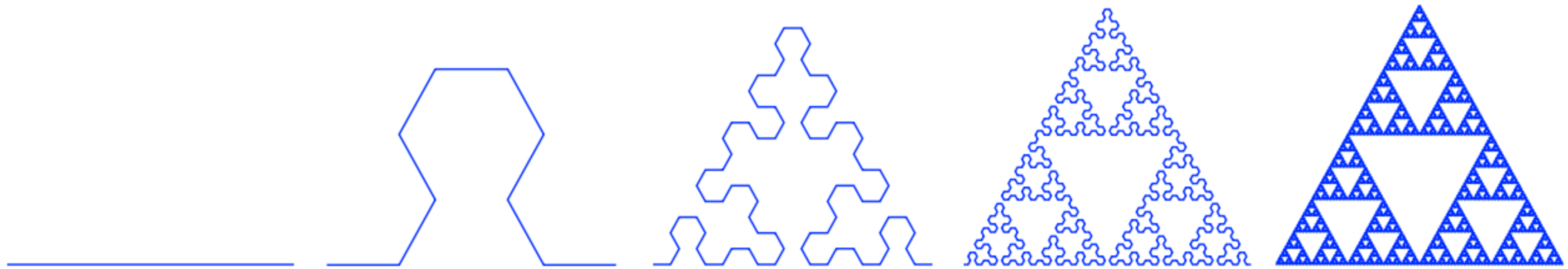
La récursivité

La récursivité

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

Exemple : le triangle de Sierpinski



La récursivité

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

Exemple : *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.

La récursivité

```
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

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

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

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

<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

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.

Pile d'exécution

Attention : 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

Rappel : Itérer : répéter n fois un processus en faisant changer la valeur des variables jusqu'a obtention du résultat.

Un calcul itératif se programme par une boucle (**pour** ou **tant que**)

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

Exercice:

Donner un algorithme (itératif et récursif) qui donne la somme 1+2+3+…..+n

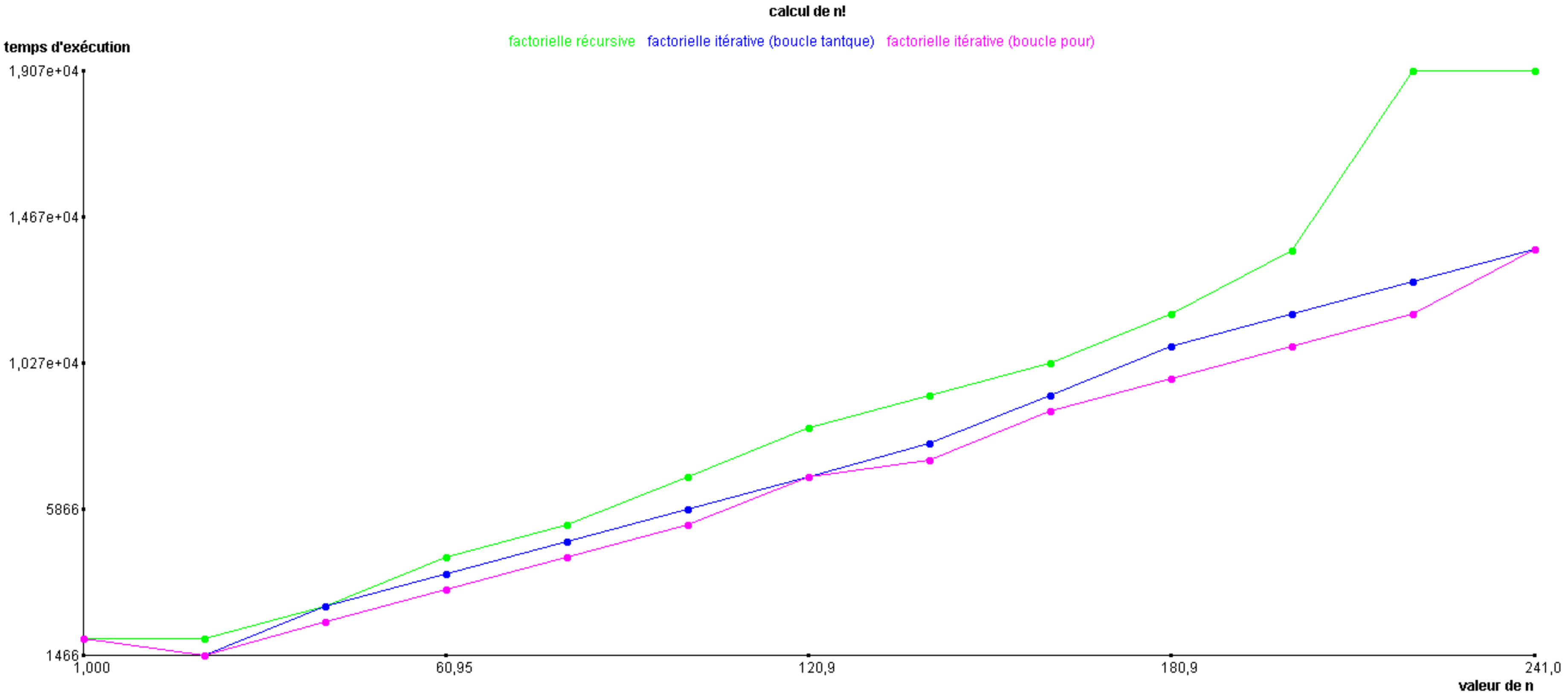
Exemple :

```
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

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



Récursif versus itératif

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

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.

La récursivité

Récursivité simple

Revenons à la fonction puissance $x \to x^n$.

Cette fonction peut être définie récursivement :

$$x^n = \begin{cases} 1 & si\ n = 0 \\ x * x^{n-1} & si\ n \geq 1 \end{cases}$$

La récursivité

- **Récursivité multiple**

  Une définition récursive peut contenir plus d'un appel récursif.

Nous voulons calculer ici les combinaisons $C^n{}_p$ en se servant de la relation de Pascal :

$$C_n^p = \begin{cases} 1 & \text{si } p = 0 \text{ ou } p = n; \\ C_{n-1}^p + C_{n-1}^{p-1} & \text{sinon.} \end{cases}$$

La récursivité

- **Récursivité mutuelle**

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

Ça peut être le cas pour la définition de la parité :

$$\text{pair}(n) = \begin{cases} \textit{vrai} & \text{si } n = 0; \\ \text{impair}(n-1) & \text{sinon}; \end{cases} \quad \text{et} \quad \text{impair}(n) = \begin{cases} \textit{faux} & \text{si } n = 0; \\ \text{pair}(n-1) & \text{sinon.} \end{cases}$$

La récursivité

- **Récursivité mutuelle**

```
// 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
```

La récursivité

- **Récursivité imbriquée**

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

La fonction d'Ackermann est définie comme suit :

$$A(m,n) = \begin{cases} n+1 & \text{si } m = 0 \\ A(m-1,1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m-1, A(m, n-1)) & \text{sinon} \end{cases}$$

La récursivité

Exercice:

Ecrire une fonction qui calcule les valeurs de la série de Fibonacci, définie par :

$$u(0) = 0$$

$$u(1) = 1$$

$$u(n) = u(n-1) + u(n-2)$$

Ecrivez cette fonction sous forme itérative et sous forme récursive. Laquelle des deux variantes est préférable ici ?

La récursivité

```
static int fiboIteratif(int n){
    if ((n == 0) || (n == 1)){
        return n;
    }else{
        int moinsDeux = 0;
        int moinsUn = 1;
        int nouveau;
        for (int i=2; i<n; i++){
            nouveau = moinsDeux + moinsUn;
            moinsDeux = moinsUn;
            moinsUn = nouveau;
        }
        return moinsDeux + moinsUn;
    }
```

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

La forme récursive est plus facile à écrire et plus proche de la définition de la fonction, mais elle est moins efficace que la version itérative

La récursivité

Principe et dangers de la récursivité

- **Principe et intérêt :**

Les mêmes que ceux de la démonstration par récurrence en mathématiques.

On doit avoir :

– un certain nombre de cas dont la résolution est connue, ces « cas simples » formeront les cas d’arrêt de la récursion;

– un moyen de se ramener d’un cas « compliqué » à un cas « plus simple ».

Récursivité terminale et non terminale

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).

Exemple de non terminalité : 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

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

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

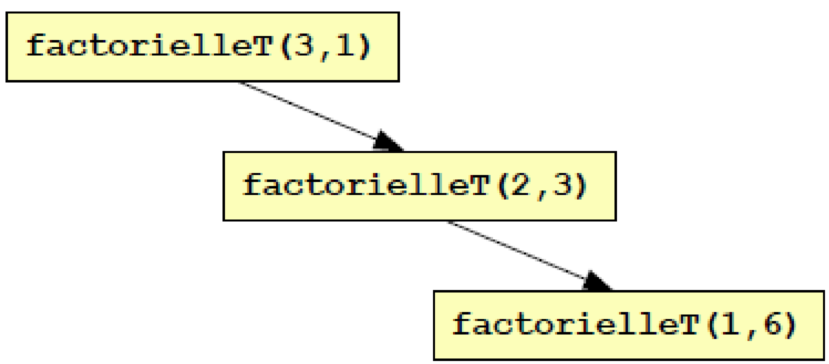
```
factorielleNT(1)
début
     si (1 = 1) alors retourne 1;
     sinon retourne 1*factorielleNT(0);
     finsi
fin
```

Récursivité terminale et non terminale

Exemple de terminalité :

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.

Intérêt de la récursivité terminale

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

Dérécursiver, c'est transformer un algorithme récursif en un algorithme équivalent ne contenant pas d'appels récursifs.

Récursivité terminale

Définition :

Un algorithme est dit récursif terminal s'il ne contient aucun traitement après un appel récursif.

Dérécursivation

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

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

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.

Dérécursivation

Remarques

- Les programmes itératifs sont souvent plus efficaces.
- mais les programmes récursifs sont plus faciles à écrire.
- Les compilateurs savent, la plupart du temps, reconnaître les appels récursifs terminaux, et ceux-ci n'engendrent pas de surcoût par rapport à la version itérative du même programme.
- Il est toujours possible de dérécursiver un algorithme récursif.

Complexité et récursivité

Exemple : calcul récursif de la factorielle

```
// cette fonction renvoie n! (n est supposé supérieur ou égal à 1)
fonction avec retour entier factorielle2(entier n)
début
        si (n = 1) alors
             retourne 1;
         sinon
             retourne n*factorielle2(n-1);
         finsi
fin
```

Paramètre de complexité : la valeur de n

Il n'y a qu'un seul cas d'exécution (pas de cas au pire ou au mieux)

Si $n \neq 1$, le calcul de la factorielle de **n** coûte une comparaison d'entiers, le calcul de la factorielle de **n-1** et une multiplication d'entiers.

Si $n = 1$, le calcul de la factorielle coûte **une comparaison** d'entiers.

Factorielle récursive

On pose une équation de récurrence :

appelons c(n) la complexité

$$c(n) = ce + c(n-1) + oe \text{ si } n \neq 1$$

$$c(1) = ce$$

On résoud cette équation de récurrence :

$$c(n) = n*ce + (n-1)*oe = O(n)$$

La complexité de la factorielle récursive est donc linéaire, comme celle de la factorielle itérative.

A l'exécution, la fonction récursive est un peu moins rapide (pente de la droite plus forte) du fait des appels récursifs

Complexité et récursivité

En général, dérécursiver un algorithme ne change pas la forme de sa complexité, pas plus que passer en récursivité terminale!

Il existe diverses techniques pour la résolution des équations de récurrence (méthode des fonctions génératrices et décomposition des fractions rationnelles, transformée en Z, …).

Theoreme : soit T(n) une fonction définie par l'e=équation de récurrence suivante, où $b \geq 2$, $k \geq 0$, $a > 0$, $c > 0$ et $d > 0$:

$$T(n) = a*T(n/b) + c*n^k$$

si $a > b^k$	alors	$T(n)=\Theta(n\log_b(a))$
si $a = b^k$	alors	$T(n)=\Theta(n^k*\log(n))$
si $a < b^k$	alors	$T(n)=\Theta(n^k)$

En conclusion

-Les algorithmes récursifs sont simples (c'est simplement une autre façon de penser).

-Les algorithmes récursifs permettent de résoudre des problèmes complexes.

-Il existe deux types de récursivités :

-**terminale**, qui algorithmiquement peuvent être transformée en algorithme non récursif.

-**non terminale**

-Les algorithmes récursifs sont le plus souvent plus gourmands en ressource que leurs équivalents itératifs.