Introduction à la programmation en Java

Faculté des Sciences de Nice

Licence Math-Info 2006-2007

Module L1I1

Frédéric MALLET

Jean-Paul ROY

Où en sommes-nous ?

- Nous savons rédiger le texte d'une classe d'objets, avec ses **champs**, ses **constructeurs**, ses **méthodes**.
- Nous pouvons exprimer qu'une méthode a ou n'a pas de résultat.
- Nous savons utiliser les collections de taille fixe (les tableaux) et les collections de taille variable (`ArrayList`)
- Nous savons construire et transformer du texte (`String`)
- Le graphisme tortue [polaire et cartésien] n'a plus de secrets…
- Nous connaissons les 3 formes de boucles !
  - le `for` si on connaît exactement le nombre de répétitions :

```
for (initialisation ;  condition_booléenne ; continuation )
{   corps ;  }
```

  - le `do…while` si on ne connaît pas le nombre de répétitions mais qu'on est sûr d'exécuter le corps au moins une fois

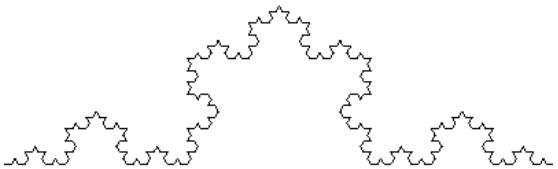
```
do { corps } while (condition_booléenne) ;
```

  - le `while` dans les autres cas

```
while (condition_booléenne) { corps; }
```


COURS 11

La Récursivité



Raisonner par récurrence

- Le **principe de récurrence** (en anglais : *induction principle*) est une technique majeure en mathématiques. Il permet à la fois de :
  - **démontrer** des théorèmes
  - **construire** des objets mathématiques

Démontrer des théorèmes	Démonstration
♦ Pour tout $n \geq 0$, on a : $\sum_{k=0}^{n} k = \frac{n(n+1)}{2}$	En deux temps : • Si n=0, alors c'est vrai : 0=0 • Si n>0, **supposons** le théorème vrai pour n-1 : $\sum_{k=0}^{n-1} k = \frac{(n-1)n}{2}$ (HR) Alors $\sum_{k=0}^{n} k = \left(\sum_{k=0}^{n-1} k\right) + n$ $= \frac{(n-1)n}{2} + n = \frac{n(n+1)}{2}$

Construire des objets	Construction
♦ L'ensemble **N** des entiers naturels	• $0 \in \mathbf{N}$ • Si $i \in \mathbf{N}$, alors $s(i) \in \mathbf{N}$ Notation : s(0)=1, s(s(0))=2,…
♦ Un espace vectoriel de dimension $n \geq 1$.	• Un e.v. de dimension 1 est une droite vectorielle. • Un e.v. $\vec{E}$ de dimension $n > 1$ est la somme directe $\vec{D} \oplus \vec{F}$ d'une droite vectorielle $\vec{D}$ et d'un e.v. de dimension n-1 $\vec{F}$.
♦ Un ensemble à n éléments.	• Un ensemble à 0 élément est $\varnothing$ • Un ensemble E à $n \geq 1$ élément est la réunion $\{\omega\} \cup F$ où F est un ensemble à n-1 éléments.

♦ Le principe de récurrence est bien adapté aux raisonnements sur des objets définis… par récurrence. Exemple typique : les entiers naturels !

♦ Bien comprendre le *moteur à deux temps* d'une récurrence :

1) Prouver (ou construire, ou programmer) le CAS DE BASE, en général n=0 mais pas toujours : c'est « le cas le plus simple ».
2) Supposer la preuve (ou la construction, ou la programmation) effectuée pour le rang n-1, et prouver alors (ou construire, ou programmer) pour le rang n.

HR

♦ Et vérifier un point capital : à force de passer de n à n-1, **on doit finir par converger vers le cas de base**, sinon gare !…

$$\begin{cases} 0! = 1 \\ n! = n \times (n-1)! \end{cases}$$ BIEN

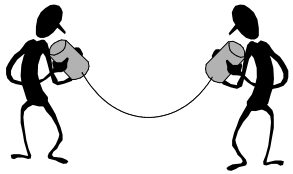
$$\begin{cases} 0! = 1 \\ n! = \dfrac{(n+1)!}{n+1} \end{cases}$$ MAL

Deux clés pour réussir en récurrence

1) Le **principe de récurrence forte** est souvent utilisé en programmation, mais aussi en maths. Il remplace l'hypothèse de récurrence (HR) : « supposons vrai POUR n-1 » par « supposons vrai JUSQU'À n-1 » !

2) Si ça résiste, ne pas hésiter à **modifier (voire à généraliser) la propriété à prouver** !

IL EST SOUVENT ET PARADOXALEMENT PLUS FACILE DE PROUVER (OU CONSTRUIRE OU PROGRAMMER) UN « TRUC » GENERAL QU'UN « TRUC » PARTICULIER !



Qu'on se le dise !

♦ Exemple : Prouver que $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} \notin \mathbf{N}$, pour tout $n \geq 2$

Le passage de n-1 à n résiste (vérifiez-le !). Regardons les premiers termes:

$$H_2 = \frac{3}{2}\ (OK),\ H_3 = \frac{11}{6},\ H_4 = \frac{25}{12}, \ldots$$

Généralisons ce qu'il faut démontrer : prouvons un « truc » plus difficile:

$$H_n = \frac{\text{impair}}{\text{pair}} \quad \text{pour tout } n \geq 2$$

SUPPOSONS donc (HR) que H_k = impair/pair, pour tout $k \leq n-1$, et montrons que H_n = impair/pair.

• Si n est **impair** : $H_n = H_{n-1} + 1/n$ = impair/pair + 1/impair = impair/pair. **OK!**
• Si n est **pair** : $n = 2k$ et $H_n = H_{2k} = [1+1/3+\ldots+1/(2k-1)] + [1/2+1/4+\ldots+1/(2k)]$
D'où H_n = ?/impair + 1/2 H_k = ?/impair + impair/pair = impair/pair. **OK!**

Ne pas hésiter à GENERALISER !

HR

Quel rapport avec la programmation ?

♦ C'est presque pareil… inutile d'en faire tout un plat ! La seule difficulté est d'enlever aux mathématiques le monopole de la récurrence.
♦ Peut-on programmer par récurrence ? Oui.
♦ Est-ce plus facile qu'avec une boucle ? Cela dépend des problèmes …
♦ Est-ce plus efficace qu'une boucle ? Pas forcément, mais la facilité de conception peut l'emporter sur le temps d'exécution. A négocier…

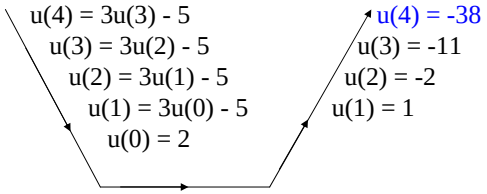
♦ L'exemple typique : une suite définie par récurrence en Analyse.

$$\begin{cases} u_0 = 2 \\ u_n = 3u_{n-1} - 5 \quad si \quad n > 0 \end{cases}$$

```
static int u (int n)    // n ≥ 0
{
   if (n == 0) return 2;
   else return 3 * u(n - 1) - 5;
}
```


♦ Comment ça marche ? Ce n'est pas votre problème, les compilateurs sont plus ou moins « intelligents » ! Votre problème, c'est la **rigueur** de l'hypothèse de récurrence : pour calculer u(n) avec n > 0, je *prétends savoir calculer* u(n-1)…

♦ Bon, ok : exécution en Java : `int n = u(4);`




```
static public int u (int n)
{
   if (n == 0) return 2;
   return 3 * u(n - 1) - 5;
}
```

♦ Il y a donc des **calculs en attente**, le calcul sera fait « en profondeur » [dans une **pile**, cf. cours du semestre 2]. Le compilateur s'en charge…

♦ L'erreur classique : penser la récurrence comme une « boucle », en essayant de visualiser l'histoire du calcul pas à pas. Raisonner plutôt de manière statique : traiter le cas de base, puis le passage de n-1 à n.

♦ Autre exemple typique : la factorielle n! d'un entier n ≥ 0

$$0! = 1 \quad et \quad n! = n \times (n-1)! \quad si \quad n > 0$$

```
static int facRec(int n)          // n ≥ 0
{
   if (n == 0) return 1;
   return n * facRec(n-1);
}
```

♦ Ce n'est pas le seul schéma *récursif* possible :

programmé par récurrence

```
static int facRec2(int n, int acc)    // n ≥ 0
{
   if (n == 0) return acc;
   return facRec2(n-1, acc*n);
}
```

facRec2(5,1)
= facRec2(4,5)
= facRec2(3,20)
= facRec2(2,60)
= facRec2(1,120)
= facRec2(0,120)
= 120

Notes personnelles

♦ Il serait malvenu de poser un **assert** pour se protéger du cas n < 0 juste avant le if, car assert serait exécuté chaque fois. Dans ce cas, on utilise un « lanceur » qui va passer la main à la méthode récursive elle-même :

```
static public int fac (int n)
{
   assert n >= 0 : "fac(" + n + ") n'existe pas!";
   return facRec(n);
}


static private int facRec (int n)   // n est ≥ 0
{
   if (n == 0) return 1;
   return n * facRec(n-1);
}
```

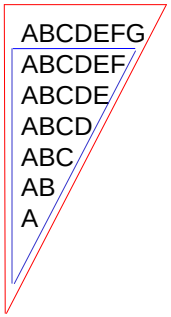
♦ Pour lancer facRec2(n,f), on écrirait aussi un lanceur facRec2(n) qui se protègerait par un **assert** et demanderait le calcul de facRec2(n,1)…

Voir des sous-schémas récursifs…

♦ La programmation d'une méthode récursive (avec ou sans résultat) se ramène souvent à la visualisation, au sein d'un calcul, d'un sous-calcul « isomorphe ». Deux exemples :

La factorielle : `n! = 1 * 2 * 3 * . . . * (n-1) * n`

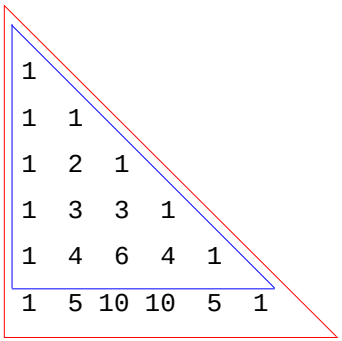
L 'épluchage d'un mot :



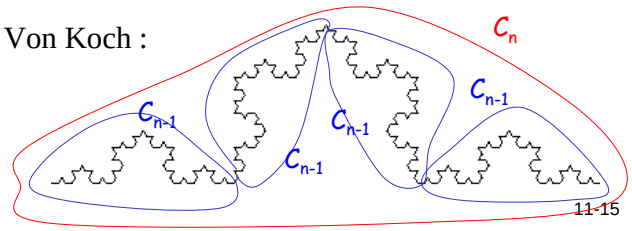

```
static void eplucheMot(String str)
{
   if (!str.equals("")) {
     System.out.println(str);
     eplucheMot(str.substring(0,str.length()-1));
   }
}
```

Terminaison : str.length() est strictement décroissante !

Le triangle de Pascal :



La courbe fractale de Von Koch :

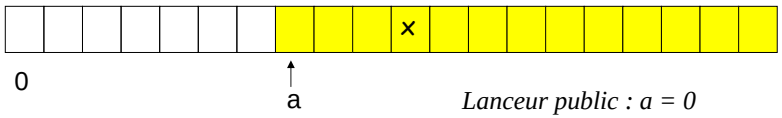


Récursivité et tableaux

♦ Pour travailler dans un sous-tableau, il est usuel de passer en paramètres les indices entre lesquels on travaille, plutôt que construire effectivement le sous-tableau !

♦ Exemple : **recherche séquentielle** d'un élément dans un tableau non trié.

```
/** retourne le rang de la première apparition de x dans le tableau tab,
 *  à partir de l'indice a inclus. Retourne -1 en cas d'échec. */
static int indexOfSeq(int x, int[] tab, int a)
{
   if (a >= tab.length) return -1;
   if (tab[a] == x)     return a;
   return indexOfSeq(x, tab, a+1);
}
```

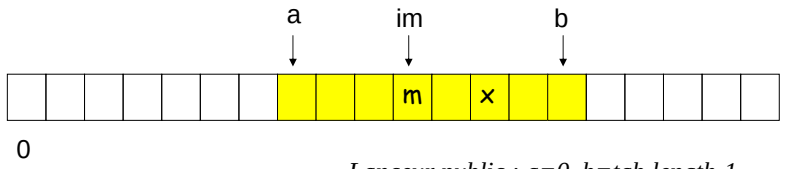


Lanceur public : a = 0

♦ Exemple : **recherche dichotomique** d'un élément dans un tableau trié.

♦ à force de diviser l'intervalle de recherche en deux, on est conduit à généraliser le problème : chercher dans un intervalle d'indices [a,b]:

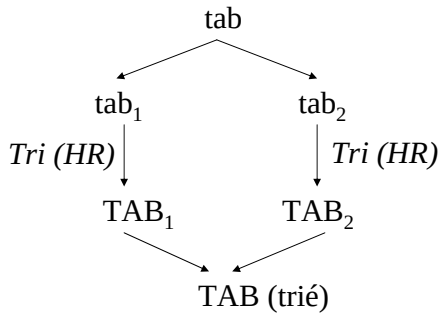
```
/** Retourne le rang d'une apparition de x dans le tableau croissant tab,
 *  parmi les indices entre a et b inclus. Retourne -1 si échec.
 */
static int indexOfDicho(int x, int[] tab, int a, int b)
{
   if (a > b) return -1;                  // échec !
   int im = (a + b) / 2, m = tab[im];     // on regarde le milieu
   if (x == m) return im;
   if (x < m) return indexOfDicho(x, tab, a, im-1);
   return indexOfDicho(x, tab, im+1, b); // si x > m
}
```



Lanceur public : a=0, b=tab.length-1

♦ Exemple : **trier récursivement un tableau !**

♦ Reprenons l'idée de **dichotomie** : on partage le tableau ***tab*** en deux parties ***tab1*** et ***tab2*** (pas forcément égales !) et on trie séparément chacune des parties par récurrence…



♦ Pour éviter l'étape de fusion des tableaux triés ***TAB$_1$*** et ***TAB$_2$***, il suffit de s'arranger pour que ces parties soient *déjà en place* dans le résultat, donc que :

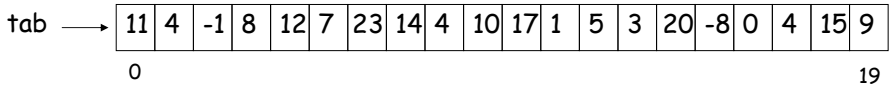
$$\forall x_1 \in TAB_1,\ \forall x_2 \in TAB_2 : x_1 \leq x_2$$

♦ L'algorithme **quicksort** (tri par pivot) consiste à adopter pour la séparation en tab$_1$ et tab$_2$ la stratégie suivante :

♦ Soit p un élément quelconque du tableau, nommé « pivot ». On choisit:

- tab1 = le sous tableau des éléments ≤ p
- tab2 = le sous tableau des éléments > p.

♦ Exemple, soit **tab** un tableau de 20 entiers :



♦ Prenons comme pivot le premier élément 11. Le **partitionnement** regroupe en tête les éléments ≤ 11, suivis du pivot, puis de ceux > 11. La méthode récursive générale chargée de ce travail sera :

```
static void partition (int[] tab, int a, int b)
```

Travail dans [a,b] avec pivot en tête !

http://en.wikipedia.org/wiki/Quicksort

Notes personnelles

*/** partitionne tab[a..b] et retourne l'indice du pivot à la fin */*

```
static int partition(int[] tab, int a, int b) {
   int ip = a, p = tab[a];  // ∀i<ip, tab[i]<pivot
   echanger(tab, ip, b);
   for(int i = a; i < b; i++) { // ip doit être le + gd possible
     if (tab[i] <= p) {          // On trouve un élément inférieur au pivot
       echanger(tab, ip, i);
       ip ++;                    // on peut donc augmenter ip
     }
   }
   echanger(tab, ip, b);
   return ip;
}
```

11	4	-1	8	12	7	23	14	4	10	11	1	5	3	20	-8	0	4	15	9

0 … 19

partition(tab, 0, 19)

9	4	-1	8	7	4	10	11	1	5	3	-8	0	4	11	12	23	14	15	20

0 … 19

0 … 19

11	4	-1	8	12	7	23	14	4	10	11	1	5	3	20	-8	0	4	15	9

L'élément ip est en **bleu**, le pivot en **rouge**, l'élément i en jaune

9	4	-1	8	12	7	23	14	4	10	11	1	5	3	20	-8	0	4	15	11
9	4	-1	8	12	7	23	14	4	10	11	1	5	3	20	-8	0	4	15	11
9	4	-1	8	12	7	23	14	4	10	11	1	5	3	20	-8	0	4	15	11
9	4	-1	8	12	7	23	14	4	10	11	1	5	3	20	-8	0	4	15	11
9	4	-1	8	12	7	23	14	4	10	11	1	5	3	20	-8	0	4	15	11
9	4	-1	8	12	7	23	14	4	10	11	1	5	3	20	-8	0	4	15	11
9	4	-1	8	7	12	23	14	4	10	11	1	5	3	20	-8	0	4	15	11
9	4	-1	8	7	12	23	14	4	10	11	1	5	3	20	-8	0	4	15	11
9	4	-1	8	7	12	23	14	4	10	11	1	5	3	20	-8	0	4	15	11
9	4	-1	8	7	4	23	14	12	10	11	1	5	3	20	-8	0	4	15	11
9	4	-1	8	7	4	10	14	12	23	11	1	5	3	20	-8	0	4	15	11

0 … 19

11	4	-1	8	12	7	23	14	4	10	11	1	5	3	20	-8	0	4	15	9

L'élément ip est en **bleu**, le pivot en **rouge**, l'élément i en jaune

9	4	-1	8	7	4	10	14	12	23	11	1	5	3	20	-8	0	4	15	11
9	4	-1	8	7	4	10	11	12	23	14	1	5	3	20	-8	0	4	15	11
9	4	-1	8	7	4	10	11	1	23	14	12	5	3	20	-8	0	4	15	11
9	4	-1	8	7	4	10	11	1	5	14	12	23	3	20	-8	0	4	15	11
9	4	-1	8	7	4	10	11	1	5	3	12	23	14	20	-8	0	4	15	11
9	4	-1	8	7	4	10	11	1	5	3	12	23	14	20	-8	0	4	15	11
9	4	-1	8	7	4	10	11	1	5	3	12	-8	14	20	23	0	4	15	11
9	4	-1	8	7	4	10	11	1	5	3	12	-8	0	20	23	14	4	15	11
9	4	-1	8	7	4	10	11	1	5	3	12	-8	0	4	23	14	20	15	11

9	4	-1	8	7	4	10	11	1	5	3	12	-8	0	4	11	14	20	15	23

- La méthode **`echanger(tab, i, j)`** permute les éléments d'indices **i** et **j** à l'intérieur du tableau **tab** (cf. TP).

11	4	-1	8	12	7	23	14	4	10	17	1	5	3	20	-8	0	4	15	9
				i									j						
11	4	-1	8	3	7	23	14	4	10	17	1	5	12	20	-8	0	4	15	9

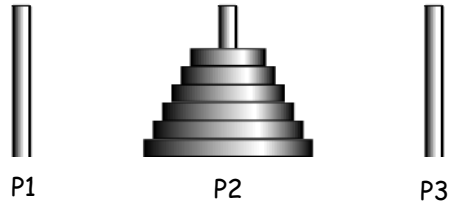
- L'algorithme de tri proprement dit de **tab[a..b]** effectue la partition puis invoque les hypothèses de récurrence sur chacune des parties :

```
static void quicksort(int[] tab, int a, int b) {
   if (b > a) {
      int ip = partition(tab, a, b);
      quicksort(tab, a, ip - 1);       // HR1
      quicksort(tab, ip + 1,  b);      // HR2
   }
}
```

Lanceur avec tab, a = 0, b = tab.length-1

Jouer par récurrence !

♦ Le jeu des **Tours de Hanoï** est connu depuis l'Antiquité. On dispose de 3 piliers P1, P2 et P3 :



♦ Il faut faire passer tous les disques de P2 vers P1, avec les règles :
- On ne bouge qu'un seul disque à la fois ;
- On ne peut poser un disque que sur un disque plus grand ;
- On peut poser un disque sur un pilier vide.

♦ Pour 3 disques, c'est facile :

P2->P1, P2->P3, P1->P3, P2->P1, P3->P2, P3->P1, P2->P1 (7 coups !)

♦ Mais pour 6 disques ???

♦ On se propose de faire calculer la solution (plus de 50 coups !) par Java et … par récurrence sur le nombre n de disques !

♦ Soit donc à résoudre le problème à n disques avec les piliers P1, P2, P3 :

```
static void hanoi (int n, String dst, String src, String tmp)
```

♦ Pour n=0, c'est facile : rien à faire !!!

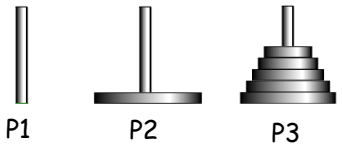
```
if (n == 0) { }
else ...
```

♦ Si n>0, **HYPOTHESE DE RECURRENCE** : supposons que l'on sache résoudre le problème à n-1 disques. Montrons qu'alors on sait le résoudre pour n disques !

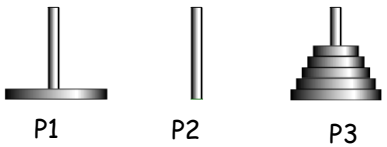
♦ Je peux donc prendre n-1 disques d'un seul coup et les déplacer sur un pilier libre, à condition d'avoir un pilier intermédiaire vide.

♦ Déplaçons donc les n-1 disques au sommet de src(P2) vers tmp(P3) :

```
if (n == 0) { }
else {
   hanoi(n-1, tmp, src, dst);
   ...
```



♦ Le grand disque sur src(P2) peut alors migrer immédiatement vers dst(P1) :

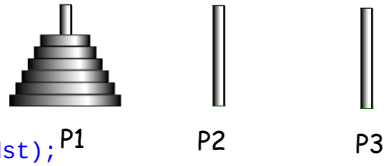



```
if (n == 0) { }
else {
   hanoi(n-1, tmp, src, tmp);
   System.out.println(src + "->" + dst);
   ...
```


Notes personnelles

♦ Et par une seconde hypothèse de récurrence, je peux prendre les n-1 disques de tmp(P3) et les déplacer sur dst(P1) en me servant de src(P2) comme intermédiaire !…

```
if (n == 0) { }
else {
   hanoi(n-1, tmp, src, dst);
   System.out.println(src + "->" + dst);
   hanoi(n-1, dst, tmp, src);
}
```




```
static void hanoi(int n, String dst, String src, String tmp)
{
   if (n > 0) {
     hanoi(n-1, tmp, src, dst);
     System.out.println(src + "->" + dst);
     hanoi(n-1, dst, tmp, src);
   }
}
```

Récurrence double !

```
hanoi(7,"P1","P2","P3");
P2->P1, P2->P3, P1->P3, P2->P1, P3->P2, P3->P1, P2->P1 ...
```

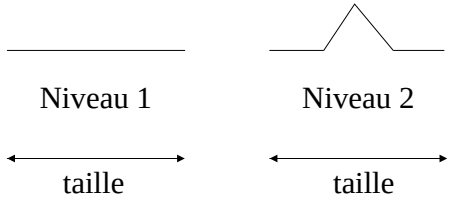

Une courbe fractale : Von Koch (1904)

♦ Une telle courbe est « invariante d'échelle » : si l'on zoome sur l'une de ses parties, on retrouve la même forme que la courbe toute entière…

♦ Une manière simple d'en dessiner consiste à définir une suite d'approximations de niveaux 1, 2, 3, 4, …

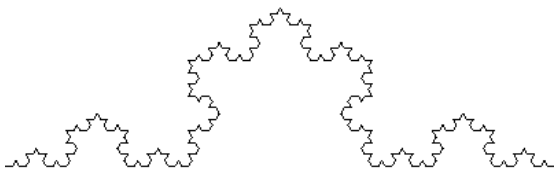
♦ On passe de la courbe de niveau n à la courbe de niveau n+1 par une règle uniforme, donc il s'agit d'une suite de courbes définies par récurrence !

♦ La véritable courbe fractale serait la courbe limite, impossible à atteindre sur un ordinateur, elle ne vit que dans l'esprit du matheux…



Et l'on procède de manière identique sur chacun des 4 segments du niveau 2 pour passer au niveau 3…


```
void vonKoch(Turtle t, int niveau, double taille)
{
   if (niveau == 1) t.forward(taille);
   else {
           vonKoch(t, niveau-1, taille/3);
           t.left(60);
           vonKoch(t, niveau-1, taille/3);
           t.right(120);
           vonKoch(t, niveau-1, taille/3);
           t.left(60);
           vonKoch(t, niveau-1, taille/3);
        }
}
```



La courbe de niveau n est constituée de 4 courbes de niveau n-1.

Récursivité et Itération

♦ Deux univers antagonistes.

♦ Pour programmer un algorithme, commencer par se positionner :

- va-t-on construire une boucle ?
- va-t-on plutôt penser par récurrence ?

♦ C'est une affaire très discutée ! Il y a des tenants des deux écoles. Seule la pratique permet de « sentir » si le problème est récursif : peut-on le réduire à un problème identique portant sur des données plus petites ?

♦ Par exemple, réduire le calcul de n! à celui de (n-1)!

♦ Ou bien voir n! comme une accumulation dans une variable ?

♦ Mais quand même : certaines récurrences cachent des boucles, et d'autres pas !…

La récursivité enveloppée (la « vraie »)

♦ Reprenons la 1ère version de la factorielle (page 11-11) :

```
static int facRec(int n)            // n ≥ 0
{
   if (n == 0) return 1;
   return n * facRec(n-1);
}
```

L'appel récursif (là où la fonction se rappelle elle-même) est **enveloppé** (suivi) par une multiplication qui sera mise en attente (« empilée ») le temps que facRec(n-1) soit disponible !

♦ La plupart des récurrences sont de ce type. Il n'est en général pas immédiat de les traduire en itérations (while, for, etc.).

♦ Si les opérations en attente sont en trop grand nombre, BOUM !

La récursivité terminale (la « fausse »)

♦ Reprenons la 2ème version de la factorielle (page 10-11) :

```
static int facRec2(int n, int f)    // n ≥ 0
{
   if (n == 0) return f;
   return facRec2(n-1, f*n);
}
```

L'appel récursif (là où la fonction se rappelle elle-même) n'est **pas enveloppé** (pas suivi) par une autre opération. Il est en position « terminale ». Il est alors facile de le traduire en boucle while :

```
tant que n ≠ 0
   (n, f) = (n-1, f*n);
résultat = f;
```


♦ Hélas, le langage Java ne permet pas les affectations multiples, comme

```
(n, f) = (n-1, f*n);
```

qui permettrait de passer de (3, 20) à (2, 60) directement. Il faut donc le faire en deux temps, mais alors **attention !**

~~`n = n-1;`~~
~~`f = f * n;`~~ ou bien

```
f = f * n;
n = n-1;
```

?

et la version « dé-récursivée » (rendue itérative) de `facRec2` serait :

```
static int facRec2(int n)    // n ≥ 0
{
   int f = 1;
   while (n > 0) {
     f = f * n;
     n = n - 1;
   }
   return f;
}
```


Notes personnelles