Récursivité

On appelle «fonction récursive» toute fonction qui fait appel à elle même. Nous allons voir que l'écriture de fonctions récursives permet d'écrire de façon simple des algorithmes qui sont beaucoup moins simples à écrire de façon itérative. La récursivité est une façon un peu plus abstraite de résoudre un problème que la manière itérative. Elle conduit à écrire des programmes très sobres en nombres de lignes de codes mais pas toujours très efficaces en terme de temps d'exécution.

Prenons l'exemple de la fonction qui calcule la somme des entiers de 1 à n. L'algorithme itératif est une représentation abstraite de la façon dont un enfant de primaire réaliserait l'opération.

$somme :=$

$$\textbf{proc}(n{::}nonnegint)\,\textbf{local}s,\ i;\ s := 0\,;\ \textbf{for}\,i\,\textbf{from}\,0\,\textbf{to}\,n\,\textbf{do}\,s := s + i\,\textbf{od}\,;\ \text{return}(s)\,\textbf{end}$$

L'idée de l'algorithme itératif est que, pour calculer la somme des entiers de 1 à n, on calcule d'abord la somme des entiers de 1 à $n-1$ puis on lui ajoute n, ce qui donne le programme suivant :

$$somrec := \textbf{proc}(n)\,\text{return}(n + \text{somrec}(n-1))\,\textbf{end}$$

Le problème est que le programme précédent ne s'arrête jamais, la fonction `sommrec` fait indéfiniment appel à elle-même. Maple s'en aperçoit car il limite le nombre d'appels d'une fonction à elle même comme le montre l'exemple suivant :

```
> somrec(10);
Error, (in somrec) too many levels of recursion
```

Pour que le programme fonctionne correctement, il est nécessaire de ne travailler qu'avec des entiers positifs ou nuls et de retourner la valeur 0 pour 0, ce qui nous donne le programme suivant qui marche merveilleusement bien :

```
somrec := proc(n::nonnegint)
    if n = 0 then return 0 else return n + somrec(n-1) end if
end proc
> somrec(10);
```

55

On peut visualiser les appels successifs de la fonction `sommrec` grâce à la fonction Maple «`trace`».

```
{--> enter somrec, args = 5
```

précise que la fonction «`somrec`» a été appelée avec l'argument 5, quant à :

```
<-- exit somrec (now at top level) = 15}
```

il précise que l'on quitte la fonction «`somrec`» et qu'elle retourne la valeur 15.

```
> trace(somrec); somrec(5);
```

somrec

```
{--> enter somrec, args = 5

{--> enter somrec, args = 4

{--> enter somrec, args = 3

{--> enter somrec, args = 2

{--> enter somrec, args = 1

{--> enter somrec, args = 0

<-- exit somrec (now in somrec) = 0}

<-- exit somrec (now in somrec) = 1}

<-- exit somrec (now in somrec) = 3}

<-- exit somrec (now in somrec) = 6}

<-- exit somrec (now in somrec) = 10}

<-- exit somrec (now at top level) = 15}
```

15

Il faut voir la programmation récursive comme une nouvelle façon de formuler les algorithmes. Les différents problèmes ne se prêtent pas tous à la programmation récursive. Par contre pour certains problèmes les solutions proposées avant même l'avènement des ordinateurs sont des solutions qui font naturellement appel à la récursivité.

Exemple : calcul du PCDG suivant l'algorithme d'Euclide. L'algorithme d'Euclide repose sur le résultat suivant :

$$\mathrm{PGCD}(a,b) = \begin{cases} a \text{ si } b = 0 \\ \mathrm{PGCD}(b, a \bmod b) \text{ sinon} \end{cases}$$

Il conduit aux versions suivantes, la première récursive et la deuxième itérative :

PGCD1 := **proc**(*a*::*integer*, *b*::*integer*)
if $b = 0$ **then return** a **else return** PGCD1(b, $a \bmod b$) **end if**
end proc

Version itérative :

PGCD2 := **proc**(*a*::*integer*, *b*::*integer*)
local x, y, z;
$x := a$; $y := b$; **while** $y \neq 0$ **do** $z := y$; $y := x \bmod y$; $x := z$ **end do** ; **return** x
end proc

Voici un exemple où on est heureux de pouvoir disposer de la récursivité. Il s'agit d'un problème proposé par le mathématicien Lucas à la fin du XIX^e siècle. Le problème était à peu près formulé de la manière suivante : «Dans un monastère bouddhiste du Vietnam dans la province de Hanoï, on trouve au milieu d'une cour intérieure trois pieux sur lesquels sont placés des disques de différentes tailles. Au départ, vers l'an mille de notre ère, les vingt disques étaient sur le premier pieu rangés sous forme

pyramidale. Les moines de ce temple ont la lourde tâche de déplacer les disques du premier pieu sur le second, à raison de 1 par jour, et en faisant en sorte de ne pas poser un disque sur un disque plus petit et en ne se servant que des trois pieux disponibles. Il est dit que le jour où le dernier disque aura été déplacé sera celui de la fin du monde.» La question posée par Lucas était de savoir quand aurait lieu la fin du monde. La question que nous allons nous poser est de décrire tous les mouvements à effectuer pour déplacer n disques d'un pieu à un autre en respectant les contraintes imposées aux moines.

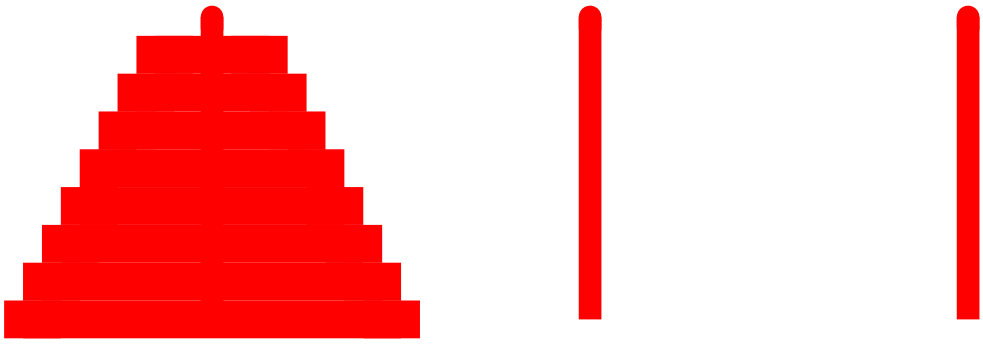
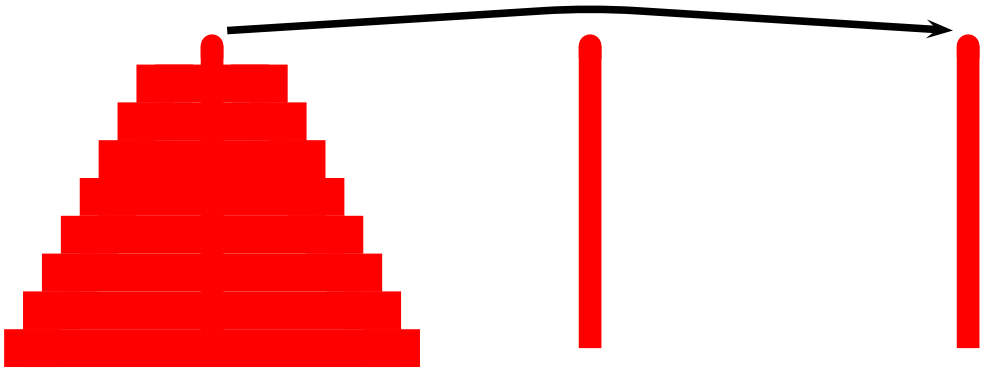


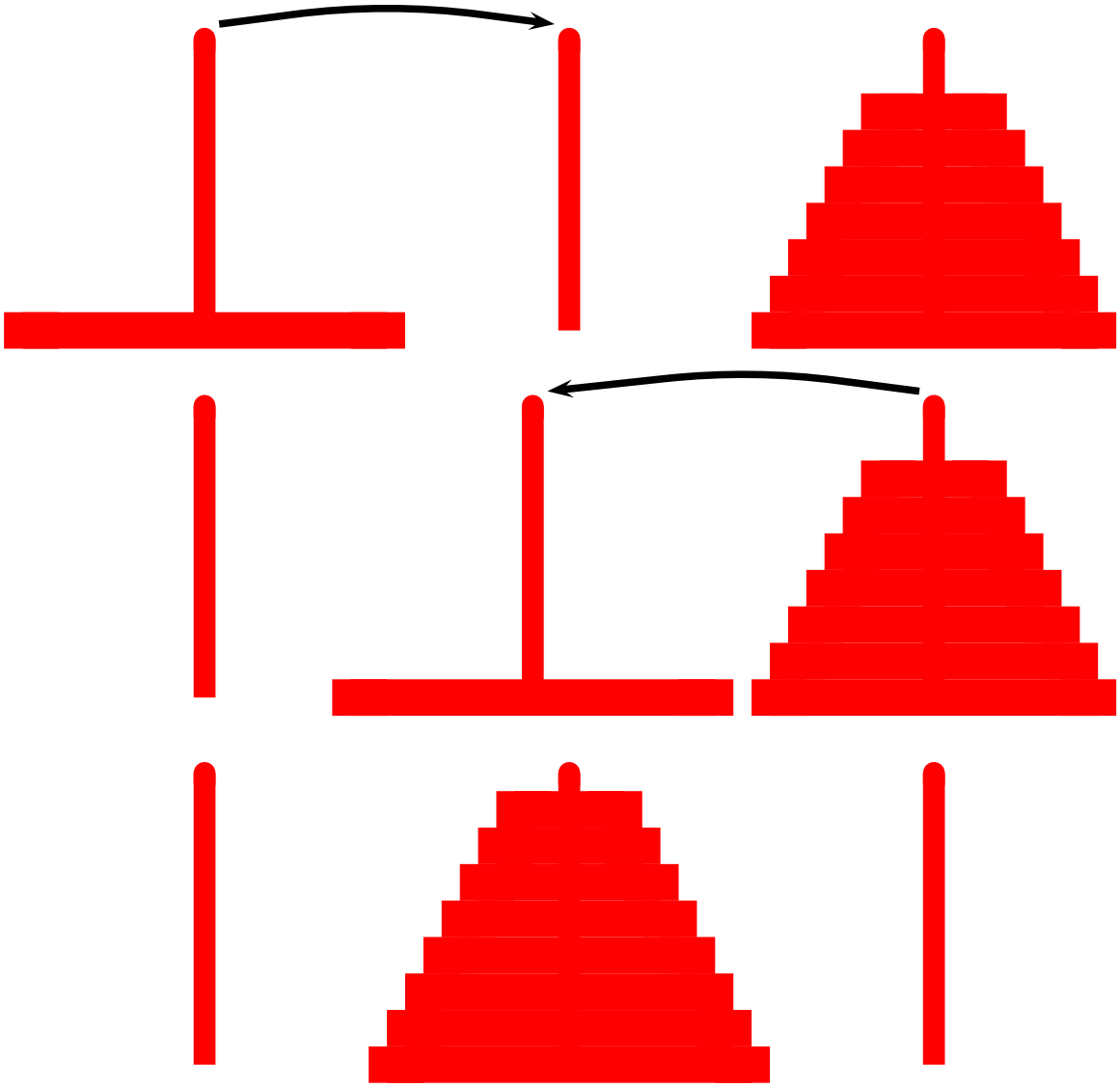
FIG. 1 – *Position initiale des disques.*

L'idée est de procéder par étapes en ramenant le problème à l'ordre n à un problème à l'ordre $n-1$. On peut remarquer que pour déplacer les n disques du premier pieu sur le deuxième, on peut procéder de la manière suivante :

- déplacer les $(n-1)$ premiers disques du pieu de gauche sur le pieu de droite,
- déplacer le dernier disque du premier pieu sur le deuxième
- déplacer les $(n-1)$ disques qui sont sur le pieu de gauche vers le pieu du milieu

ce qui schématiquement nous donne :





Tout ceci nous conduire à écrire de la façon la plus synthétique le programme `hanoï` suivant :

```
> hanoi := proc(n :: nonnegint,a,b,c)
  if n = 1 then printf('deplacer %s sur %s\n',a,b)
           else hanoi(n-1,a,c,b);
                hanoi(1,a,b,c);
                hanoi(n-1,c,b,a);
  fi;
  end;
```

hanoi := **proc**(*n*::*nonnegint*, *a*, *b*, *c*)
if *n* = 1 **then** printf('*dplacer %s sur %s\n*', *a*, *b*)
else hanoi(*n* − 1, *a*, *c*, *b*) ; hanoi(1, *a*, *b*, *c*) ; hanoi(*n* − 1, *c*, *b*, *a*)
end if
end proc

```
> hanoi(3,gauche,centre,droite);
```

```
déplacer gauche sur centre
déplacer gauche sur droite
déplacer centre sur droite
déplacer gauche sur centre
déplacer droite sur gauche
```

```
déplacer droite sur centre
déplacer gauche sur centre
```

Un exemple de mauvaise utilisation de la récursivité nous est donné par l'étude de la suite de Fibonacci (mathématicien du XIII^e S.). Elle est définie par : $u_0 = u_1 = 1$ et $u_{n+2} = u_{n+1} + u_n$. Cette définition conduit naturellement au programme récursif suivant :

fibbo := **proc**(*n*::*nonnegint*)
if $n = 0$ **or** $n = 1$ **then return** 1 **else return** fibbo$(n-1)$ + fibbo$(n-2)$ **end if**
end proc

Pour calculer u_5, il faut calculer u_4 et u_3 puis, pour calculer u_4 il faut de nouveau calculer u_3 et u_2 et ainsi de suite. En fait, on démontre facilement que pour calculer u_n, on fait u_n appels à la fonction «`fibbo`» or $u_n \sim \left(\frac{1+\sqrt{5}}{2}\right)^n$ ce qui nous donne une complexité exponentielle pour la fonction «`fibbo`».

En Maple, on peut contourner ce problème en utilisant l'option «`remember`» qui stocke dans une table toutes les valeurs calculées d'une fonction. Ainsi, avant d'effectuer le corps du programme, Maple commence par regarder dans la table si la fonction n'a pas déjà été appelée pour l'argument donné en entrée. Si c'est le cas, il récupère la valeur dans la table sinon il exécute la fonction.

fibbo_rem := **proc**(*n*::*nonnegint*)
option *remember*;
if $n = 0$ **or** $n = 1$ **then return** 1
else return fibbo_rem$(n-1)$ + fibbo_rem$(n-2)$
end if
end proc

Comparons enfin les temps d'exécution de chacune des fonctions :

```
> time(fibbo(25));
```

39.650

```
> time(fibbo_rem(25));
```

.016

En conclusion, la récursivité oui, mais il faut faire attention à ne pas générer un trop grand nombre d'appels récursifs sous peine de voir Maple refuser l'exécution du programme avec un

```
Error, (in somrec) too many levels of recursion
```

Un usage raisonné de la récursivité est tout à fait profitable comme le montre l'exemple des tours de Hanoï.