Programmation récursive/impérative

Une introduction

Guillaume CONNAN

IREM de Nantes

4 janvier 2010

Sommaire

Sommaire

Sommaire

La programmation impérative est la plus répandue (Fortran, Pascal, C,...) et la plus proche de la machine réelle. Son principe est de modifier à chaque instant l'état de la mémoire via les affectations. Historiquement, son principe a été initié par John Von Neumann et Alan Turing.

Par exemple, lorsqu'on entre avec XCAS :

```
a:=2
```

cela signifie que la case mémoire portant le petit « fanion » a contient à cet instant la valeur 2.

On peut changer le contenu de cette case mémoire un peu plus tard :

```
a:=4
```

et cette fois la case a contient la valeur 4.

On peut changer le contenu de cette case mémoire un peu plus tard :

```
a:=4
```

et cette fois la case a contient la valeur 4.

Le caractère chronologique de ce type de programmation est donc primordial. Regardons la suite d'instructions suivante :

```
a:=1;
b:=2;
a:=a+b; // maintenant a contient 3
b:=a-b; // maintenant b contient 3-2=1
a:=a-b; // maintenant a contient 3-1=2
```

Listing 1 – attention aux affectations

Le caractère chronologique de ce type de programmation est donc primordial. Regardons la suite d'instructions suivante :

```
a:=1;
b:=2;
a:=a+b; // maintenant a contient 3
b:=a-b; // maintenant b contient 3-2=1
a:=a-b; // maintenant a contient 3-1=2
```

Listing 2 – attention aux affectations

Danger

Il faut donc être extrêmement attentif à la chronologie des instructions données. C'est un exercice intéressant en soi mais pas sans danger (c'est d'ailleurs la source de la plupart des « bugs » en informatique).

Question : est-ce que

```
a:=3;
a:=a+1;
```

dit la même chose que

```
a:=a+1;
a:=3;
```

La motivation première de ce type de fonctionnement a été pratique : les ordinateurs étant peu puissants, le souci principal du concepteur de programme a longtemps été d'économiser au maximum la mémoire. Au lieu d'accumuler des valeurs dans différents endroits de la mémoire (*répartition spatiale*), on les stocke au même endroit mais à différents moments (*répartition temporelle*). C'est le modèle des machines à états (machine de TURING et architecture de VON NEUMANN) qui impose au programmeur de connaître à tout instant l'état de la mémoire centrale.

Sommaire

En classe de mathématique, cet aspect technique nous importe peu. Il nous permet cependant d'illustrer dynamiquement la notion de fonction :

```
y:=x^2+1;

x:=1;    y;    // ici y=2
x:=-5;   y;    // ici y=26
x:=1/2;  y;    // ici y=5/4
```

Listing 3 – affectation et image par une fonction

La fonction $f : x \mapsto x^2 + 1$ dépend de la variable x. On affecte une certaine valeur à la variable ce qui détermine la valeur de son image.

```
y:=x^2+1;

x:=1;    y;    // ici y=2
x:=-5;   y;    // ici y=26
x:=1/2;  y;    // ici y=5/4
```

Listing 4 – affectation et image par une fonction

La fonction $f : x \mapsto x^2 + 1$ dépend de la variable x. On affecte une certaine valeur à la variable ce qui détermine la valeur de son image.

Remarque

Il est toutefois à noter qu'en fait on a travaillé ici sur des expressions numériques et non pas des fonctions.

On peut enchaîner des affectations :

```
y:=x^2+1;
z:=y+3;

x:=1;    y; z;   /* ici y:=2 et z=5 */
x:=-5    y; z;   // ici y=26  et z=29;
x:=1/2; y; z;   // ici y=5/4 et z=17/4;
```

Listing 5 – affectation et composition de fonctions

et illustrer ainsi la composition de fonctions.

On peut enchaîner des affectations :

```
y:=x^2+1;
z:=y+3;

x:=1;    y; z;   /* ici y:=2 et z=5 */
x:=-5    y; z;   // ici y=26  et z=29;
x:=1/2; y; z;   // ici y=5/4 et z=17/4;
```

Listing 6 – affectation et composition de fonctions

et illustrer ainsi la composition de fonctions.

Avec un peu de mauvaise foi (car le cas est simple et artificiel) mais pas tant que ça puisque de tels problèmes sont source de milliers de bugs sur des milliers de programmes plus compliqués, nous allons mettre en évidence un problème majeur de ce genre de programmation quand on n'est pas assez attentif.

On créé à un certain moment une variablea à laquelle on affecte une valeur :

```
a:=2;
```

On introduit plus tard une procédure (une sorte de fonction informatique des langages impératifs) qu'on nomme f et qui effectue certains calculs en utilisant a :

```
f(x):={
  a:=a+1;
  return(x+a)
}
```

Listing 7 – exemple de procédure

On introduit plus tard une procédure (une sorte de fonction informatique des langages impératifs) qu'on nomme f et qui effectue certains calculs en utilisant a :

```
f(x):={
  a:=a+1;
  return(x+a)
}
```

Listing 8 – exemple de procédure

On peut donc construire un objet qui **ressemble** à une fonction d'une variable (mais qui n'en est évidemment pas) et qui à une même variable renvoie plusieurs valeurs. En effet, une procédure ne dépend pas uniquement des arguments entrés et donc peut changer de comportement au cours du programme : c'est ce qu'on appelle en informatique des *effets de bord*.

Certains langages utilisent le terme fonction mais nous préférerons « procédure » pour ne pas confondre avec les fonctions mathématiques.

On a en fait plutôt construit ici une fonction de deux variables :

$$g \ : \ (a,x) -> x+a$$

Le premier appel de $f(5)$ est en fait $g(3,5)$ et le deuxième $g(4,5)$, etc. mais on a pu le faire en utilisant une forme qui **laisse à penser** que f est une fonction qui renvoie une infinité d'images différentes d'un même nombre.

Informatiquement, a et x jouent des rôles différents puisque x est un argument de la procédure f et a est une variable globale donc « extérieure » à f (on dirait un *paramètre*). D'ailleurs, XCAS envoie un avertissement pour nous le rappeler :

```
// Warning: a, declared as global variable(s) compiling f
```

Danger

Il faudra donc veiller à éviter le recours à des variables globales.

Sommaire

Les langages fonctionnels bannissent totalement les affectations.

Ils utilisent des fonctions *au sens mathématique du terme*, c'est-à-dire qu'une fonction associe une unique valeur de sortie à une valeur donnée en entrée.

Les fonctions peuvent être testées une par une car *elles ne changent pas selon le moment où elles sont appelées dans le programme.* On parle de *transparence référentielle*.

Le programmeur n'a donc pas besoin de savoir dans quel état se trouve la mémoire. C'est le logiciel qui s'occupe de la gestion de la mémoire.

Les langages fonctionnels bannissent totalement les affectations.
Ils utilisent des fonctions *au sens mathématique du terme*, c'est-à-dire qu'une fonction associe une unique valeur de sortie à une valeur donnée en entrée.
Les fonctions peuvent être testées une par une car *elles ne changent pas selon le moment où elles sont appelées dans le programme.* On parle de *transparence référentielle*.
Le programmeur n'a donc pas besoin de savoir dans quel état se trouve la mémoire. C'est le logiciel qui s'occupe de la gestion de la mémoire.

Les langages fonctionnels bannissent totalement les affectations.
Ils utilisent des fonctions *au sens mathématique du terme*, c'est-à-dire qu'une fonction associe une unique valeur de sortie à une valeur donnée en entrée.
Les fonctions peuvent être testées une par une car *elles ne changent pas selon le moment où elles sont appelées dans le programme.* On parle de *transparence référentielle*.
Le programmeur n'a donc pas besoin de savoir dans quel état se trouve la mémoire. C'est le logiciel qui s'occupe de la gestion de la mémoire.

Les langages fonctionnels bannissent totalement les affectations.
Ils utilisent des fonctions *au sens mathématique du terme*, c'est-à-dire qu'une fonction associe une unique valeur de sortie à une valeur donnée en entrée.
Les fonctions peuvent être testées une par une car *elles ne changent pas selon le moment où elles sont appelées dans le programme.* On parle de *transparence référentielle*.
Le programmeur n'a donc pas besoin de savoir dans quel état se trouve la mémoire. C'est le logiciel qui s'occupe de la gestion de la mémoire.

Historiquement, ce style de programmation est né du λ-calcul développé par Alonzo Church juste avant la publication des travaux d'Alan Turing qui a d'ailleurs été un étudiant de Church à Princeton...

La **récursivité** est le mode habituel de programmation avec de tels langages : il s'agit de fonctions qui s'appellent elles-mêmes. Pendant longtemps, la qualité des ordinateurs et des langages fonctionnels n'était pas suffisante et limitaient les domaines d'application de ces langages.

On utilise également en programmation fonctionnelle des *fonctions d'ordre supérieur*, c'est-à-dire des fonctions ayant comme arguments d'autres fonctions : cela permet par exemple de créer des fonctions qui à une fonction dérivable associe sa fonction dérivée.

Sommaire

On place un certain nombre d'élèves en les classant par ordre décroissant de taille.

Le plus grand voudrait connaître sa propre taille mais chaque élève ne peut mesurer que son écart de taille avec son voisin.

Seul le plus petit connaît sa taille.

On place un certain nombre d'élèves en les classant par ordre décroissant de taille.
Le plus grand voudrait connaître sa propre taille mais chaque élève ne peut mesurer que son écart de taille avec son voisin.
Seul le plus petit connaît sa taille.

On place un certain nombre d'élèves en les classant par ordre décroissant de taille.
Le plus grand voudrait connaître sa propre taille mais chaque élève ne peut mesurer que son écart de taille avec son voisin.
Seul le plus petit connaît sa taille.

Le plus grand mesure son écart avec son voisin et garde le nombre en mémoire.

Le voisin fait de même avec son propre voisin et ainsi de suite.

Enfin l'avant dernier fait de même puis le plus petit lui transmet sa taille : il peut donc calculer sa propre taille et transmet la au suivant et ainsi de suite. Chacun va ainsi pouvoir calculer sa taille de proche en proche....

Le plus grand mesure son écart avec son voisin et garde le nombre en mémoire.
Le voisin fait de même avec son propre voisin et ainsi de suite.
Enfin l'avant dernier fait de même puis le plus petit lui transmet sa taille : il peut donc calculer sa propre taille et transmet la au suivant et ainsi de suite. Chacun va ainsi pouvoir calculer sa taille de proche en proche....

Le plus grand mesure son écart avec son voisin et garde le nombre en mémoire.
Le voisin fait de même avec son propre voisin et ainsi de suite.
Enfin l'avant dernier fait de même puis le plus petit lui transmet sa taille : il peut donc calculer sa propre taille et transmet la au suivant et ainsi de suite. Chacun va ainsi pouvoir calculer sa taille de proche en proche....

Sommaire

Si nous reprenons notre exemple précédant, nous pouvons faire exprimer aux élèves que l'entier suivant n est égal à $1+$ l'entier suivant $n-1$: on ne sort pas des limites du cours de mathématiques et on n'a pas besoin d'introduire des notions d'informatique en travaillant sur cette notion.

On peut écrire un algorithme ainsi :

si $n = 0$ **alors**
 | 1
sinon
 | 1+ successeur de $n - 1$

La récursivité n'est pas l'apanage des langages fonctionnels. Par exemple XCAS permet d'écrire des programmes récursifs.

En français :

```
successeur(k):={
 si k==0 alors 1
 sinon 1+successeur(k-1)
 fsi
}:;
```

Listing 9 – exemple de programme récursif

En anglais :

```
successeur(k):={
 if(k==0)then{1}
 else{1+successeur(k-1)}
}:;
```

Listing 10 – exemple de programme récursif en anglais

Que se passe-t-il dans le cœur de l'ordinateur lorsqu'on entre `successeur(3)` ?

- On entre comme argument 3 ;
- Comme 3 n'est pas égal à 0, alors`successeur(3)` est stocké en mémoire et vaut 1+`successeur(2)` ;
- Comme 2 n'est pas égal à 0, alors`successeur(2)` est stocké en mémoire et vaut 1+`successeur(1)` ;
- Comme 1 n'est pas égal à 0, alors`successeur(1)` est stocké en mémoire et vaut 1+`successeur(0)` ;

- Cette fois,`successeur(0)` est connu et vaut 1 ;
- `successeur(1)` est maintenant connu et vaut `1+successeur(0)` c'est-à-dire 2 ;
- `successeur(2)` est maintenant connu et vaut `1+successeur(1)` c'est-à-dire 3 ;
- `successeur(3)` est maintenant connu et vaut `1+successeur(2)` c'est-à-dire 4 : c'est fini !

C'est assez long comme cheminement mais ce n'est pas grave car *c'est l'ordinateur qui effectue le « sale boulot »* ! Il stocke les résultats intermédiaires dans une *pile* et n'affiche finalement que le résultat.

Comme il calcule vite, ce n'est pas grave. L'élève ou le professeur ne s'est occupé que de la définition récursive (mathématique !) du problème.

C'est assez long comme cheminement mais ce n'est pas grave car *c'est l'ordinateur qui effectue le « sale boulot »* ! Il stocke les résultats intermédiaires dans une *pile* et n'affiche finalement que le résultat. Comme il calcule vite, ce n'est pas grave. L'élève ou le professeur ne s'est occupé que de la définition récursive (mathématique !) du problème.

XCAS calcule facilement`successeur(700)` mais `successeur(7000)` dépasse les possibilités de calcul du logiciel.

Et oui, un langage impératif ne traite pas efficacement le problème de pile, c'est pourquoi pendant longtemps seuls les algorithmes impératifs ont prévalus.

XCAS calcule facilement`successeur(700)` mais `successeur(7000)` dépasse les possibilités de calcul du logiciel.
Et oui, un langage impératif ne traite pas efficacement le problème de pile, c'est pourquoi pendant longtemps seuls les algorithmes impératifs ont prévalus.

Utilisons à présent le langage fonctionnel OCAML, développé par l'INRIA. Même s'il intègre des aspects impératifs pour faciliter l'écriture de certains algorithmes, il est en premier lieu un langage fonctionnel qui en particulier gère très efficacement la mémoire de l'ordinateur pour éviter sa saturation lors d'appels récursifs.

Le programme s'écrit comme en mathématique (mais en anglais...) :

```
# let rec successeur(k)=
  if k=0 then 1
  else 1+successeur(k-1);;
```

Listing 11 – successeur d'un entier

Alors par exemple :

```
# successeur(36000);;
- : int = 36001
```

Mais

```
# successeur(3600000);;
Stack overflow during evaluation (looping recursion?).
```

La *pile* où sont stockés les résultats intermédiaires créés par la récursion est en effet saturée.

Aujourd'hui, les langages fonctionnels sont très efficaces et gèrent de manière intelligente la *pile*. Ils le font soit automatiquement, soit si la fonction utilise une *récursion terminale*, c'est-à-dire si l'appel récursif à la fonction n'est pas *enrobé* dans une autre fonction.

Ici, ce n'est pas le cas car l'appel récursif `successeur(k-1)` est enrobé dans la fonction $x \mapsto 1 + x$.

La *pile* où sont stockés les résultats intermédiaires créés par la récursion est en effet saturée.
Aujourd'hui, les langages fonctionnels sont très efficaces et gèrent de manière intelligente la *pile*. Ils le font soit automatiquement, soit si la fonction utilise une *récursion terminale*, c'est-à-dire si l'appel récursif à la fonction n'est pas *enrobé* dans une autre fonction.
Ici, ce n'est pas le cas car l'appel récursif `successeur(k-1)` est enrobé dans la fonction $x \mapsto 1 + x$.

La *pile* où sont stockés les résultats intermédiaires créés par la récursion est en effet saturée.
Aujourd'hui, les langages fonctionnels sont très efficaces et gèrent de manière intelligente la *pile*. Ils le font soit automatiquement, soit si la fonction utilise une *récursion terminale*, c'est-à-dire si l'appel récursif à la fonction n'est pas *enrobé* dans une autre fonction.
Ici, ce n'est pas le cas car l'appel récursif `successeur(k-1)` est enrobé dans la fonction $x \mapsto 1 + x$.

Dans un premier temps, on peut laisser de côté ce problème de récursion terminale au lycée et se contenter de travailler sur de petits entiers. Ainsi, on peut choisir de travailler avec OCAML ou XCAS par exemple, même si ce dernier n'est pas un langage fonctionnel.

Sommaire

Définissons une fonction qui à deux fonctions f et g associe sa composée $f \circ g$:

```
# let compose=function
    (f,g) -> function x->f(g(x));;
```

Listing 12 – composition de fonction

OCAML devine le type des objets introduits en répondant :

```
val compose : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b = <fun>
```

```
# let compose=function
   (f,g) -> function x->f(g(x));;
```

Listing 13 – composition de fonction

OCAML devine le type des objets introduits en répondant :

```
val compose : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b = <fun>
```

On peut schématiser ce qu'a compris OCAML :

$$f : {}'a \mapsto {}' b \qquad g : {}'c \mapsto {}' a \qquad f \circ g : {}'c \mapsto {}' b$$

Créons deux fonctions numériques :

```
# let carre = function
      x -> x*x;;

# let double = function
     x -> 2*x;;
```

Listing 14 – création de fonctions simples

Composons-les :

```
# compose(double,carre)(2);;
- : int = 8
# compose(carre,double)(2);;
- : int = 16
```

En effet, le double du carré de 2 est 8 alors que le carré du double de 2 vaut 16...

On peut faire la même chose avec XCAS :

```
compose(f,g):={
  x->f(g(x))
}
```

Listing 15 – composée de fonctions avec XCAS

```
compose(x->x^2,x->2*x)(2)
```

Sommaire

Sommaire

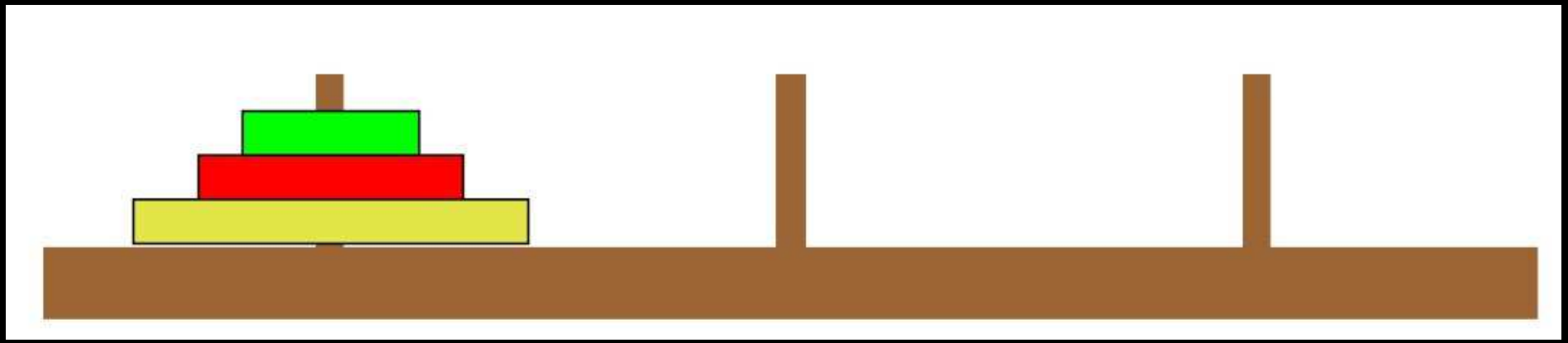
Sommaire

Ce casse-tête a été posé par le mathématicien français Édouard Lucas en 1883.
Le jeu consiste en une plaquette de bois où sont plantés trois piquets.

- on ne déplace qu'un disque à la fois ;
- on ne peut poser un disque que sur un disque de diamètre supérieur.

Ce casse-tête a été posé par le mathématicien français Édouard LUCAS en 1883.
Le jeu consiste en une plaquette de bois où sont plantés trois piquets.

- on ne déplace qu'un disque à la fois ;
- on ne peut poser un disque que sur un disque de diamètre supérieur.



Sommaire

En réfléchissant récursivement, c'est enfantin !... Pour définir un algorithme récursif, il nous faut régler un cas simple et pouvoir simplifier un cas compliqué.

- *cas simple* : s'il y a zéro disque, il n'y a rien à faire !
- *simplification d'un cas compliqué* : nous avons n disques au départ sur le premier disque et nous savons résoudre le problème pour $n-1$ disques. Appelons A, B et C les piquets de gauche à droite. Il suffit de déplacer les $n-1$ disques supérieurs de A vers B (hypothèse de récurrence) puis on déplace le plus gros disque resté sur A en C. Il ne reste plus qu'à déplacer vers C les $n-1$ disques qui étaient en B. (hypothèse de récurrence).

- *cas simple* : s'il y a zéro disque, il n'y a rien à faire !
- *simplification d'un cas compliqué* : nous avons n disques au départ sur le premier disque et nous savons résoudre le problème pour $n-1$ disques. Appelons A, B et C les piquets de gauche à droite. Il suffit de déplacer les $n-1$ disques supérieurs de A vers B (hypothèse de récurrence) puis on déplace le plus gros disque resté sur A en C. Il ne reste plus qu'à déplacer vers C les $n-1$ disques qui étaient en B. (hypothèse de récurrence).

On voit bien la récursion : le problème à n disques est résolu si on sait résoudre le cas à $n-1$ disques et on sait quoi faire quand il n'y a plus de disques.

On sait pourquoi cet algorithme va réussir mais on ne sait pas comment s'y prendre étape par étape.

On va donc appeler à l'aide l'ordinateur en lui demandant d'écrire les mouvements effectués à chaque étape.

On voit bien la récursion : le problème à n disques est résolu si on sait résoudre le cas à $n-1$ disques et on sait quoi faire quand il n'y a plus de disques.
On sait pourquoi cet algorithme va réussir mais on ne sait pas comment s'y prendre étape par étape.
On va donc appeler à l'aide l'ordinateur en lui demandant d'écrire les mouvements effectués à chaque étape.

On voit bien la récursion : le problème à n disques est résolu si on sait résoudre le cas à $n-1$ disques et on sait quoi faire quand il n'y a plus de disques.
On sait pourquoi cet algorithme va réussir mais on ne sait pas comment s'y prendre étape par étape.
On va donc appeler à l'aide l'ordinateur en lui demandant d'écrire les mouvements effectués à chaque étape.

On commence par créer une fonction qui affichera le mouvement effectué :

```
let mvt depart arrivee=
print_string
("Déplace un disque de la tige "^départ^" vers la tige "^
    arrivée);
print_newline();;
```

Listing 16 – mouvement élémentaire

Le programme proprement dit :

```
let rec hanoi a b c= function
   | 0 -> ()  (*0 disque : on ne fait rien*)
   | n -> hanoi a c b (n-1); (*n-1 disques sont placés dans
        l'ordre de a vers b*)
          mvt a c; (*on déplace le disque restant en a vers
               c*)
          hanoi b a c (n-1) (*n-1 disques sont placés dans
              l'ordre de b vers c*);;
```

Listing 17 – résolution récursive du problème des tours de Hanoï

Par exemple, dans le cas de 3 disques :

```
# hanoi "A" "B" "C" 3;;
Deplace un disque de la tige A vers la tige C
Deplace un disque de la tige A vers la tige B
Deplace un disque de la tige C vers la tige B
Deplace un disque de la tige A vers la tige C
Deplace un disque de la tige B vers la tige A
Deplace un disque de la tige B vers la tige C
Deplace un disque de la tige A vers la tige C
- : unit = ()
```

Sommaire

Prenez un papier et un crayon. Dessinez trois piquets A, C et B et les trois disques dans leur position initiale sur le plot A.
Déplacez alors les disques selon la méthode suivante :

- déplacez le plus petit disque vers le plot suivant selon une permutation circulaire A-C-B-A ;
- un seul des deux autres disques est déplaçable vers un seul piquet possible.

Prenez un papier et un crayon. Dessinez trois piquets A, C et B et les trois disques dans leur position initiale sur le plot A.
Déplacez alors les disques selon la méthode suivante :

- déplacez le plus petit disque vers le plot suivant selon une permutation circulaire A-C-B-A ;
- un seul des deux autres disques est déplaçable vers un seul piquet possible.

Réitérez (en informatique à l'aide de boucles...) ce mécanisme jusqu'à ce que tous les disques soient sur C dans la bonne position.
On voit donc bien ici comment ça marche ; il est plus délicat de savoir pourquoi on arrivera au résultat.
Le programme est lui-même assez compliqué à écrire : vous trouverez une version en C (235 lignes...) à cette adresse :

http ://files.codes-sources.com/fichier.aspx ?id=38936&f=HANOI%5cHANO