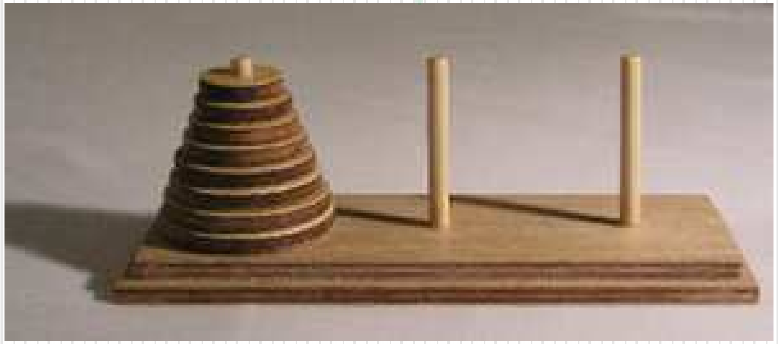
ALGORITHMIQUE II

Récurrence et Récursivité



Récurrence

- **Suite récurrente:** la définition d'une suite est la donnée
  - d'un terme général défini en fonction du (ou des) terme(s) précédant(s)
  - D'un terme initial qui permet d'initialiser le calcul
- **Principe de récurrence :**

  Soit P un prédicat (ou propriété) sur IN qui peut être soit vrai soit faux (on écrira souvent P(n) à la place de P(n) = vrai).

  On suppose que
  - P(0) vrai
  - $\forall n \in$ IN, $P(n) \Rightarrow P(n+1)$

  Alors , pour tout $n \in$ IN, P(n) est vrai.

  Si on considère le prédicat suivant

  P(n) : je sais résoudre le problème pour n

  alors le principe de récurrence nous dit que si je sais résoudre le Pb pour n=0 et que si je sais exprimer la solution pour n en fonction de la solution pour n+1 alors je sais résoudre le Pb pour n'importe quel n.

Récursivité

- Examples:

1. Puissance

$$\begin{cases} a^0 = 1 \\ a^{n+1} = a\ a^n \end{cases}$$

Ou bien

$$\begin{cases} a^0 = 1 \\ a^n = a\ a^{n-1} \quad n>0 \end{cases}$$

2. Factoriel

$$\begin{cases} 0! = 1 \\ n! = n\ (n-1)! \quad , \quad n \geq 1 \end{cases}$$

3. Suite de Fibonacci

$$\begin{cases} F_0 = F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \quad , \quad n \geq 2 \end{cases}$$

Récursivité

- Un algorithme (ou fonction) est dit récursif s'il est défini en fonction de lui-même.
- Exemples

```
Fonction puissance(x : réel, n : entier) : réel
début
        si  n = 0 alors retourner 1
        sinon retourner (x * puissance(x , n-1))
fin
```

```
Factoriel (n)
début
        si n = 0 alors retourner(1)
        sinon  retourner (n*factoriel(n-1))
fin
```

Récursivité

- *fact (n)*

  début

  si n = 0 alors retourner(1)

  *sinon retourner(n*fact(n-1))*

  fsi

  fin

- Le déroulement de l'appel de fact(3):

  3*fact(2)

  Fact(3) 3*2=6 2*fact(1)

  6 2*1=2 1*fact(0)

  1*1=1

- La condition n = 0 est appelée **test d'arrêt** de la récursivité.
- Il est impératif de prévoir un test d'arrêt dans une fonction récursive, sinon l'exécution ne s'arrêtera jamais.
- L'appel récursif est traité comme n'importe appel de fonction.

Récursivité

- L'appel d'une fonction (récursive ou non) se fait dans un **contexte d'exécution** propre (pile d'exécution), qui contient :
  - L'adresse mémoire de l'instruction qui a appelé la fonction (adresse de retour)
  - Les valeurs des paramètres et des variables locales à la fonction.
- L'exécution d'une fonction récursive se fait par des appels successifs à la fonction jusqu'à ce que la condition d'arrêt soit vérifiée, et à chaque appel, les valeurs des paramètres et l'adresse de retour sont mis (empilés) dans la pile d'exécution.

Récursivité

- L'ordre des instructions par rapport à un appel récursif est important.
- Exemple:

afficher(n)

début

si n > 0 alors

١

afficher(n div 10)

٢

fsi

fin

écrire(n mod 10)

- L'algorithme récursif afficher(n) permet d'afficher les chiffres d'un entier, strictement positif, selon la disposition de l'instruction écrie(n mod 10):

- Si l'instruction est placée en ١ , les chiffres sont affichés dans l'ordre inverse

- Si elle est placée en ٢, alors les chiffres seront affichés dans le bon ordre

Pour n = 123, on a :

١ → 3 2 1

٢ → 1 2 3

Type de récursivité

- Récursivité simple: Une fonction récursive contient un seul appel récursif.
- Récursivité multiple: une fonction récursive contient plus d'un appel récursif (exemple suite de Fibonacci).
- Récursivité mutuelle(ou croisée): Consiste à écrire des fonctions qui s'appellent l'une l'autre.

  Exemple

Récursivité mutuelle

```
Pair(n)
début
   si  n = 0 alors
         retourner vrai
   sinon
    retourner (impair(n-1))
   fsi
fin
```

```
Impair(n)
début
   si  n = 0 alors
         retourner (faux)
   sinon
      retourner (pair(n-1))
   fsi
fin
```

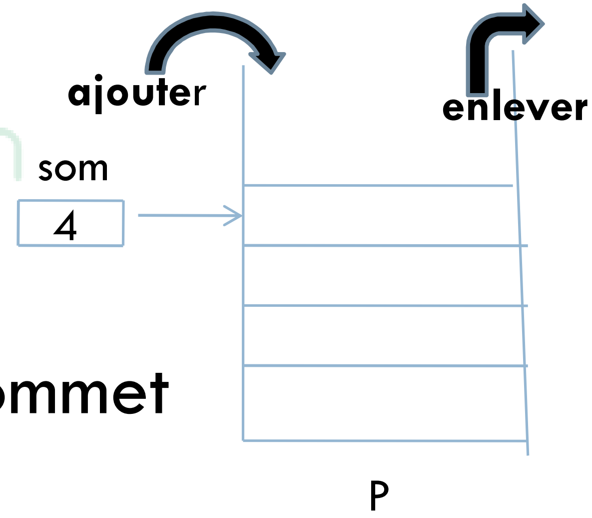
Un peu de Structures de Données

- **Notion de pile.**

  Une pile est une structure pour représenter une suite d'éléments avec la contrainte qu'on ne peut ajouter, ou enlever, un élément que d'un même côté de la pile (dit sommet de la pile).

- Exemple pile d'assiettes.
- Une pile peut être représentée

  par un tableau et un indice de sommet



Notion de Pile

- **<u>Opérations définies sur les piles:</u>**
  - **initialiser(p : Pile)** //Crée une pile vide.
  - **sommet(p : Pile) : élément**// Renvoie l'élément au sommet de la pile p, <u>sous la condition que p soit non vide.</u>
  - **empiler(x : élément, p : Pile)** // ajoute x au sommet de la pile p.
  - **dépiler(p : Pile)** // supprime l'élément au sommet de la pile p, <u>sous la condition que p soit non vide</u>.
  - **pileVide(p : Pile) : booléen** // retourne vrai si p est vide.

Notion de Pile

- **Exemple.**

  Une expression e est dite bien parenthésée (on se limite au '(' et ')') si :

1. Le nombre de parenthèses ouvrantes ($|e|_($) est égal au nombre de parenthèses fermantes ($|e|_)$) dans e.
2. Tout préfixe (partie gauche) u de e vérifie: $|u|_(- |u|_(\geq 0$.

Algorithme: on parcourt l'expression e (de gauche à droite). A chaque rencontre d'une parenthèse ouvrante on l'empile, et à chaque rencontre d'une parenthèse fermente on dépile.

Si on arrive à la fin de e avec une pile vide, l'expression e est bien parenthésée sinon e n'est pas bien parenthésée.

- l'expression (()())() est bien parenthée.
- l'expression ())(n'est pas bien parenthésée.

Transformation du récursif en itératif : « Dérécursivation »

- **Schéma d'un algorithme récursif:**

```
algoR(X)
début
        A
    si C(X) alors
        B;
        algoR(φ(X));
        D;
    sinon
        E;
    fsi;
fin
```

Où :
X : liste de paramètres
C : condition d'arrêt portant sur X
A, B, D, E : bloc d'instructions (éventuellement vide)
$\varphi(X)$: transformation des paramètres

Transformation du récursif en itératif : « Dérécursivation »

```
algoR(X)
Début
        A
    si C(X) alors
        B ;
        algoR(φ(X));
        D ;
    sinon
        E ;
    fsi;
fin
```

- Algorithme itératif équivalent.

```
algoI(X)
p : Pile
début
    initialiser(p);
        A ;
    tantque C(X) faire
        B ;
        empiler(X, p);
        X := φ (X);
        A ;
    ftantque;
        E ;
    tantque (non pileVide(p))  faire
        X := sommet(p);
        dépiler(p);
        D ;
    ftantque
fin
```

Dérécusivation

- Exemple.

afficherR(n)

début

si n > 0 alors

afficher(n div 10)

écrire(n mod 10);

fsi

fin

$A = B = E = \varnothing$

```
afficherI(n)
p : Pile;
Début
    initialiser(p);
    tanque  n > 0 faire
        empiler(n, p);
        n := n div 10;
    ftantque
    tantque (non pileVide(p)) faire
        n := sommet(p);
        dépiler(p);
        écrire(n mod 10);
    ftanque
fin
```

Transformation du récursif en itératif : « Dérécursivation »

- **Récursivité terminale:**

  La récursivité est dite <u>terminale</u> si la dernière instruction exécutée est un appel récursive; (Cas où D = $\emptyset$). Il est claire, dans ce cas, d'éliminer la pile dans la version itérative. (On dépile pour ne rien faire dans la 2ème boucle).

- La récursivité d'une fonction **F**(X) est aussi dite terminale lorsqu'elle se termine par l'instruction retourner(**F**(φ(X))). On ajoute, dans ce cas, un paramètre à la liste X pour contenir le résultat de retour d'un appel récursive, comme le montre l'exemple suivant:

Exemple

fonction FACR(*n*) ;
début
  si *n*=0 alors retourner (1)
  sinon retourner (*n** FACR(*n*-1)) ;
fin.

RÉCURSIVE

fonction FACR'(*n*, *r*)
début
  si *n*=0 **alors retourner** (*r*)
  sinon retourner (FACR'(*n*-1, *n***r*)) ;
fin.

RÉCURSIVITÉ TERMINALE

fonction FACI'(*n*, *r*) ;
début
  tant que *n* > 0 **faire**
  { *r* := *n** *r* ; *n* := *n*-1 ; }
  retourner (*r*) ;
fin.

fonction FACI(*n*) ;
début
  r := 1 ;
  tant que *n* > 0 **faire**
  { *r* := *n** *r* ; *n* := *n*-1 ; }
  retourner (*r*) ;
fin.

ITÉRATIVE

Complexité des algorithmes récursifs

La complexité des algorithmes récursifs est souvent exprimée par une équation de récurrence.

- Exemple1. Complexité de l'algorithme récursif pour calculer n! (l'opération dominante est la multiplication)

  Soit T(n) le coût de FACR(n). T(n) vérifie l'équation:

$$\begin{cases} T(0) = 0 \\ T(n) = T(n-1) + 1 \end{cases}$$

  la solution de cette équation est :

  $T(n) = n = 1 + 1 + \ldots + 1$ (n fois)

Complexité des algorithmes récursifs

- Exemple2.
- Tri par sélection

```
 sel_rec(T,n)
début
  si n > 1 alors
       k← max {i ∈{1,2,…,n} /  T[i] ≥ T[j] , j=1,2,…,n et j ≠ i}
       échanger(T[k],T[n])
       sel_rec(T,n-1)
  fsi;
fin
```

- Complexité : T(n) vérifie :

$$\begin{cases} T(1) = 0 \\ T(n) = T(n-1) + n \end{cases}$$

$T(n) = n+T(n-1) = n+(n-1)+T(n-2) =\ldots= n+(n-1)+\ldots+2 +C(1) = \frac{n(n+1)}{2} - 1$

$T(n) = O(n^2)$

On ouvre une parenthèse

Encore un peu de structures de données

- **Notion d'arbre binaire**

  On introduit cette notion pour savoir interpréter les arbres d'appels dans le cas d'une récursivité double(où il y a deux appels récursifs).

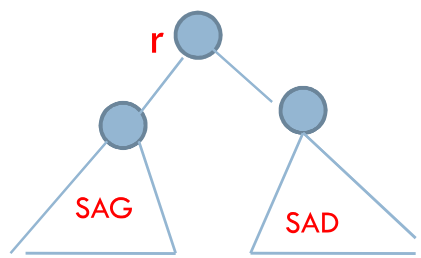
  - Les arbres sont utilisés pour représenter une suite d'éléments.

  - un arbre est un ensemble de nœuds, chaque nœud représente un élément de la suite.

  <u>Définition récursive d'un arbre binaire</u>:

- un arbre binaire est:
  - Soit vide
  - Soit formé :
    - D'un nœud (appelé racine)
    - D'un sous-arbre gauche , noté SAG, qui est un arbre binaire)
    - D'un sous-arbre droit, noté SAD, qui est aussi un arbre binaire

(les deux sou-arbres gauche et droit sont disjoints).

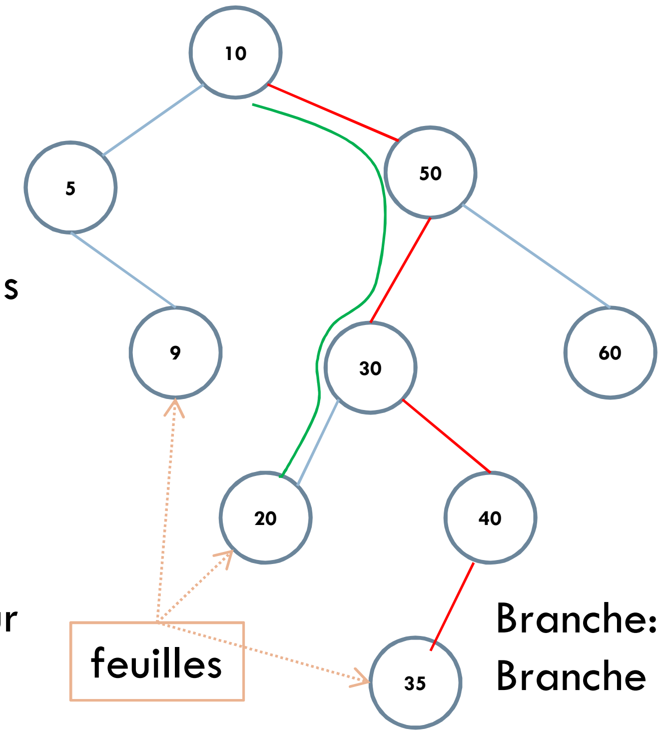


Arbre binaire: terminologie

Soit A un arbre binaire de racine r.

- La racine du SAG (resp. SAD) de A est appelé fils gauche (resp. fils droit) de r et r est appelé père.
- Un nœud qui n' a pas de fils est appelé feuille.
- un chemin de l'arbre est une suite de nœuds $n_1, n_2, \ldots n_k$ où n_{i+1} est un fils (gauche ou droit) de n_i, $1 \leq i \leq k-1$.
- Longueur d'un chemin = nombre de nœuds, constituant le chemin, - 1
- Une branche est un chemin de la racine à une feuille.
- La hauteur d'un arbre est la longueur de la plus longue branche de l'arbre.

$$h(A) = \begin{cases} -1 \text{ si } A = \emptyset \\ 1 + \max(h(SAG(A)), h(SAD(A)) \end{cases}$$



Branche:

Branche plus longue:

Hauteur = 4

Arbre binaire

- Résultat (utile pour la complexité sur les arbres binaires de recherche):

la hauteur h d'un arbre binaire de taille n (n est le nombre de nœuds de l'arbre) vérifie: $1 + \lceil \log_2 n \rceil \leq h \leq n$

- la hauteur est, en moyenne, un O(log n) et un O(n) dans le pire des cas.

- Les algorithmes sur les arbres binaires se ramènent souvent aux algorithmes de parcours de ces arbres.

Algorithmes de parcours

puis on ferme la parenthèse

Parcours(a : Arbre)

début

si a $\neq \emptyset$ alors

١

parcours(SAG(a));

٢

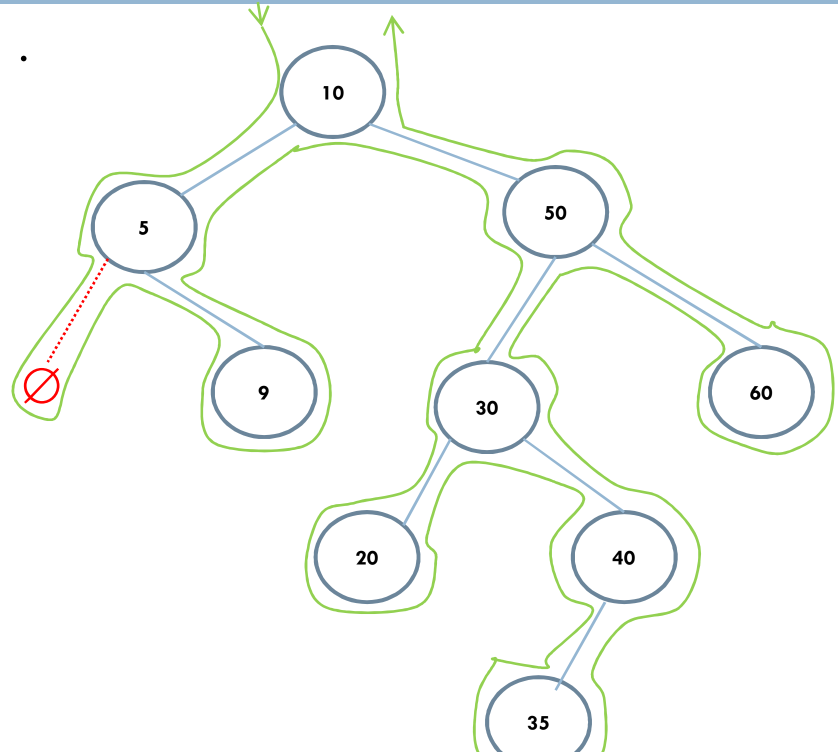
parcours(SAD(a));

٣

fsi;

fin

Traiter(racine(a))



Selon la disposition de l'instruction traiter(racine(a)), on distingue 3 types de parcours:

١ : parcours préfixe.

٢ : parcours infixe.

٣ : parcours postfixe.

On passe 3 fois sur un nœud .

Parcours préfixe: on traite un nœud lorsqu'on le rencontre pour la 1er fois.

10-5-9-50-30-20-40-35-

Parcours infixe: " " " " la 2è fois.

5-9-10-20-30-35-40-50-60

Parcours posfixe: " " " le quitte pour la dernière fois.

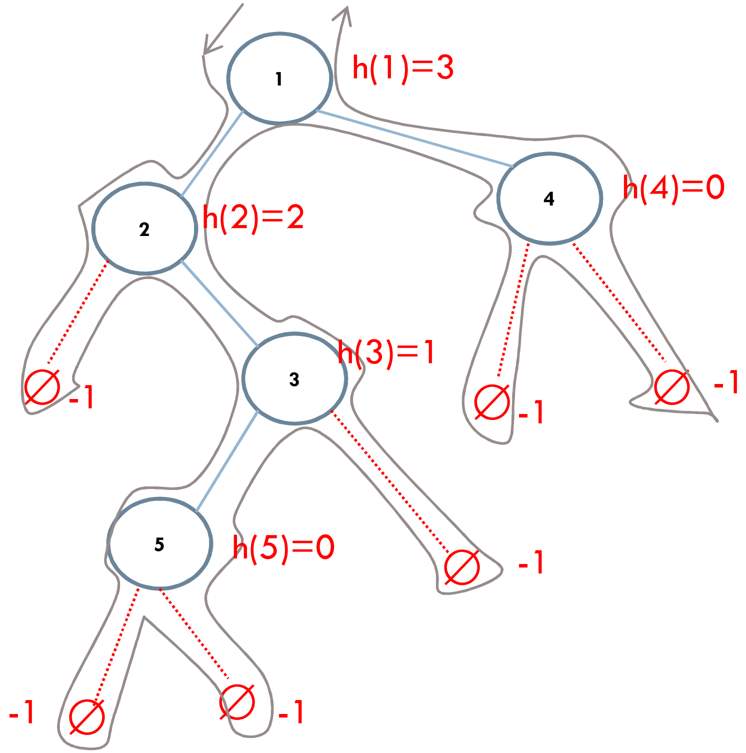
9-5-20-35-40-30-60-50-

Récursivité double
&
Arbre des appels récursifs

- Exemple1: calcul de la hauteur d'un arbre binaire.

```
    h(a : arbre)
début
    si a = ∅ alors retourner -1
    sinon
            h1 := h(SAG(a));
            h2 : h(SAD(a));
            retourner(1+max(h1,h2));
    fsi;
fin
```

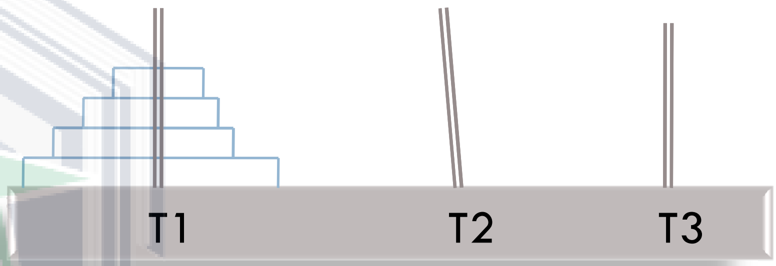
- Un arbre est donné par sa racine
- appel de h(1):



Récursivité double
&
Arbre des appels récursifs

- Exemple2 : **tours de Hanoï** (**Occupation des moines de Hanoï**)

  Le jeu consiste à faire passer les disques de la tour T1 à la tours T2, en ne déplaçant qu'un seul disque à la fois, et en utilisant la tour intermédiaire T3 de telle sorte qu'à aucun moment un disque ne soit empilé sur un disque de plus petite dimension.



La solution semble difficile, et pourtant une solution récursive existe.

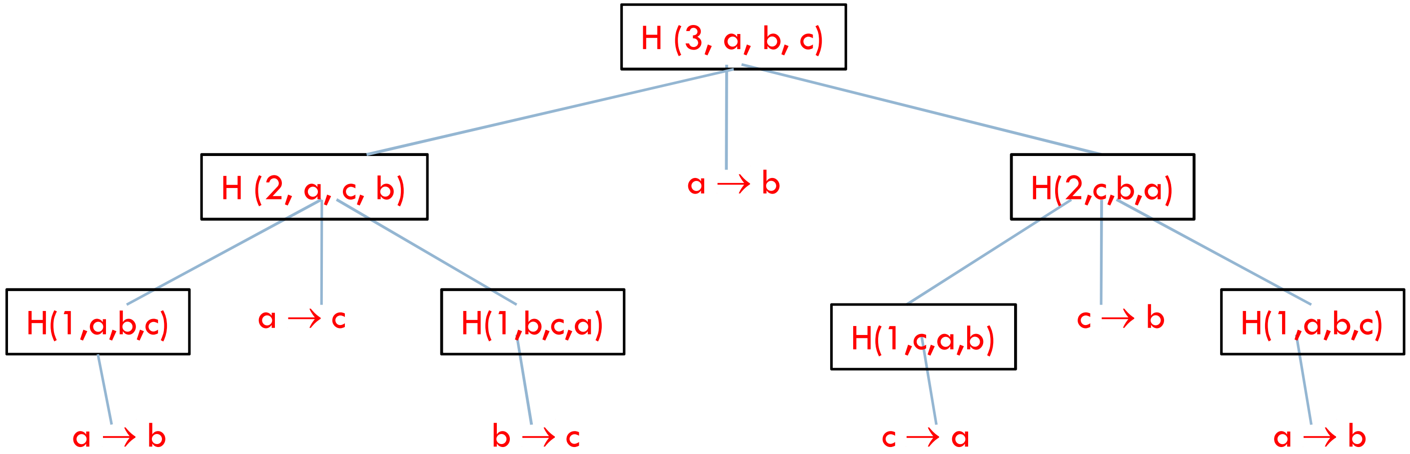
Soit n le nombre de disques à déplacer. Si n=1 la solution est triviale.
Si on sait transférer n-1 disques alors on sait en transférer n.
Il suffit de transférer les n-1 disques supérieurs de la tours T1 vers la tours T3, de déplacer le disque le plus grand de T1 vers T2, puis de transférer les n-1 disques de T3 vers T2. Ceci se traduit par l'algorithme récursif suivant:

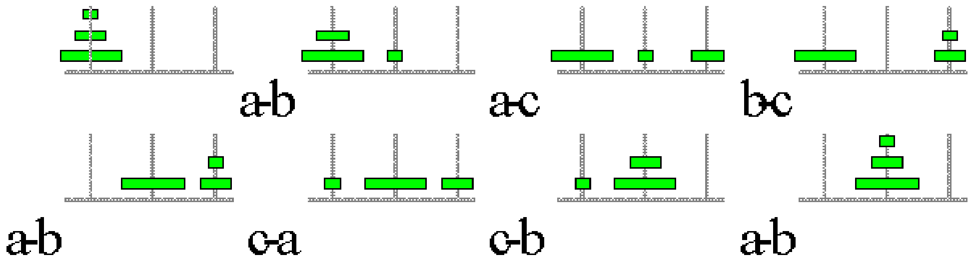
Tours de Hanoï

```
H(n,T1,T2,T3)
début
   si n = 1 alors écrire(T1, '→',T2)
   sinon
         H(n-1,T1,T3,T2);
         écrire(T1,'→',T2)
         H(n-1,T3,T2,T1);
   fsi
fin
```

```
H(n,T1,T2,T3)
début
  si n = 1 alors écrire(T1, '→',T2)
  sinon
      H(n-1,T1,T3,T2);
      écrire(T1,'→',T2)
      H(n-1,T3,T2,T1);
  fsi
fin
```

Arbre des appels de H(3,a,b,c)





Tours de Hanoï : Complexité

Soit T(n) le temps pour déplacer les n disques. T(n) vérifie l'équation :

$$\begin{cases} T(1) = 1 \\ T(n) = 2\,T(n-1) + 1 \end{cases}$$

On a :

$T(2) = 2 + 1$

$T(3) = 2(2 + 1) + 1 = 2^2 + 2 + 1$

On montre, par récurrence, que

$T(n) = 1 + 2 + \dots + 2^{n-1} = 2^n - 1$

- Sachant que $T(10) = 2^{10} - 1 = 1023$ et une année $\cong 0.3 \times 10^8$ secondes, il faudrait, pour les moines, 10^{10} siècles pour pouvoir déplacer 64 disques!