Fonctions récursives

IAP1

Catherine Dubois

1. Notion de récursivité et de récurrence

Une *définition récursive* est une définition dans laquelle intervient le nom qu'on est *en train* de définir.

Exemples :

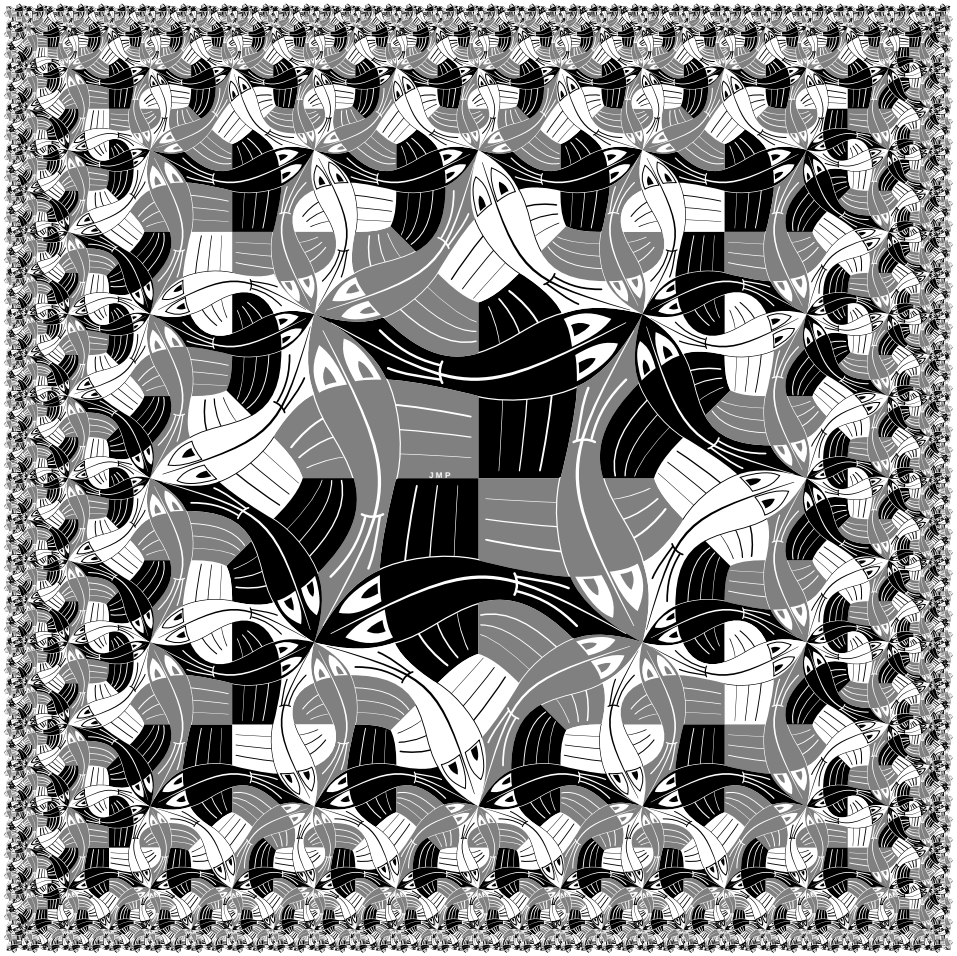
x est *descendant* de y ssi
- soit, x est fils ou fille de y
- soit, x est fils ou fille de z et z est *descendant* de y

Une *expression* OCaml est une constante, un identificateur, un couple d'*expressions* ... etc

suites récurrentes

tableaux d'Escher, fugues de Bach, textes, dessins sur les boites de Vache qui rit, Les Mille et une nuits, circuits électriques ....

Recommencer la même chose, retrouver un même motif,
suggérer l'infini (par une description finie) ...



Un chien vint dans l'office
Et prit une andouillette
Alors à coups de louche
Le chef le mit en miettes

Les autres chiens ce voyant
Vite vite l'ensevelirent [...]
Au pied d'une croix en bois blanc
Où en passant on pouvait lire :

Un chien vint dans l'office
Et prit une andouillette
Alors à coups de louche [etc.]

Samuel Beckett
En attendant Godot

En informatique :

- fonctions récursives (plus largement algorithmes)
  Elles s'appellent elles-mêmes de manière directe ou indirecte (f appelle g qui appelle h qui appelle f)
  Possibilité existante dans la plupart des langages les plus récents (Pascal, C, Ada, OCaml ...)
- types de données récursifs (ex: listes, arbres, ...)
  Une *liste* est soit vide, soit de la forme *element :: liste*
  Un *arbre binaire* est soit vide, soit formé d'une racine, d'un sous-arbre gauche et d'un sous-arbre droit, qui sont eux-mêmes des *arbres binaires*
  Les types récursifs s'expriment directement en OCaml mais pas en C (utilisation de pointeurs - voir cours IAP2)

```
let rec fact n = if n=0 then 1                    0! = 1
                 else n * fact (n - 1);;          n! = n × (n-1)
```

Récursion primitive

En C :
```
int fact(int n)
{if (n==0) return 1;
 else return (n*fact(n-1));
}
```

```
let rec MacCarthy (n) =
  if n > 100 then (n-10) else MacCarthy (MacCarthy (n+11));
```

Récursion imbriquée - nested recursion

```
let rec somlist l = match l with [] -> 0
                               | x::r -> x + (somlist r);;
```

Récursion primitive structurelle

Définissons les fonctions `pair` et `impair`
`pair n` = `true` si n pair
`pair n` = `false` sinon
de même, `impair n` = `true` si n est impair
`impair n` = `false` sinon.

$\Longrightarrow$ Fonctions mutuellement récursives (définition simultanée)

```
let rec pair n = if n=0 then true else  impair (n-1)
and impair n = if n=0 then false else pair (n-1) ;;
```

2. Déroulement d'un appel à une fonction récursive

`fact(3) = 3 * fact(2)` (d'après la définition)

`----------`

calcul suspendu ⇒ appel récursif

`fact(2) = 2 * fact(1)`

`----------`

calcul suspendu

`fact(1) = 1 * fact(0)`

`----------`

calcul suspendu

`fact(0) = 1`

Puis reprise des calculs suspendus

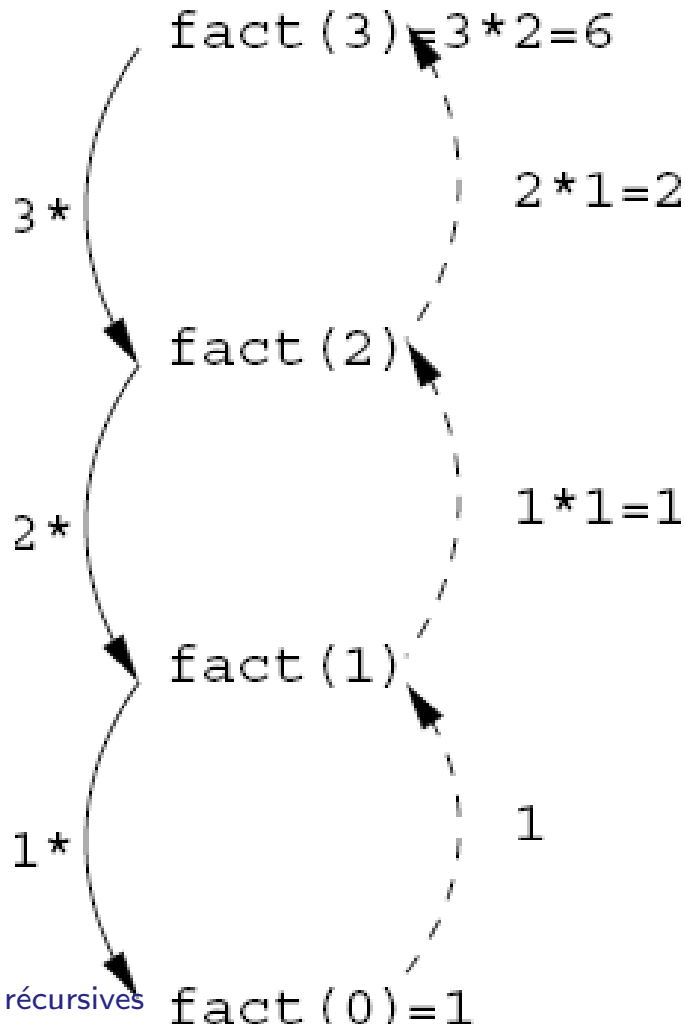
`fact(1) = 1 * 1 = 1`

`fact(2) = 2 * 1 = 2`

`fact(3) = 3 * 2 = 6`

On peut suivre ce déroulement avec la commande `trace`

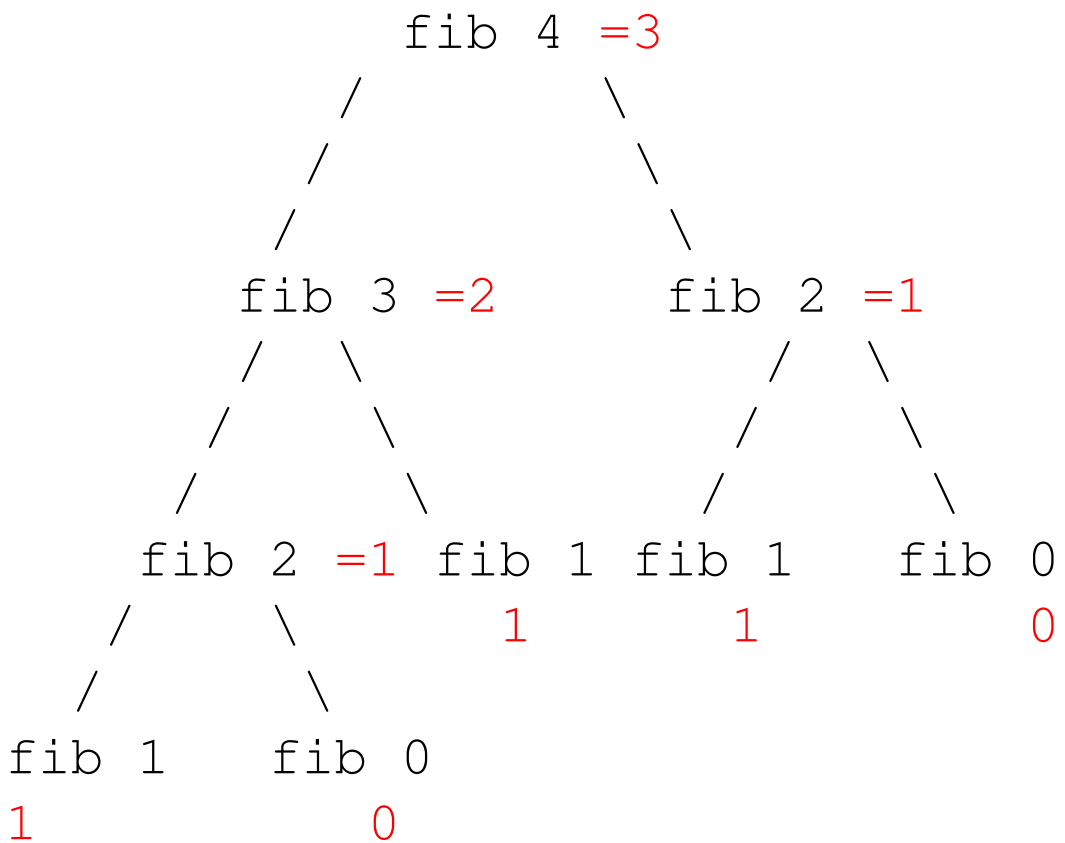
```
# #trace fact;;
fact is now traced.
# fact 3;;
fact <-- 3          |
fact <-- 2          |
fact <-- 1          |    Appels récursifs
fact <-- 0          |
fact --> 1               Valeur de fact(0)
fact --> 1               Valeur de fact(1)
fact --> 2               Valeur de fact(2)
fact --> 6               Valeur de fact(3)
- : int = 6
```


fact(3)=3*2=6
3*
2*1=2
fact(2)
2*
1*1=1
fact(1)
1*
1
fact(0)=1

fib n = nième nombre de Fibonacci

```
let rec fib n = if n=0 then 0
                else if n=1 then 1
                     else (fib (n-1)) + (fib (n-2));;
```

Suivons le calcul de `fib 4` :



```
let rec g = function  n -> n * g (n - 1);;
```

elle boucle pour tout argument : ABSENCE DE CAS DE BASE

```
let rec f n = if n=0 then 1 else f (n + 1);;
```

elle boucle pour tout argument strictement positif : le cas de base existe mais on s'en ELOIGNE

```
let rec h n = if n=0 then failwith "h: " else h (n - 1);;
```

elle échoue pour tout argument positif : le cas de base existe mais c'est un cas d'échec

Une conjecture mathématique célèbre affirme que la fonction définie par

```
let rec conj n =
  if n = 1 then 1 else
  if n mod 2 = 0 then conj (n/2) else conj (3*n+1);;
```

termine pour tout entier et donc est égale à la fonction constante qui vaut 1 partout.

Et que pensez vous de cette définition ?

Le jour où les lendemains des jours de congé seront des jours de congé, on aura fait un grand pas dans la civilisation des loisirs

⇒ **nécessité de cas particuliers non récursifs sur lesquels on aboutit** (le cas général doit *tendre* vers le cas de base)
On parle alors de définition récursive *bien fondée* (parfois difficile à montrer)

4. Terminaison

Soit f une fonction écrite en Ocaml

Démontrer qu'elle est définie sur D_f (ou qu'elle termine sur D_f) c'est démontrer la propriété suivante :

$$\forall x \in D_f, \exists y \in A. f(x) = y$$

- `fact` est définie sur N :

Démonstration : par récurrence sur N

- `somlist` est définie pour toutes les listes :
Démonstration : par récurrence sur la longueur des listes ou par induction structurelle sur les listes

Principe de récurrence sur les listes : induction structurelle

soit P une propriété portant sur les listes

$$(P([]) \wedge (\forall l.P(l) \Rightarrow \forall e.P(e :: l))) \Leftrightarrow \forall l P(l)$$

Analogie avec le principe de récurrence sur N

Disponible sur tout ensemble défini inductivement, cf plus tard et cours logique

Ici les listes sont définies inductivement : on se donne la liste vide et un moyen de construire des listes à partir d'une liste existante, d'un élément et de l'opérateur `::`

Reprenons la démonstration de la terminaison de `somlist` par induction structurelle sur la liste argument de la fonction

Posons P(l) la propriété $\exists y \in N.somlist(l) = y$.
On veut montrer $\forall l.P(l)$.

1ère étape : montrons $P([])$.

Par définition de la fonction, $somlist([]) = 0$, donc $P([])$ est vraie (il suffit de prendre $y = 0$).

2ème étape : soit l une liste. Montrons que $P(l)$ implique que pour tout e, on a $P(e :: l)$
L'hypothèse d'induction est ici $P(l)$, soit $\exists y_0 \in N.somlist(l) = y_0$.
Soit e un élément quelconque. Montrons $P(e :: l)$ i.e.
$\exists y \in N.somlist(e :: l) = y$.
Par définition de la fonction, $somlist(e :: l) = e + somlist(l) = e + y_0$ (d'après l'hypothèse d'induction). On a donc $P(e :: l)$, il suffit de prendre $y = e + y_0$.

On a donc démontré en utilisant le principe d'induction structurelle sur les listes que $\forall l.\exists y \in N.somlist(l) = y$.

- `puissance` est définie pour tout exposant naturel

```
let rec puissance (a,n) =
   if n=0 then 1
   else let y = puissance (a, (n / 2)) in
            if n mod 2 = 0 then  y*y else  y*y*a;;
```

L'argument est divisé par 2 à chaque appel
Démonstration : récurrence généralisée sur N (pour a donné)
⇒ A chaque appel, les arguments décroissent pour aboutir au cas de base

Plus généralement NOTION D'ORDRE BIEN FONDE

Soit $<$ une relation d'ordre sur un ensemble E

C'est une relation d'ordre bien fondée si il n'existe pas de suites infinies décroissantes (au sens de cet ordre)

Principe de récurrence bien fondée sur (E,$<$) (induction bien fondée) :

$$(\forall x \in E.(\forall y \in E, y < x \Rightarrow P(y)) \Rightarrow P(x)) \Leftrightarrow \forall x \in E.P(x)$$

De manière générale :

Pour prouver qu'une fonction récursive f termine sur son domaine de définition (i.e. $\forall x \in D_f . \exists y . f(x) = y$) :

- Déterminer son domaine de définition
- Exhiber un ordre (strict) bien fondé sur le domaine de définition
- Montrer que les arguments des appels récursifs restent dans le domaine de définition de la fonction et sont plus petits *(au sens de l'ordre bien fondé)* que l'argument de l'appel initial

(voir la démonstration faite au tableau)

Terminaison : problème indécidable, difficile mais fondamental
On sait le résoudre pour certaines situations ou familles de fonctions

Comment trouver un ordre bien fondé ?

$<$ est un ordre bien fondé sur N, pas sur Z

La relation "être un sous-terme strict" est un ordre bien fondé

L'ordre préfixe sur les mots d'un alphabet est un ordre bien fondé.
($u < v$ *ssi* $\exists u'.v = uu'$)

+ Quelques théorèmes (boite à outils de construction) :

• si $<_1$ est un ordre bien fondé sur E_1, si $<_2$ est un ordre bien fondé sur E_2, alors l'ordre lexicographique $<_{1,2}$ est ordre bien fondé sur $E_1 \times E_2$

Rappel : définition de l'ordre lexico sur $E_1 \times E_2$:
$x_1, x_2 \in E_1, y_1, y_2 \in E_2$,
$(x_1, y_1) <_{1,2} (x_2, y_2)$ *ssi* $x_1 <_1 x_2 \vee (x_1 = x_2 \wedge y_1 <_2 y_2)$

```
let rec f (x,y) =
  if x =0 || y=0 then 0
  else if x=1 then f(x,y-1)
       else f(x-1,y)
```

La fonction f termine sur N^2. Pour la démonstration utiliser l'ordre lexicographique sur N^2 construit à partir de l'ordre usuel sur N.

```
let rec ack (m,p) = match (m,p) with
   (0, _) -> p+1
| (_, 0) -> ack (m-1, 1)
| _ ->  ack (m-1,ack (m,p-1));;
```

- soit m une fonction de E dans N

Si on définit la relation $<_E$ dans E par :
$x <_E y$ *ssi* $m(x) < m(y)$
alors $<_E$ est un ordre bien fondé sur E.

m est appelée *mesure*

Exemple : $E =$ ensemble des listes finies et $m =$ longueur

Rq : on peut généraliser à toute fonction m à valeurs dans un ensemble muni d'un ordre bien fondé.

On peut utiliser ces principes de récurrence pour démontrer d'autres propriétés : correction par rapport à une spécification ou autres

Voir exercices TD.

5. Conception d'algorithmes récursifs

Idée principale : *Diviser pour régner*

- paramétrer le problème (déterminer les différents éléments dont dépend la solution)
- rechercher un ou plusieurs cas de base et donner leur solution respective
- donner la décomposition du cas général en sous-problèmes identiques au problème de départ et s'assurer que les valeurs des paramètres des sous-problèmes *tendent vers les valeurs des paramètres correspondant aux cas particuliers*

• Application 1 : le calcul du reste dans la division euclidienne de deux nombres naturels

Spécification du problème Pour a entier naturel et b entier naturel non nul, on veut calculer r tel que : $\exists q.\ a = b \times q + r$ avec $0 \leq r < b$.

```
Interface
reste  : int * int -> int
arguments a : nombre à diviser
          b : diviseur
précondition : a entier naturel, b entier naturel non
postcondition :  ∃ q ≥ 0, a = b*q + resultat et
                 0 ≤ resultat < b.
```

Cas de base C'est le cas $a < b$, le résultat est alors a (on prend $q = 0$)

Cas général ($a \geqslant b$)

Si on sait calculer le reste de $a - b$ (qui est bien un entier positif) par b, on sait aussi calculer le reste de a par b. C'est le même !

En effet, posons r= reste de $a - b$ par b et q le quotient correspondant
$a - b = q \times b + r$ avec $0 \leq r < b$.
$a = (q + 1) \times b + r$ avec $0 \leq r < b$

```
let rec reste (a, b) =
   if a < b then a else reste (a-b, b);;
```

- Terminaison de la fonction sur $N \times N^*$:

L'ordre bien fondé qui nous intéresse est l'ordre lexicographique sur $N \times N^*$ (à partir de l'ordre naturel sur N)

si $a > b$ alors $a - b \in N$

Décroissance de l'argument de l'appel récursif :

$$(a - b, b) < (a, b) \text{ car } a - b < a \ (b \text{ n'est pas nul})$$

- Correction (partielle) de la fonction par rapport à la spécification

Prouver que $reste(a, b)$ vérifie $\exists q.\ a = b \times q + reste(a, b)$ et $0 \leq reste(a, b) < b$

Par induction bien fondée avec l'ordre lexico sur $N \times N^*$.

- Application 2 : renverser une liste

Spécification du problème Pour *l* liste finie quelconque, calculer la liste miroir

liste miroir de `[a1;a2; ... ;an]` = `[an; ... ;a2;a1]`

Cas de base C'est le cas `[ ]`, le résultat est alors `[ ]`

Cas général la liste *l* est de la forme *a* :: *r*
Si on sait renverser la liste *r*, on sait aussi renverser la liste *l* : il suffit d'ajouter au bout du miroir de *r* l'élément *a*

En effet, si r = `[a2; ... ;an]` alors son miroir est `[an; ... ;a2]`.
Ajoutons a au bout : `[an ... a2; a]`,
c'est bien le miroir de *l* = `[a;a2 ... ;an]`

```
let rec renverser l = match l with
                [] -> []
            | a::r -> (renverser r) @ [a];;
val renverser : 'a list -> 'a list = <fun>
```

- Terminaison de la fonction :

La liste est par définition finie, donc la suite des restes est finie décroissante (au sens du nombre des éléments)
Plus formellement : on choisit l'ordre bien fondé suivant :

$$l_1 <_l l_2 \text{ ssi longueur de } l_1 < \text{longueur de } l_2$$

On peut aussi remarquer que les appels récursifs se font sur des sous-termes stricts : récursion structurelle (critère syntaxique pour la terminaison)

- Prouver que la longueur de *renverser l* est la même que celle de *l*

Par récurrence sur la taille de la liste

$P(n)$ *: pour toute liste de longueur n,* $lg(renverserl) = lg(l)$

Démarche : Montrer $P(0)$ *et Montrer* $P(n+1)$ *en supposant* $P(n)$

Autre méthode : par récurrence sur les listes (Principe d'induction structurelle des listes)

Induction structurelle sur les listes (finies) :

$$P([]) \wedge (\forall l.(P(l) \Rightarrow \forall a.P(a :: l)) \Leftrightarrow \forall l.P(l)$$

Démonstration :

$P(l)$: lg(renverser(l))=lg(l)

- Démontrons $P([])$: lg(renverser([])) = lg([])

- Supposons $P(l)$: $lg(renverser(l)) = lg(l)$
Soit a, un élement.
Il faut montrer $P(a :: l)$ soit $lg(renverser(a :: l)) = lg(a :: l)$
$lg(renverser(a :: l)) = lg(renverser(l)@[a]) = lg(renverser(l)) + lg([a]) = lg(l) + 1 = lg(a :: l)$

QED

- Application 3 : les tours de Hanoi

Spécification du problème

64 anneaux de taille différente sont empilés du plus grand au plus petit sur une tige A.

On veut les empiler sur la tige B en utilisant une tige C comme intermédiaire en respectant les règles suivantes :

- on ne peut déplacer que l'anneau au sommet d'une tige
- un anneau ne peut être posé que sur une tige vide ou sur un anneau plus grand

Écrire un algorithme qui décrit les déplacements à effectuer.

On va calculer la liste des déplacements : on représentera le déplacement de x vers y par (x,y)

Les paramètres :

- `n` nombre d'anneaux à déplacer,
- `dep` la tige de départ (sur laquelle sont empilés les N anneaux,
- `arr` la tige d'arrivée et
- `inter` la tige qui servira d'intermédiaire

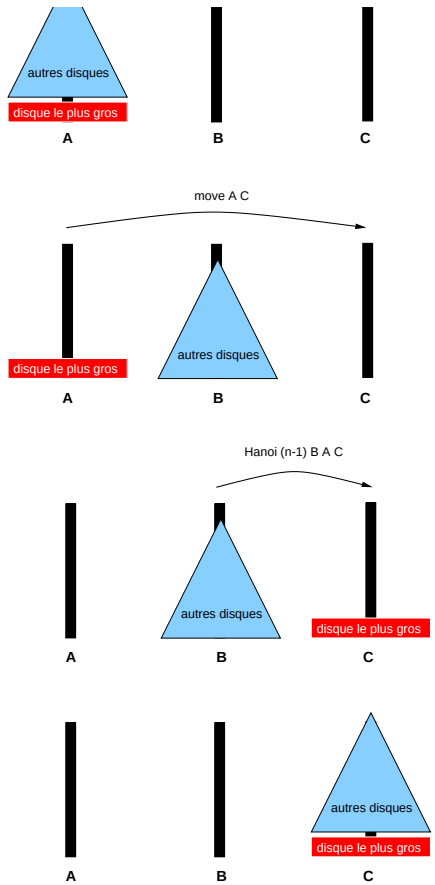
`hanoi(n,dep,arr,inter)` calcule la liste des déplacements pour transférer les `n` anneaux empilés du plus petit au plus grand sur la tige `dep` vers la tige `arr` en utilisant `inter` come tige intermédiaire.

```
hanoi :  int*tige*tige*tige -> (tige*tige) list
```

avec `type tige = A | B | C`

pb à résoudre : `hanoi(64, A, B, C)`

Illustration graphique de la décomposition récursive du cas général `hanoi (n, A, C, B)`



Cas général n anneaux avec n $>$ 0

Déplacer les n-1 plus petits de dep vers inter en se servant de arr

Déplacer le plus grand anneau de dep vers arr

Déplacer les n-1 petits disques de inter vers arr en se servant de dep

Cas de base 0 anneau, pas de déplacement

Le code

```
#let rec hanoi (n,dep,arr,inter) =
 if n=0 then []
 else hanoi(n-1,dep,inter,arr) @
           [(dep, arr)] @ hanoi(n-1,inter,arr,dep);;

hanoi : int * 'a * 'a * 'a -> ('a * 'a) list = <fun>

# hanoi(3,A,B,C);;
- : (tige * tige) list =
[(A, B); (A, C); (B, C); (A, B); (C, A); (C, B); (A, B)]
```

Terminaison

Le nombre de disques diminue de 1 unité à chaque appel récursif

6. Schémas de fonctions récursives

une fonction (algo) récursive (directe) est de la forme générale :

```
let rec f x = if  cas de base then  exp sans f
              else  ... (f  exp)....
```

ou

```
let rec f x = match x with
               cas de base →  exp sans f
              | cas général → ... (f  exp)....
```

- Récursivité non primitive

Si appels récursifs imbriqués : `f (f x)`

Exemple : la fonction d'Ackermann

```
let rec ack (n,p) = match (n,p) with
                      (0,p) -> p+1
                    | (m,0) -> ack(m-1,1)
                    | (m,p) -> ack(m-1, ack(m, p-1));
```

- Récursivité (primitive) terminale

```
let rec f x = if c x then g x
              else f (h x)
```

Aucune autre action derrière l'appel récursif

Exemple : la fonction reste

Tous les appels récursifs s'évaluent à la même valeur : le résultat recherché.

Pas de calcul en suspens, Implémentation optimisée (pas de sauvegarde de contexte, OCaml la transforme en boucle)

- Récursivité (primitive) non terminale

```
let rec f x = if c x then g x
              else k (f (h x), (j x))
```

Exemple : la fonction fact (k = multiplication, j = id, h = prédécesseur)
Calculs mis en suspens, faits pendant la remontée des appels récursifs.
Ne pas chercher d'algorithmes itératifs équivalents sauf si problèmes de performance (complexe dans le cas général : gestion de piles)
On peut parfois transformer une fonction non terminale en une fonction récursive terminale (plus efficace mais souvent moins lisible et moins naturelle)

7. De non terminale à terminale

- Exemple 1 : la factorielle

```
let rec fact n = if n=0 then 1 else n * fact (n - 1);;
```

Autre version :

```
(* Interface fact2 :
type : int*int -> int
arguments : n, acc      précondition : n entier naturel
postcondition : fact2(n,acc) = acc * n!   *)
let rec fact2 (n,acc) = if n=0 then acc
                        else fact2 (n-1, acc*n);;
acc accumulateur : on y accumule les calculs intermédiaires

let fact n = fact2 (n,1);;

fact 4 = fact2(4,1) = fact2(3,1*4) = fact2(2,1*4*3)
       = fact2(1,1*4*3*2) = fact2(0,1*4*3*2*1)=1*4*3*2*1
```

Généralisation : transformation systématique si *op* est une opération binaire infixe associative et possède un élément neutre *e*

soit p une fonction à valeurs dans les booléens, g est une fonction

```
let rec f x = if p x then e
              else (f (g x)) op x;;
```

Autre version :

```
let rec f2 (x,y) = if p x then y
                   else f2 (g x, x op y);;

let f1 x = f2 (x,e);;
```

On se donne également une relation d'ordre bien fondé $<$ telle que $g(x) < x$ pour tout x tel que $p(x) = \mathit{false}$.
on a $f2(x, e) = f(x)$.
Indication : démontrer par induction bien fondée sur $<$ la propriété plus générale : $\forall x, y. f2(x, y) = f(x)\ \mathit{op}\ y$.

- Exemple 2 : renverser une liste

```
let rec renverser l = match l with
  [] -> []
| a::r -> (renverser r) @ [a];;
```

Autre version :

```
let rec renverser2 (l,acc) = match l with
  [] -> acc
| x::r -> renverser2 (r,x::acc);;

let renverser l = renverser2 (l, []);;

renverser [1;2;3] = renverser2 ([1;2;3], [])
                  = renverser2 ([2;3], 1::[])
                  = renverser2 ([3], 2::1::[])
                  = renverser2 ([], 3::2::1::[])
                  =  3::2::1::[]
```

renverser2 (l, acc) calcule (miroir de l) @ acc

On a remplacé la concaténation (coûteuse) par :: (gratuit)

Dans la 1ère version : pour une liste de n éléments : n concaténations

or la concaténation demande le parcours de la 1ère liste

$\Rightarrow$ complexité en n^2.

2ème version : 1 seul parcours de la liste à renverser

$\Rightarrow$ complexité en n.