IA41

Concepts fondamentaux en Intelligence Artificielle et langages dédiés

CM #9b

Etude de problèmes classiques en LISP

Fabrice LAURI

utbm

Plan

- Résolution de problèmes et représentation des données

- Problème #1 : les tours de Hanoi

- Problème #2 : le problème des cruches d'eau

- Problème #3 : le problème de la monnaie

- Problème #4 : le fermier, le renard, l'oie et le grain

Résolution de problèmes

Faire face à un problème signifie que :

- l'on se trouve dans une certaine situation
- que l'on désire atteindre un certain but
- que l'on est pas immédiatement conscient de la suite d'actions à accomplir pour atteindre ce but, c'est-à-dire pour **résoudre le problème**

Pour résoudre un problème, il faut alors :

- (apprendre à) le comprendre dans ses moindres détails
- concevoir un plan, c-à-d :
  - déterminer un lien entre les données (la situation) et les inconnues (le but)
  - considérer éventuellement des sous-problèmes
- mettre le plan à exécution
- examiner la solution.

Représentation du problème

Une résolution efficace nécessite une bonne représentation du problème.

A partir de l'énoncé du problème, déterminer :

- les paramètres constitutifs de l'univers de recherche
- les opérations permises pour passer d'une situation à une autre.

Les exemples de problèmes qui suivent illustrent le fait qu'un problème peut devenir facile à résoudre dès lors que l'on utilise la **bonne représentation**.

Plus précisément, dans ces problèmes, une situation intermédiaire semble avoir plusieurs successeurs possibles, mais un seul d'entre eux mérite d'être considéré.

Problème #1
Les tours de Hanoi

Un peu d'histoire...

Ce puzzle (jeu mathématique) fût inventé par le mathématicien français *Edouard Lucas* en 1883.

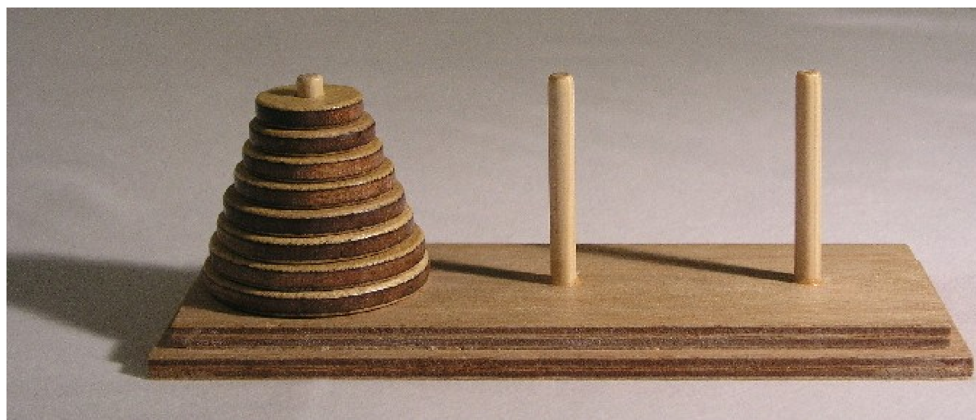
Ce puzzle fût inspiré d'une légende indienne, qui évoque l'existence d'un temple construit au commencement de l'univers.
Dans une vaste salle de ce temple se trouvent trois poteaux, sur lesquels sont disposés au total 64 disques.
Les prêtres du temple, agissant sur la commande d'une prophétie ancestrale, déplace chaque jour l'un des disques selon des règles précises.
Selon la légende, lorsque tous les disques seront placés sur le troisième poteaux, la fin du monde sera imminente.

A raison d'un mouvement de disque chaque seconde :

2^{64}-1 secondes ≈ **585 billion d'années**

Définition du problème

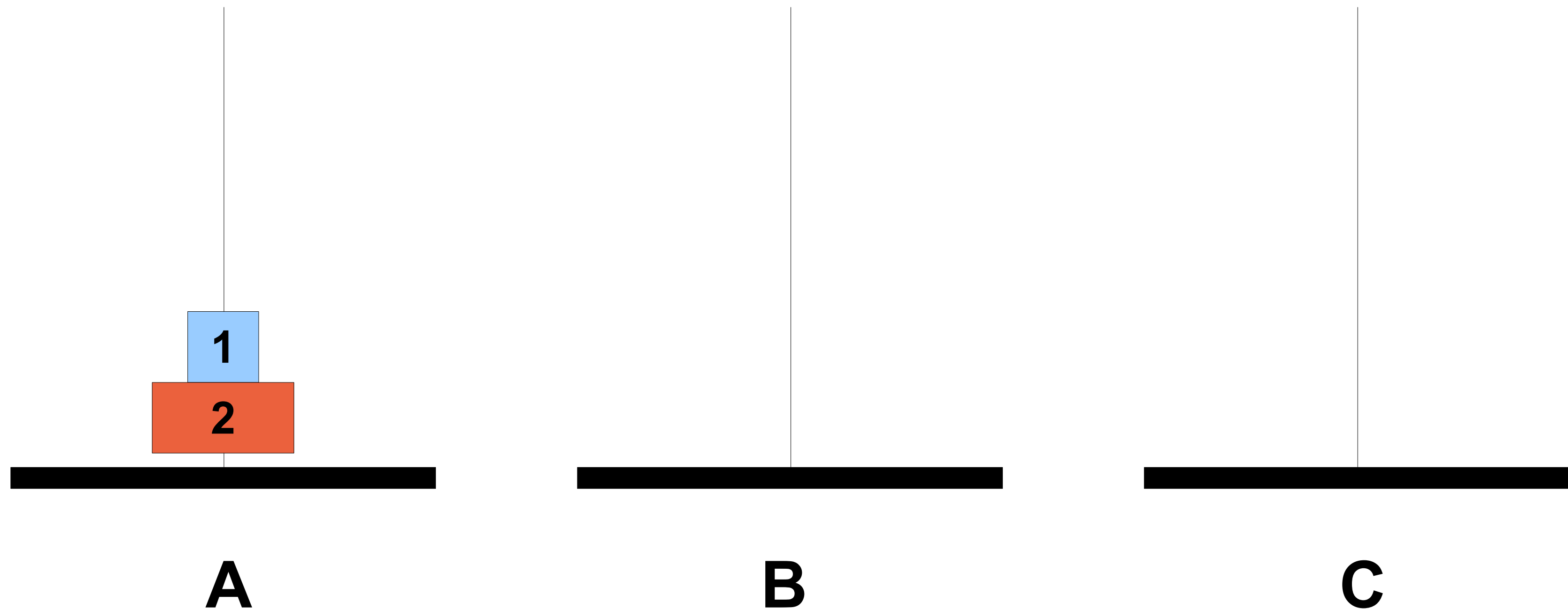
Il se compose de trois piquets de bois, notés A, B et C.
On dispose sur le premier piquet un certain nombre de disques.
Les disques sont tous de diamètres différents, et sont disposés sur le piquet A de telle sorte à former une pyramide, le plus grand disque formant la base de la pyramide.



Le but de ce puzzle est de reconstituer la pyramide de disques du piquet A sur le piquet C, en sachant que :

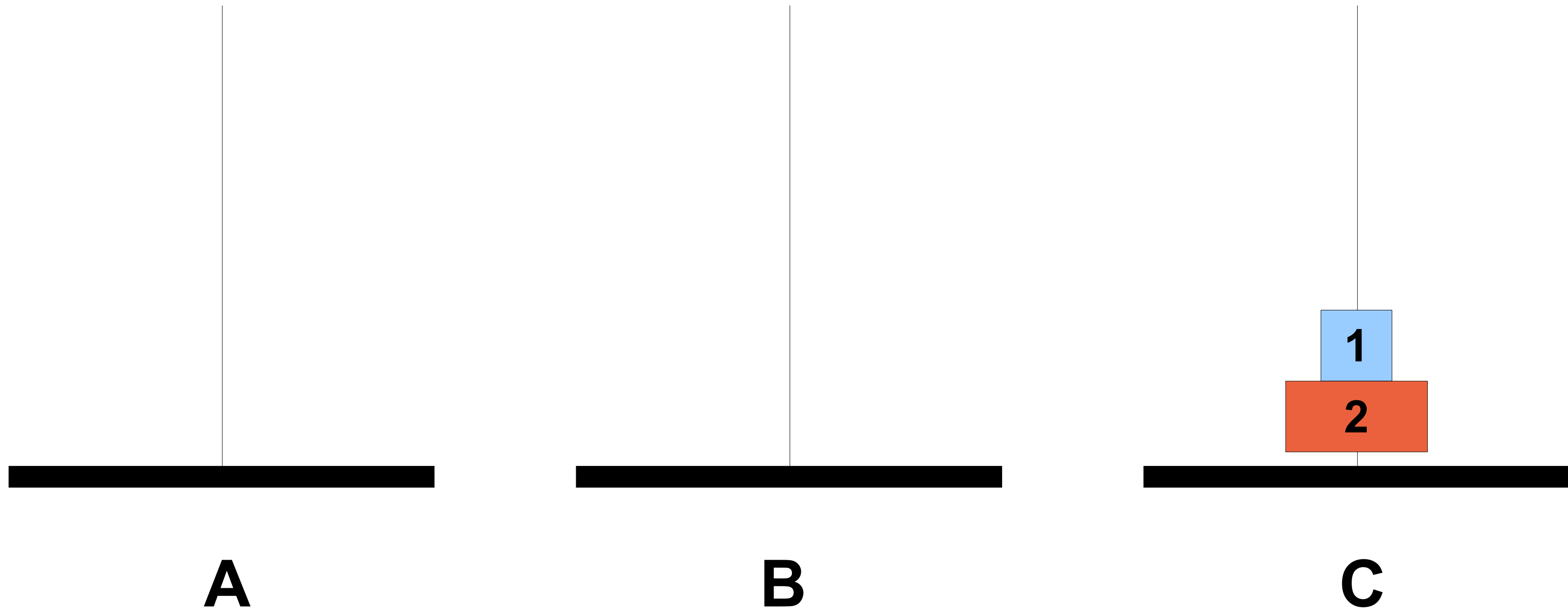
- un seul disque peut être déplacé à la fois d'un piquet à un autre
- un disque ne peut être placé que sur un disque plus grand

Résolution optimale du puzzle avec 2 disques (en 3 étapes)



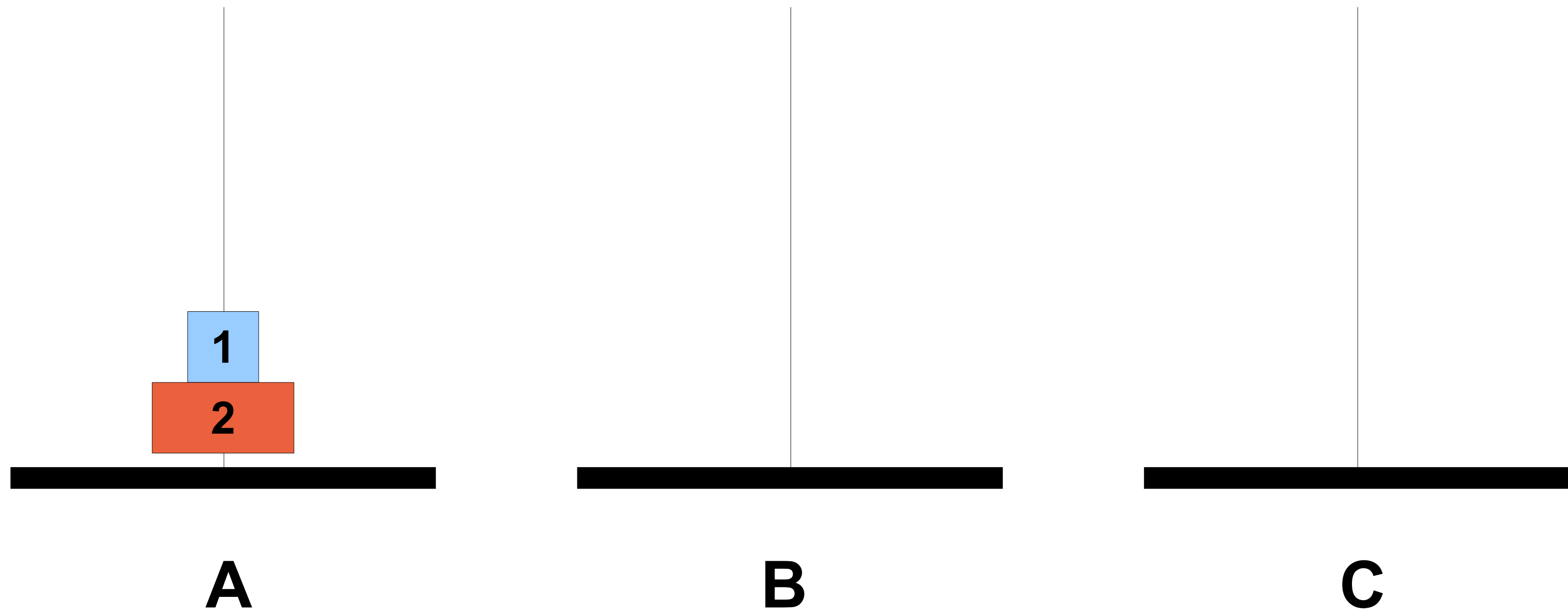
Position initiale des disques

Résolution optimale du puzzle avec 2 disques (en 3 étapes)



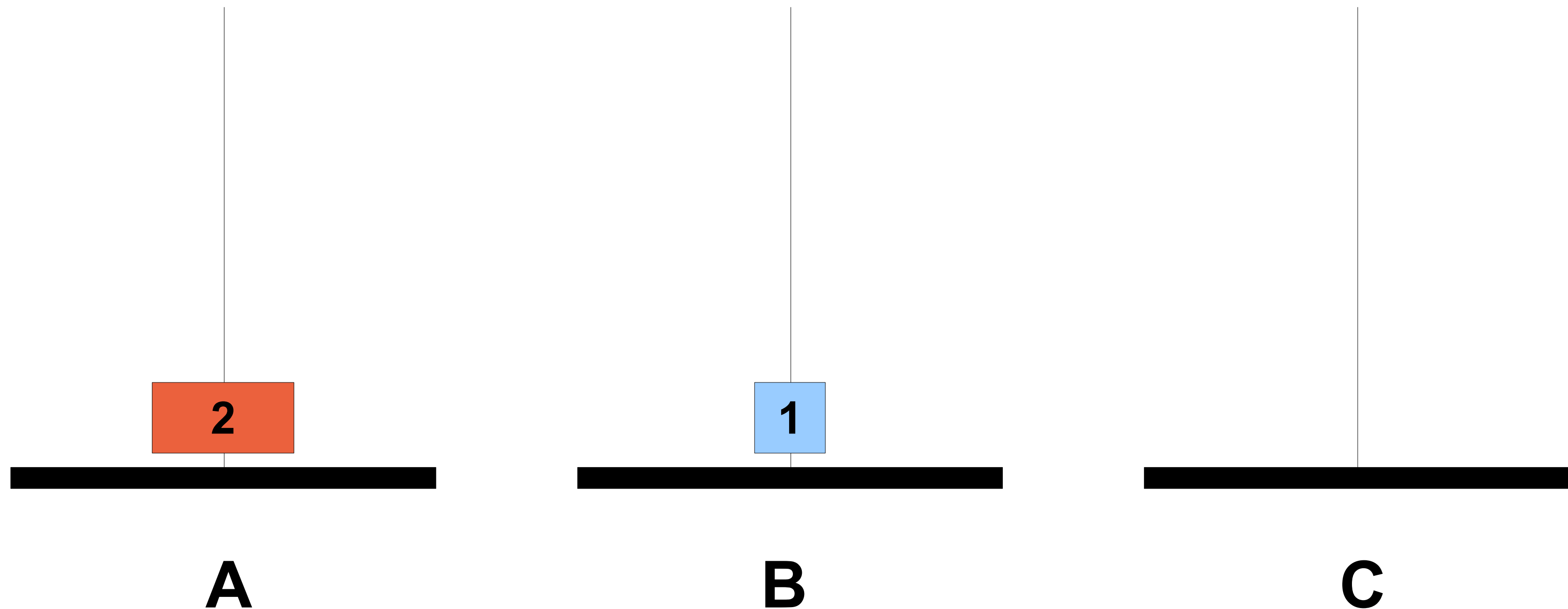
Position finale des disques (but à atteindre)

Résolution optimale du puzzle avec 2 disques (en 3 étapes)



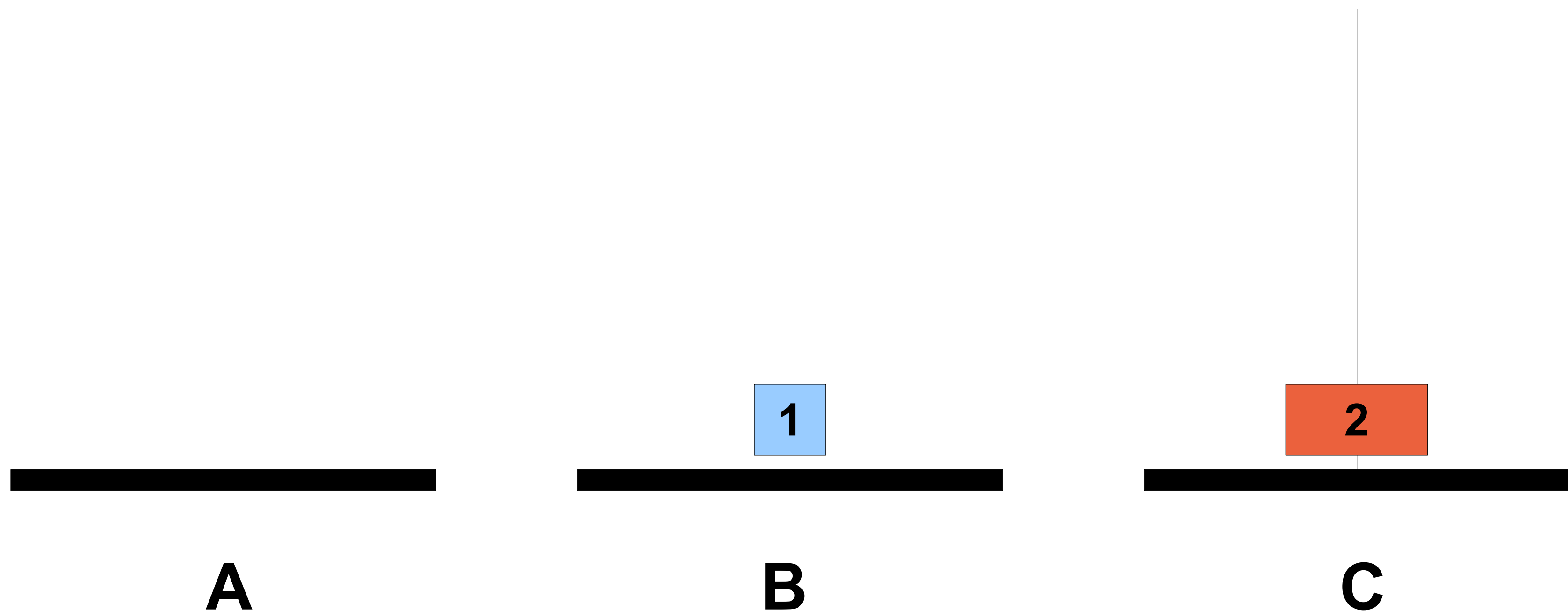
Position initiale des disques

Résolution optimale du puzzle avec n=2



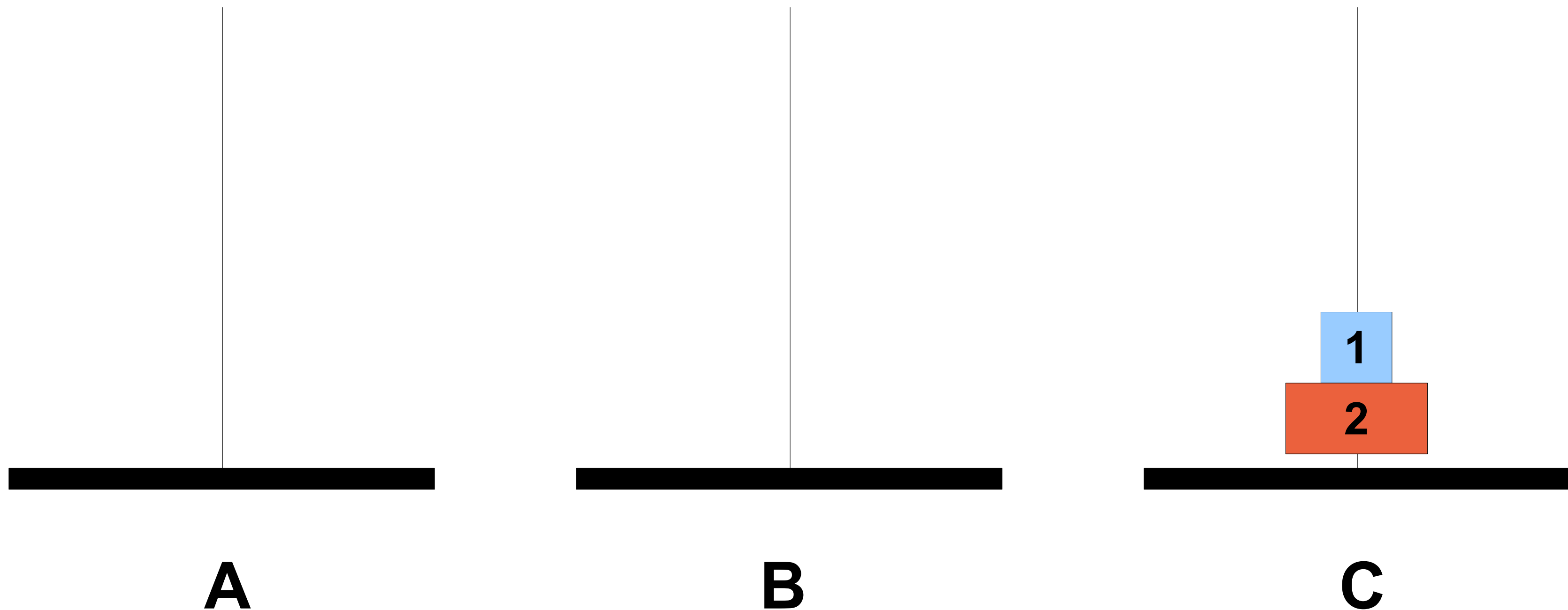
Etape 1 : déplacer 1 de A vers B

Résolution optimale du puzzle avec n=2



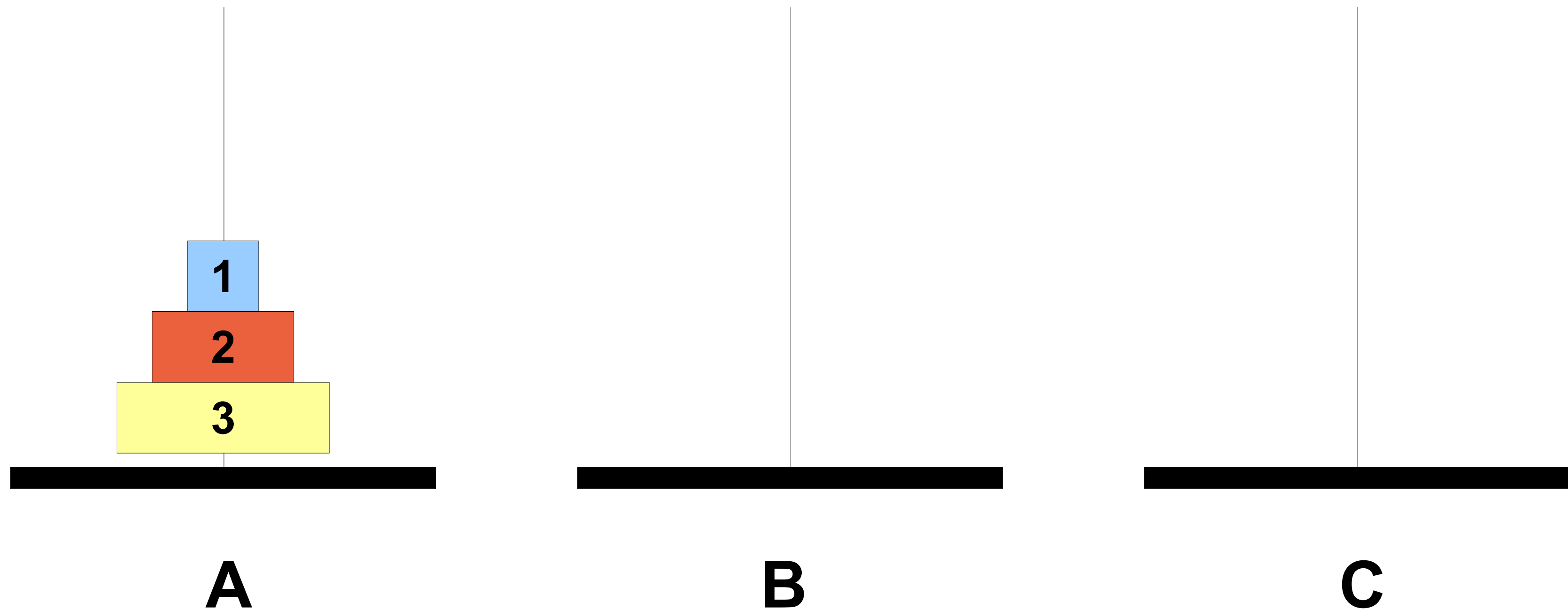
Etape 2 : déplacer 2 de A vers C

Résolution optimale du puzzle avec n=2



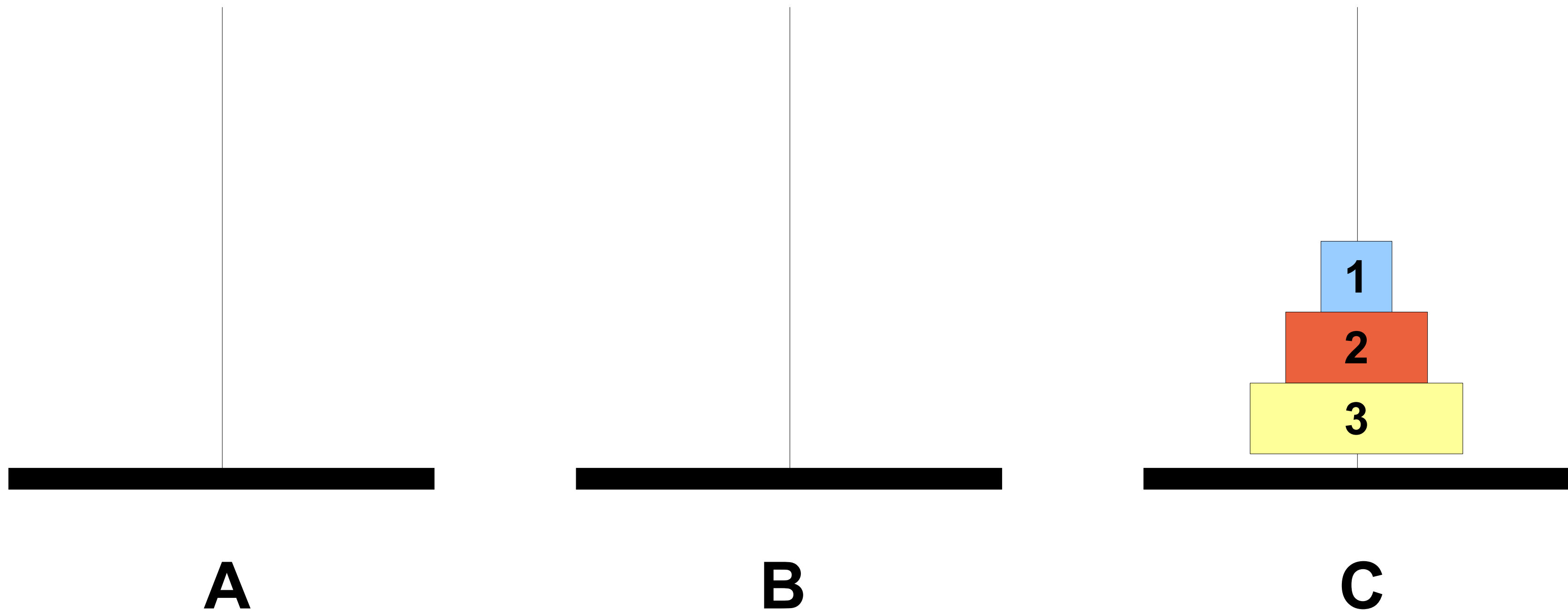
Etape 3 : déplacer 1 de B vers C

Résolution optimale du puzzle avec 3 disques (en 7 étapes)



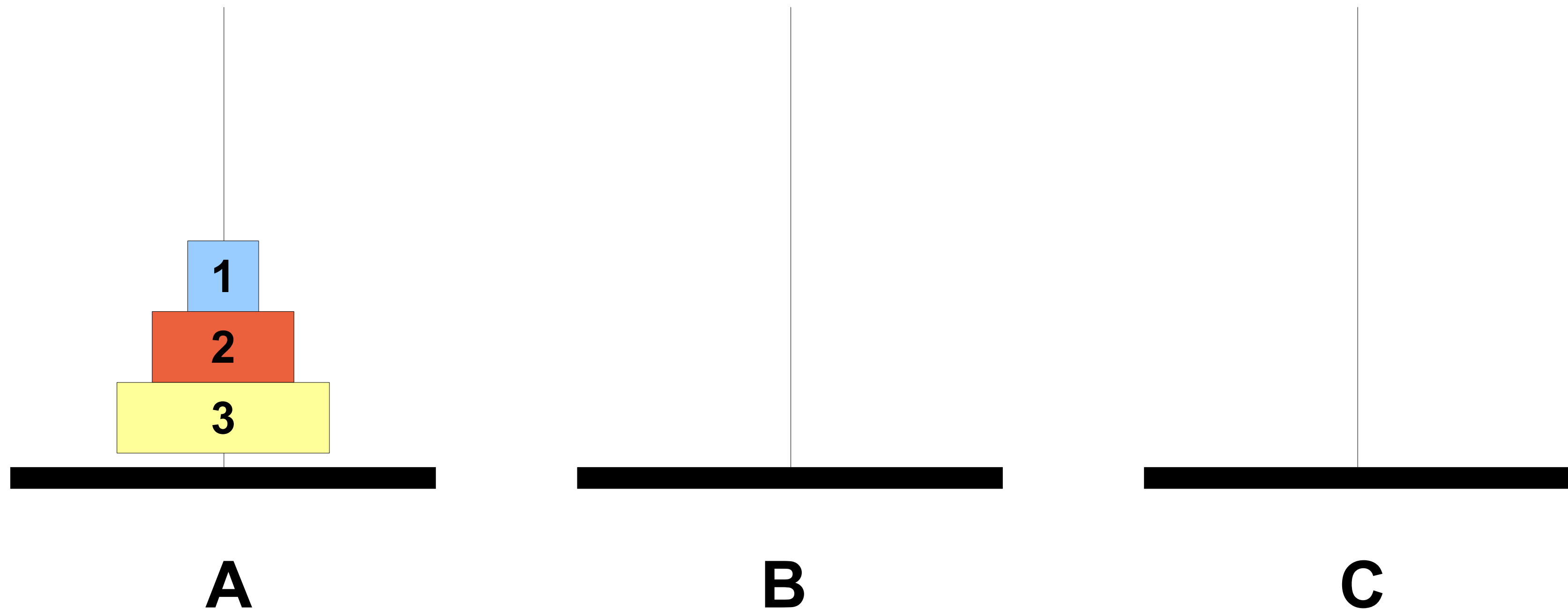
Position initiale des disques

Résolution optimale du puzzle avec 3 disques (en 7 étapes)



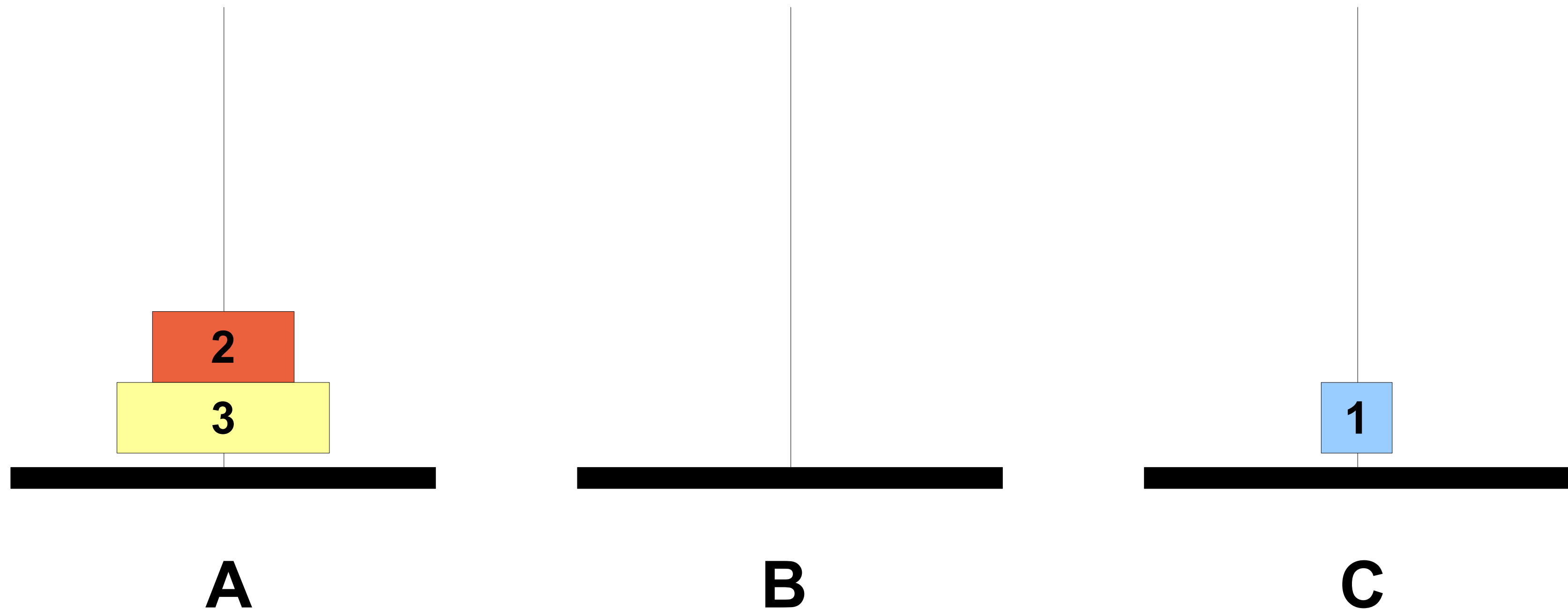
Position finale des disques (but à atteindre)

Résolution optimale du puzzle avec 3 disques (en 7 étapes)



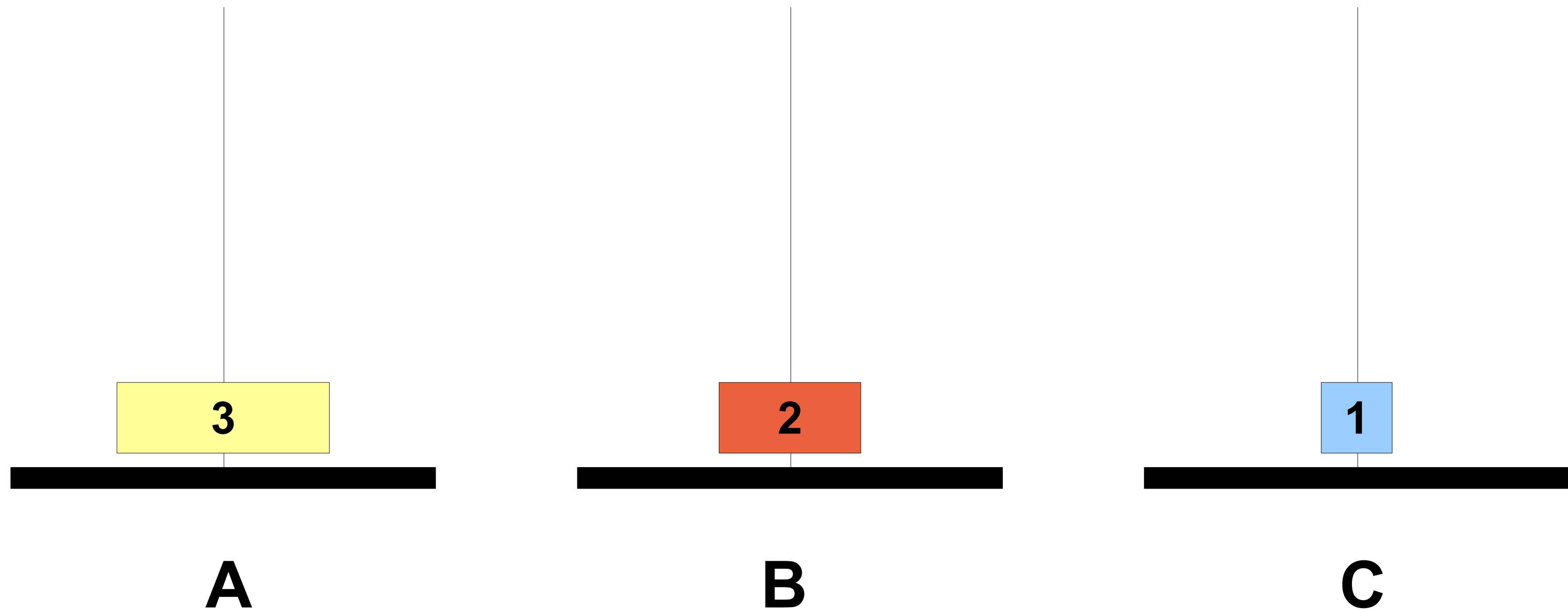
Position initiale des disques

Résolution optimale du puzzle avec n=3



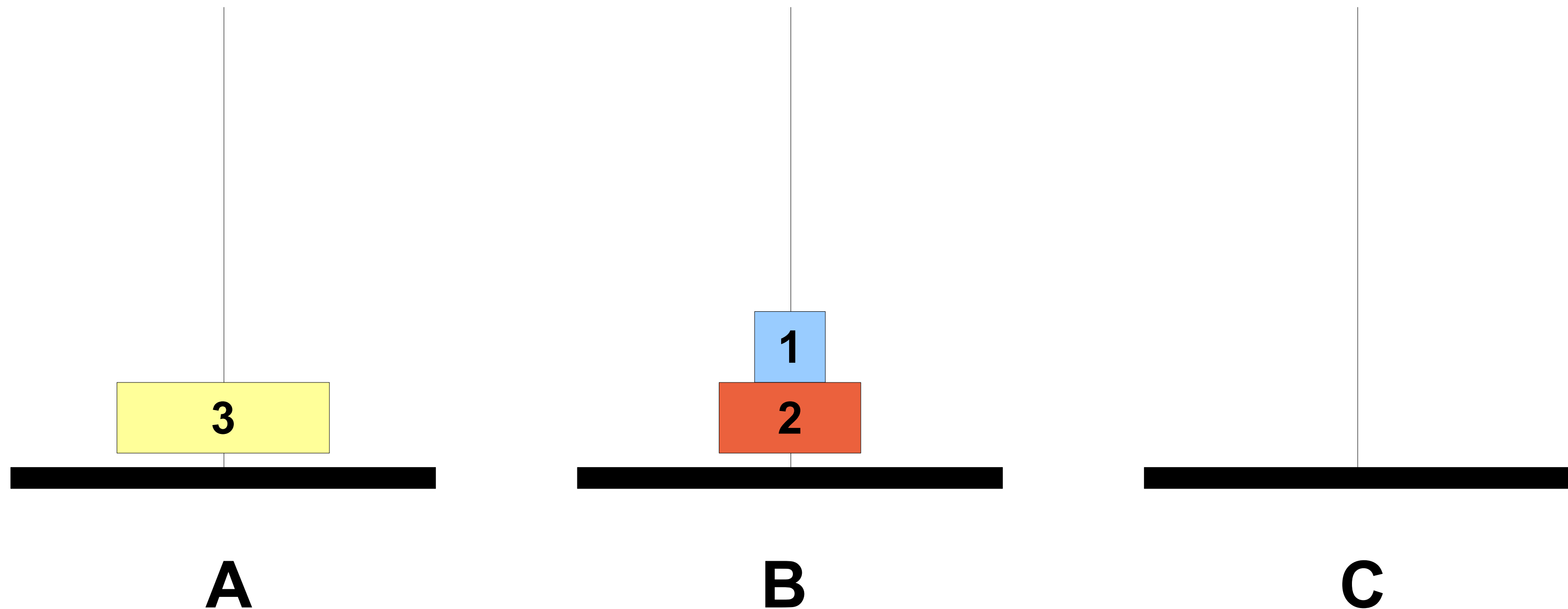
Etape 1 : déplacer 1 de A vers C

Résolution optimale du puzzle avec n=3



Etape 2 : déplacer 2 de A vers B

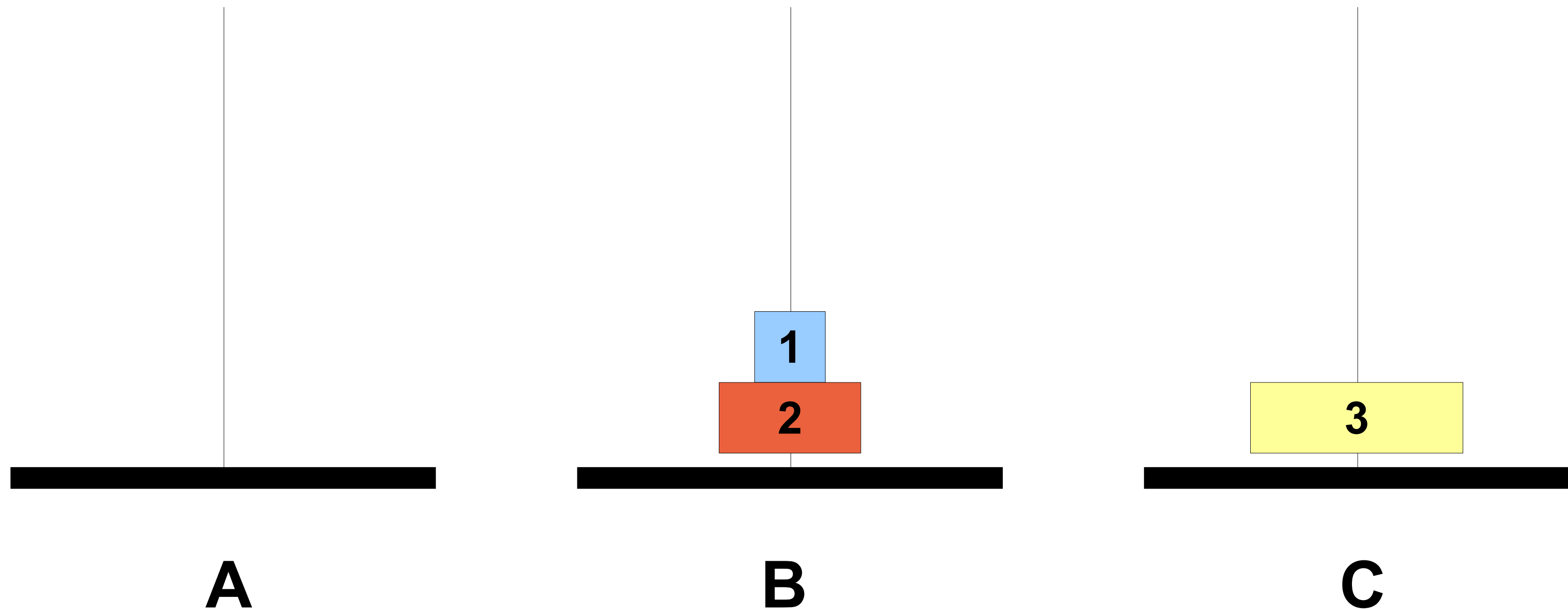
Résolution optimale du puzzle avec n=3



Etape 3 : déplacer 1 de C vers B

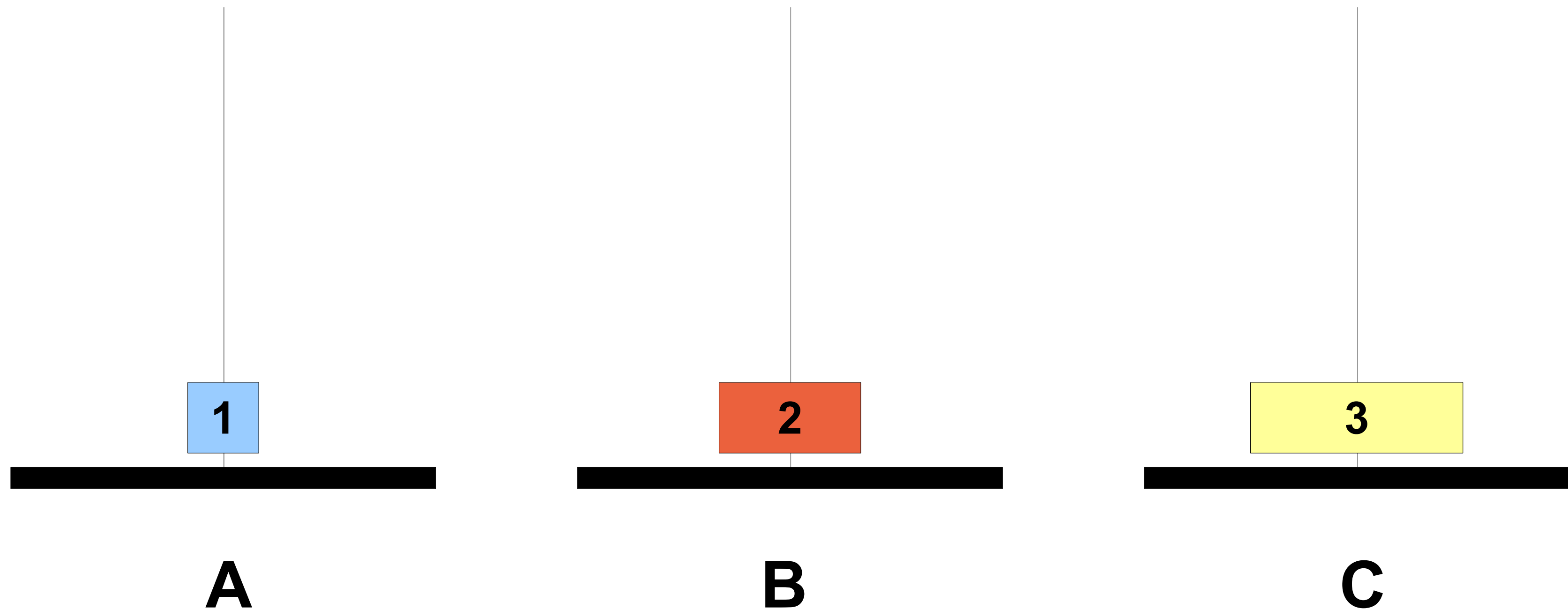
Pyramide 1-2 déplacée de A à B

Résolution optimale du puzzle avec n=3



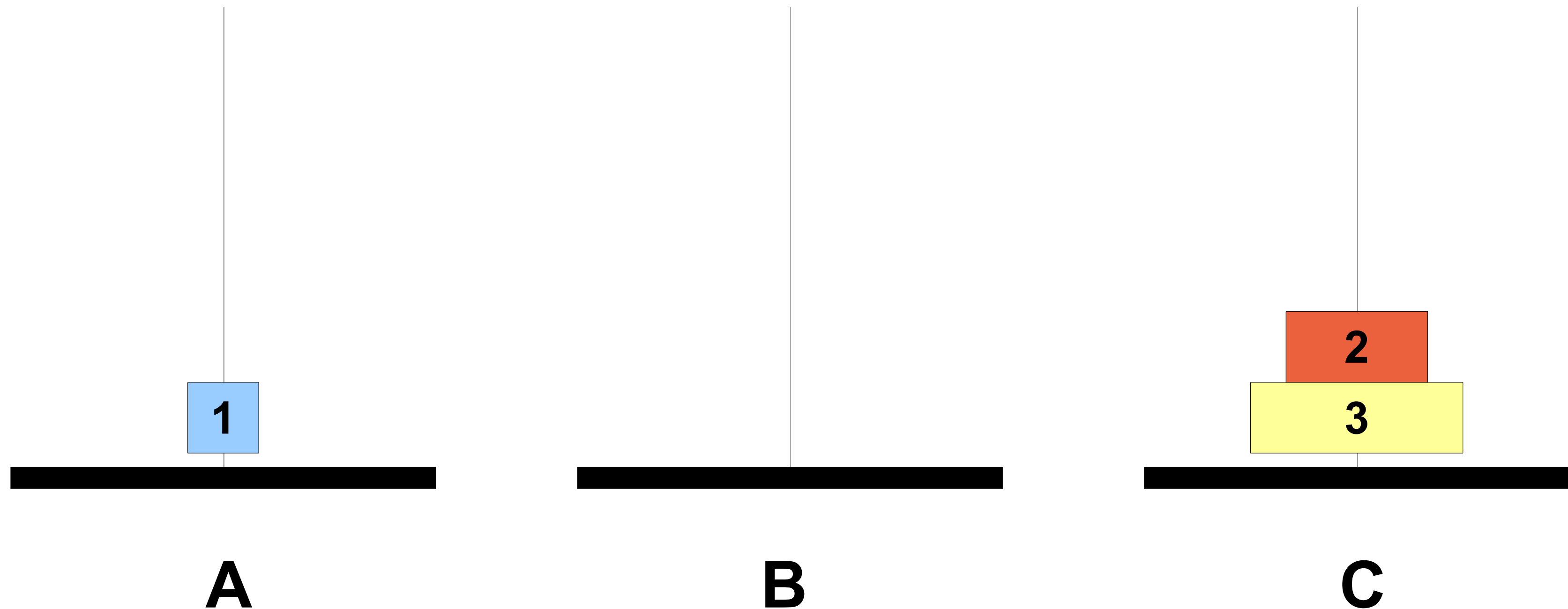
Etape 4 : déplacer 3 de A vers C

Résolution optimale du puzzle avec n=3



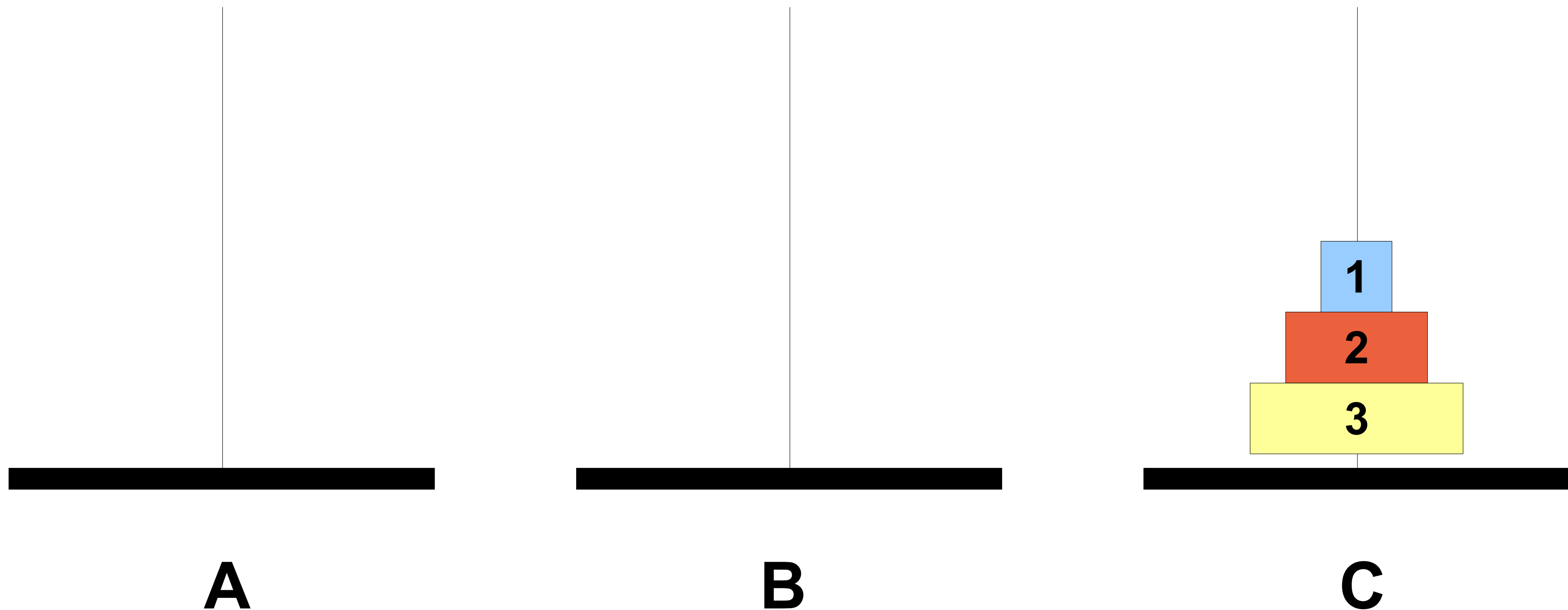
Etape 5 : déplacer 1 de B vers A

Résolution optimale du puzzle avec n=3



Etape 6 : déplacer 2 de B vers C

Résolution optimale du puzzle avec n=3



Etape 7 : déplacer 1 de A vers C

Pyramide 1-2-3 reconstituée sur le piquet C

Résolution du problème

Déplacer n disques du **piquet A** au **piquet C** consiste à suivre les étapes suivantes :

1) déplacer les n-1 disques au dessus du disque n du **piquet A** au **piquet B**, en utilisant le **piquet C** comme **piquet intermédiaire**,

2) déplacer le piquet n du **piquet A** vers le **piquet C**,

3) déplacer les n-1 disques du **piquet B** vers le **piquet C**, en utilisant le **piquet A** comme **piquet intermédiaire**.

→ la **solution est récursive.**

Solution récursive

Soit la fonction *solveHanoi (n A B C)* prenant 4 arguments :
- n est le nombre de disques à déplacer
- A, B et C sont respectivement le piquet de départ, le piquet intermédiaire et le piquet d'arrivée.

En LISP, celle-ci s'écrirait ainsi :

```
(defun solveHanoi (n A B C)
    (if (= n 0) nil
        (solveHanoi (1- n) A C B)    ; déplacer les n-1 disques de A
                                       vers B (C=piquet intermédiaire)

        (moveDisc n A C)             ; déplacer le piquet n de A vers C

        (solveHanoi (1- n) B A C)))  ; déplacer les n-1 disques de B
                                       vers C (A piquet intermédiaire)
```

Trace de la fonction pour n=3

Etape 1 : Déplacer Disque 1 de A vers C
Etape 2 : Déplacer Disque 2 de A vers B
Etape 3 : Déplacer Disque 1 de C vers B

→ Disques 1 et 2 sur piquet B

Etape 4 : Déplacer Disque 3 de A vers C
Etape 5 : Déplacer Disque 1 de B vers A
Etape 6 : Déplacer Disque 2 de B vers C
Etape 7 : Déplacer Disque 1 de A vers C

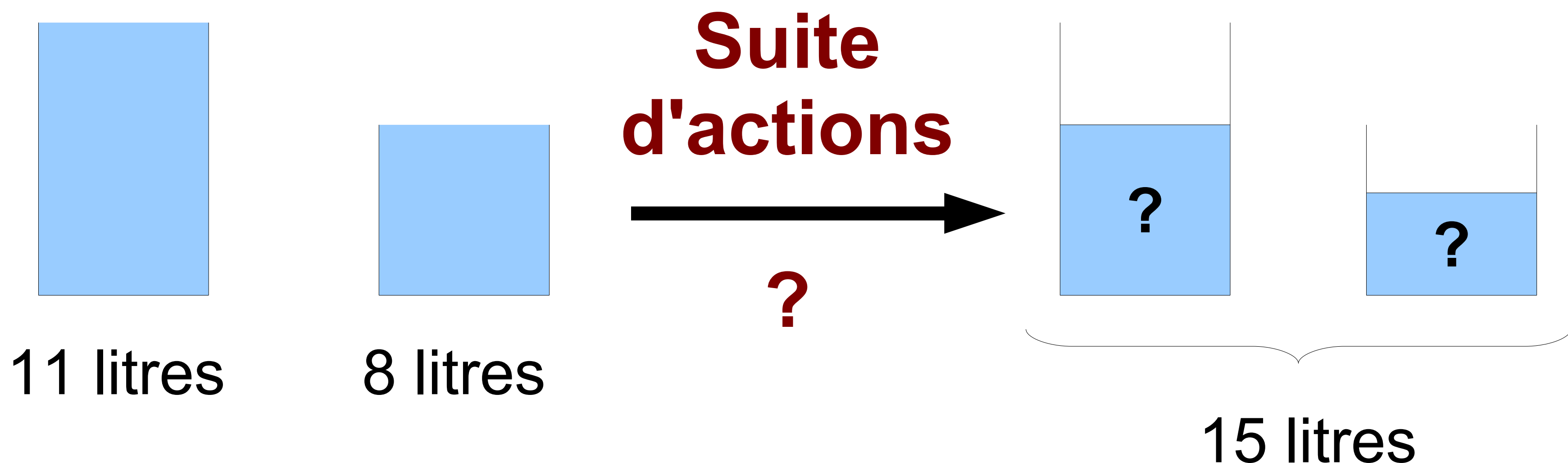
→ Disques 1, 2 et 3 sur piquet C

Problème #2
Le problème des cruches d'eau

Définition du problème

On dispose d'une source inépuisable d'eau et de deux cruches dont les contenances exprimées en litres sont respectivement **a** et **b**.
On désire prélever exactement **c** litres d'eau en utilisant uniquement ces deux cruches, en sachant qu'une cruche ne permet de prélever que sa contenance nominale.

a, **b** et **c** sont des nombres naturels distincts tels que **a** < **b** et **c** < **a+b**.



Représentation du problème

Une représentation possible d'une situation est d'utiliser une paire dont les composants sont les contenus de la petite cruche (notée *PC*) et de la grande cruche (notée *GC*), respectivement. Par exemple (2 4) signifiera que PC contient actuellement 2 litres et que GC contient 4 litres.

On peut dresser une taxinomie des situations :
- Une situation est dite *initiale* si l'une des deux cruches est pleine et l'autre vide.
- On dira qu'une situation est *intéressante* si l'une des cruches est pleine ou vide tandis que l'autre n'est ni pleine ni vide.
- Une situation sera *convenable* si elle est initiale ou intéressante.

Ainsi, les situations (8 0) ou (0 11) sont les deux situations initiales possibles.

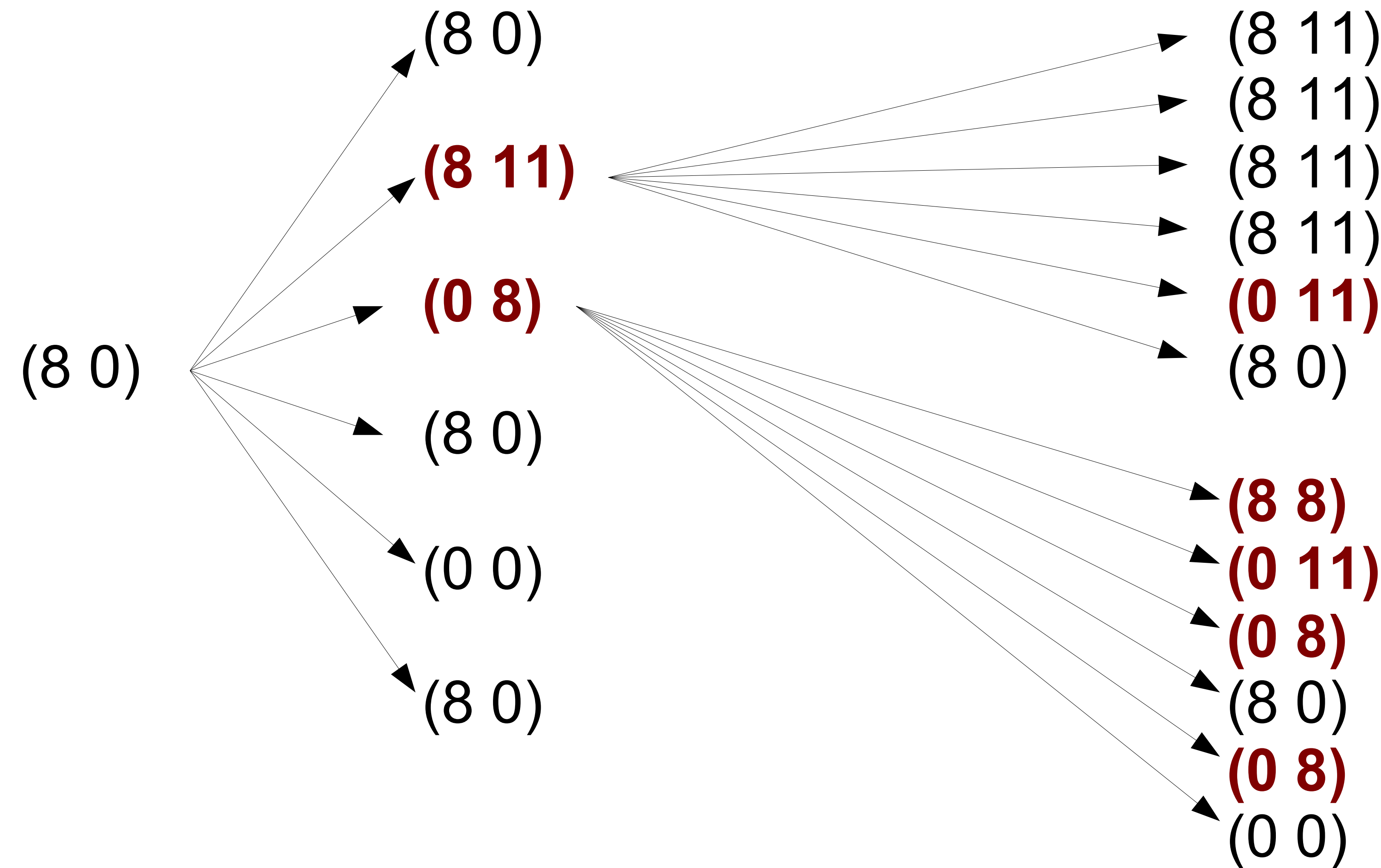
Représentation du problème : les opérations valides

D'emblée, on remarque 6 opérations possibles :
- RP : remplir la petite cruche (PC)
- RG : remplir la grande cruche (GC)
- TPG : transvaser PC dans GC (seulement si GC non pleine)
- TGP : transvaser GC dans PC (seulement si PC non pleine)
- VP : vider PC
- VG : vider GC

Que se passe t'il alors lorsque :
- je remplis une cruche
- je transvase une cruche

Exemple

A partir de la position initiale (8 0), on peut atteindre en deux coups les positions suivantes :



Analyse du problème (1/2)

1. Après remplissage, une cruche a autant d'eau que le permet sa contenance maximale.

2. Transvaser une cruche dans une autre ne modifie pas le contenu total des deux cruches.

3. Avant et après chaque opération, au plus une des deux cruches d'eau peut avoir un contenu autre que nul ou maximal. En effet, toute opération (y compris le transvasement) a pour effet de remplir ou de vider une cruche.

4. Il est évident qu'à la première étape, seules deux possibilités sont intéressantes : remplir la petite cruche ou remplir la grande cruche.

Analyse du problème (2/2)

5. On observe aussi qu'à chaque étape ultérieure, une seule des possibilités est intelligente.

   En effet, exactement une étape sur deux est un transvasement, en provenance de la cruche pleine s'il y en a une, ou à destination de la cruche vide sinon.

   Ce transvasement a pour effet soit de vider la cruche de départ, soit de remplir la cruche d'arrivée.

   Dans le premier cas, l'étape suivante consiste à remplir la cruche vide.
   Dans le second cas, elle consiste à vider la cruche pleine.

Définition d'une fonction contextuelle

```
(defun next_sit (p g sit op)      ; situation convenable suivante
                                    atteignable en un transvasement
                                    ou un remplissage/vidage
  (let ((p1 (car sit)) (g1 (cadr sit)))
      (if (eq op 'trans)
         (cond      ;; transvasement
           ((= p1 p) (let ((dg (- g g1))) (if (> p dg) (cons (- p dg) (list g))
(cons 0 (list (+ g1 p))))))
           ((= p1 0) (if (> p g1) (cons g1 (list 0)) (cons p (list (- g1 p)))))
           ((= g1 g) (cons p (list (- g1 (- p p1)))))
           ((= g1 0) (cons 0 (list p1))))
         (cond      ;; remplissage cruche vide ou vidage cruche pleine
           ((= p1 p) (cons 0 (list g1)))         ;; vidage PC
           ((= p1 0) (cons p (list g1)))         ;; remplissage PC
           ((= g1 g) (cons p1 (list 0)))         ;; remplissage GC
           ((= g1 0) (cons p1 (list g)))))))     ;; vidage GC
```

Définition de la fonction de résolution

```
(defun swapValue (op)          ;; transvasement ↔ rempl./vidage
  (if (eq op 'trans) 'rp_vd 'trans))

(defun solveR (p g sit op n)   ;; génération des situations
                                   convenables atteignables
                                   à partir de sit
  (if (= n (+ (car sit) (cadr sit)))
    t
    (let ((new_sit (next_sit p g sit op)))
     (solveR p g (print new_sit) (swapValue op) n))))

(defun solve (p g sit n)       ;; résolution du problème
  (solveR p g sit 'trans n))
```

Trace de la solution

(solve 8 11 '(8 0) 15)

(0 8)
(8 8)
(5 11)
(5 0)
(0 5)
(8 5)
(2 11)
(2 0)

(0 2)
(8 2)
(0 10)
(8 10)
(7 11)
(7 0)
(0 7)
(8 7)

Problème #3
Le problème de la monnaie

Définition du problème

On dispose d'un ensemble d'espèces monétaires d'une certaine coupure en quantité illimitée.

Le problème de la monnaie consiste à déterminer de combien de manières différentes une certaine somme S peut être payée.

Par exemple, à l'aide de pièces de **1 Euro** et de **5 Euros**, on peut payer la somme de **13 Euros** de **3** manières différentes :

- 13 pièces de 1 Euro
- 8 pièces de 1 Euro et 1 de 5 Euros
- 3 pièces de 1 Euro et 2 de 5 Euros

Représentation du problème

Les données du problème sont les espèces monétaires disponibles et la somme S à payer. Les espèces peuvent être spécifiées dans une liste.

On doit ainsi définir la fonction :
(money <liste> <somme>)

Par exemple :

(money '(1 5) 13) → 3

Stratégie de résolution (récursive)

Si on doit payer une somme de S Euros, et que l'on dispose entre autres de pièces de x Euros, 2 types de solutions existent :

- celles qui n'utilisent pas cette pièce
- celles qui l'utilisent au moins une fois.

On a donc la définition récursive suivante :

$$\text{money } l, s = \text{money } l \backslash \{x\}, s + \text{money } l, s-x$$

Cas de base : liste vide ou somme <= 0

Cas général : liste non vide ou somme > 0

$\{ l = () \text{ ou } s < 0 \} \rightarrow \text{money } l, s = 0$

$\{ s = 0 \} \rightarrow \text{money } l, s = 1$

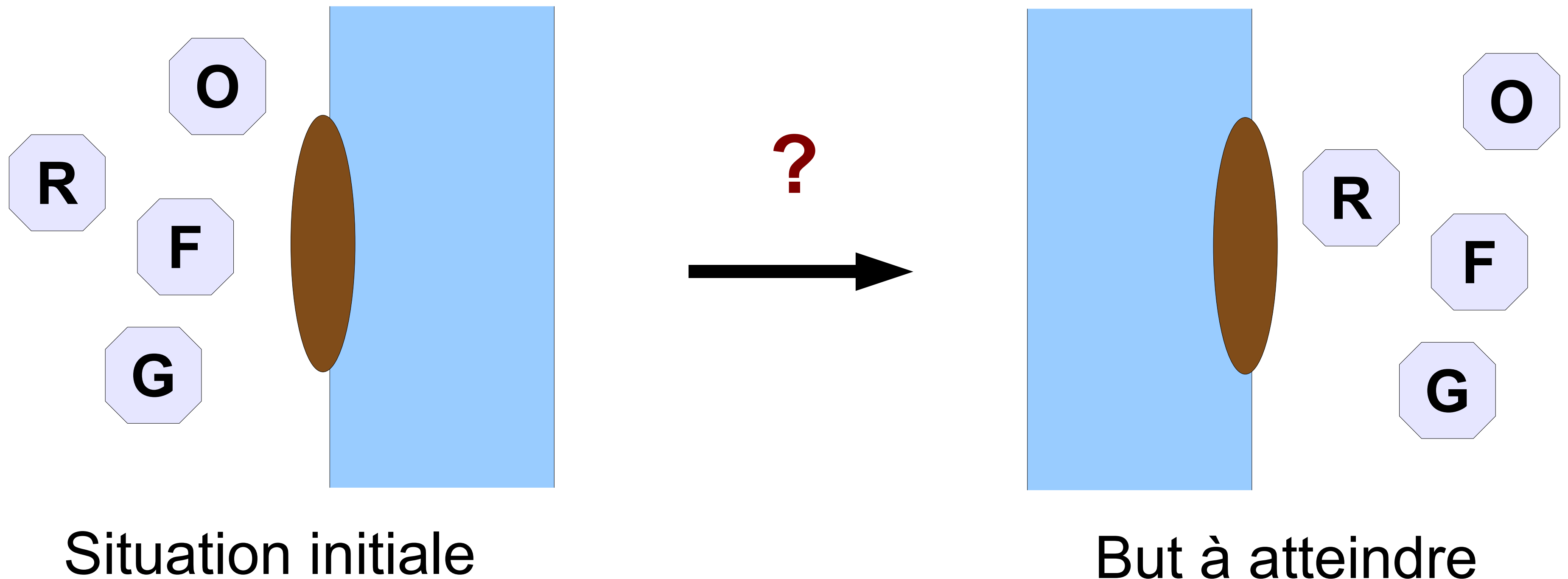
$\{ l \neq () \} \rightarrow \text{money } l, s = \text{money reste } l, s + \text{money } l, s\text{-tete } l$

Problème #4
Le fermier, le renard, l'oie et le grain

Définition du problème

Un fermier possède un renard, une oie et du grain et veut traverser une rivière avec toute cette marchandise. Il ne dispose que d'un bateau dans lequel il ne peut transporter qu'un seul objet à la fois, en plus de lui-même. Il ne peut laisser seuls le renard et l'oie, car le renard mangerait l'oie, ni l'oie et le grain, qui serait alors picoré par l'oie.

Comment ce fermier peut-il donc accomplir sa traversée ?



Représentation du problème

Une représentation possible d'une situation est d'utiliser une liste composée de deux sous-listes, chacune d'elle contenant les entités présentes sur l'une des rives.
Par exemple, la liste ((F O G) (R)) indiquera que le Fermier, l'Oie et le Grain sont présents sur la rive gauche tandis que le Renard est sur la rive droite.

On peut dresser une taxinomie des situations :

- Une situation est dite *initiale* si le Fermier, le Renard, l'Oie et le Grain sont sur la rive gauche. Ainsi, ((F R O G) ()) est la situation initiale.
- On dira qu'une situation est invalide si :
    - le Renard et l'Oie sont seuls présents sur une rive ou
    - l'Oie et le Grain sont seuls présents sur une rive

Représentation du problème : les opérations valides

D'emblée, on remarque 4 opérations possibles :
- traverser avec le renard
- traverser avec l'oie
- traverser avec le grain
- traverser tout seul

A chaque fois que le Fermier traverse la rivière, la situation résultante doit être valide.

Définition de quelques fonctions contextuelles (1/6)

```
(defun getLeftRiver (l)                    ; objets sur la rive gauche
  (car l))

(defun getRightRiver (l)                   ; objets sur la rive droite
  (cadr l))

(defun isOnLeftRiver (l)                   ; fermier sur la rive gauche ?
  (member 'F (getLeftRiver l)))

(defun isOnRightRiver (l)                  ; fermier sur la rive droite ?
  (member 'F (getRightRiver l)))
```

Définition de quelques fonctions contextuelles (2/6)

```
(defun isObjectOnLeftRiver (l e)   ; objet e sur la rive gauche ?
  (member e (getLeftRiver l)))

(defun isObjectOnRightRiver (l e) ; objet e sur la rive droite ?
  (member e (getRightRiver l)))


(defun isGoalReached (l)             ; situation finale atteinte ?
  (let ((rr (getRightRiver l)))
    (and (member 'F rr) (member 'R rr) (member 'O rr) (member
'G rr) t)))
```

Définition de quelques fonctions contextuelles (3/6)

```
(defun isPositionValid (l)          ; position valide ?
  (let ((lr (getLeftRiver l)) (rr (getRightRiver l)))
    (and (isRiverSafe lr) (isRiverSafe rr) t)))

(defun isRiverSafe (l)              ; rive sans risque ?
  (cond ((member 'F l) t)
     ((and (member 'R l) (member 'O l)) nil)
     ((and (member 'O l) (member 'G l)) nil)
     (t t)))
```

Définition de quelques fonctions contextuelles (4/6)

```
(defun traverseWith (l e)          ; traverser la rive avec l'objet e
  (if (and (isOnLeftRiver l) (isObjectOnLeftRiver l e))
      (list (remove 'F (remove e (getLeftRiver l)))
        (append (list 'F e) (getRightRiver l)))
    (if (and (isOnRightRiver l) (isObjectOnRightRiver l e))
      (list (append (list 'F e) (getLeftRiver l))
        (remove 'F (remove e (getRightRiver l)))) l)))

(defun traverseWithNothing (l) ; traverser la rive tout seul
  (if (isOnLeftRiver l) (list (remove 'F (getLeftRiver l))
      (cons 'F (getRightRiver l)))
    (list (cons 'F (getLeftRiver l)) (remove 'F (getRightRiver l)))))
```

Définition de quelques fonctions contextuelles (5/6)

```
(defun traverse (l)      ; traverser la rive de la meilleure manière
  (let ((p1 (traverseWith l 'R))
        (p2 (traverseWith l 'O))
        (p3 (traverseWith l 'G))
        (p4 (traverseWithNothing l)))
    (cond ((and (isPositionValid p4) (isANewPosition p4))
      (print p4))
            ((and (isPositionValid p1) (isANewPosition p1))
      (print p1))
            ((and (isPositionValid p2) (isANewPosition p2))
      (print p2))
           ((and (isPositionValid p3) (isANewPosition p3))
      (print p3)))))
```

Définition de quelques fonctions contextuelles (6/6)

```
(defun isANewPosition (l)    ; nouvelle situation ?
  (not (member (sortPosition l) history :test 'equal)))


(defun sortRiver (l)          ; réarrange les éléments sur une rive
  (cond ((null l) l)
          ((member 'F l) (cons 'F (sortRiver (remove 'F l))))
          ((member 'R l) (cons 'R (sortRiver (remove 'R l))))
          ((member 'O l) (cons 'O (sortRiver (remove 'O l))))
          ((member 'G l) (cons 'G (sortRiver (remove 'G l))))))

(defun sortPosition (l)     ; réarrange les élements sur les 2 rives
  (append (list (sortRiver (getLeftRiver l))) (list (sortRiver
(getRightRiver l)))))
```

Définition de la fonction de résolution

```
(setf start '((F R O G) ()))          ; situation initiale
(setf history ())                     ; historique initial vide

(defun solve (l)                      ; résolution (récursive)
  (if (isGoalReached l) t
    (progn (insertIntoHistory l) (solve (traverse l)))))

(defun insertIntoHistory (e)          ; ajoute e dans l'historique
  (setf history (cons (sortPosition e) history)))
```

Trace de la solution

((F R O G) ()) ; situation initiale :
Fermier, Renard, Oie et Grain sur la rive gauche

((R G) (F O))

((F R G) (O))

((G) (F R O))

((F O G) (R))

((O) (F G R))

((F O) (G R))

(() (F R O G)) ; situation finale :
Fermier, Renard, Oie et Grain sur la rive droite