Récursivité et listes

MP2I - Informatique

Anthony Lick
Lycée Janson de Sailly

Introduction

La récursivité

Définition (récursivité)

Une **fonction récursive** est une fonction qui s'appelle elle-même.

Exemple

Il y a deux manières de calculer la factorielle : avec un produit ou par récurrence.

$$n! = \prod_{i=1}^{n} i \qquad n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

```
1  let fact n =
2    let f = ref 1 in
3    for i=1 to n do
4      f := !f * i
5    done ;
6    !f
7  ;;
```

```
1  let rec fact_rec n = match n with
2    | 0 -> 1
3    | _ -> n * fact_rec (n-1)
4  ;;
```

La récursivité

```
1  let rec fact_rec n = match n with
2    | 0 -> 1
3    | _ -> n * fact_rec (n-1)
4  ;;
```

Définition

Attention : il faut utiliser le mot-clé **rec** lors de la définition, qui indique au compilateur Caml que l'objet qu'on est en train de définir est une fonction récursive.

On distingue clairement deux cas dans cette fonction :

- le cas $n = 0$, appelé **cas terminal** ;
- le cas $n > 0$, qui produit un **appel récursif** à la fonction `fact_rec`.

Pratique de la récursivité

Ce qui se passe “en interne” : la pile d’appels

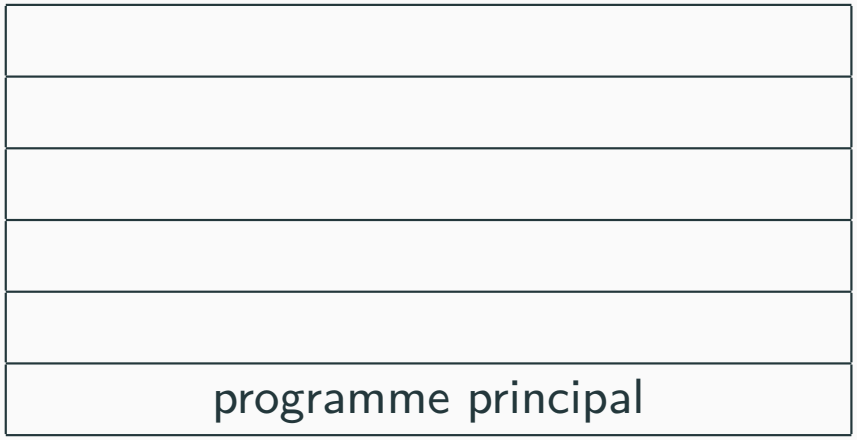
- En informatique, la **pile d’appels** (**call stack** en anglais) est une structure de données de type pile, qui sert à enregistrer des informations au sujet des fonctions actives dans un programme.
- Une **fonction active** est une fonction dont l’exécution n’est pas encore terminée.
- L’utilisation principale de la pile d’appels est de garder la trace de l’endroit où chaque fonction active doit retourner à la fin de son exécution.
- En pratique, lorsqu’une fonction est appelée par un programme, son **adresse de retour** (adresse de l’instruction qui suit l’appel) est empilée sur la pile d’appels.

La pile d'appels

- En plus d'emmagasiner des adresses de retour, la pile d'exécution stocke aussi d'autres valeurs, comme les variables locales de la fonction, les paramètres de la fonction, etc...
- En particulier, lors d'appels imbriqués c'est à dire lorsqu'une fonction f appelle une fonction g, ce qui est relatif à l'appel de la fonction g est placé juste au dessus de ce qui est relatif à la fonction f.
- Lorsque g termine son exécution, ce qui est relatif à l'exécution de g est dépilé. Comme l'adresse de retour est contenu dans la pile d'appel, l'exécution de f peut reprendre juste après l'endroit où g a été appelée.

La récursivité

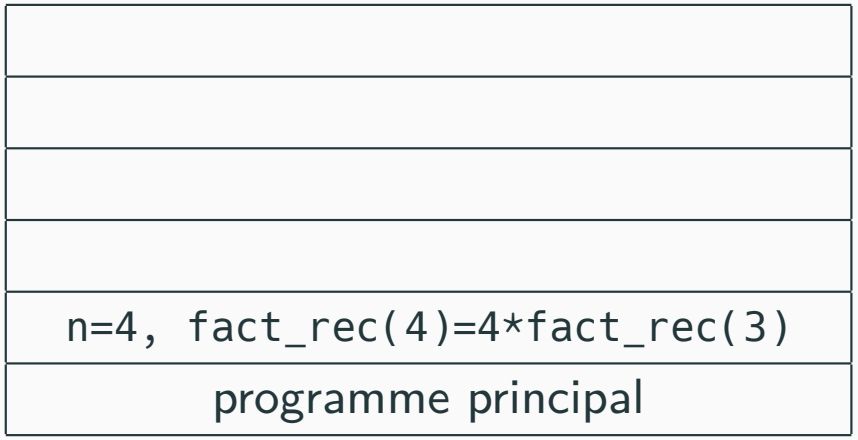
```
1  let rec fact_rec n = match n with
2    | 0 -> 1
3    | _ -> n * fact_rec (n-1)
4  ;;
5
6  (* programme principal *)
7  ...
8  fact_rec 4
```



Une fonction récursive est essentiellement une fonction qui s'appelle elle-même, ainsi les appels successifs de f s'empilent dans la pile d'appels.

La récursivité

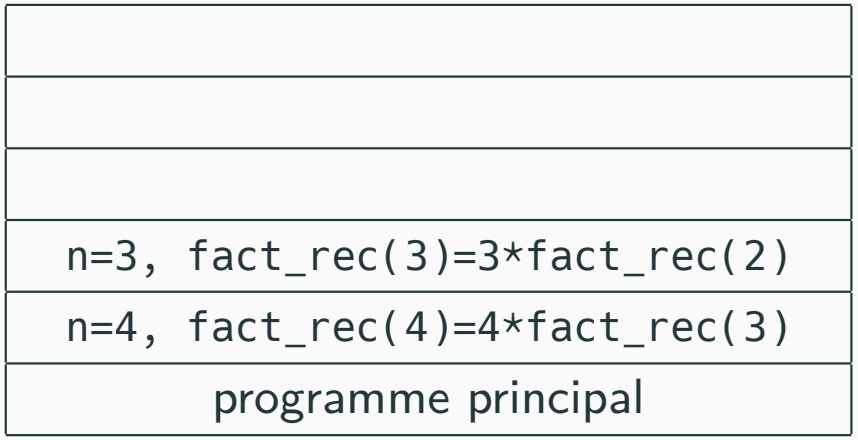
```
1  let rec fact_rec n = match n with
2    | 0 -> 1
3    | _ -> n * fact_rec (n-1)
4  ;;
5
6  (* programme principal *)
7  ...
8  fact_rec 4
```



Une fonction récursive est essentiellement une fonction qui s'appelle elle-même, ainsi les appels successifs de f s'empilent dans la pile d'appels.

La récursivité

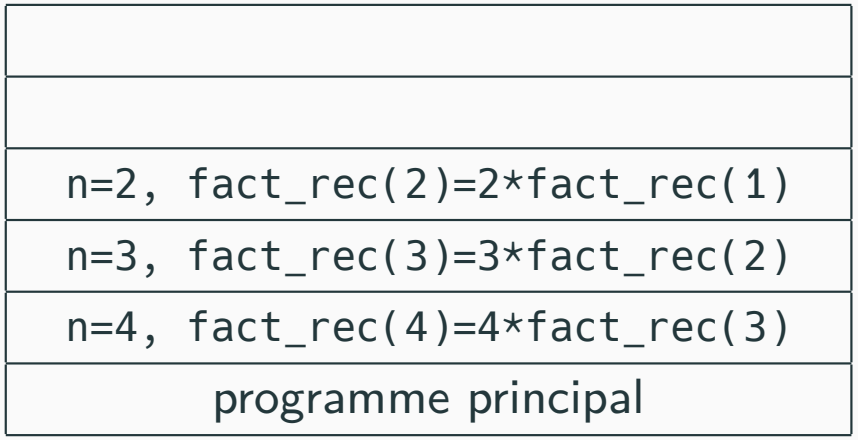
```
1  let rec fact_rec n = match n with
2    | 0 -> 1
3    | _ -> n * fact_rec (n-1)
4  ;;
5
6  (* programme principal *)
7  ...
8  fact_rec 4
```



Une fonction récursive est essentiellement une fonction qui s'appelle elle-même, ainsi les appels successifs de f s'empilent dans la pile d'appels.

La récursivité

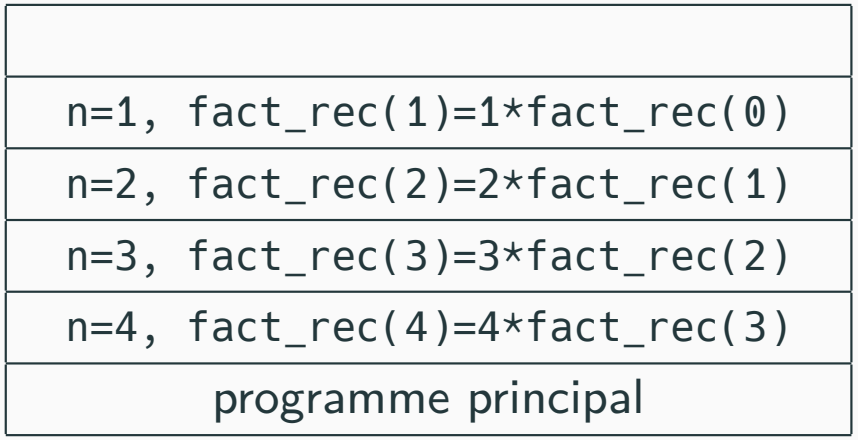
```
1  let rec fact_rec n = match n with
2    | 0 -> 1
3    | _ -> n * fact_rec (n-1)
4  ;;
5
6  (* programme principal *)
7  ...
8  fact_rec 4
```



Une fonction récursive est essentiellement une fonction qui s'appelle elle-même, ainsi les appels successifs de f s'empilent dans la pile d'appels.

La récursivité

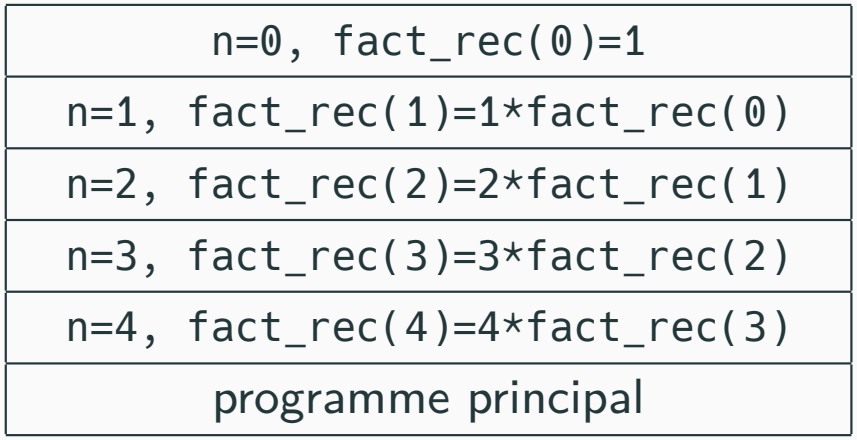
```
1  let rec fact_rec n = match n with
2    | 0 -> 1
3    | _ -> n * fact_rec (n-1)
4  ;;
5
6  (* programme principal *)
7  ...
8  fact_rec 4
```



Une fonction récursive est essentiellement une fonction qui s'appelle elle-même, ainsi les appels successifs de f s'empilent dans la pile d'appels.

La récursivité

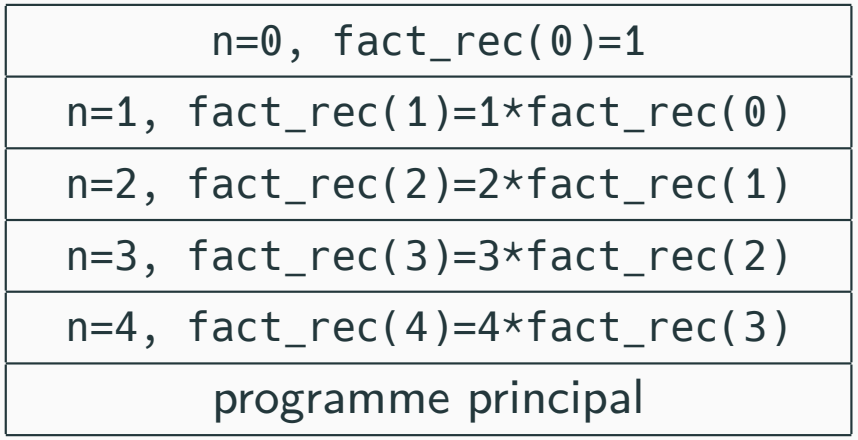
```
1  let rec fact_rec n = match n with
2    | 0 -> 1
3    | _ -> n * fact_rec (n-1)
4  ;;
5
6  (* programme principal *)
7  ...
8  fact_rec 4
```



Une fonction récursive est essentiellement une fonction qui s'appelle elle-même, ainsi les appels successifs de f s'empilent dans la pile d'appels.

La récursivité

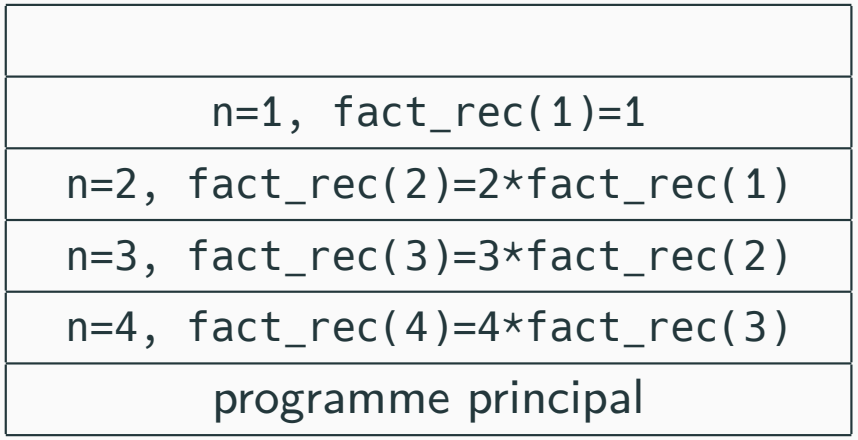
```
1  let rec fact_rec n = match n with
2    | 0 -> 1
3    | _ -> n * fact_rec (n-1)
4  ;;
5
6  (* programme principal *)
7  ...
8  fact_rec 4
```



Une fois arrivé à un cas terminal (ne produisant plus d'appel récursif), le nombre d'éléments de la pile d'appels se réduit.
Dans le cas de la fonction factorielle, comme celle-ci ne rappelle qu'une fois, dès lors qu'on a commencé à dépiler on ne s'arrête plus.

La récursivité

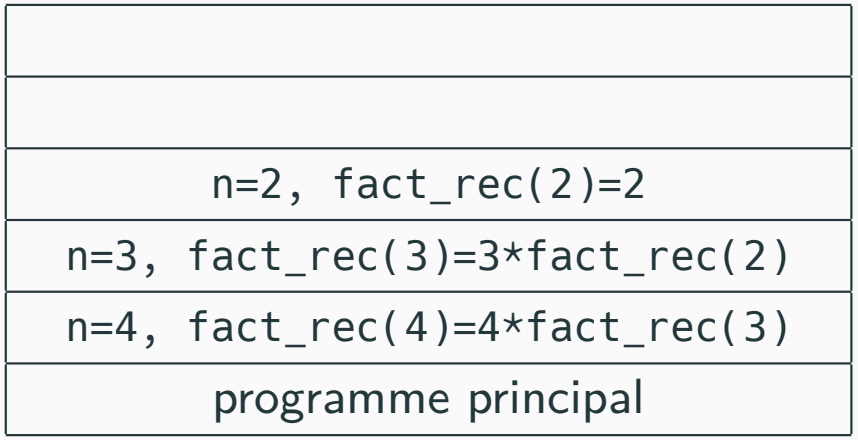
```
1  let rec fact_rec n = match n with
2    | 0 -> 1
3    | _ -> n * fact_rec (n-1)
4  ;;
5
6  (* programme principal *)
7  ...
8  fact_rec 4
```



Une fois arrivé à un cas terminal (ne produisant plus d'appel récursif), le nombre d'éléments de la pile d'appels se réduit.
Dans le cas de la fonction factorielle, comme celle-ci ne rappelle qu'une fois, dès lors qu'on a commencé à dépiler on ne s'arrête plus.

La récursivité

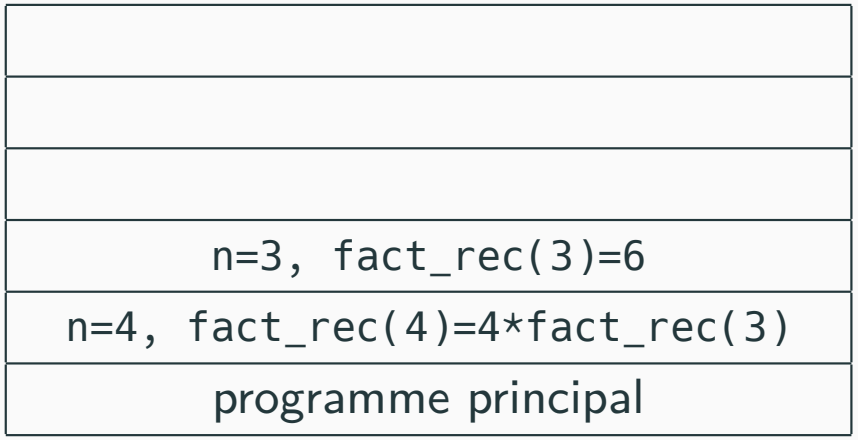
```
1  let rec fact_rec n = match n with
2    | 0 -> 1
3    | _ -> n * fact_rec (n-1)
4  ;;
5
6  (* programme principal *)
7  ...
8  fact_rec 4
```



Une fois arrivé à un cas terminal (ne produisant plus d'appel récursif), le nombre d'éléments de la pile d'appels se réduit.
Dans le cas de la fonction factorielle, comme celle-ci ne rappelle qu'une fois, dès lors qu'on a commencé à dépiler on ne s'arrête plus.

La récursivité

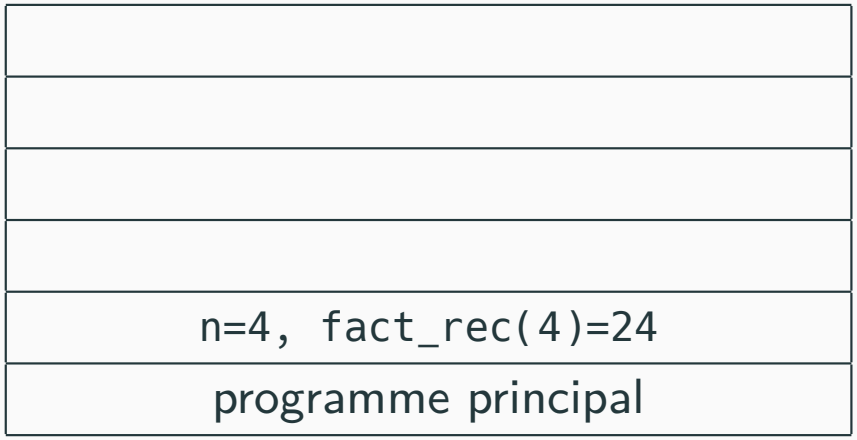
```
1  let rec fact_rec n = match n with
2    | 0 -> 1
3    | _ -> n * fact_rec (n-1)
4  ;;
5
6  (* programme principal *)
7  ...
8  fact_rec 4
```



Une fois arrivé à un cas terminal (ne produisant plus d'appel récursif), le nombre d'éléments de la pile d'appels se réduit.
Dans le cas de la fonction factorielle, comme celle-ci ne rappelle qu'une fois, dès lors qu'on a commencé à dépiler on ne s'arrête plus.

La récursivité

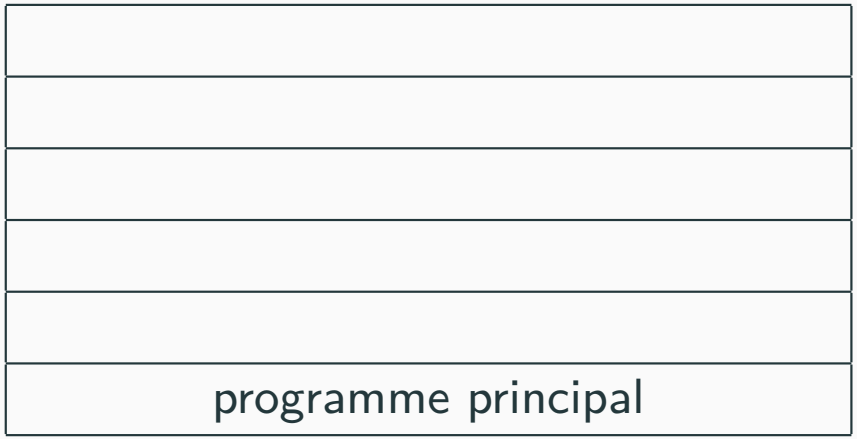
```
1  let rec fact_rec n = match n with
2    | 0 -> 1
3    | _ -> n * fact_rec (n-1)
4  ;;
5
6  (* programme principal *)
7  ...
8  fact_rec 4
```



Enfin, la récursion s'arrête lorsqu'on dépile l'élément correspondant au premier appel de la fonction.

La récursivité

```
1  let rec fact_rec n = match n with
2    | 0 -> 1
3    | _ -> n * fact_rec (n-1)
4  ;;
5
6  (* programme principal *)
7  ...
8  fact_rec 4
```



Enfin, la récursion s'arrête lorsqu'on dépile l'élément correspondant au premier appel de la fonction.

Tracer les appels de fonction

La fonction #trace

On peut vérifier le comportement expliqué précédemment en **traçant** la fonction `fact_rec`.

L'instruction **`#trace`** permet d'imprimer à l'écran les entrées et sorties d'une fonction.
(pour arrêter de la tracer, il faut utiliser **`#untrace`**).

```
# #trace fact_rec ;;
fact_rec is now traced.
# fact_rec 4 ;;
fact_rec <-- 4
fact_rec <-- 3
fact_rec <-- 2
fact_rec <-- 1
fact_rec <-- 0
fact_rec --> 1
fact_rec --> 1
fact_rec --> 2
fact_rec --> 6
fact_rec --> 24
- : int = 24
```

Stack Overflow

Taille de la pile d'appels

On voit que le nombre d'appels imbriqués réalisés par une fonction récursive peut être important : il faut stocker ces appels, ce qui est coûteux en mémoire.

En pratique, on peut voir la pile d'appels comme une pile de capacité finie : si la pile est pleine, un appel supplémentaire produit un dépassement de capacité : le fameux **stack overflow** en anglais.

```
1  # fact_rec 1000000 ;;
2  Stack overflow during evaluation (looping recursion?).
```

Récursivité terminale

```
1  let rec somme n = match n with
2    | 0 -> 0
3    | _ -> n + somme (n-1)
4  ;;
```

Exemple

Voici un autre exemple de fonction récursive : le calcul de la somme des entiers de 0 à n, pour $n > 0$.

Dans le dernier test ci-dessous, la capacité de la pile d'appels a été dépassée : le calcul n'a pu aboutir par manque de mémoire.

Quelques tests

```
1  #somme 100 ;;
2  - : int = 5050
3  #somme 1000 ;;
4  - : int = 500500
5  # somme 1000000 ;;
6  Stack overflow during evaluation (looping recursion?).
```

récursivité terminale

```
1  let rec somme_avec_acc n acc = match n with
2    | 0 -> acc
3    | _ -> somme_avec_acc (n-1) (n+acc)
4  ;;
```

```
1  #somme_avec_acc 100 0 ;;
2  - : int = 5050
3  #somme_avec_acc 1000 0 ;;
4  - : int = 500500
5  #somme_avec_acc 1000000 0 ;;
6  - : int = 500000500000
```

Exemple

Considérons alors une autre fonction ci-dessus, qui utilise un **accumulateur acc** comme deuxième argument, dans lequel se font les additions. Observons quelques tests avec **acc=0**.

Cette fonction permet donc de calculer la somme des entiers de 0 à n, mais ne semble pas souffrir du grossissement de la pile d'appels. En effet, cette fonction est **récursive terminale**.

récursivité terminale

Définition

Une fonction f récursive est dite **récursive terminale** si tout appel récursif est le dernier calcul réalisé par f.

Exemple

La fonction `somme_avec_acc` respecte bien cette définition : lorsque l'on n'est pas dans le cas terminal $n = 0$, la fonction calcule $n - 1$ et $n + acc$ avant de réaliser l'appel récursif, qui est alors le dernier calcul effectué lors de cet appel de fonction.

récursivité terminale

Intérêt

L'intérêt d'avoir une fonction f **récursive terminale** est qu'un appel récursif à f ne nécessite pas d'empilement sur la pile d'appels : l'appel récursif effectué peut prendre la place de l'appel en cours dans la pile d'appels, puisqu'il n'y a plus d'instructions à effectuer une fois l'appel récursif terminé.

Certains compilateurs, dont le compilateur Caml, sont capables de détecter les fonctions récursives terminales, et d'en tirer profit pour diminuer les empilements dans la pile d'appels.

D'autres langages, comme Python par exemple, ne gèrent pas la récursivité terminale.

Utilisation d'un accumulateur

En Caml, on fait donc souvent une transformation similaire à la précédente pour transformer une fonction récursive en fonction récursive terminale utilisant un **accumulateur**.

Il reste un petit détail : il est un peu pénible de devoir passer 0 en paramètre de la fonction pour la valeur initiale de l'accumulateur.

récursivité terminale

```
1  let somme2 n =
2    let rec aux n acc = match n with
3      | 0 -> acc
4      | _ -> aux (n-1) (acc+n)
5    in aux n 0
6  ;;
```

Utilisation d'une fonction auxiliaire

Une solution est de définir la fonction somme_avec_acc comme interne à une autre fonction, qui se contentera de l'appeler avec acc=0.

La fonction somme_avec_acc n'existant plus dans le programme principal, on peut lui donner un nom plus court (souvent aux, pour auxiliaire).

Exemples de fonctions récursives

Algorithme d'Euclide

```
1  let rec pgcd a b = match b with
2    | 0 -> a
3    | _ -> pgcd b (a mod b)
4  ;;
```

Exemple

```
1  # pgcd 1898615 16586155318 ;;
2  - : int = 1
```

Exemple : Algorithme d'Euclide

Une méthode récursive de calcul du PGCD de deux entiers positifs est la suivante :

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

Cette méthode mène à la fonction récursive terminale ci-dessus.

Exemples de fonctions récursives

Exemple : Algorithme d'exponentiation rapide

L'algorithme d'exponentiation rapide se reformule très facilement en faisant usage de la récursivité.

En effet, pour $n \geq 0$, on a :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ \left(x^{n/2}\right)^2 & \text{si } n \text{ est pair} \\ x \times \left(x^{\lfloor n/2 \rfloor}\right)^2 & \text{si } n \text{ est impair} \end{cases}$$

Exemples de fonctions récursives

Algorithme d'exponentiation rapide

```
let rec expo_rapide x n = match n with
  | 0 -> 1
  | _ -> let y = expo_rapide x (n/2) in
         if n mod 2 = 0 then y*y else y*y*x
;;
```

Exemple : Algorithme d'exponentiation rapide

La fonction correspondante ci-dessus n'est pas récursive terminale, ce qui n'est pas gênant car ici seulement $O(\ln n)$ appels récursifs sont effectués.

Récursivité croisée

Récursivité croisée

Rappel : **and** permet de définir simultanément plusieurs objets.

Il est possible de définir simultanément plusieurs fonctions qui s'appellent mutuellement (on parle de **récursivité mutuelle** ou **croisée**).

Exemple

```
let rec pair n = match n with
  | 0 -> true
  | _ -> impair (n-1)
and impair n = match n with
  | 0 -> false
  | _ -> pair (n-1)
;;
```

Tests

```
# pair 5 ;;
- : bool = false
# pair 6 ;;
- : bool = true
# impair 5 ;;
- : bool = true
```

Récursivité croisée

Moyenne arithmético-géométrique

```
let rec u n = match n with
  | 0 -> u0
  | _ -> (u (n-1) +. v (n-1))/. 2.
and v n = match n with
  | 0 -> v0
  | _ -> sqrt (u (n-1) *. v (n-1))
;;
```

Tests avec u0=1. et v0=4.

```
#u 3 ;;
- : float = 2.24303398875
#v 3 ;;
- : float = 2.24302317183
#u 4 ;;
- : float = 2.24302858029
#v 4 ;;
- : float = 2.24302858028
```

Exemple

Dans le code ci-dessus, on suppose que `u0` et `v0` sont des variables globales définies au préalable.

Les deux fonctions définissent alors deux suites $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$, dont on pourrait montrer qu'elles sont adjacentes : leur limite est appelée la moyenne arithmético-géométrique de u_0 et v_0.

Récursivité croisée

Définition

Pour prendre en compte la récursivité croisée, on peut donner une autre définition de la récursivité.

Une fonction f est dite **récursive** lorsque dans la pile d'appels peuvent se trouver simultanément plusieurs appels à f.

Remarque

Notons que les deux fonctions u et v ont malheureusement une grande complexité, comme on va le voir maintenant sur un autre exemple.

Chevauchement des appels récursifs

Exemple

Considérons la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ définie par :

$$F_n = \begin{cases} 1 & \text{si } n = 0 \text{ ou } n = 1 \\ F_{n-2} + F_{n-1} & \text{sinon} \end{cases}$$

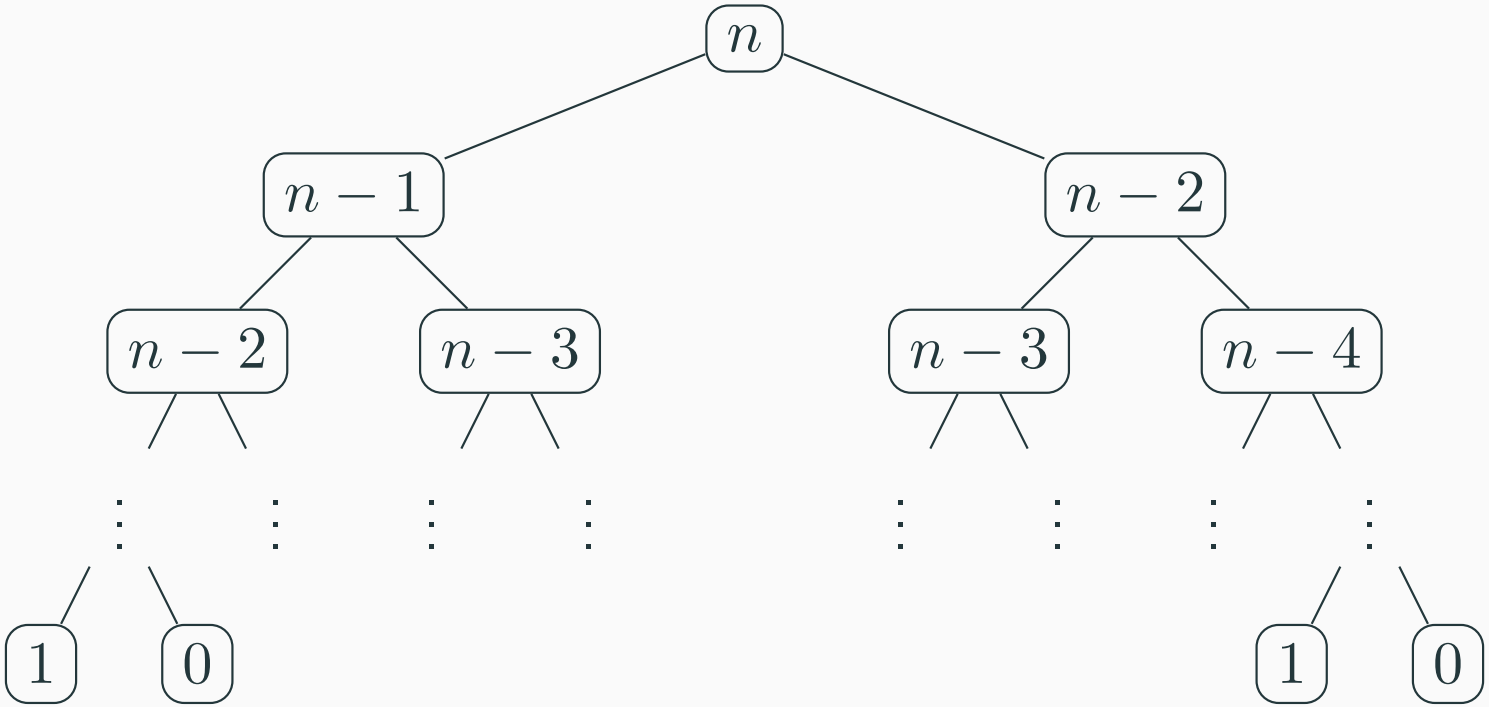
Une transcription en Caml s'obtient aisément :

```
1  let rec fibo n = match n with
2   | 0 | 1 -> 1
3   | _ -> fibo (n-1) + fibo (n-2)
4 ;;
```

Chevauchement des appels récursifs

Problème

Le problème de l'algorithme précédent est le nombre d'appels récursifs effectués.



Chevauchement des appels récursifs

Notons A_n le nombre d'appels récursifs nécessaires pour le calcul de F_n.

Alors, la suite (A_n) vérifie la relation de récurrence :

$$A_0 = A_1 = 0$$
$$A_n = 2 + A_{n-1} + A_{n-2}, \text{ pour } n \geq 2$$

On a alors

$$A_n + 2 = (A_{n-1} + 2) + (A_{n-2} + 2)$$

Chevauchement des appels récursifs

Autrement dit, la suite $(A_n + 2)_{n,\in\mathbb{N}}$ coïncide avec la suite $(2F_n)_{n\in\mathbb{N}}$.

On peut donner une expression explicite de A_n :

$$\forall n \in \mathbb{N}, \quad A_n = \frac{2}{\sqrt{5}} \left[\left(\frac{\sqrt{5}+1}{2} \right)^{n+1} - \left(\frac{1-\sqrt{5}}{2} \right)^{n+1} \right] - 2$$

On a donc $A_n \sim C \times \left(\frac{\sqrt{5}+1}{2} \right)^n$, avec $C > 0$.

Ainsi, le calcul de F_n par cette méthode est de complexité exponentielle, puisque $\frac{\sqrt{5}+1}{2} > 1$.

Chevauchement des appels récursifs

Chevauchement

On parle ici de chevauchement des appels récursifs : la fonction, bien que correcte, nécessite de réaliser plusieurs fois les mêmes calculs pour aboutir.

Elle est impraticable pour des grands n.

Déjà le calcul de F_{42} nécessite quasiment 10 secondes sur mon ordinateur personnel, et chaque incrémentation de n demande une multiplication du temps de calcul par environ 1.618.

```
# let a=Sys.time() in let _ = fibo 42 in Sys.time() -. a ;;
- : float = 9.90973699999999802
# let a=Sys.time() in let _ = fibo 43 in Sys.time() -. a ;;
- : float = 16.0324019999999976
```

Chevauchement des appels récursifs

Comment éviter ce phénomène ?

Le même phénomène fait que les fonctions u et v précédentes sont inefficaces.

Donnons deux méthodes pour éviter ce problème :

- utiliser une fonction itérative, mais on perd la formulation récursive ;
- stocker les valeurs déjà calculées dans un dictionnaire.

En conclusion, il faut faire attention à ne pas faire des appels récursifs qui se recoupent, sous peine de voir la complexité exploser !

Un exemple plus complet : les tours de Hanoï

Avantage de la récursivité

Avantage

On a déjà vu un avantage des fonctions récursives : leur formulation est **plus simple** que leur équivalent impératif.

On va maintenant voir un exemple de problème pour lequel une solution récursive est très adaptée, mais pour lequel une solution impérative n'est pas facile à trouver.

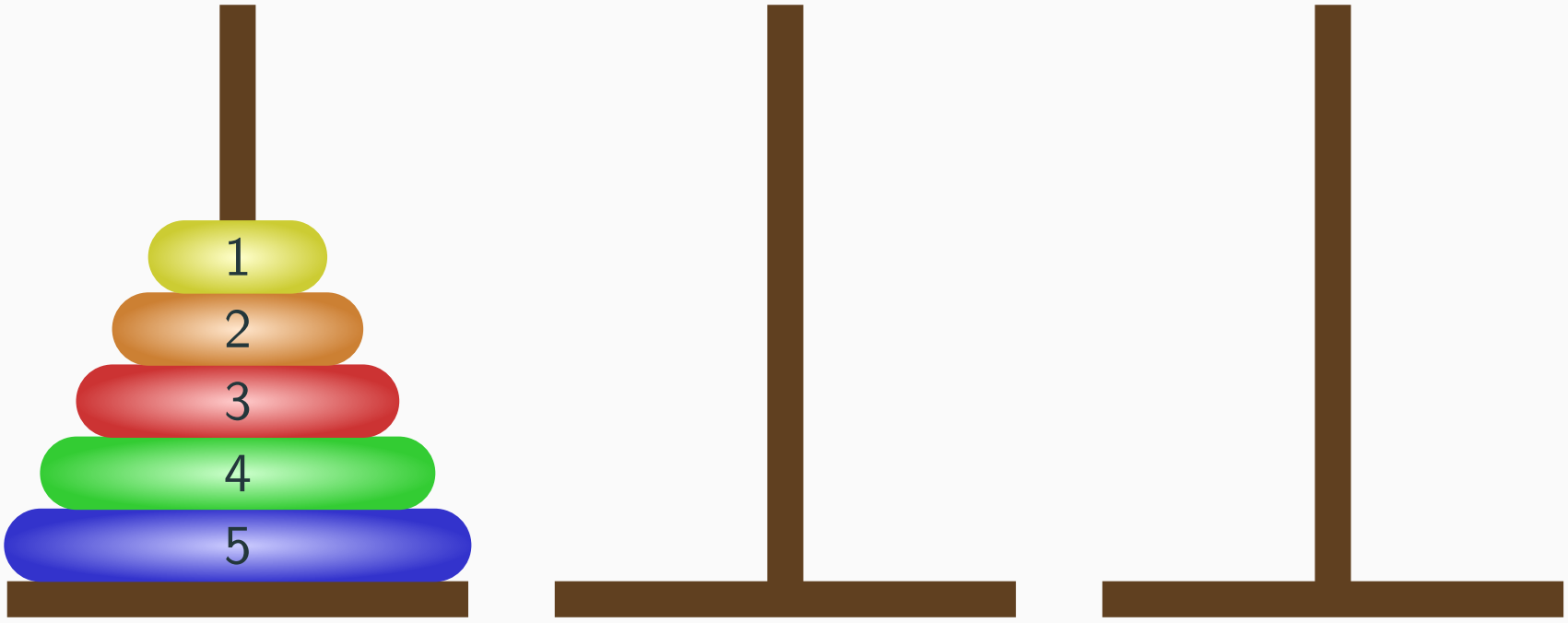
Les tours de Hanoï

Problème

On dispose de n disques troués en leur centre, numérotés de 1 à n, de diamètres croissants.

On se donne également 3 piquets.

Initialement, tous les disques sont enfilés sur le premier piquet, du plus grand au plus petit.

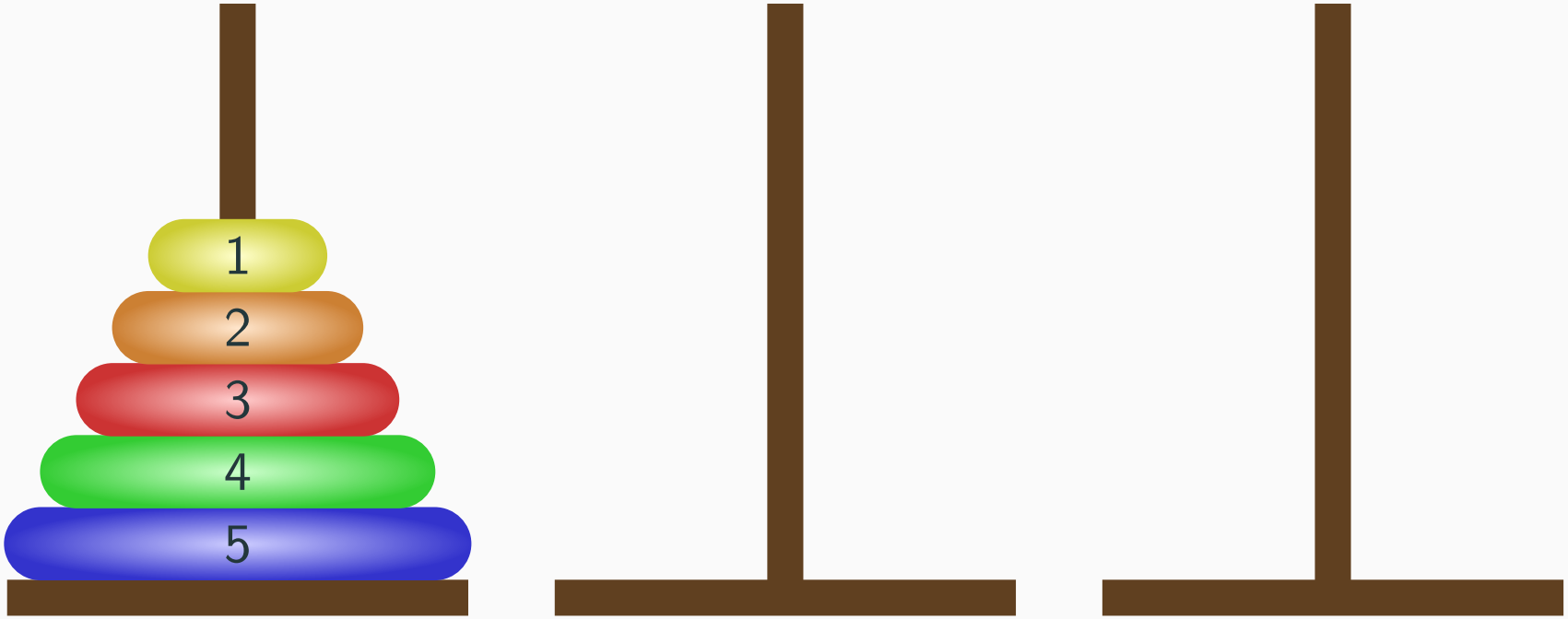


Les tours de Hanoï

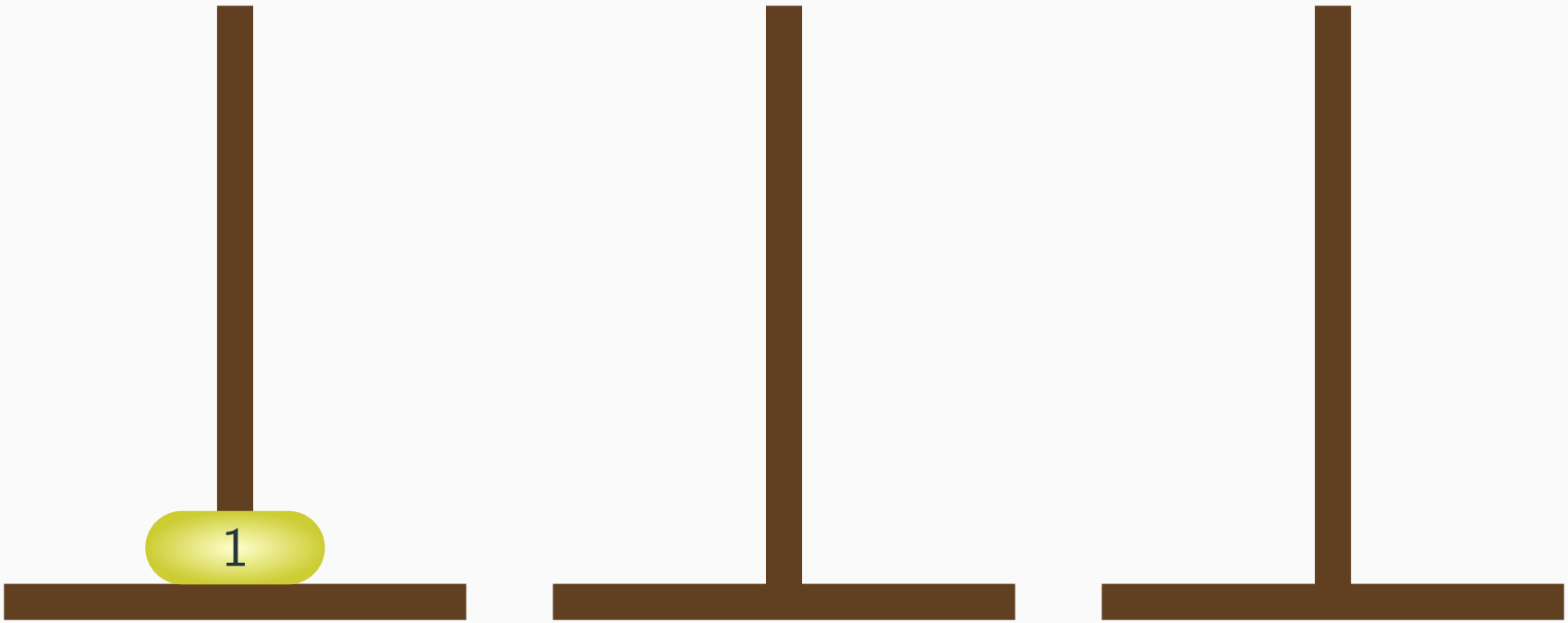
Problème

Le but du jeu est de déplacer les disques sur le troisième piquet, en suivant les règles suivantes :

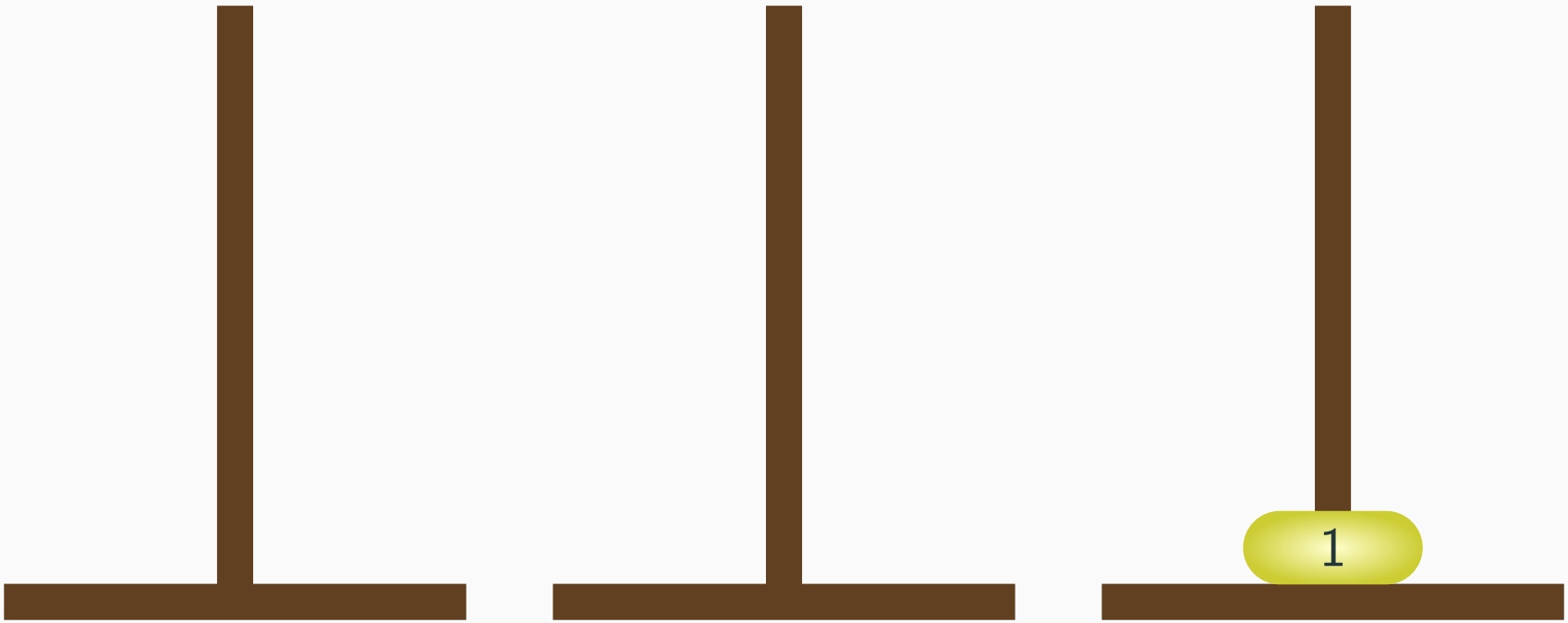
- déplacer les disques un par un, d'un piquet à un autre ;
- un disque ne doit jamais être posé sur un disque de diamètre inférieur.



Tour de Hanoï – 1 Disque



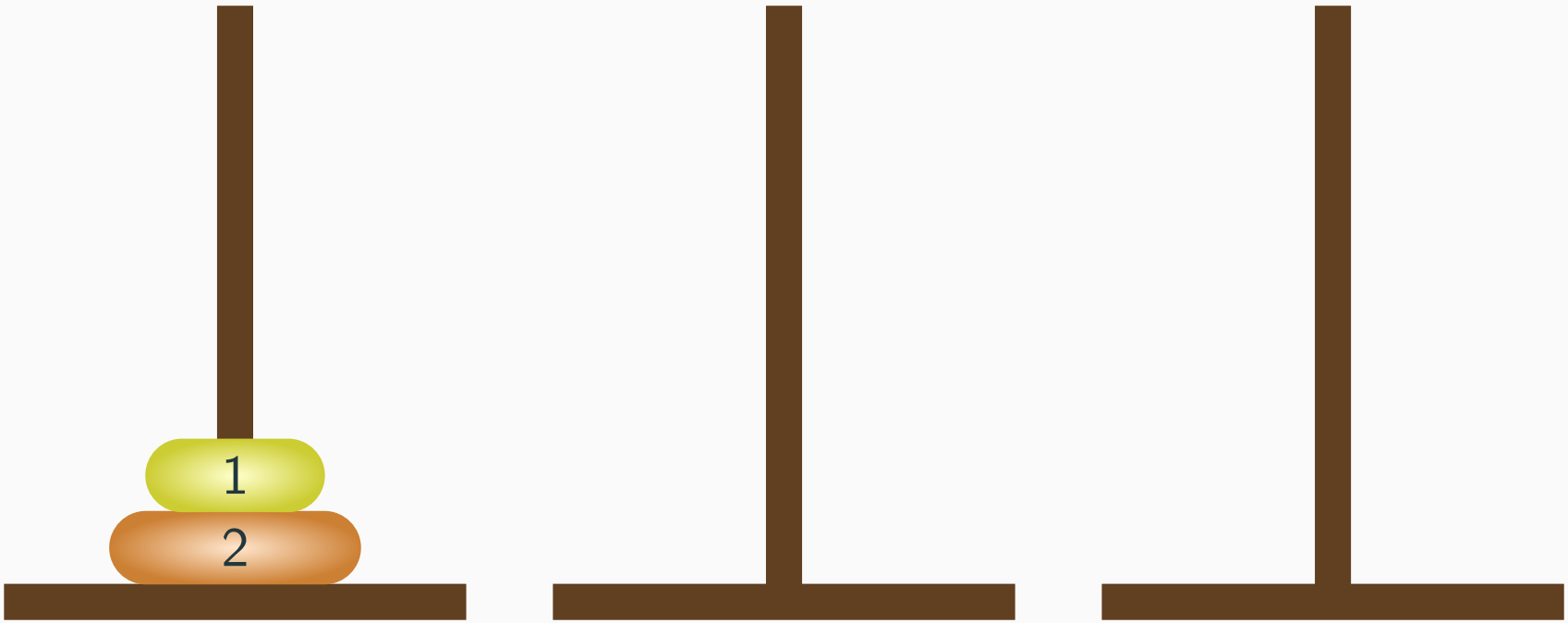
Tour de Hanoï – 1 Disque



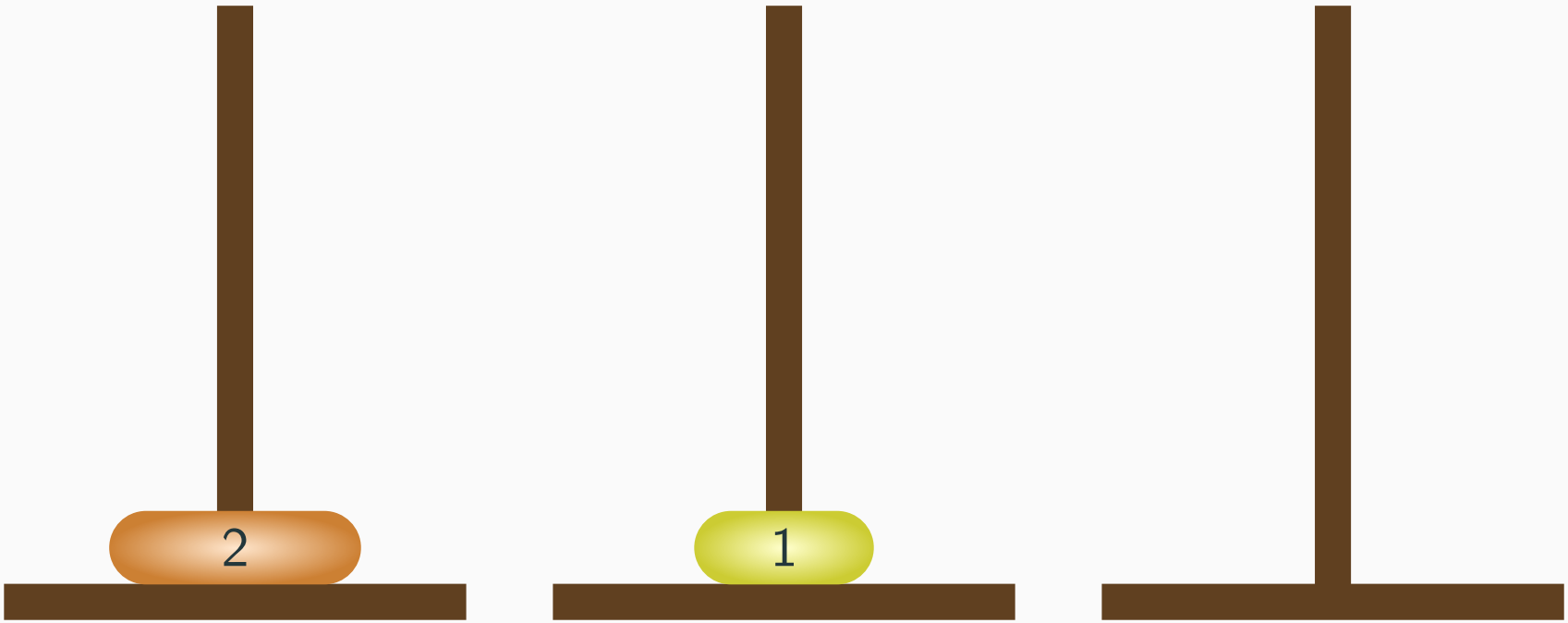
Déplacer le disque du piquet 1 au piquet 3.

Tour de Hanoï – 1 Disque

Tour de Hanoï – 2 Disques

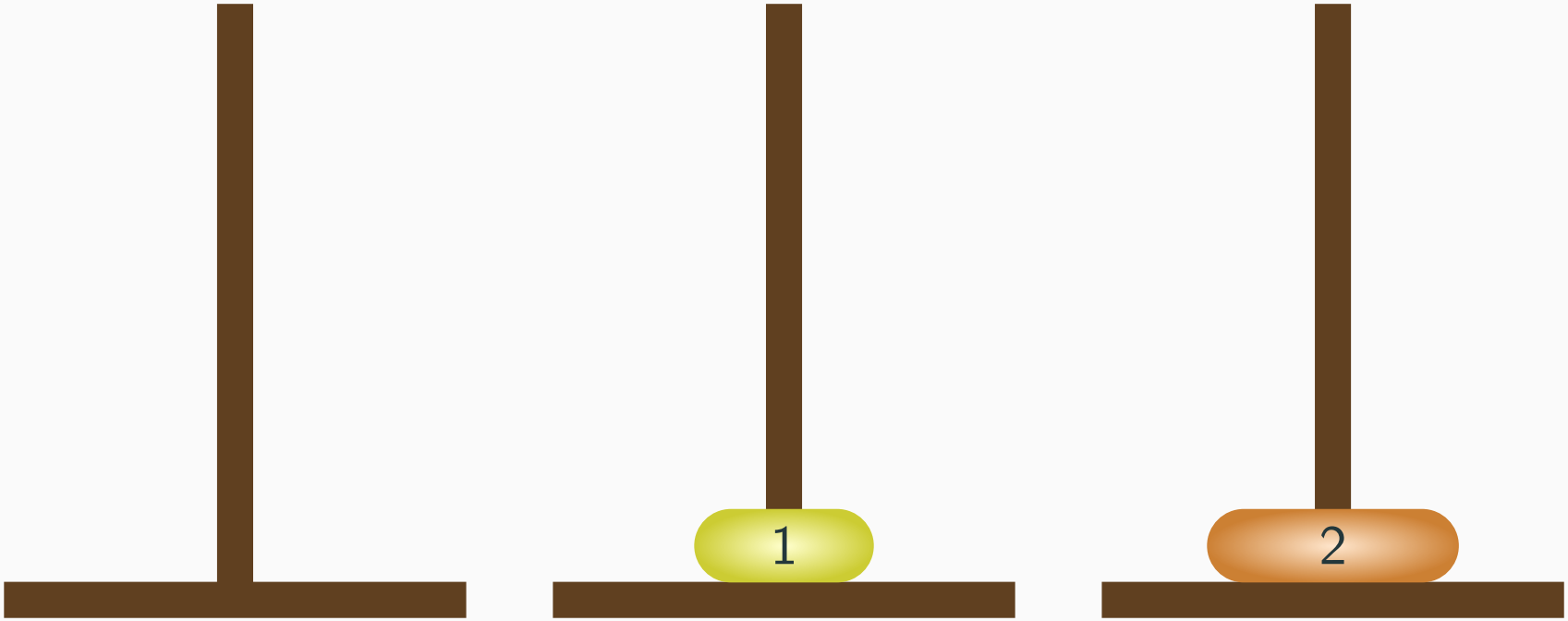


Tour de Hanoï – 2 Disques



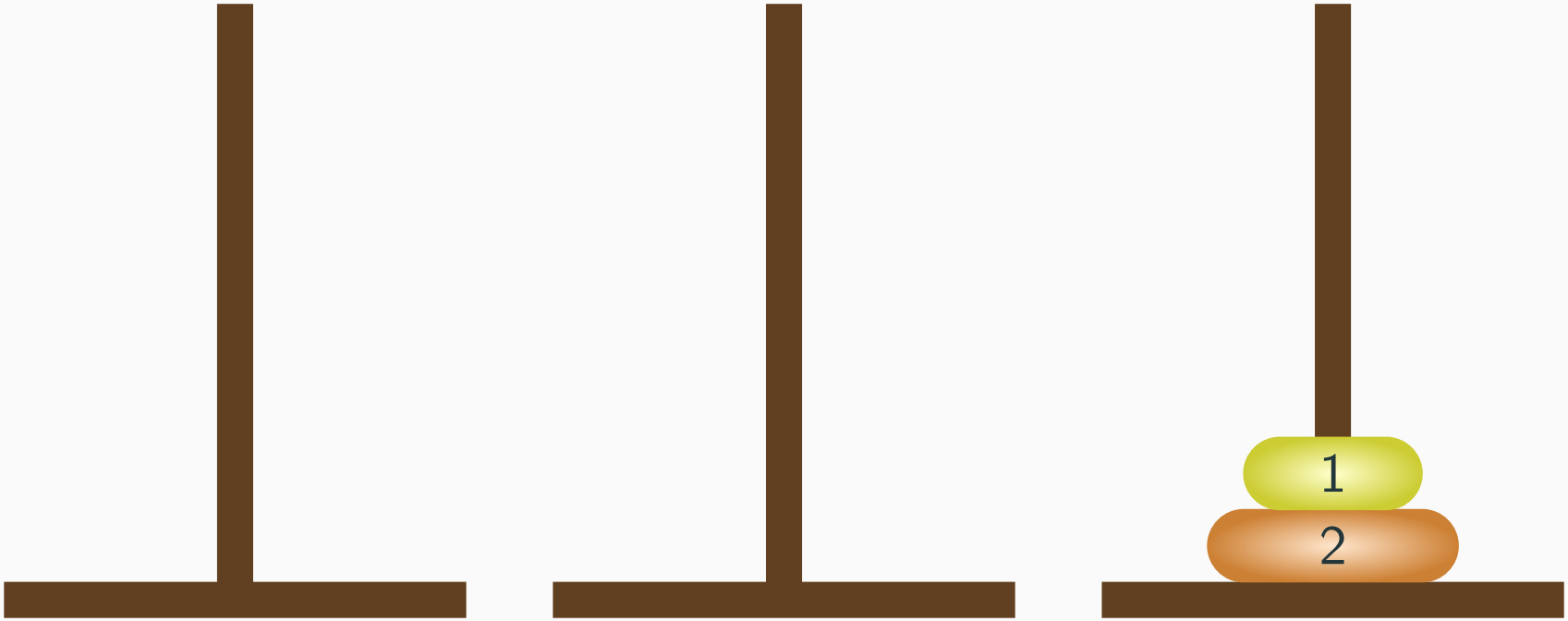
Déplacer le disque du piquet 1 au piquet 2.

Tour de Hanoï – 2 Disques



Déplacer le disque du piquet 1 au piquet 3.

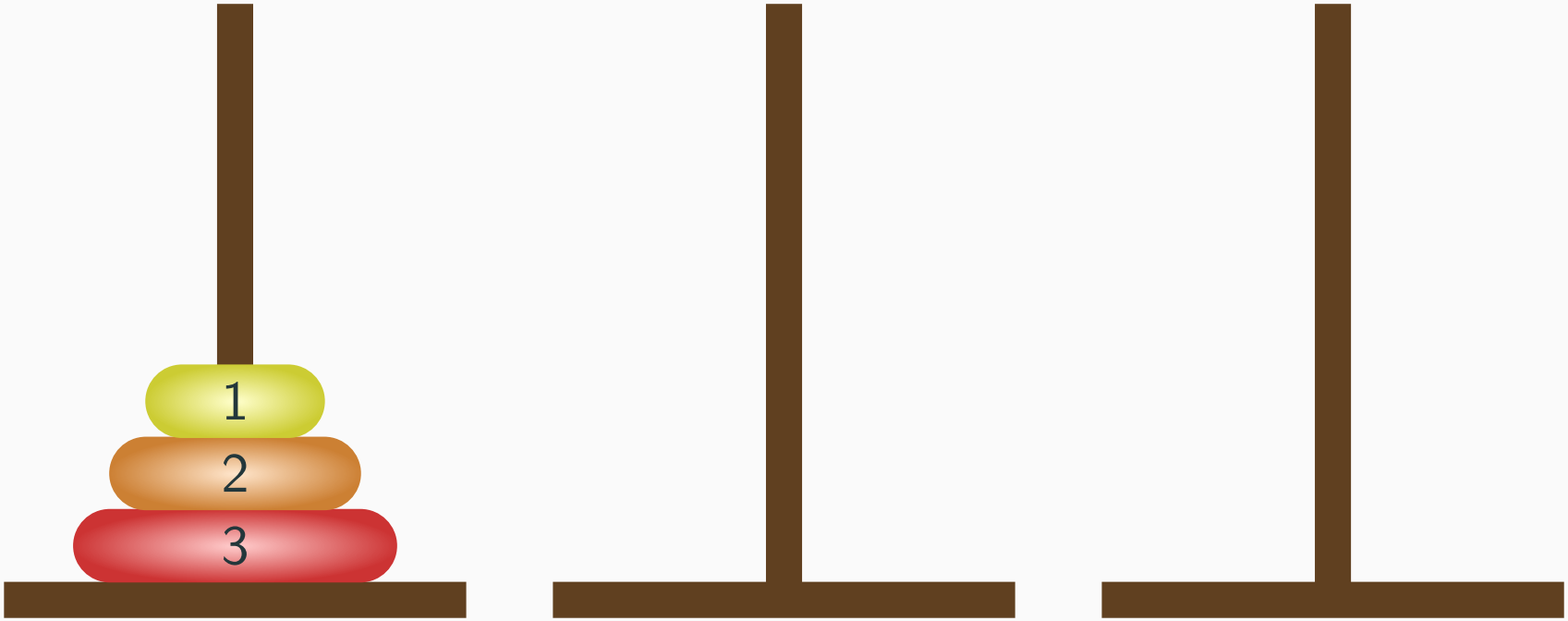
Tour de Hanoï – 2 Disques



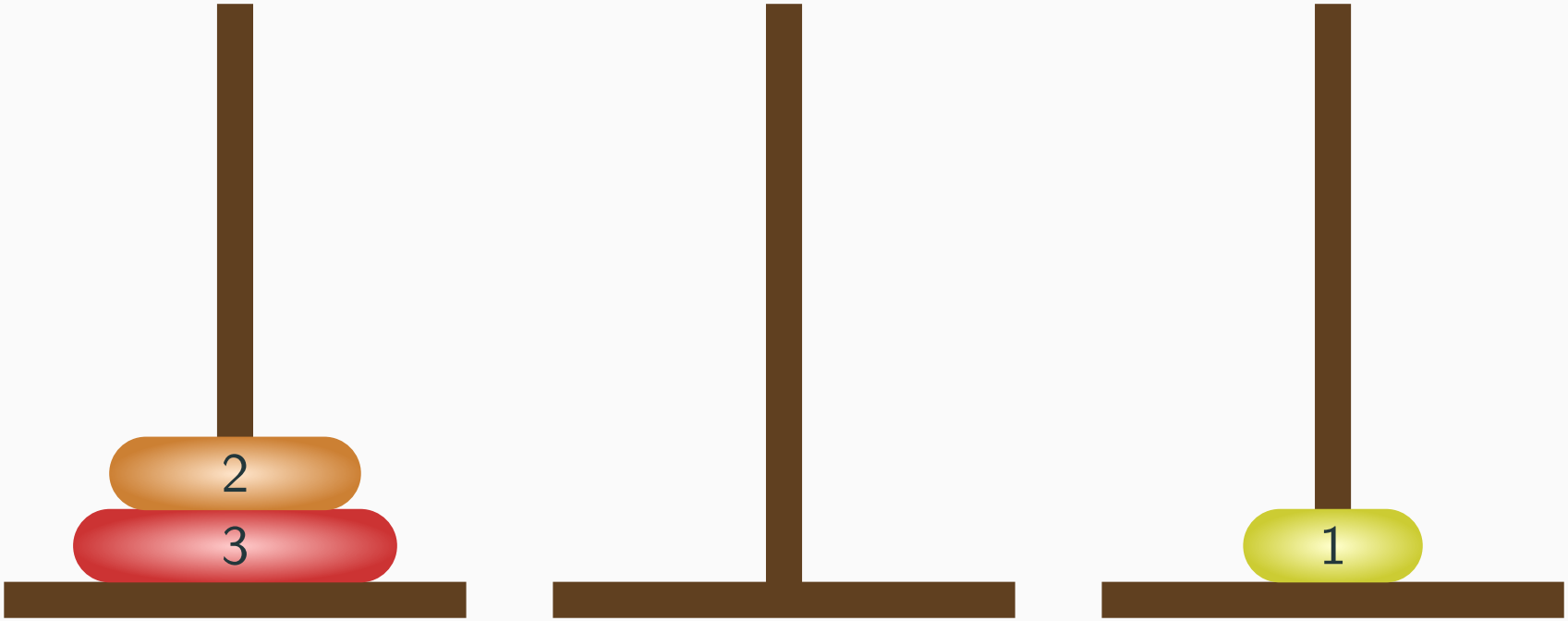
Déplacer le disque du piquet 2 au piquet 3.

Tour de Hanoï – 2 Disques

Tour de Hanoï – 3 Disques

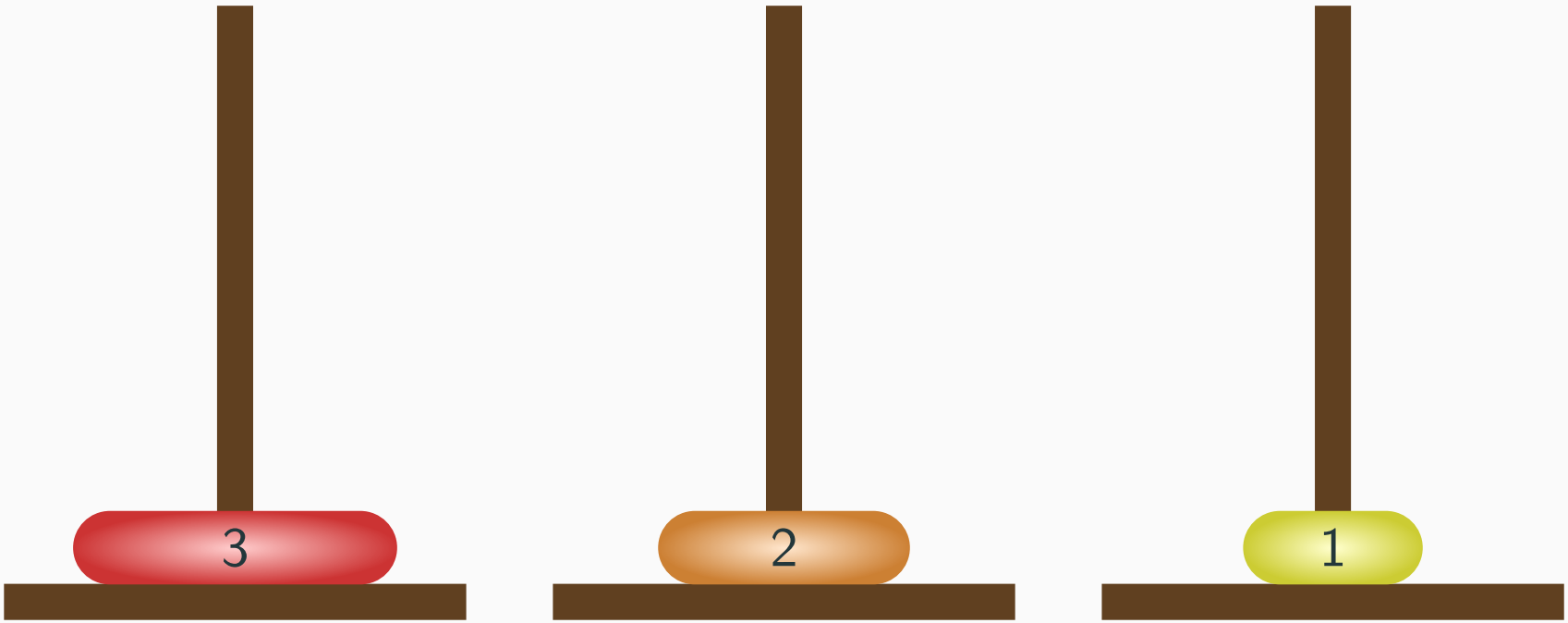


Tour de Hanoï – 3 Disques



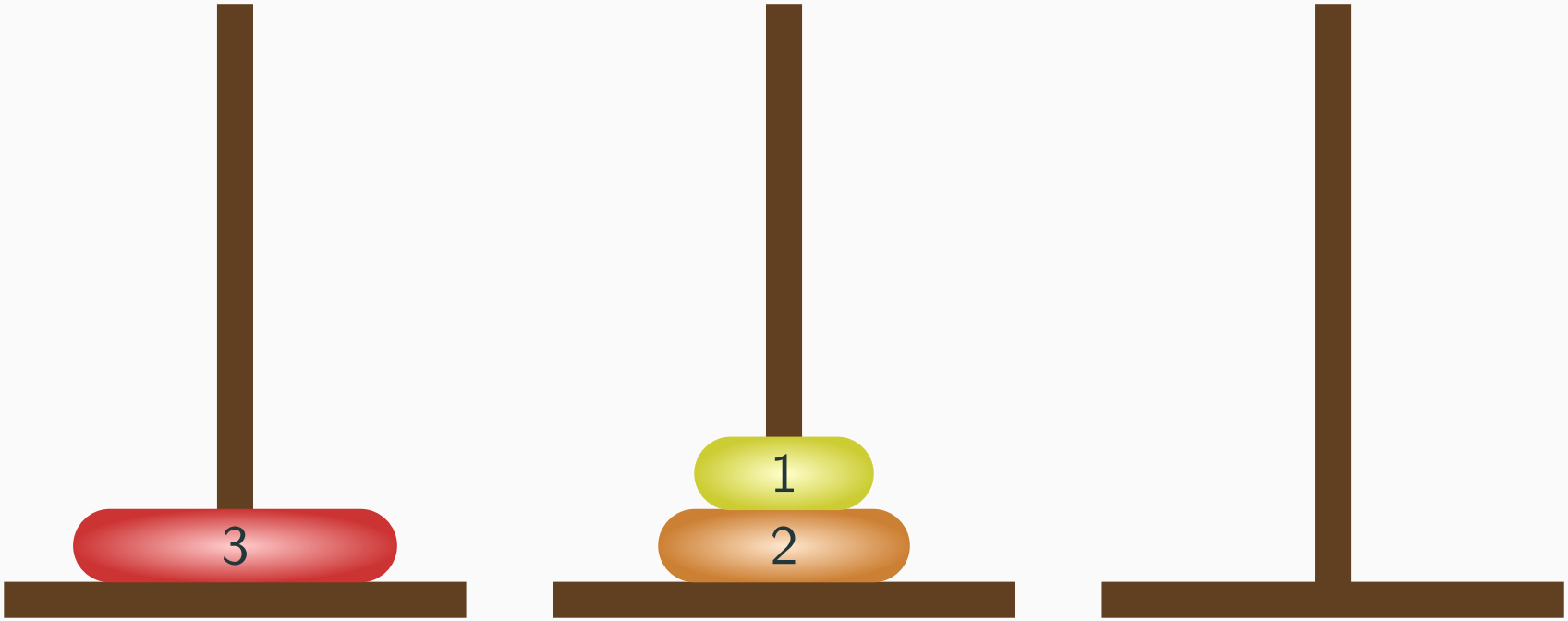
Déplacer le disque du piquet 1 au piquet 3.

Tour de Hanoï – 3 Disques



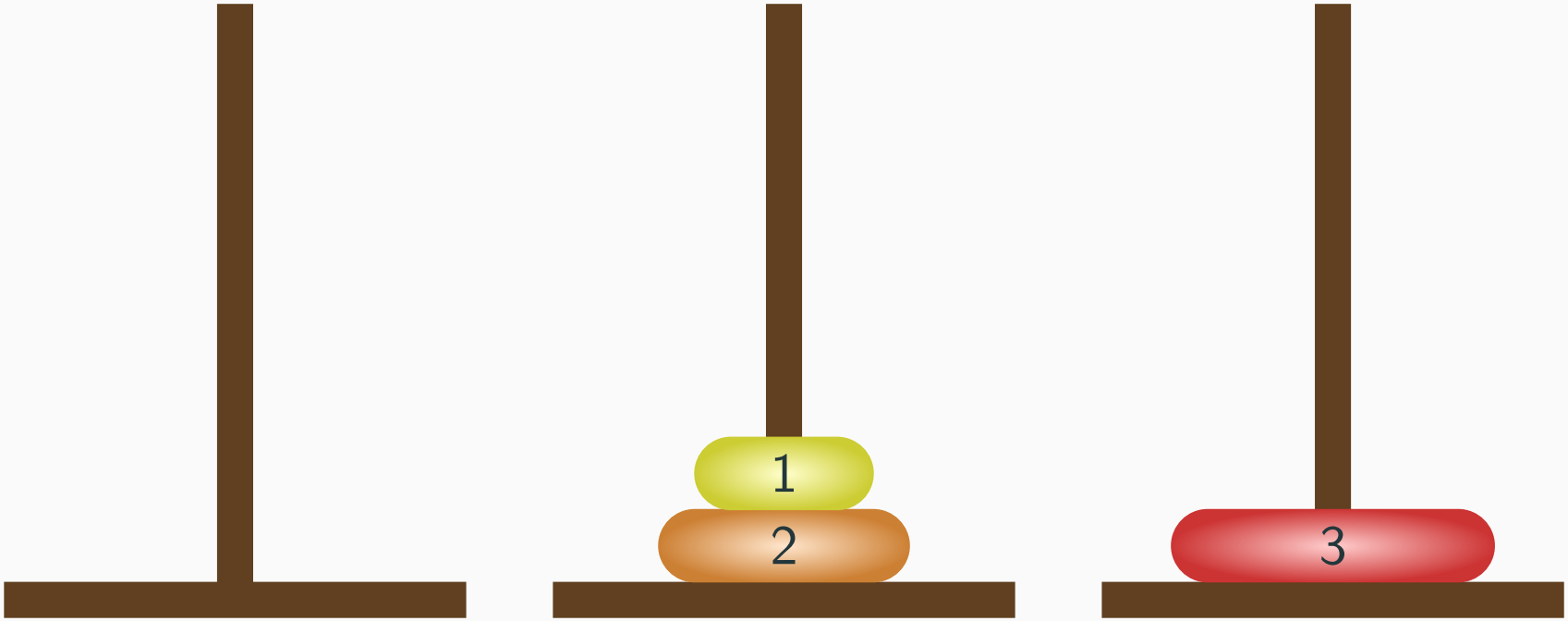
Déplacer le disque du piquet 1 au piquet 2.

Tour de Hanoï – 3 Disques



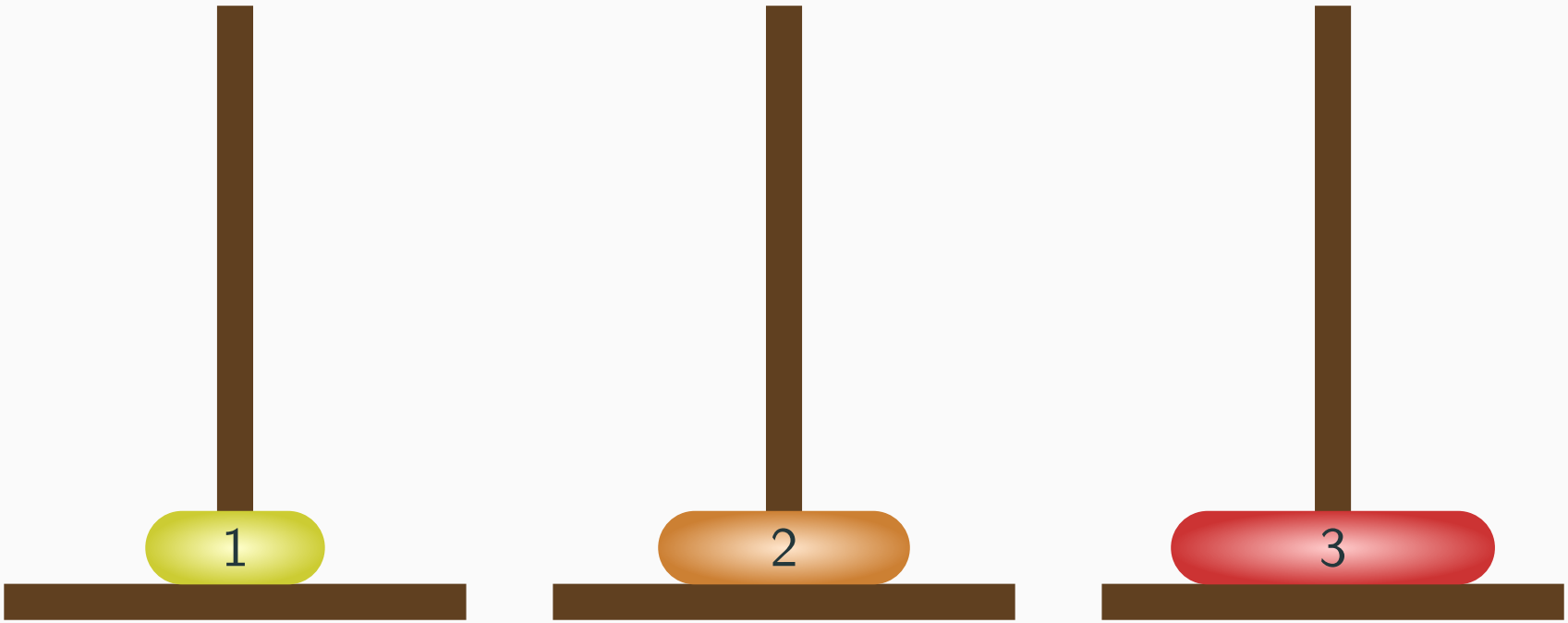
Déplacer le disque du piquet 3 au piquet 2.

Tour de Hanoï – 3 Disques



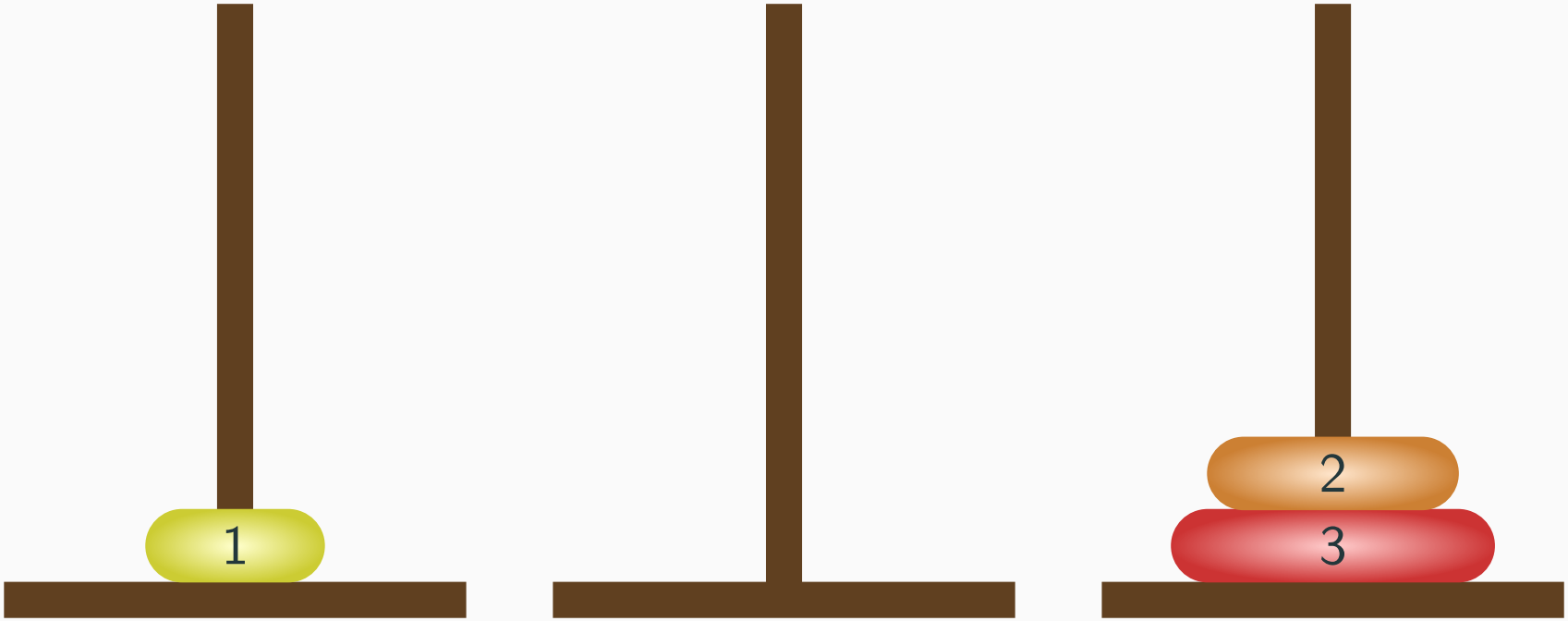
Déplacer le disque du piquet 1 au piquet 3.

Tour de Hanoï – 3 Disques



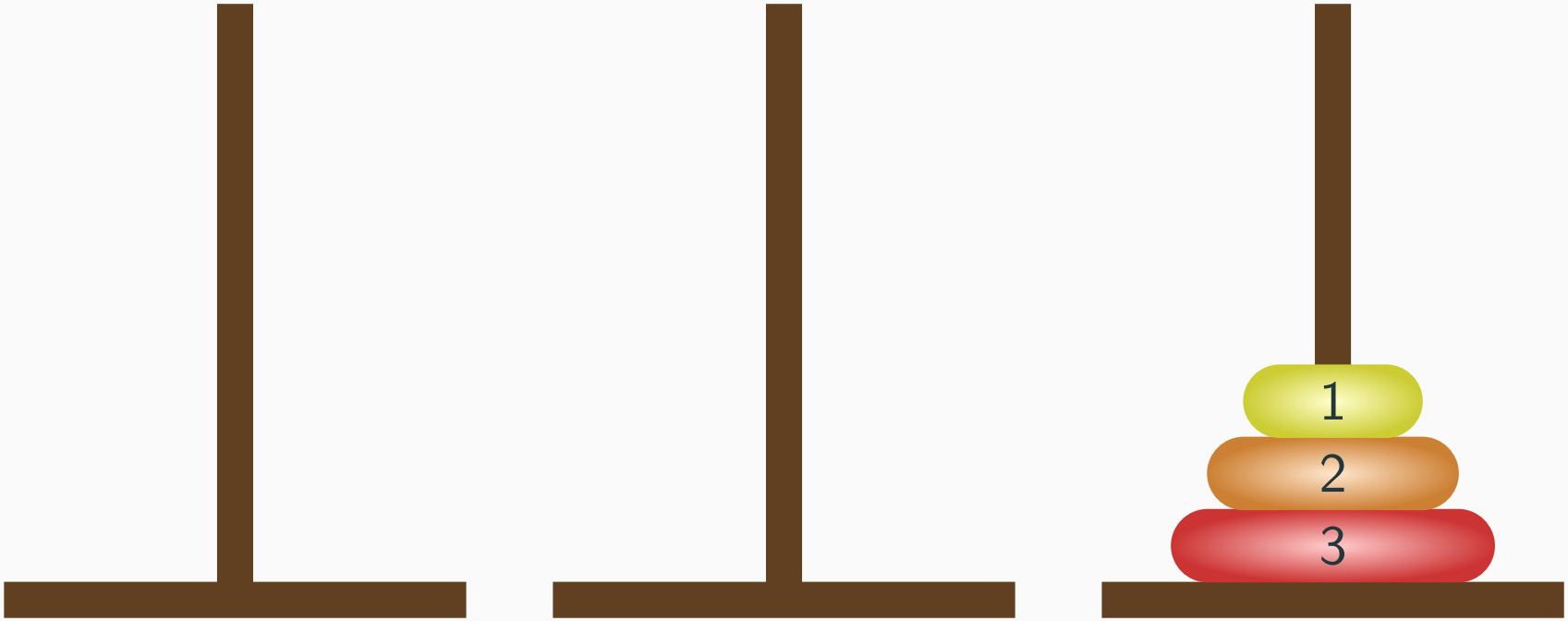
Déplacer le disque du piquet 2 au piquet 1.

Tour de Hanoï – 3 Disques



Déplacer le disque du piquet 2 au piquet 3.

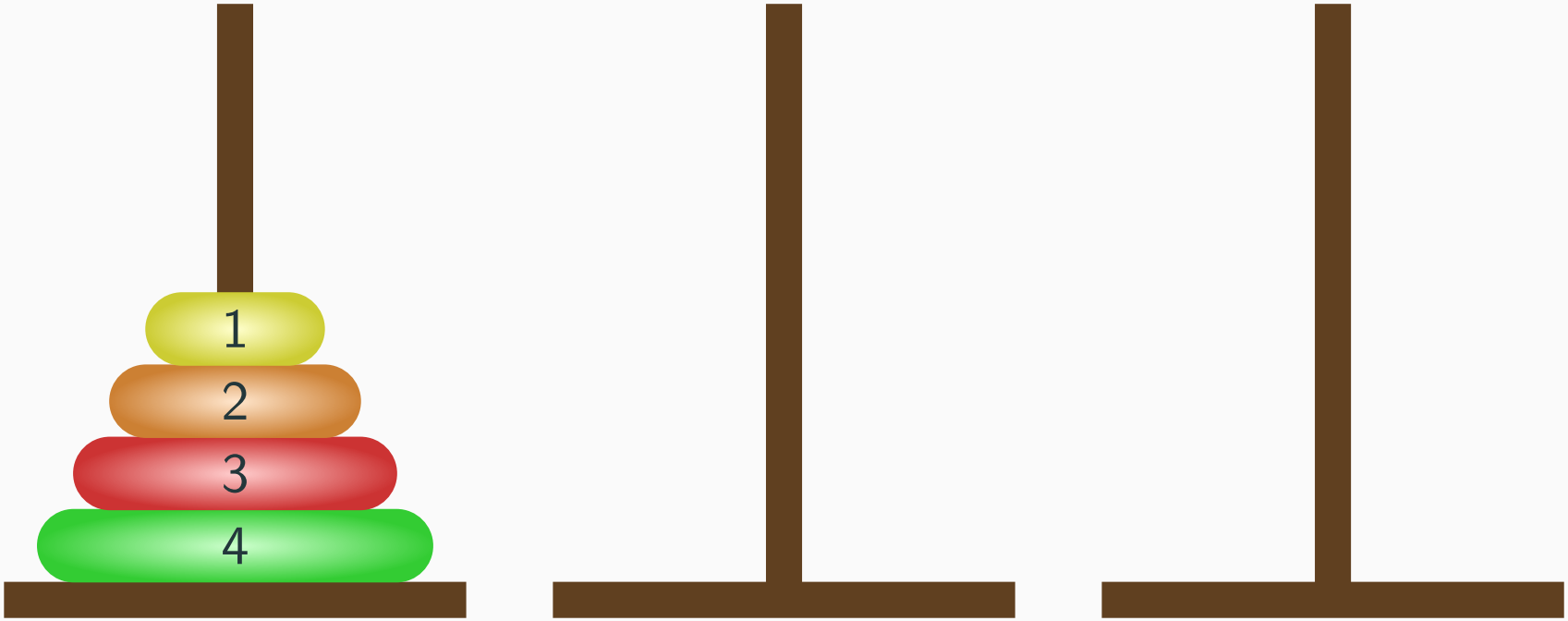
Tour de Hanoï – 3 Disques



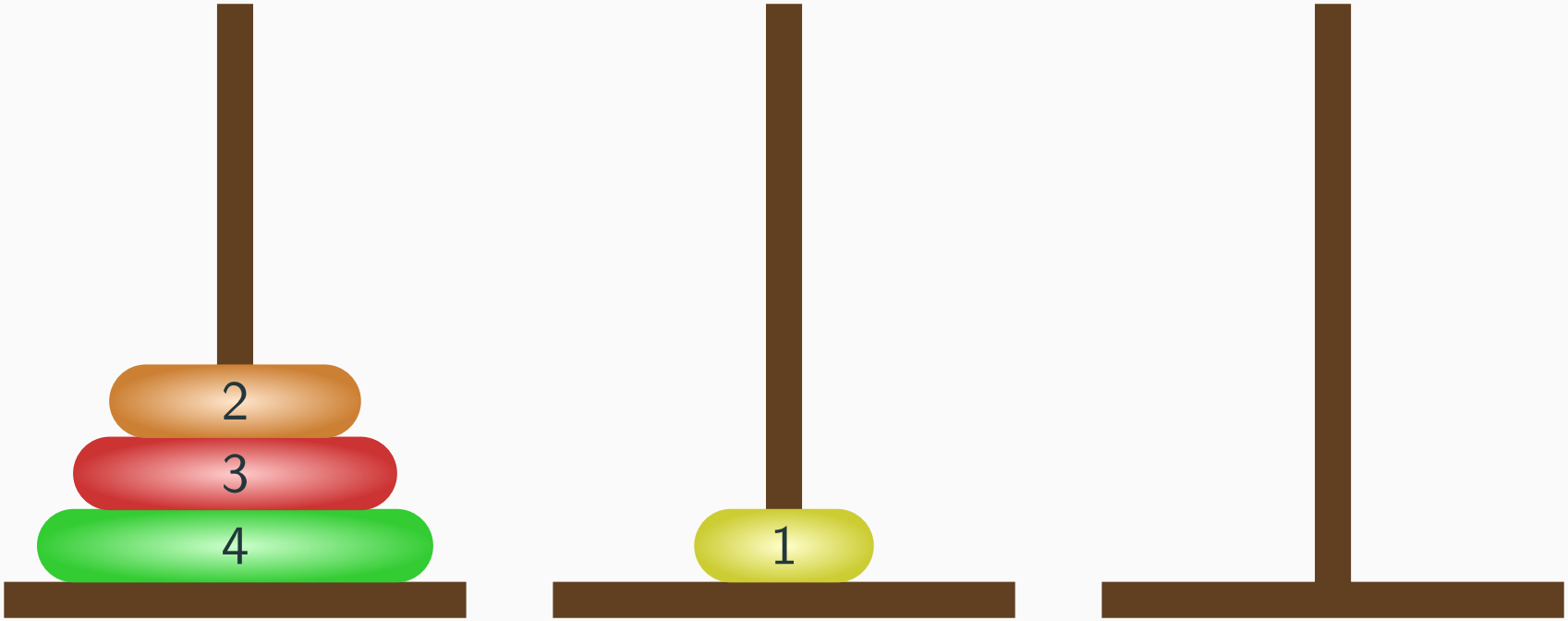
Déplacer le disque du piquet 1 au piquet 3.

Tour de Hanoï – 3 Disques

Tour de Hanoï – 4 Disques

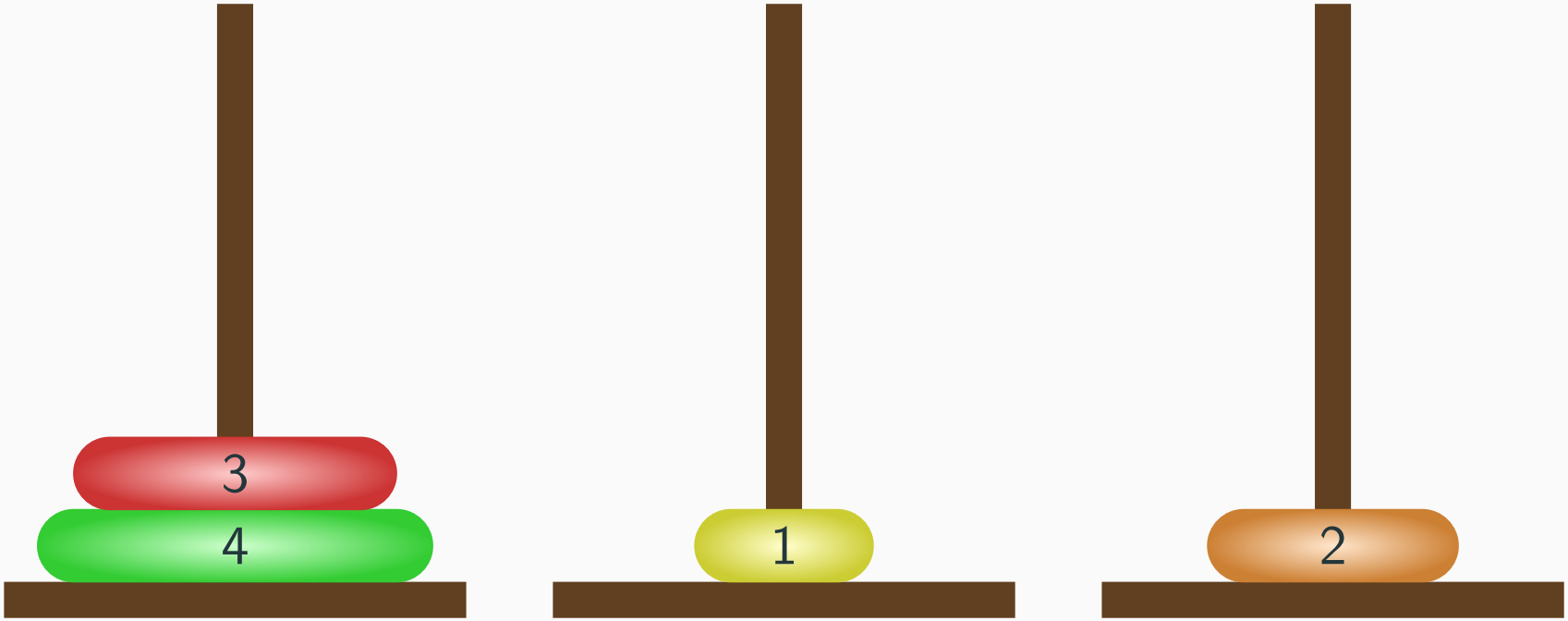


Tour de Hanoï – 4 Disques



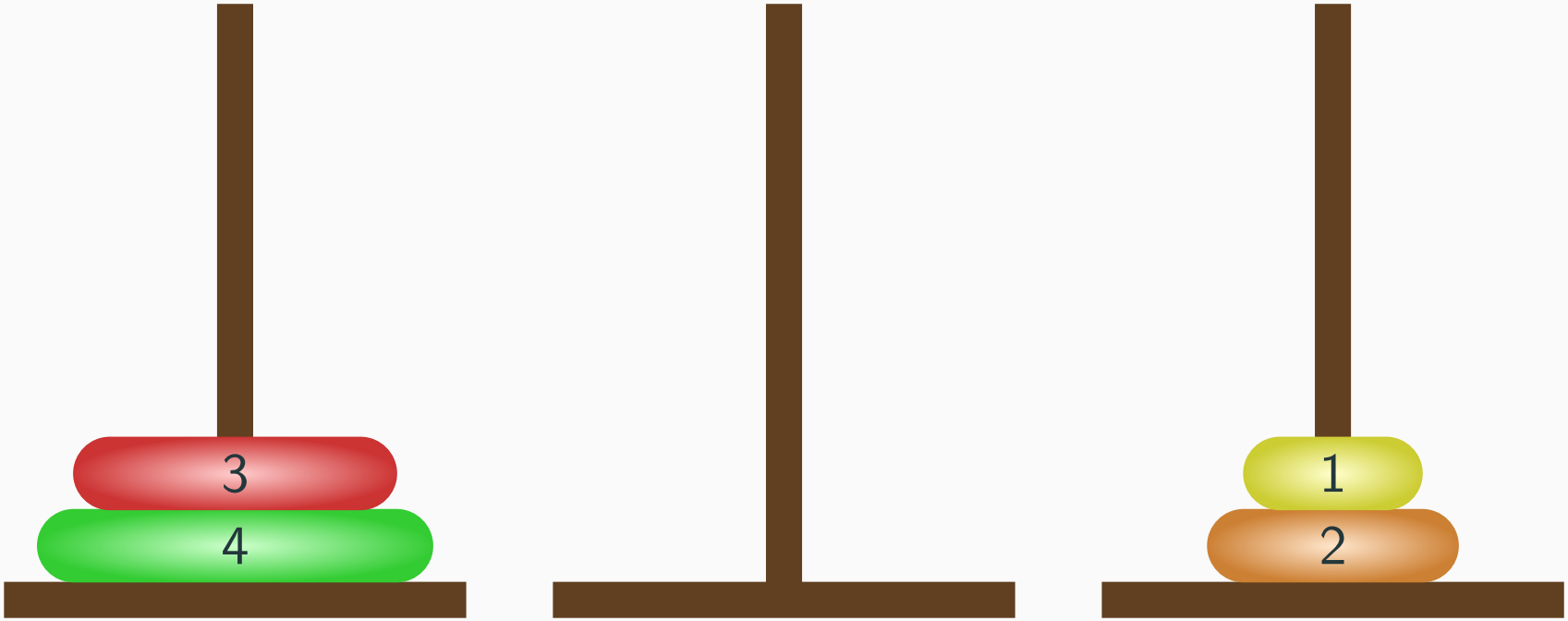
Déplacer le disque du piquet 1 au piquet 2.

Tour de Hanoï – 4 Disques



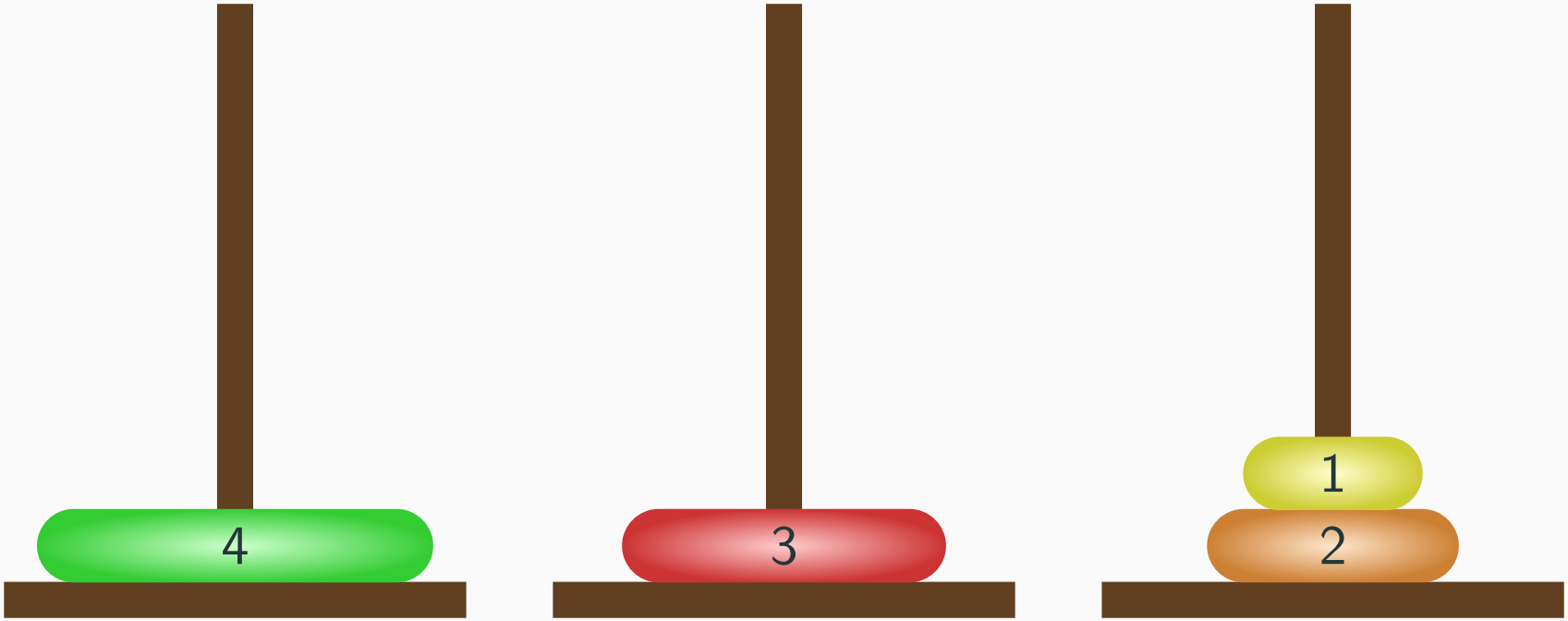
Déplacer le disque du piquet 1 au piquet 3.

Tour de Hanoï – 4 Disques



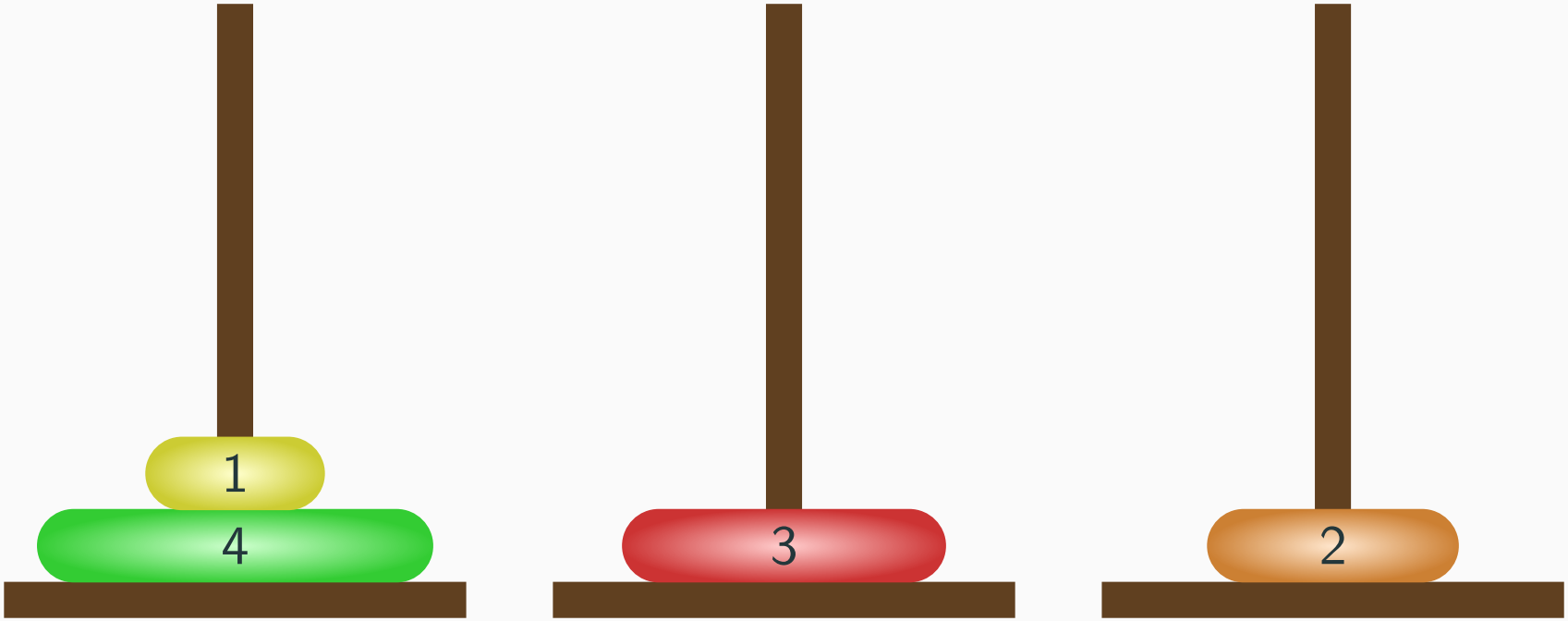
Déplacer le disque du piquet 2 au piquet 3.

Tour de Hanoï – 4 Disques



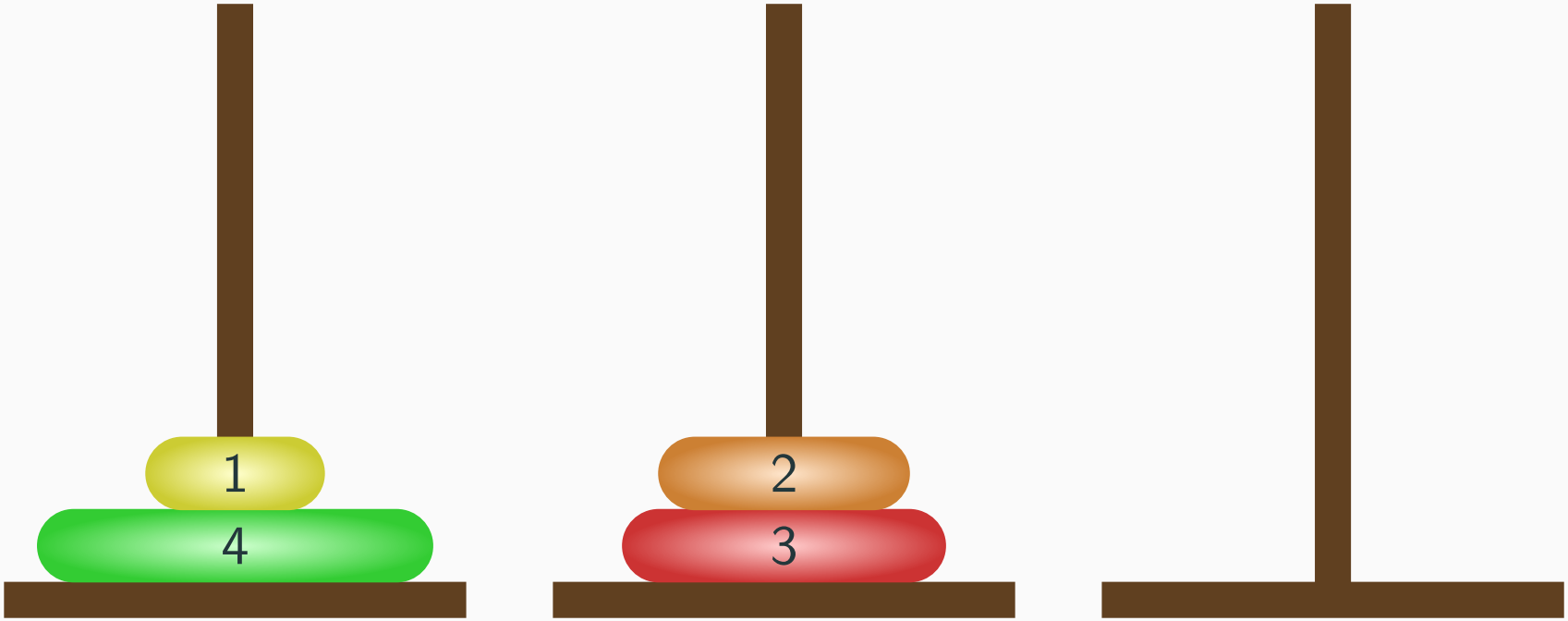
Déplacer le disque du piquet 1 au piquet 2.

Tour de Hanoï – 4 Disques



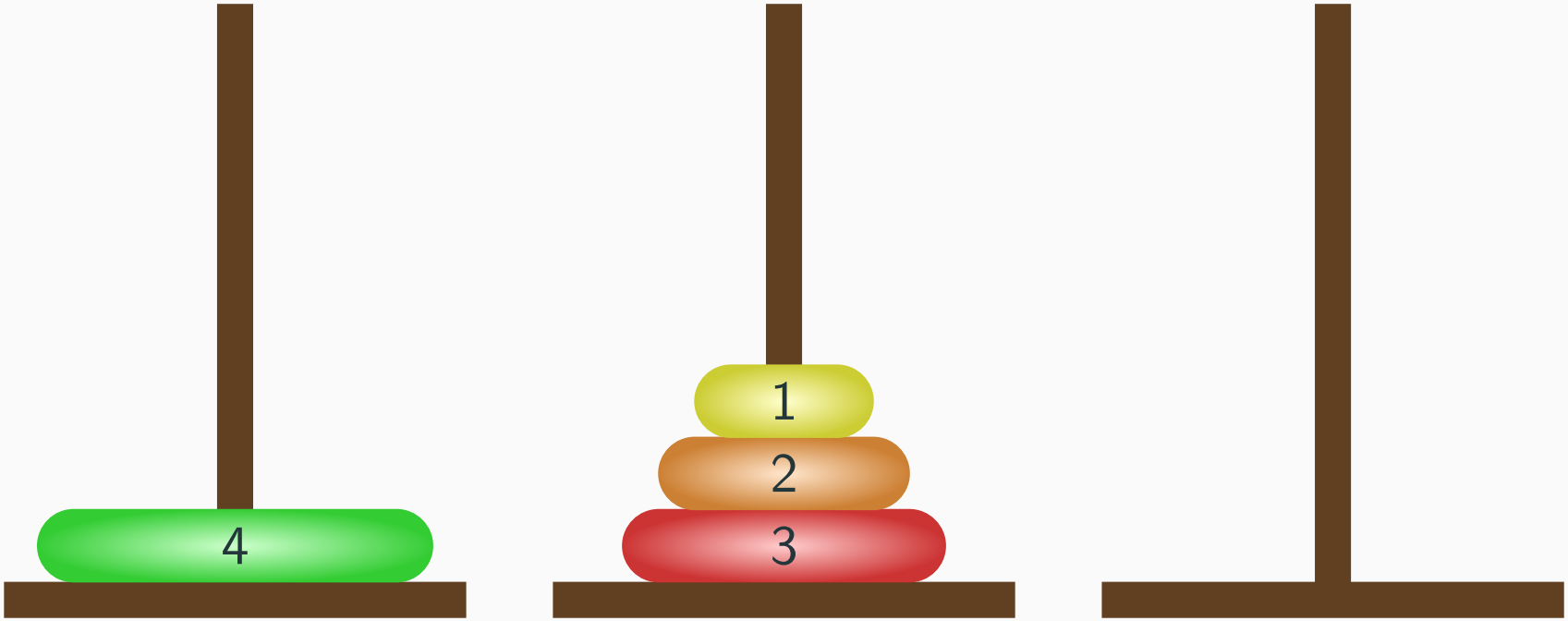
Déplacer le disque du piquet 3 au piquet 1.

Tour de Hanoï – 4 Disques



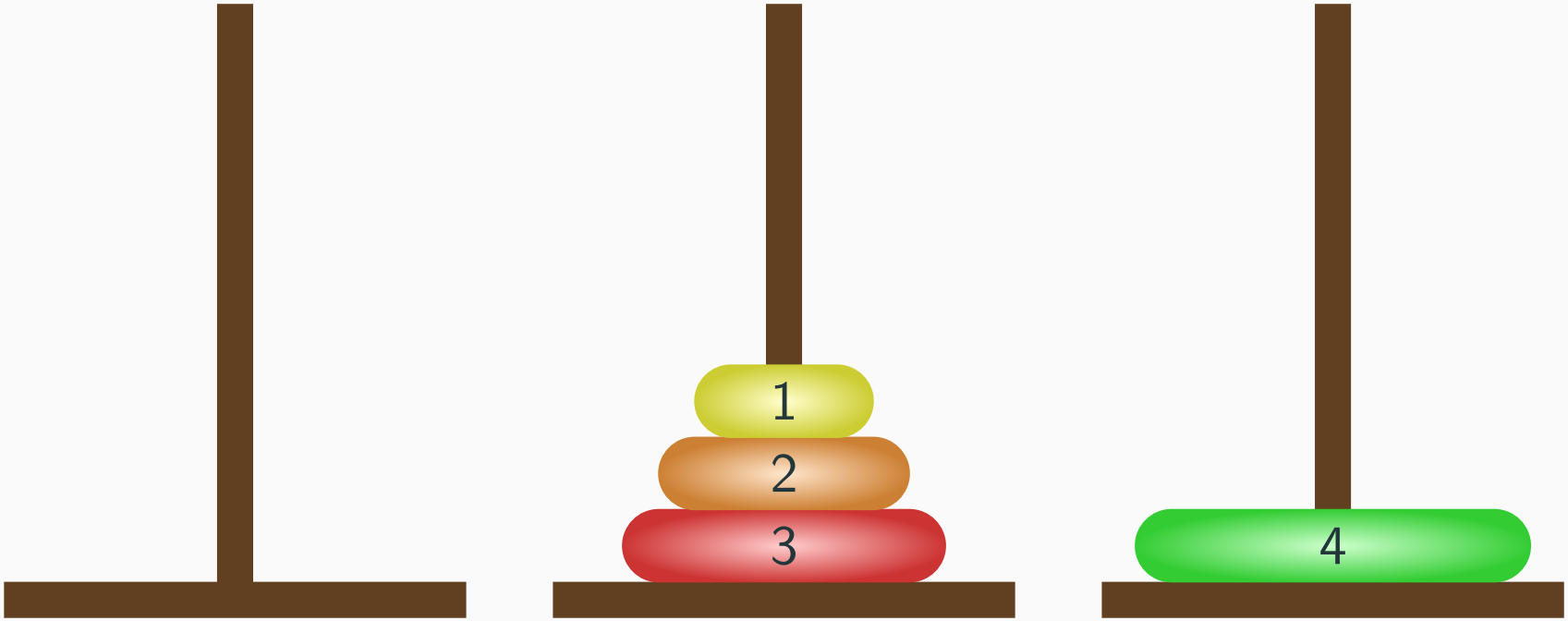
Déplacer le disque du piquet 3 au piquet 2.

Tour de Hanoï – 4 Disques



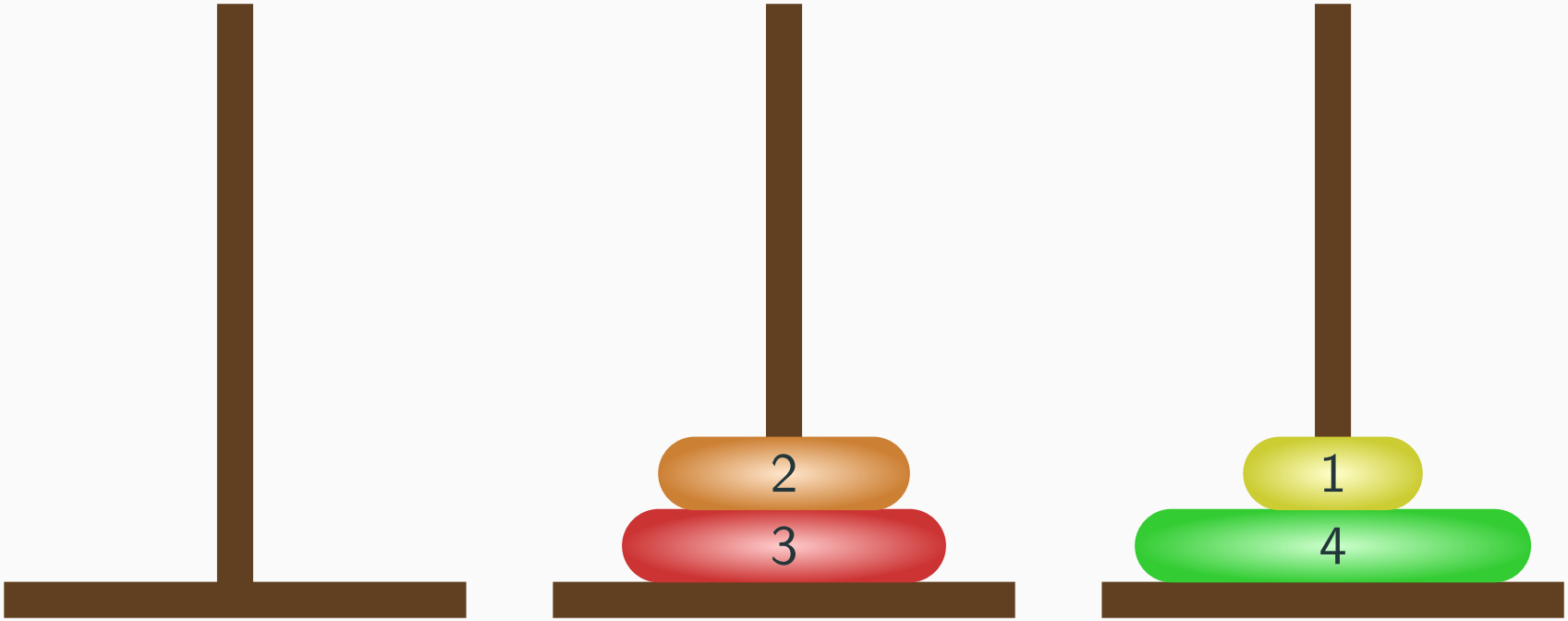
Déplacer le disque du piquet 1 au piquet 2.

Tour de Hanoï – 4 Disques



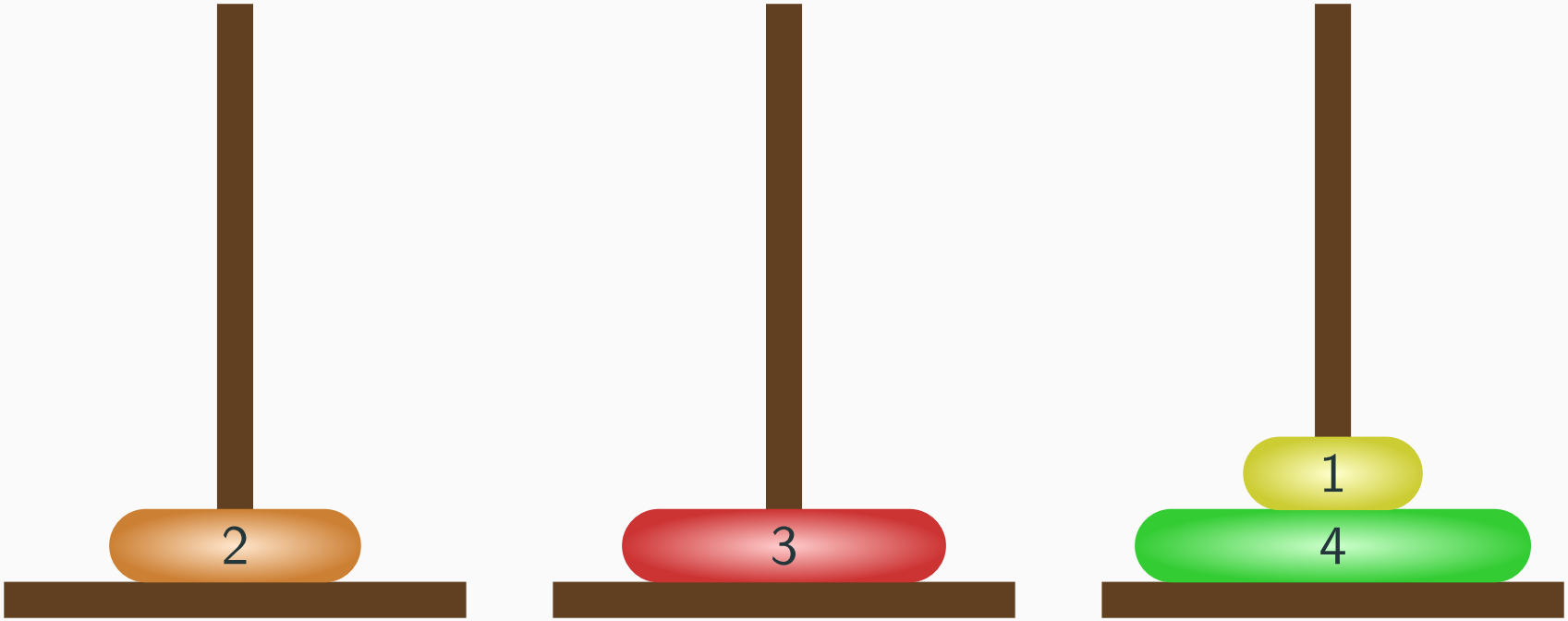
Déplacer le disque du piquet 1 au piquet 3.

Tour de Hanoï – 4 Disques



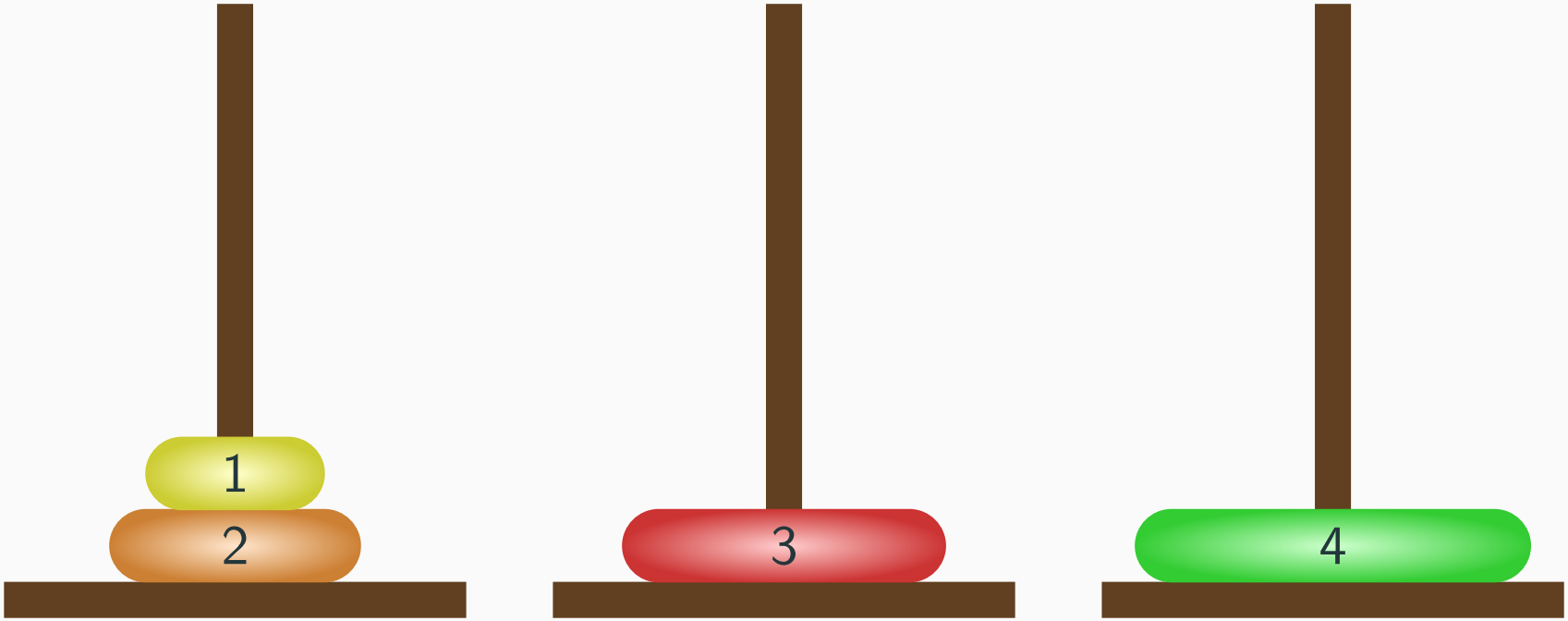
Déplacer le disque du piquet 2 au piquet 3.

Tour de Hanoï – 4 Disques



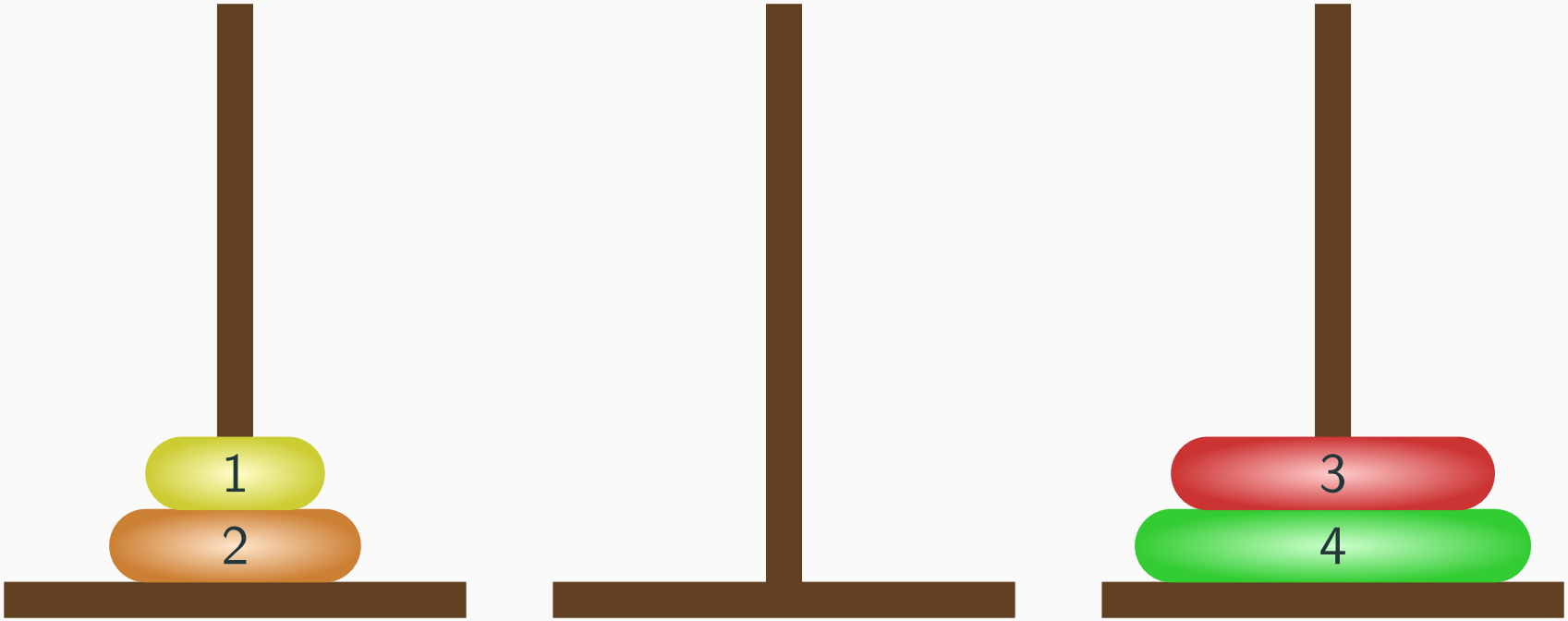
Déplacer le disque du piquet 2 au piquet 1.

Tour de Hanoï – 4 Disques



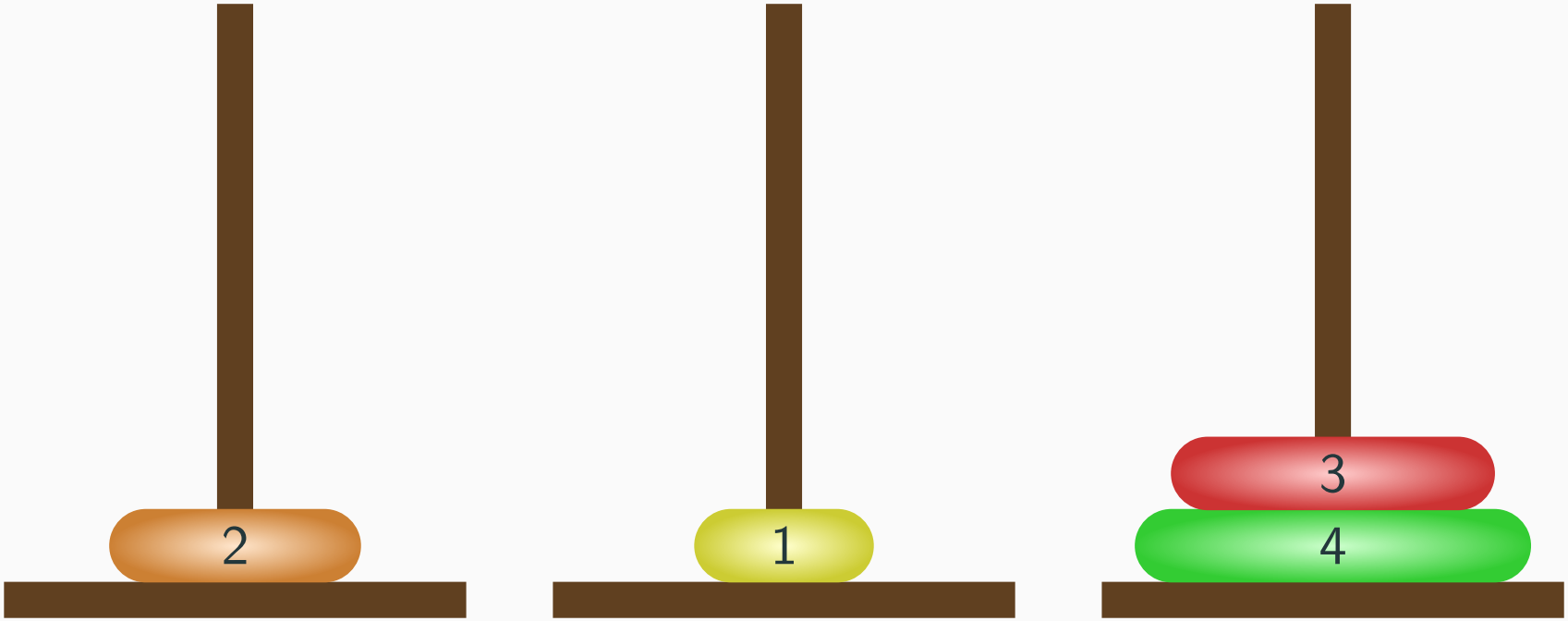
Déplacer le disque du piquet 3 au piquet 1.

Tour de Hanoï – 4 Disques



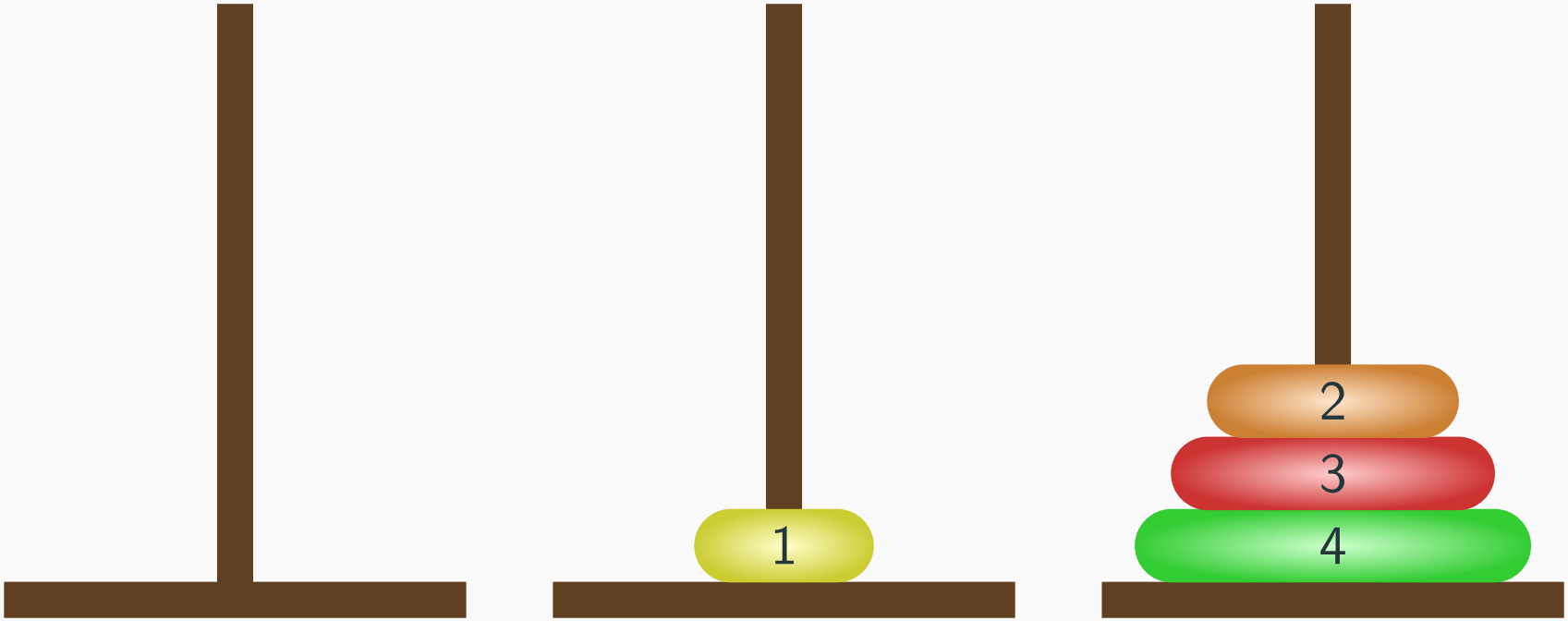
Déplacer le disque du piquet 2 au piquet 3.

Tour de Hanoï – 4 Disques



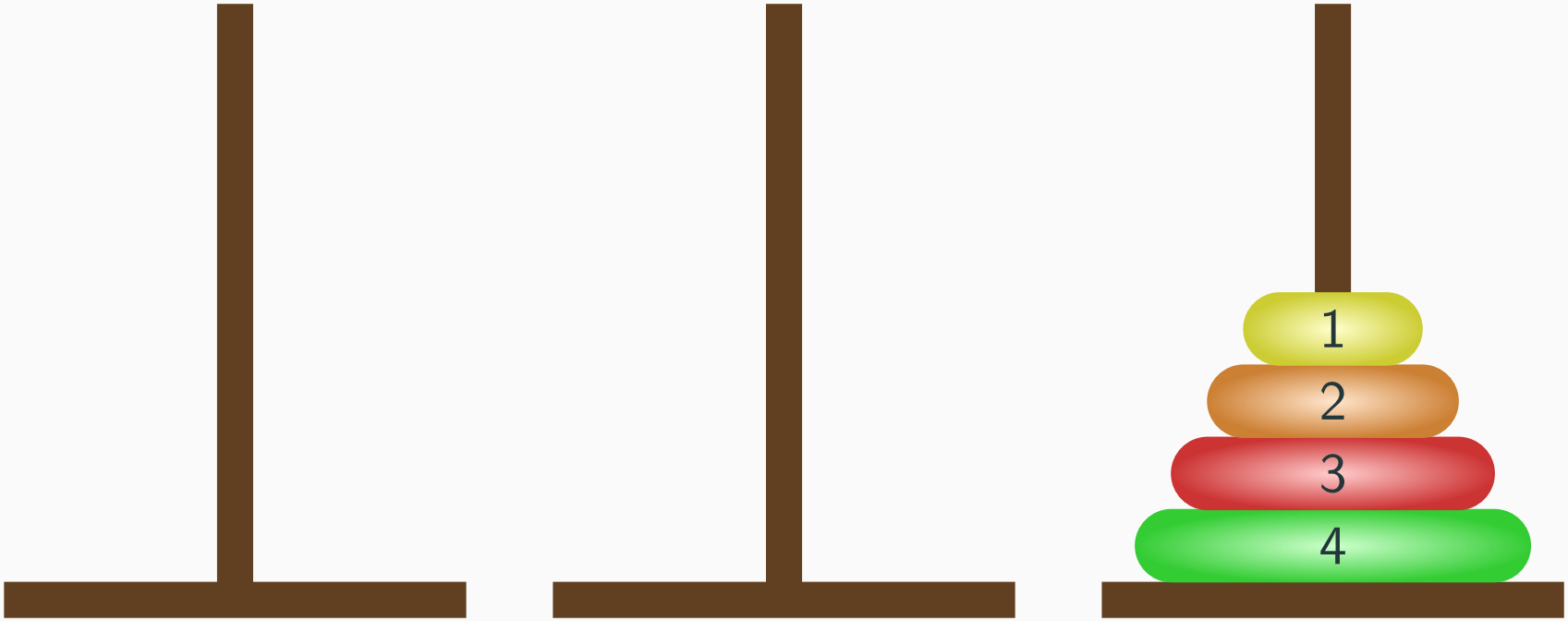
Déplacer le disque du piquet 1 au piquet 2.

Tour de Hanoï – 4 Disques



Déplacer le disque du piquet 1 au piquet 3.

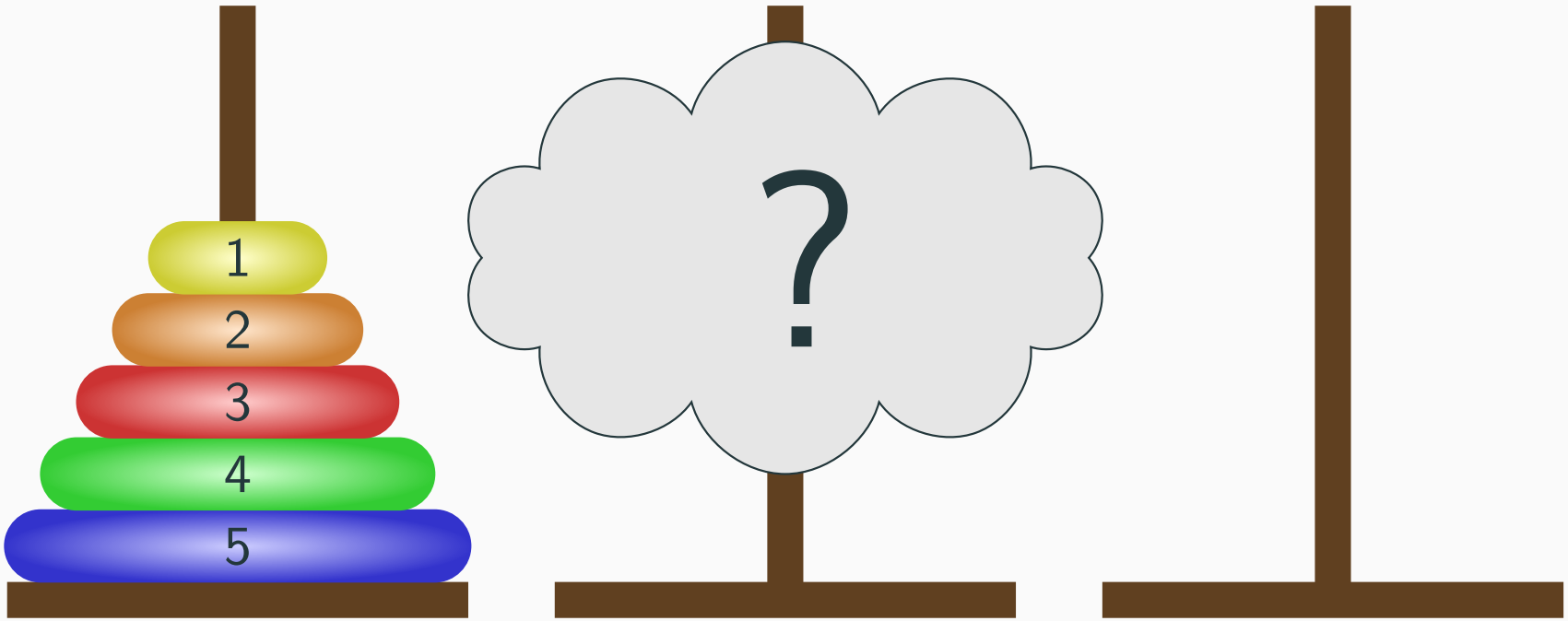
Tour de Hanoï – 4 Disques



Déplacer le disque du piquet 2 au piquet 3.

Tour de Hanoï – 4 Disques

Tour de Hanoï – 5 Disques



Tours de Hanoï

Recherche d'algorithme

On cherche un algorithme qui nous donne les mouvements de disques à effectuer pour résoudre le jeu, pour tout n.

Il n'est pas évident de résoudre ce problème de manière impérative.

En revanche, le problème devient très facile lorsqu'on pense à la récursivité.

Tours de Hanoï

Algorithme

Soit i, j, et k les numéros des piquets : $\{i, j, k\} = \{1, 2, 3\}$, et soit $n \in \mathbb{N}$.

Pour faire passer n disques du piquet i au piquet j :

- si $n = 0$, il n'y a rien à faire ;
- pour $n \geq 1$, il suffit de :
  - faire passer les $n - 1$ disques numérotés de 1 à $n - 1$ de i vers k ;
  - de déplacer le disque n de i vers j ;
  - de refaire passer les $n - 1$ disques de k vers j.

  Le fait de travailler avec des disques plus petits permet de respecter la deuxième règle.

Tours de Hanoï

```
let deplacement i j =
  print_string (i^" -> "^j^"\n")
;;
```

```
let hanoi n =
  let rec aux i j k n = match n with
    | 0 -> ()
    | _ -> aux i k j (n-1) ; deplacement i j ; aux k j i (n-1)
  in aux "1" "3" "2" n ;
;;
```

Exemple

```
# hanoi 3 ;;
1 -> 3
1 -> 2
3 -> 2
1 -> 3
2 -> 1
2 -> 3
1 -> 3
- : unit = ()
```

Tours de Hanoï

Solution optimale

Il est facile de montrer que le nombre de mouvements produits est $2^n - 1$.

C'est optimal.

Complexité

La fonction `hanoi` a une complexité en $O(2^n)$.

Conclusion

Comme on le voit sur cet exemple, les fonctions où plusieurs appels récursifs sont nécessaires ne sont pas vraiment faciles à traduire de façon itérative (à moins d'utiliser une pile pour essentiellement réécrire la récursivité).

C'est un avantage des fonctions récursives.

Structure de liste chaînée

Structure de liste chaînée

Types récursifs

La **liste chaînée** est l'exemple le plus simple de **type récursif**, ou **type défini par induction** : le type se retrouve dans la définition.

Caml est extrêmement pratique pour définir et manipuler ces types, comme on va le voir par la suite.

Naturellement, les fonctions qui manipulent des types récursifs sont le plus souvent récursives.

Structure de liste chaînée

Définition (liste chaînée)

Une **liste chaînée** d'éléments de type t est :

- soit la **liste vide** (souvent notée `[]`) ;
- soit la donnée d'un élément x de type t et d'une liste chaînée l d'éléments de type t, qu'on notera $\mathrm{Cons}(x, l)$.

Dans ce dernier cas, on appelle x la **tête** de la liste, et l la **queue** de la liste (qui est une liste).

Structure de liste chaînée

Remarque

Cette manière **récursive** de définir le type liste chaînée est essentiellement ce qu'on va faire en Caml.

Une définition équivalente (mais **inductive**) est la suivante :

Définition

L'ensemble des **listes chaînées** d'éléments de type t est le plus petit ensemble C tel que :

- la **liste vide `[]`** $\in C$.
- si x est de type t et $l \in C$, alors $\mathrm{Cons}(x, l) \in C$.

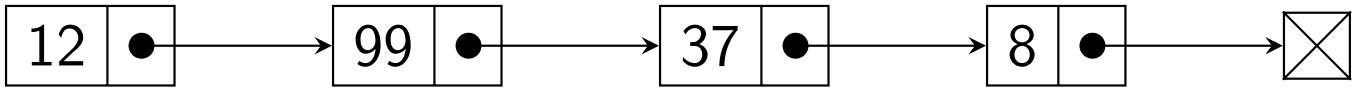
Structure de liste chaînée

Exemple

Voici une liste constituée des éléments 12, 99, 37 et 8, à savoir :

$$\mathrm{Cons}(12, \mathrm{Cons}(99, \mathrm{Cons}(37, \mathrm{Cons}(8, \texttt{[]}))))$$

Elle peut être représentée par le schéma suivant :



Le type `list` en Caml

Le type `list`

En Caml, les **listes chaînées** sont implémentées par le type `list`.

On peut définir une liste en donnant la séquence des éléments de la liste, entre `[]`, et séparés par des points-virgules.

Le type `list` en Caml

```
# [12; 99; 37; 8] ;;
- : int list = [12; 99; 37; 8]
```

Exemple

Voici la liste précédente en Caml.

On remarque que le type est **`int list`**.

Les listes chaînées en Caml sont **homogènes**, comme les tableaux (tous les éléments d'une liste sont du même type).

Le type `list` en Caml

Cons

L'opérateur Cons pour construire des listes chaînées existe en Caml : c'est l'opérateur `::` qui est un opérateur **infixe** (il s'utilise comme **+** par exemple).

`x::q` donne une `'a list` si `x` est de type `'a` et `q` est de type `'a list`.

La version préfixe permet de de se convaincre de ces types :

```
1  # let pref_cons (x,q) = x::q ;;
2  val pref_cons : 'a * 'a list -> 'a list = <fun>
```

Le type `list` en Caml

Exemple

On peut ajouter 12 à la liste $[99; 37; 8]$ comme ceci :

```
# 12::[99; 37; 8] ;;
- : int list = [12; 99; 37; 8]
```

Le type `list` en Caml

Tête et queue

Inversement, à partir d'une liste non vide, on peut accéder à sa **tête** et à sa **queue** via les fonctions **`hd`** (head) et **`tl`** (tail) du module **`List`**.

```
# List.hd ;;
- : 'a list -> 'a = <fun>
# List.tl ;;
- : 'a list -> 'a list = <fun>
# let q=["a"; "b"; "c"] ;;
val q : string list = ["a"; "b"; "c"]
# List.hd q ;;
- : string = "a"
# List.tl q ;;
- : string list = ["b"; "c"]
```

Le type `list` en Caml

Filtrage

Ceci dit, les fonctions **`hd`** et **`tl`** sont rarement utilisées, on préfère fonctionner par **filtrage** sur les listes.

L'opérateur **`::`** peut être utilisé comme motif de filtrage, voici comment recoder **`hd`** et **`tl`** :

```
1  let tete l = match l with
2    | [] -> failwith "liste vide"
3    | x::_ -> x
4  ;;
5
6  let queue l = match l with
7    | [] -> failwith "liste vide"
8    | _::q -> q
9  ;;
```

Le type `list` en Caml

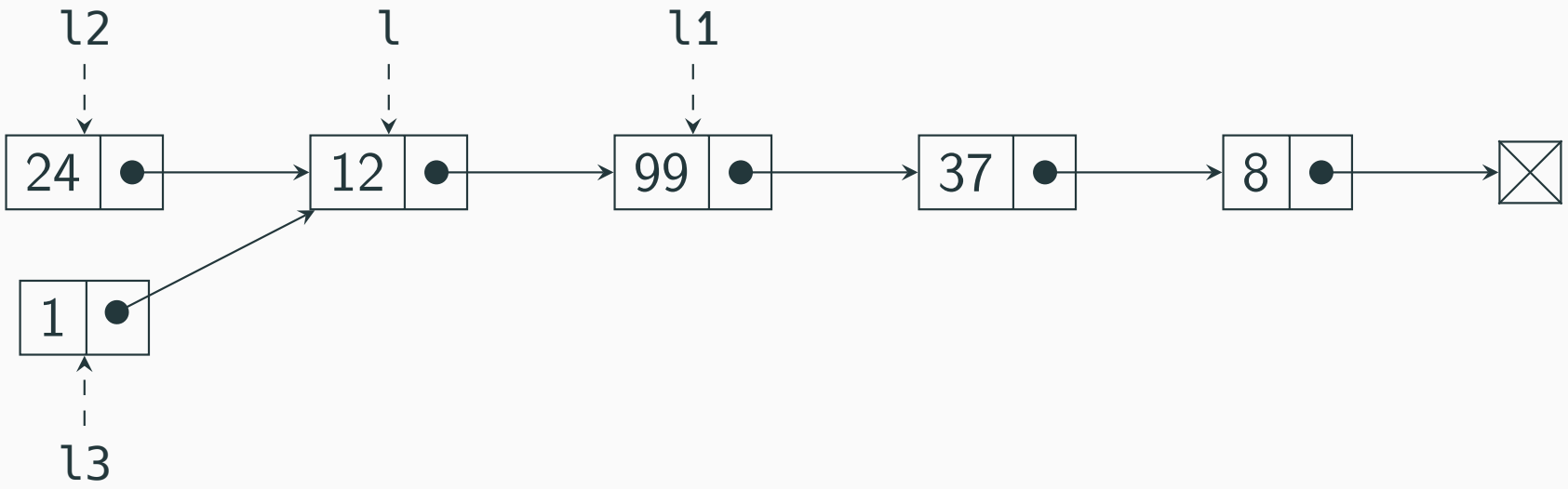
Complexité des opérations

Les trois opérations `hd`, `tl` et `::` ont toutes une **complexité constante**.

Ceci peut paraître étonnant car on crée de nouvelles listes via ces opérations, mais elles ne sont pas recopiées entièrement.

Les éléments sont **partagés** au maximum pour diminuer la complexité (en temps comme en mémoire).

Le type `list` en Caml



Exemple

Voici une représentation de la liste `l=[12; 99; 37; 8]`, et des listes `l1=List.tl l`, `l2=24::l` et `l3=1::l`.

Résumé et avertissement

`list` vs `array`

On n'a parlé que de l'élément de **tête** dans cette représentation.

En effet, lorsqu'on manipule une liste chaînée, **seul l'élément en tête de liste est accessible**.

Les listes chaînées (`list`) sont très différentes des tableaux (`array`) :

- Un tableau est de taille fixée, et on peut accéder et modifier ses éléments en temps constant (on parle de structure **mutable**).
- Une liste chaînée est **immuable** (ou non mutable, ou persistante), l'accès à la tête et la construction de nouvelles listes avec `List.tl` et `::` se font en temps constant.

Exemples de fonctions sur les listes

```
1  let rec parcours l = match l with
2    | [] -> ()
3    | x::q -> parcours q
4  ;;
```

Parcours d'une liste

Le type `list` étant récursif, les fonctions sur les listes sont en général récursives.

La fonction ci-dessus se contente de parcourir la liste passée en argument, en entier, et rien d'autre.

Mais elle est l'archétype d'une fonction sur une liste :

- elle est récursive ;
- le filtrage comporte le motif de la liste vide `[]`, et le motif d'une liste non vide `x::q`.

Exemples de fonctions sur les listes

Longueur d'une liste

La fonction suivante calcule la longueur d'une liste (elle existe déjà en OCaml : **`List.length`**).

```
1  let rec longueur l = match l with
2    | [] -> 0
3    | _::q -> 1 + longueur q
4  ;;
```

Remarques

Attention : la complexité de cette fonction est **linéaire** en la taille de la liste.

C'est également le cas de **`List.length`**, mais la version de OCaml est **récursive terminale**.

Exemples de fonctions sur les listes

Tester l'appartenance d'un élément à une liste

La fonction suivante implémente la fonction **`List.mem`** de Caml (pour "member" en anglais).

```
1  let rec appartient x l = match l with
2    | [] -> false
3    | y::q -> y=x || appartient x q
4  ;;
```

Exemples de fonctions sur les listes

Un exemple un peu plus complexe

L'opérateur `::` peut être utilisé plusieurs fois dans un motif de filtrage.

La fonction suivante teste si une liste possède deux éléments consécutifs égaux.

```
1  let rec deux_egaux l = match l with
2    | [] | [_] -> false
3    | x::y::q when x=y -> true
4    | x::q -> deux_egaux q
5  ;;
```

Exemples de fonctions sur les listes

Miroir d'une liste

La fonction **`List.rev`** de Caml renvoie la liste "miroir" de la liste passée en argument.

En voici une implémentation, où on utilise une **fonction auxiliaire** et un **accumulateur** qu'on remplit au fur et à mesure que l'on lit la liste passée en entrée.

```
1  let miroir l =
2    let rec aux acc l = match l with
3      | [] -> acc
4      | x::q -> aux (x::acc) q
5    in aux [] l
6  ;;
```

Concaténation de listes

Concaténation

L'opérateur `@` permet de concaténer deux listes : `l1@l2` rajoute la liste `l2` à la fin de `l1`.

Attention cependant, on essaie si possible de ne pas l'utiliser car l'opération `l1@l2` s'effectue en $O(n_1)$, où n_1 est la taille de `l1`. En effet, il faut parcourir la première liste et en faire une copie, avant de "brancher" `l2` à la fin.

```
# let l1 = [1;2;3] in
  let l2 = [4;5;6] in
  let l = l1@l2 in
  (l1,l2,l) ;;
- : int list * int list * int list = ([1; 2; 3], [4; 5; 6], [1; 2; 3; 4; 5; 6])
```

Ajout d'un élément dans une liste

Important

Soit `l` une liste de taille n, et `x` un élément.

- Rajouter `x` à gauche de `l` avec `x::l` se fait en $O(1)$.
  C'est ce qu'on fera donc naturellement.
- Rajouter `x` à droite `l` avec `l@[x]` se fait en $O(n)$.
  Il faudra donc essayer de l'éviter.

Construction de listes

Il existe plusieurs manières de construire des listes.

Une possibilité est de donner la liste des éléments “à la main”.

Sinon, il faut la construire via une structure itérative ou en faisant usage de récursivité.

Donnons deux méthodes pour construire la liste des entiers de 0 à $n-1$.

Construction de listes

Fonction récursive

La fonction suivante convient par exemple.

On se force à écrire une fonction **récursive terminale** ici.

```
1  let liste_ent n =
2    let rec aux acc k = match k<0 with
3      | true -> acc
4      | false -> aux (k::acc) (k-1)
5    in aux [] (n-1)
6  ;;
```

Construction de listes

Fonction impérative

Une autre possibilité est d'utiliser une boucle, et une référence vers une liste.

Rappel : une liste est **immuable**, mais on peut changer l'objet pointé par une référence.

```
1  let liste_ent n =
2    let l = ref [] in
3    for i = n-1 downto 0 do
4      l := i::!l
5    done ;
6    !l
7  ;;
```

Une implémentation personnalisée des listes

Implémentation personnalisée

Il est très facile de réécrire une implémentation personnalisée des listes Caml, à l'aide d'un **type somme**.

```
1  type 'a liste = Vide
2                | Cons of 'a * 'a liste
3  ;;
```

Une implémentation personnalisée des listes

Opérateurs

Voici la définition de l'opérateur **cons** et des fonctions **tete** et **queue**, analogues de `::`, `List.hd` et `List.tl`.

On donne aussi le type des fonctions écrites.

```
1  let tete l = match l with
2    | Vide -> failwith "Vide"
3    | Cons (x, _) -> x
4  ;;
5
6  let queue l = match l with
7    | Vide -> failwith "Vide"
8    | Cons (_, q) -> q
9  ;;
10
11 let cons (x,q) = Cons(x,q) ;;
```

```
1  tete : 'a liste -> 'a
2  queue : 'a liste -> 'a liste
3  cons : 'a * 'a liste -> 'a liste
```

Une implémentation personnalisée des listes

Remarque

C'est quasiment de cette manière que sont implémentées les listes en Caml.

On remarquera que **`hd`**, **`tl`** et la version préfixe de **`::`** ont les mêmes types que ce qu'on obtient, en remplaçant **`liste`** par **`list`**.

Implémentation de structures de pile et de file à l'aide de listes chaînées

Implémentation de piles et de files

On va implémenter une structure de **pile** et une structure de **file** à l'aide de listes chaînées.

La différence avec l'implémentation que l'on a vu sur les tableaux est que la taille peut varier : il est peu coûteux (complexité $O(1)$) d'ajouter un élément en tête d'une liste pour obtenir une nouvelle liste. On peut donc réaliser des piles et files **non bornées** : leur capacité est virtuellement infinie.

Implémentation de structures de pile et de file à l'aide de listes chaînées

Polymorphisme

On n'aura pas non plus à préciser un élément à la création pour spécifier le type (comme on a dû le faire pour les tableaux), la liste vide étant **polymorphe**.

```
# [] ;;
- : 'a list = []
```

Structure de pile

Pile

La structure de **pile** avec son seul sommet accessible est très similaire à une liste chaînée avec son élément de tête accessible.

Ainsi une liste chaînée est quasiment une pile.

On réalise ici une structure mutable, constituée d'un unique champ (modifiable) contenant une liste.

```
1  type 'a pile = {mutable contenu: 'a list} ;;
```

Structure de pile

```
1   let creer_pile () = {contenu=[]} ;;
2   let pile_vide p = p.contenu = [] ;;
3   let empiler p x = p.contenu <- x::p.contenu ;;
4
5   let sommet p = match p.contenu with
6     | [] -> failwith "pile vide"
7     | x::_ -> x
8   ;;
9
10  let depiler p = match p.contenu with
11    | [] -> failwith "pile vide"
12    | x::q -> p.contenu <- q ; x
13  ;;
```

Complexité des opérations

Toutes les opérations se font en $O(1)$.

Structure de pile

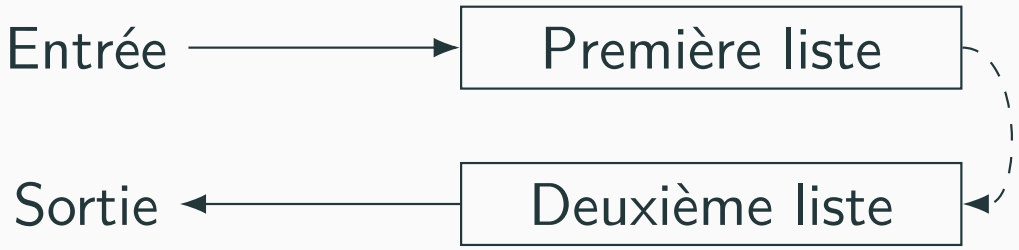
File

Implémenter une structure de **file** à l'aide de listes n'est pas aussi simple qu'une structure de pile : en effet, seule la tête de liste est accessible.

Mais on peut en utiliser deux. Cela mène au type suivant :

```
1  type 'a file = {mutable entree: 'a list ; mutable sortie: 'a list} ;;
```

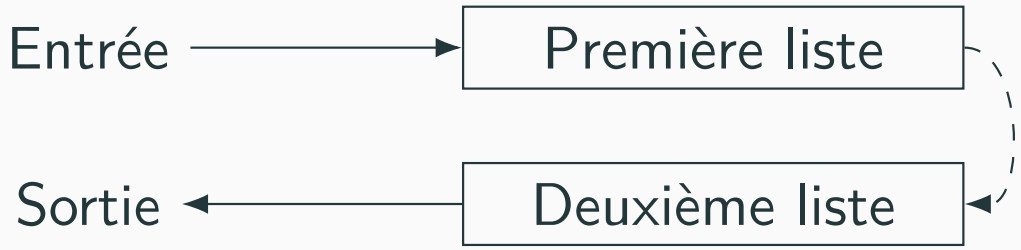
Structure de pile



Principe

La première est utilisée pour rajouter des éléments (en queue de file), la seconde pour enlever des éléments (en tête de file), comme le montre la figure ci-dessus.

Structure de pile



Principe

La seule difficulté dans l'écriture des opérations se situe dans l'opération "défiler", lorsque la seconde liste est vide : c'est l'élément en bout de la première liste qui doit sortir.

Pour garder une complexité constante en moyenne, il suffit d'abord de remplacer la deuxième liste (vide) par la première, retournée (et la première par une liste vide).

Structure de pile

```
1  let creer_file () = {entree = []; sortie = []} ;;
2  let file_vide f = f.entree = [] && f.sortie = [] ;;
3  let enfiler f x = f.entree <- x::f.entree ;;
4
5  let rec defiler f = match f.sortie with
6    | [] when f.entree = [] -> failwith "file vide"
7    | [] -> f.sortie <- List.rev f.entree ; f.entree <- [] ; defiler f
8    | x::q -> f.sortie <- q ; x
9  ;;
```

Opérations

Voici les opérations associées. Remarquez l'utilisation de la récursivité pour simplifier l'écriture de **`defiler`**.

Complexité des opérations

Toutes les opérations se font en temps **constant**, sauf lorsqu'il faut défiler alors que la seconde liste est vide.
Dans ce cas, la complexité est **linéaire** en le nombre d'éléments présents dans la file, ce qui est fâcheux.

Toutefois, ce cas-là ne se produit pas trop souvent.
En effet lorsqu'on défile alors qu'il y a n éléments dans la file (et que la seconde liste est vide), le coût est $O(n)$, mais néanmoins les $n-1$ prochaines opérations "défiler" se feront en temps **constant**.

Ainsi, sur ces n opérations, il y a un coût total $O(n)$, donc $O(1)$ en **moyenne**. On dit que la **complexité amortie** de l'opération "défiler" est $O(1)$.

Structure de pile

Exemple

```
# let f = creer_file () ;;
val f : '_a file = {entree = []; sortie = []}
# for i=0 to 999 do enfiler f i done ;;
- : unit = ()
# defiler f ;; (* cette opération est coûteuse *)
- : int = 0
# f ;; (* on peut défiler 999 fois en temps O(1) *)
- : int file =
{entree = []; sortie = [1; 2; 3; 4; 5; 6; 7; 8; 9;...]}
```

Structure de pile

Remarque

De manière générale, on peut de la même façon réaliser une structure de file dès que l'on a une implémentation d'une structure de pile : il suffit d'en utiliser deux.