Les ensembles en informatique

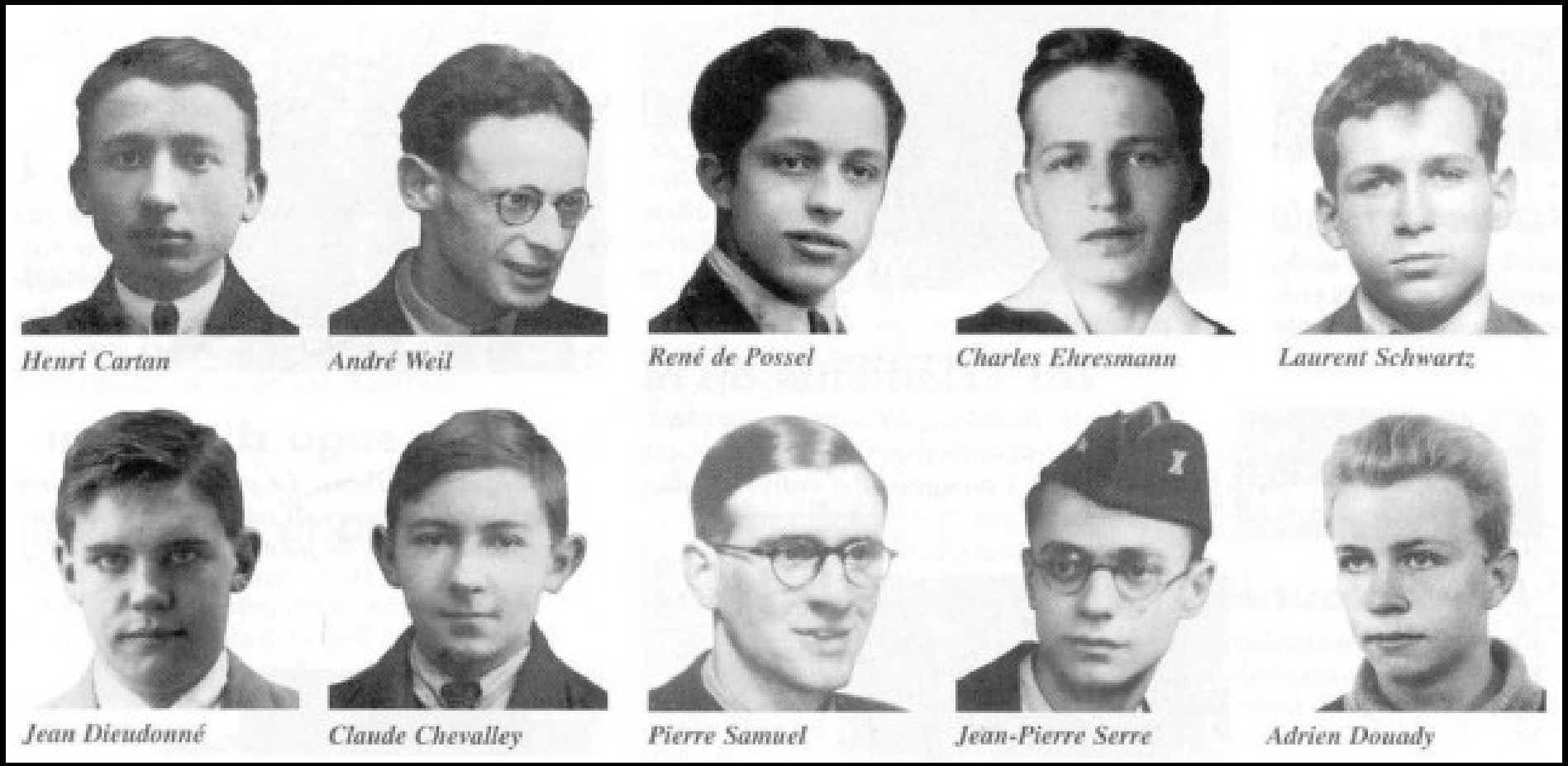
INFO1 - Semaine 38

Guillaume CONNAN

IUT de Nantes - Dpt d'informatique

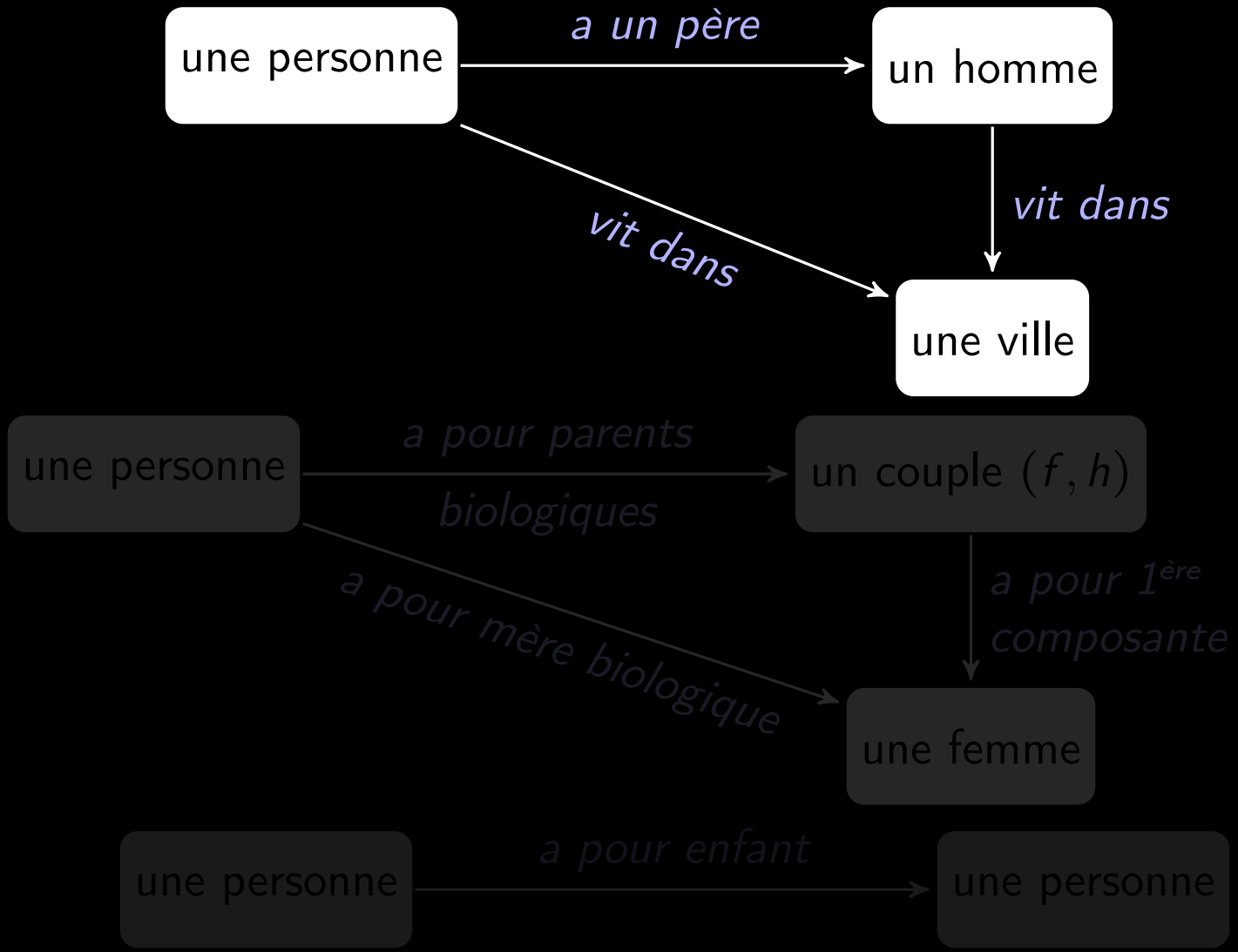
Dernière mise à jour : 28 septembre 2014 à 21:19

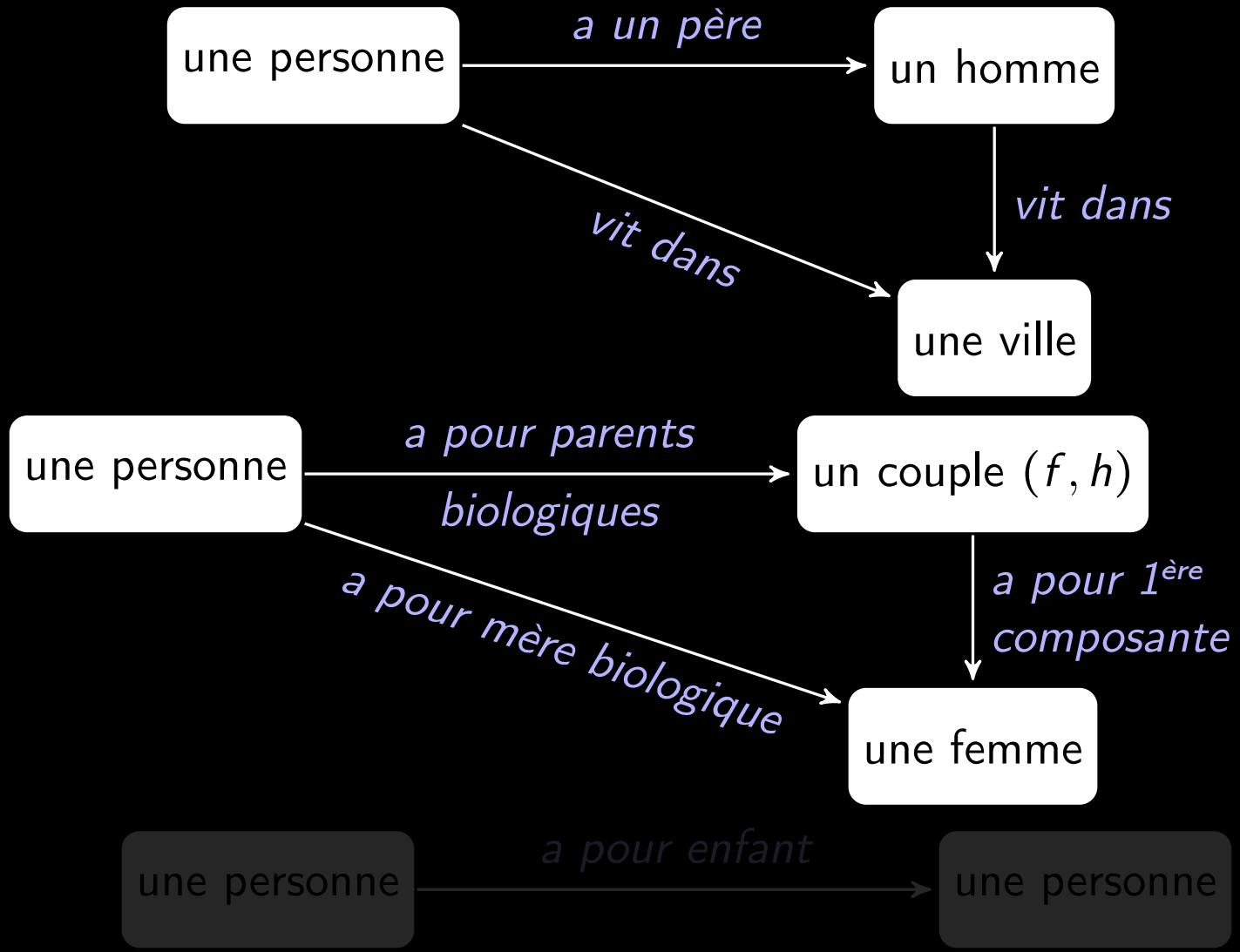
Sommaire

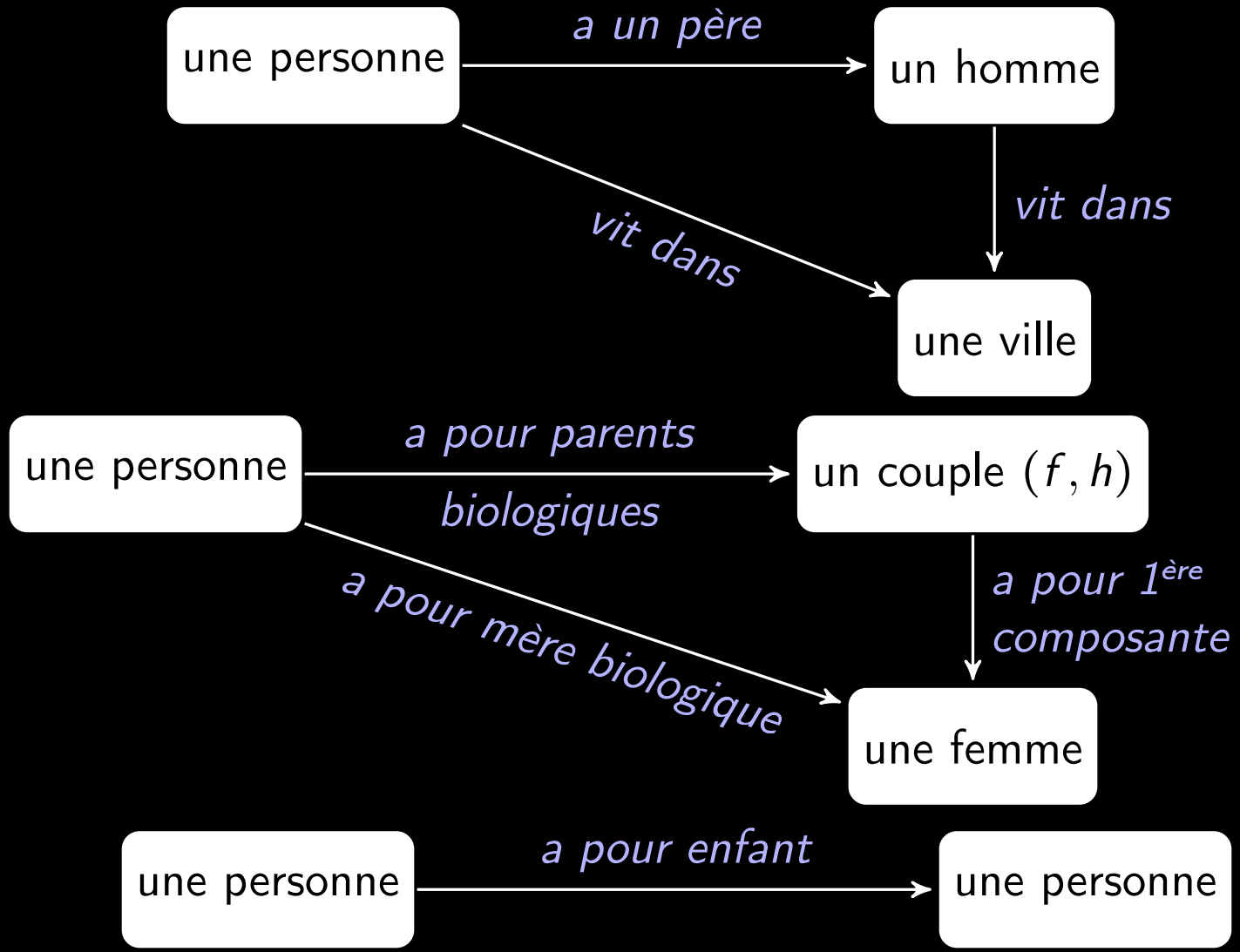

Henri Cartan
André Weil
René de Possel
Charles Ehresmann
Laurent Schwartz
Jean Dieudonné
Claude Chevalley
Pierre Samuel
Jean-Pierre Serre
Adrien Douady

...autrefois, on a pu croire que chaque branche des mathématiques dépendait d'intuitions particulières [...] ce qui eût entraîné pour chacune la nécessité d'un langage formalisé qui lui appartînt en propre. On sait aujourd'hui qu'il est possible, logiquement parlant, de faire dériver presque toute la mathématique d'une source unique, la ***Théorie des Ensembles****.*

Sommaire


une personne
a un père
un homme
vit dans
vit dans
une ville
une personne
a pour parents biologiques
un couple (f, h)
a pour mère biologique
a pour 1ère composante
une femme
une personne
a pour enfant
une personne


une personne
a un père
un homme
vit dans
vit dans
une ville
une personne
a pour parents
biologiques
un couple (f, h)
a pour mère biologique
a pour $1^{ère}$
composante
une femme
une personne
a pour enfant
une personne


une personne
a un père
un homme
vit dans
vit dans
une ville
une personne
a pour parents
biologiques
un couple (f, h)
a pour mère biologique
a pour $1^{ère}$
composante
une femme
une personne
a pour enfant
une personne

Georg Cantor
(1845 - 1918)

Haskell

```
λ> :t 'a'
'a' :: Char
```

Haskell

```
λ> :t 2 + 2 == 4
2 + 2 == 4 :: Bool
```

Haskell

```
λ> :t 3
3 :: Num a⇒a
```

Haskell

```
λ> :t 'a'
'a' :: Char
```

Haskell

```
λ> :t 2 + 2 == 4
2 + 2 == 4 :: Bool
```

Haskell

```
λ> :t 3
3 :: Num a⇒a
```

Haskell

```
λ> :t 'a'
'a' :: Char
```

Haskell

```
λ> :t 2 + 2 == 4
2 + 2 == 4 :: Bool
```

Haskell

```
λ> :t 3
3 :: Num a⇒a
```

Environnement de travail

On considère un « tout » contenant des « objets » distincts. On peut vérifier si certains de ces « objets » vérifient ou non une « propriété ».

$$G = \{ \text{étudiant} \mid (\text{étudiant est en INFO1}) \text{ ET } (\text{étudiant est un garçon}) \}$$

→ « Robert appartient à G »

Environnement de travail

On considère un « tout » contenant des « objets » distincts. On peut vérifier si certains de ces « objets » vérifient ou non une « propriété ».

$$G = \left\{ \text{étudiant} \mid (\text{étudiant est en INFO1}) \text{ ET } (\text{étudiant est un garçon}) \right\}$$

- « Robert appartient à G »
- « Robert est un élément de G »
- « G contient Robert »

Environnement de travail

On considère un « tout » contenant des « objets » distincts. On peut vérifier si certains de ces « objets » vérifient ou non une « propriété ».

$$G = \{ \text{étudiant} \mid (\text{étudiant est en INFO1}) \text{ ET } (\text{étudiant est un garçon}) \}$$

- « Robert appartient à G »
- « Robert est un élément de G »
- « G contient Robert »
- « Robert $\in G$ »
- « $G \ni$ Robert »

Environnement de travail

On considère un « tout » contenant des « objets » distincts. On peut vérifier si certains de ces « objets » vérifient ou non une « propriété ».

$$G = \left\{ \text{étudiant} \mid (\text{étudiant est en INFO1}) \text{ ET } (\text{étudiant est un garçon}) \right\}$$

- « Robert appartient à G »
- « Robert est un élément de G »
- « G contient Robert »
- « Robert $\in G$ »
- « $G \ni$ Robert »

Environnement de travail

On considère un « tout » contenant des « objets » distincts. On peut vérifier si certains de ces « objets » vérifient ou non une « propriété ».

$$G = \left\{ \text{étudiant} \mid (\text{étudiant est en INFO1}) \text{ ET } (\text{étudiant est un garçon}) \right\}$$

- « Robert appartient à G »
- « Robert est un élément de G »
- « G contient Robert »
- « Robert $\in G$ »
- « $G \ni$ Robert »

Environnement de travail

On considère un « tout » contenant des « objets » distincts. On peut vérifier si certains de ces « objets » vérifient ou non une « propriété ».

$$G = \left\{ \text{étudiant} \mid (\text{étudiant est en INFO1}) \text{ ET } (\text{étudiant est un garçon}) \right\}$$

- « Robert appartient à G »
- « Robert est un élément de G »
- « G contient Robert »
- « Robert $\in G$ »
- « $G \ni$ Robert »

Environnement de travail

On considère un « tout » contenant des « objets » distincts. On peut vérifier si certains de ces « objets » vérifient ou non une « propriété ».

$$G = \left\{ \text{étudiant} \mid (\text{étudiant est en INFO1}) \text{ ET } (\text{étudiant est un garçon}) \right\}$$

- « Robert appartient à G »
- « Robert est un élément de G »
- « G contient Robert »
- « Robert $\in G$ »
- « $G \ni$ Robert »

Entiers naturels pairs ?

Haskell

```
naturels = [0..]
pairs = [n | n←naturels, mod n 2 == 0]
petits_pairs = [n | n←[0..15], mod n 2 == 0]
petits_pairs_bis = [2*k | k←[0..7]]
```

Haskell

```
λ> petits_pairs
[0,2,4,6,8,10,12,14]
```

Entiers naturels pairs ?

Haskell

```haskell
naturels = [0..]
pairs = [n | n←naturels, mod n 2 == 0]
petits_pairs = [n | n←[0..15], mod n 2 == 0]
petits_pairs_bis = [2*k | k←[0..7]]
```

Haskell

```
λ> petits_pairs
[0,2,4,6,8,10,12,14]
```

Entiers naturels pairs ?

Haskell

```
naturels = [0..]
pairs = [n | n←naturels, mod n 2 == 0]
petits_pairs = [n | n←[0..15], mod n 2 == 0]
petits_pairs_bis = [2*k | k←[0..7]]
```

Haskell

```
λ> petits_pairs
[0,2,4,6,8,10,12,14]
```

- $2 \in \texttt{pairs}$
- $2 \in \{x | (x \in \mathbb{N}) \wedge P(x)\}$
- $2 \in \{t | (t \in \mathbb{N}) \wedge P(t)\}$
- $P(2)$
- $2 \in \{x | x \text{ est un entier naturel et } x \text{ est pair}\}$
- $3 \notin \texttt{pairs}$
- $3 \in \{x | (x \in \mathbb{N}) \wedge \neg P(x)\}$
- $3 \notin \{t | (t \in \mathbb{N}) \wedge P(t)\}$
- $\neg P(3)$

- $2 \in \texttt{pairs}$
- $2 \in \{x | (x \in \mathbb{N}) \wedge P(x)\}$
- $2 \in \{t | (t \in \mathbb{N}) \wedge P(t)\}$
- $P(2)$
- $2 \in \{x | x \text{ est un entier naturel et } x \text{ est pair}\}$
- $3 \notin \texttt{pairs}$
- $3 \in \{x | (x \in \mathbb{N}) \wedge \neg P(x)\}$
- $3 \notin \{t | (t \in \mathbb{N}) \wedge P(t)\}$
- $\neg P(3)$

- $2 \in \texttt{pairs}$
- $2 \in \{x | (x \in \mathbb{N}) \wedge P(x)\}$
- $2 \in \{t | (t \in \mathbb{N}) \wedge P(t)\}$
- $P(2)$
- $2 \in \{x | x \text{ est un entier naturel et } x \text{ est pair}\}$
- $3 \notin \texttt{pairs}$
- $3 \in \{x | (x \in \mathbb{N}) \wedge \neg P(x)\}$
- $3 \notin \{t | (t \in \mathbb{N}) \wedge P(t)\}$
- $\neg P(3)$

- $2 \in \texttt{pairs}$
- $2 \in \{x | (x \in \mathbb{N}) \wedge P(x)\}$
- $2 \in \{t | (t \in \mathbb{N}) \wedge P(t)\}$
- $P(2)$
- $2 \in \{x | x \text{ est un entier naturel et } x \text{ est pair}\}$
- $3 \notin \texttt{pairs}$
- $3 \in \{x | (x \in \mathbb{N}) \wedge \neg P(x)\}$
- $3 \notin \{t | (t \in \mathbb{N}) \wedge P(t)\}$
- $\neg P(3)$

- $2 \in \texttt{pairs}$
- $2 \in \{x | (x \in \mathbb{N}) \wedge P(x)\}$
- $2 \in \{t | (t \in \mathbb{N}) \wedge P(t)\}$
- $P(2)$
- $2 \in \{x | x \text{ est un entier naturel et } x \text{ est pair}\}$
- $3 \notin \texttt{pairs}$
- $3 \in \{x | (x \in \mathbb{N}) \wedge \neg P(x)\}$
- $3 \notin \{t | (t \in \mathbb{N}) \wedge P(t)\}$
- $\neg P(3)$

- $2 \in \texttt{pairs}$
- $2 \in \{x | (x \in \mathbb{N}) \wedge P(x)\}$
- $2 \in \{t | (t \in \mathbb{N}) \wedge P(t)\}$
- $P(2)$
- $2 \in \{x | x \text{ est un entier naturel et } x \text{ est pair}\}$
- $3 \notin \texttt{pairs}$
- $3 \in \{x | (x \in \mathbb{N}) \wedge \neg P(x)\}$
- $3 \notin \{t | (t \in \mathbb{N}) \wedge P(t)\}$
- $\neg P(3)$

- $2 \in \texttt{pairs}$
- $2 \in \{x | (x \in \mathbb{N}) \wedge P(x)\}$
- $2 \in \{t | (t \in \mathbb{N}) \wedge P(t)\}$
- $P(2)$
- $2 \in \{x | x \text{ est un entier naturel et } x \text{ est pair}\}$
- $3 \notin \texttt{pairs}$
- $3 \in \{x | (x \in \mathbb{N}) \wedge \neg P(x)\}$
- $3 \notin \{t | (t \in \mathbb{N}) \wedge P(t)\}$
- $\neg P(3)$

- $2 \in \texttt{pairs}$
- $2 \in \{x | (x \in \mathbb{N}) \wedge P(x)\}$
- $2 \in \{t | (t \in \mathbb{N}) \wedge P(t)\}$
- $P(2)$
- $2 \in \{x | x \text{ est un entier naturel et } x \text{ est pair}\}$
- $3 \notin \texttt{pairs}$
- $3 \in \{x | (x \in \mathbb{N}) \wedge \neg P(x)\}$
- $3 \notin \{t | (t \in \mathbb{N}) \wedge P(t)\}$
- $\neg P(3)$

- $2 \in \texttt{pairs}$
- $2 \in \{x | (x \in \mathbb{N}) \wedge P(x)\}$
- $2 \in \{t | (t \in \mathbb{N}) \wedge P(t)\}$
- $P(2)$
- $2 \in \{x | x \text{ est un entier naturel et } x \text{ est pair}\}$
- $3 \notin \texttt{pairs}$
- $3 \in \{x | (x \in \mathbb{N}) \wedge \neg P(x)\}$
- $3 \notin \{t | (t \in \mathbb{N}) \wedge P(t)\}$
- $\neg P(3)$

Exemple 1

Un nombre x « vérifie R » si, et seulement si, son carré vaut 2.

$$\forall x(x \in \mathbb{N} \wedge \neg R(x))$$

für Alle : 1934 par Gerhard Gentzel

Exemple 1

Un nombre x « vérifie R » si, et seulement si, son carré vaut 2.

$$\forall x(x \in \mathbb{N} \wedge \neg R(x))$$

für Alle : 1934 par Gerhard Gentzel

Exemple 1

Un nombre x « vérifie R » si, et seulement si, son carré vaut 2.

$$\forall x(x \in \mathbb{N} \wedge \neg R(x))$$

*für **A**lle* : 1934 par Gerhard Gentzel

Exemple 1

Un nombre x « vérifie R » si, et seulement si, son carré vaut 2.

$$\forall x(x \in \mathbb{N} \wedge \neg R(x))$$

*für **A**lle* : 1934 par Gerhard Gentzel

Exemple 2

Un nombre x « vérifie S » si, et seulement si, son carré vaut 4.

$$\exists x(x \in \mathbb{N} \wedge S(x))$$

Esiste almeno uno : 1897 Giuseppe Peano

$$\exists! x(x \in \mathbb{N} \wedge S(x))$$

Exemple 2

Un nombre x « vérifie S » si, et seulement si, son carré vaut 4.

$$\exists x(x \in \mathbb{N} \wedge S(x))$$

Esiste almeno uno : 1897 Giuseppe Peano

$$\exists! x(x \in \mathbb{N} \wedge S(x))$$

Exemple 2

Un nombre x « vérifie S » si, et seulement si, son carré vaut 4.

$$\exists x(x \in \mathbb{N} \wedge S(x))$$

***E**siste almeno uno* : 1897 Giuseppe Peano

$$\exists! x(x \in \mathbb{N} \wedge S(x))$$

Exemple 2

Un nombre x « vérifie S » si, et seulement si, son carré vaut 4.

$$\exists x(x \in \mathbb{N} \wedge S(x))$$

***E**siste almeno uno* : 1897 Giuseppe Peano

$$\exists! x(x \in \mathbb{N} \wedge S(x))$$

Théorème 3

- $\forall x(x \in E \rightarrow P(x)) \equiv E = \{x \in E \mid P(x)\}$
- $\exists x(x \in E \rightarrow P(x)) \equiv \{x \in E \mid P(x)\} \neq \emptyset$
- $\neg(\forall x(x \in E \rightarrow P(x))) \equiv \exists x(x \in E \rightarrow \neg P(x))$
- $\neg(\exists x(x \in E \rightarrow P(x))) \equiv \forall x(x \in E \rightarrow \neg P(x))$

Théorème 3

- $\forall x(x \in E \rightarrow P(x)) \equiv E = \{x \in E \mid P(x)\}$
- $\exists x(x \in E \rightarrow P(x)) \equiv \{x \in E \mid P(x)\} \neq \emptyset$
- $\neg(\forall x(x \in E \rightarrow P(x))) \equiv \exists x(x \in E \rightarrow \neg P(x))$
- $\neg(\exists x(x \in E \rightarrow P(x))) \equiv \forall x(x \in E \rightarrow \neg P(x))$

Théorème 3

- $\forall x(x \in E \rightarrow P(x)) \equiv E = \{x \in E \mid P(x)\}$
- $\exists x(x \in E \rightarrow P(x)) \equiv \{x \in E \mid P(x)\} \neq \varnothing$
- $\neg(\forall x(x \in E \rightarrow P(x))) \equiv \exists x(x \in E \rightarrow \neg P(x))$
- $\neg(\exists x(x \in E \rightarrow P(x))) \equiv \forall x(x \in E \rightarrow \neg P(x))$

Théorème 3

- $\forall x(x \in E \rightarrow P(x)) \equiv E = \{x \in E \mid P(x)\}$
- $\exists x(x \in E \rightarrow P(x)) \equiv \{x \in E \mid P(x)\} \neq \varnothing$
- $\neg\left(\forall x(x \in E \rightarrow P(x))\right) \equiv \exists x(x \in E \rightarrow \neg P(x))$
- $\neg\left(\exists x(x \in E \rightarrow P(x))\right) \equiv \forall x(x \in E \rightarrow \neg P(x))$

Théorème 3

- $\forall x(x \in E \rightarrow P(x)) \equiv E = \{x \in E \mid P(x)\}$
- $\exists x(x \in E \rightarrow P(x)) \equiv \{x \in E \mid P(x)\} \neq \varnothing$
- $\neg\left(\forall x(x \in E \rightarrow P(x))\right) \equiv \exists x(x \in E \rightarrow \neg P(x))$
- $\neg\left(\exists x(x \in E \rightarrow P(x))\right) \equiv \forall x(x \in E \rightarrow \neg P(x))$

Haskell

```haskell
s :: (Num a, Eq a)⇒a→ Bool
s =                    \x→ x^2 == 4
```

Haskell

```
λ> s 2
True
λ> s 2.0
True
λ> s 3
False
λ> any s [0..]
True
λ> find s [0..]
Just 2
λ> filter s [0..100]
[2]
```

Haskell

```haskell
s :: (Num a, Eq a)⇒a→ Bool
s =                    \x→ x^2 == 4
```

Haskell

```
λ> s 2
True
λ> s 2.0
True
λ> s 3
False
λ> any s [0..]
True
λ> find s [0..]
Just 2
λ> filter s [0..100]
[2]
```

Exercice 1

Dans une ville, il n'y a qu'un barbier qui rase tous ceux qui ne se rasent pas eux-mêmes et uniquement ceux-ci : qui rase le barbier ?...

Sommaire

Integer

Haskell

```
λ> 2^1000
10715086071862673209484250490600018105614048117055336074437503883
83788156958581275946729175531468251871452856923140435984577574698
85421074605062371141877954182153046474983581941267398767559165543
86542167660429831652624386837205668069376
```

Int

Haskell

```
λ> let n = 2^63 - 1 :: Int
λ> n
9223372036854775807
```

Haskell

```
λ> n * 2
-2
```

Int

Haskell

```
λ> let n = 2^63 - 1 :: Int
λ> n
9223372036854775807
```

Haskell

```
λ> n * 2
-2
```

Haskell

```
λ> maxBound :: Int
9223372036854775807
λ> minBound :: Int
-9223372036854775808
```

Haskell

```
λ> [minBound :: Int8 ..]
[-128,-127,-126,-125,-124,-123,-122,-121,-120,-119,-118,-117,-116,
-111,-110,-109,-108,-107,-106,-105,-104,-103,-102,-101,-100,-99,-9
-93,-92,-91,-90,-89,-88,-87,-86,-85,-84,-83,-82,-81,-80,-79,-78,-7
-72,-71,-70,-69,-68,-67,-66,-65,-64,-63,-62,-61,-60,-59,-58,-57,-5
-51,-50,-49,-48,-47,-46,-45,-44,-43,-42,-41,-40,-39,-38,-37,-36,-3
-30,-29,-28,-27,-26,-25,-24,-23,-22,-21,-20,-19,-18,-17,-16,-15,-1
-9,-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,1
23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,
52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,
81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,101,1
107,108,109,110,111,112,113,114,115,116,117,118,119,120,121,122,12
126,127]
```

Les booléens

Haskell

```
λ> True && False
False
λ> False && False
False
λ> False || True
True
λ> not False
True
λ> 2 + 2 > 5
False
λ> 2 + 2 /= 5
True
λ> "Iut" == "Enfer"
False
λ> "Haskell" > "C"
True
```

Char

Haskell

```
λ> minBound :: Char
'\NUL'
λ> maxBound :: Char
'\1114111'
λ> take 130 [ minBound :: Char .. ]
"\NUL\SOH\STX\ETX\EOT\ENQ\ACK\a\b\t\n\v\f\r\SO\SI\DLE\DC1\DC2\DC3\
\EM\SUB\ESC\FS\GS\RS\US
    !\"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\
\]^_`abcdefghijklmnopqrstuvwxyz{|}~\DEL\128\129"
```

Sommaire

Définition 4 (Axiome d'extensionalité)

Deux ensembles A et B sont égaux si, et seulement si, ils contiennent les mêmes éléments. On écrit alors $A = B$.

Exercice 2

On note $E = \{x \mid x \in \mathbb{Z} \wedge x^2 = 1\}$ *et* $F = \{x \mid x \in \mathbb{R} \wedge \lfloor x \rfloor = 1\}$.
Que pouvez-vous dire de E *par rapport à* F *?*

Haskell

```
λ> [1,2,3] == [1,3,2]
False
λ> 3 * 0.1 == 0.3
False
λ> 2 * 0.1 == 0.2
True
```

Définition 4 (Axiome d'extensionalité)

Deux ensembles A et B sont égaux si, et seulement si, ils contiennent les mêmes éléments. On écrit alors $A = B$.

Exercice 2

On note $E = \{x \mid x \in \mathbb{Z} \wedge x^2 = 1\}$ *et* $F = \{x \mid x \in \mathbb{R} \wedge \lfloor x \rfloor = 1\}$.
Que pouvez-vous dire de E *par rapport à* F *?*

Haskell

```
λ> [1,2,3] == [1,3,2]
False
λ> 3 * 0.1 == 0.3
False
λ> 2 * 0.1 == 0.2
True
```

Définition 4 (Axiome d'extensionalité)

Deux ensembles A et B sont égaux si, et seulement si, ils contiennent les mêmes éléments. On écrit alors $A = B$.

Exercice 2

On note $E = \{x \mid x \in \mathbb{Z} \wedge x^2 = 1\}$ *et* $F = \{x \mid x \in \mathbb{R} \wedge \lfloor x \rfloor = 1\}$.
Que pouvez-vous dire de E *par rapport à* F *?*

Haskell

```
λ> [1,2,3] == [1,3,2]
False
λ> 3 * 0.1 == 0.3
False
λ> 2 * 0.1 == 0.2
True
```

Définition 5 (inclusion)

L'ensemble X est inclus dans l'ensemble Y si, et seulement si, tous les éléments de X sont des éléments de Y. Cela se note

$$(X \subseteq Y) \leftrightarrow ((z \in X) \rightarrow (z \in Y))$$

On dira indifféremment « X *est contenu dans* Y », « Y *contient* X », « X *est un sous-ensemble de* Y », « X *est une partie de* Y ».

Démonstration ?...

$$(\varnothing \subseteq E) \leftrightarrow ((z \in \varnothing) \rightarrow (z \in E)) \text{ ?...}$$

Définition 5 (inclusion)

L'ensemble X est inclus dans l'ensemble Y si, et seulement si, tous les éléments de X sont des éléments de Y. Cela se note

$$(X \subseteq Y) \leftrightarrow ((z \in X) \rightarrow (z \in Y))$$

On dira indifféremment « X *est contenu dans* Y », « Y *contient* X », « X *est un sous-ensemble de* Y », « X *est une partie de* Y ».

Exemple ?

Tout ensemble E contient l'ensemble vide.

Démonstration ?...

$$(\varnothing \subseteq E) \leftrightarrow ((z \in \varnothing) \rightarrow (z \in E)) \text{ ?...}$$

Définition 5 (inclusion)

L'ensemble X est inclus dans l'ensemble Y si, et seulement si, tous les éléments de X sont des éléments de Y. Cela se note

$$(X \subseteq Y) \leftrightarrow ((z \in X) \rightarrow (z \in Y))$$

On dira indifféremment « X *est contenu dans* Y », « Y *contient* X », « X *est un sous-ensemble de* Y », « X *est une partie de* Y ».

Théorème 6

Tout ensemble E contient l'ensemble vide.

Démonstration ?...

$$(\varnothing \subseteq E) \leftrightarrow ((z \in \varnothing) \rightarrow (z \in E)) \text{ ?...}$$

Définition 5 (inclusion)

L'ensemble X est inclus dans l'ensemble Y si, et seulement si, tous les éléments de X sont des éléments de Y. Cela se note

$$(X \subseteq Y) \leftrightarrow ((z \in X) \rightarrow (z \in Y))$$

On dira indifféremment « X *est contenu dans* Y », « Y *contient* X », « X *est un sous-ensemble de* Y », « X *est une partie de* Y ».

Théorème 6

Tout ensemble E contient l'ensemble vide.

Démonstration ?...

$$(\varnothing \subseteq E) \leftrightarrow ((z \in \varnothing) \rightarrow (z \in E)) \text{ ?...}$$

Définition 5 (inclusion)

L'ensemble X est inclus dans l'ensemble Y si, et seulement si, tous les éléments de X sont des éléments de Y. Cela se note

$$(X \subseteq Y) \leftrightarrow ((z \in X) \rightarrow (z \in Y))$$

On dira indifféremment « X *est contenu dans* Y », « Y *contient* X », « X *est un sous-ensemble de* Y », « X *est une partie de* Y ».

Théorème 6

Tout ensemble E contient l'ensemble vide.

Démonstration ?...

$$(\varnothing \subseteq E) \leftrightarrow ((z \in \varnothing) \rightarrow (z \in E)) \text{ ?...}$$

Définition 5 (inclusion)

L'ensemble X est inclus dans l'ensemble Y si, et seulement si, tous les éléments de X sont des éléments de Y. Cela se note

$$(X \subseteq Y) \leftrightarrow ((z \in X) \rightarrow (z \in Y))$$

On dira indifféremment « X *est contenu dans* Y », « Y *contient* X », « X *est un sous-ensemble de* Y », « X *est une partie de* Y ».

Théorème 6

Tout ensemble E contient l'ensemble vide.

Démonstration ?...

$$(\varnothing \subseteq E) \leftrightarrow ((z \in \varnothing) \rightarrow (z \in E)) \text{ ?...}$$

Définition 5 (inclusion)

L'ensemble X est inclus dans l'ensemble Y si, et seulement si, tous les éléments de X sont des éléments de Y. Cela se note

$$(X \subseteq Y) \leftrightarrow ((z \in X) \rightarrow (z \in Y))$$

On dira indifféremment « X *est contenu dans* Y », « Y *contient* X », « X *est un sous-ensemble de* Y », « X *est une partie de* Y ».

Théorème 6

Tout ensemble E contient l'ensemble vide.

Démonstration ?...

$$(\varnothing \subseteq E) \leftrightarrow ((z \in \varnothing) \rightarrow (z \in E)) \text{ ?...}$$

DANGER !

Il faudra bien distinguer $\subset$, $\subseteq$ $\not\subseteq$ et $\not\subset$.

Théorème 7 (Axiome d'extensionalité version II)

$$(\forall X)(\forall Y)(((X \subseteq Y) \wedge (Y \subseteq X)) \rightarrow (X = Y))$$

Haskell

```
est_inclus :: (Eq a)⇒Ens a→ Ens a→ Bool
est_inclus ens1 ens2 = pour_tout (contient ens2) ens1
```

Haskell

```
instance (Eq a)⇒Eq (Ens a) where
  ens1 == ens2 = (est_inclus ens1 ens2) && (est_inclus ens2
      ens1)
```

Théorème 7 (Axiome d'extensionalité version II)

$$(\forall X)(\forall Y)(((X \subseteq Y) \wedge (Y \subseteq X)) \rightarrow (X = Y))$$

Haskell

```
est_inclus :: (Eq a)⇒Ens a→ Ens a→ Bool
est_inclus ens1 ens2 = pour_tout (contient ens2) ens1
```

Haskell

```
instance (Eq a)⇒Eq (Ens a) where
  ens1 == ens2 = (est_inclus ens1 ens2) && (est_inclus ens2
      ens1)
```

Théorème 7 (Axiome d'extensionalité version II)

$$(\forall X)(\forall Y)(((X \subseteq Y) \wedge (Y \subseteq X)) \rightarrow (X = Y))$$

Haskell

```
est_inclus :: (Eq a)⇒Ens a→ Ens a→ Bool
est_inclus ens1 ens2 = pour_tout (contient ens2) ens1
```

Haskell

```
instance (Eq a)⇒Eq (Ens a) where
  ens1 == ens2 = (est_inclus ens1 ens2) && (est_inclus ens2
      ens1)
```

Exercice 3

Montrez que

$$\{a\} = \{b\} \leftrightarrow a = b$$

puis que $\varnothing \neq \{\varnothing\}$ *et* $\{\varnothing\} \neq \{\{\varnothing\}\}$.

Théorème 3

Nous admettrons que si E désigne un ensemble alors l'ensemble des parties de E est un ensemble noté $P(E)$.

$$P(E) = \{X \mid X \subset E\}$$

Exercice 3

Montrez que

$$\{a\} = \{b\} \leftrightarrow a = b$$

puis que $\varnothing \neq \{\varnothing\}$ *et* $\{\varnothing\} \neq \{\{\varnothing\}\}$.

Théorème 8

Nous admettrons que si E désigne un ensemble alors l'ensemble des parties de E est un ensemble noté $P(E)$

$$P(E) = \{X \mid X \subseteq E\}$$

Exercice 3

Montrez que

$$\{a\} = \{b\} \leftrightarrow a = b$$

puis que $\varnothing \neq \{\varnothing\}$ *et* $\{\varnothing\} \neq \{\{\varnothing\}\}$.

Théorème 8

Nous admettrons que si E *désigne un ensemble alors* ***l'ensemble des parties de*** E *est un ensemble noté* $P(E)$

$$P(E) = \{X \mid X \subseteq E\}$$

Cet ensemble admet toujours au moins deux éléments : lesquels ?

Exercice 3

Montrez que

$$\{a\} = \{b\} \leftrightarrow a = b$$

puis que $\varnothing \neq \{\varnothing\}$ *et* $\{\varnothing\} \neq \{\{\varnothing\}\}$.

Théorème 8

Nous admettrons que si E *désigne un ensemble alors* ***l'ensemble des parties de*** E *est un ensemble noté* $P(E)$

$$P(E) = \{X \mid X \subseteq E\}$$

Exercice 4

Cet ensemble admet toujours au moins deux éléments : lesquels ?

Exercice 3

Montrez que

$$\{a\} = \{b\} \leftrightarrow a = b$$

puis que $\varnothing \neq \{\varnothing\}$ *et* $\{\varnothing\} \neq \{\{\varnothing\}\}$.

Théorème 8

Nous admettrons que si E *désigne un ensemble alors* ***l'ensemble des parties de*** E *est un ensemble noté* $P(E)$

$$P(E) = \{X \mid X \subseteq E\}$$

Exercice 4

Cet ensemble admet toujours au moins deux éléments : lesquels ?

Exemple 9

$E = \{D, L, M\}$

Haskell

```
λ> let e1 = lit ['D','L','M']
λ> ens_parties e1
{ { } { 'M' } { 'L' } { 'L' 'M' } { 'D' } { 'D' 'M' } { 'D'
    'L' } { 'D' 'L' 'M' } }
```

Exercice 5

Que dire alors de $P(P(E))$ *?*

Exemple 9

$E = \{D, L, M\}$

Haskell

```
λ> let e1 = lit ['D','L','M']
λ> ens_parties e1
{ { } { 'M' } { 'L' } { 'L' 'M' } { 'D' } { 'D' 'M' } { 'D'
    'L' } { 'D' 'L' 'M' } }
```

Exercice 5

Que dire alors de $P(P(E))$ *?*

Exemple 9

$E = \{D, L, M\}$

Haskell

```
λ> let e1 = lit ['D','L','M']
λ> ens_parties e1
{ { } { 'M' } { 'L' } { 'L' 'M' } { 'D' } { 'D' 'M' } { 'D'
    'L' } { 'D' 'L' 'M' } }
```

Exercice 5

Que dire alors de $P(P(E))$ *?*

Haskell

```
ens_parties ::  Eq a⇒Ens a→ Ens (Ens a)
ens_parties Vide = Ens (Vide,Vide)
ens_parties (Ens (t,q)) = union pq (applique (insere t) pq)
  where pq = ens_parties q
```

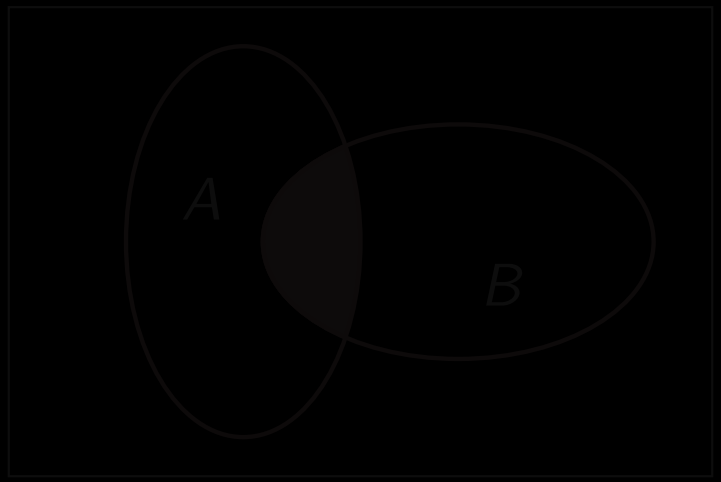
Sommaire

Définition 10

L'**intersection** des ensembles A et B est l'ensemble $A \cap B$ constitué des éléments communs à A et B.

$$A \cap B = \left\{ x \mid (x \in A) \wedge (x \in B) \right\}$$

C'est une partie de E. De plus $(x \in A \cap B) \leftrightarrow (x \in A \wedge x \in B)$.



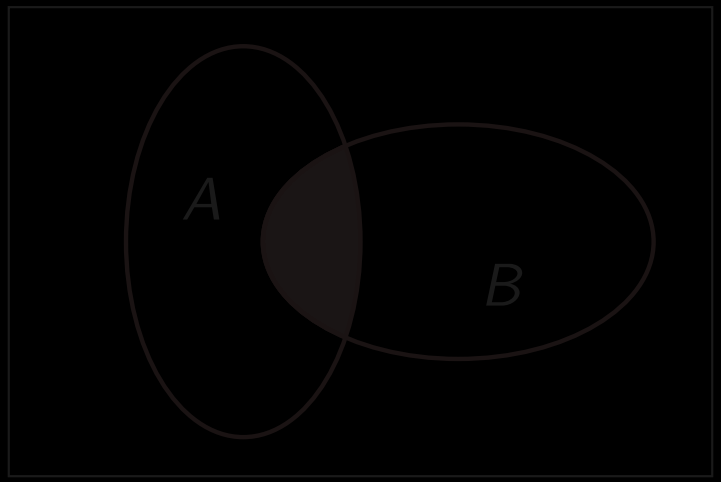
$A \cap B$

Définition 10

L'**intersection** des ensembles A et B est l'ensemble $A \cap B$ constitué des éléments communs à A et B.

$$A \cap B = \left\{ x \mid (x \in A) \wedge (x \in B) \right\}$$

C'est une partie de E. De plus $(x \in A \cap B) \leftrightarrow (x \in A \wedge x \in B)$.



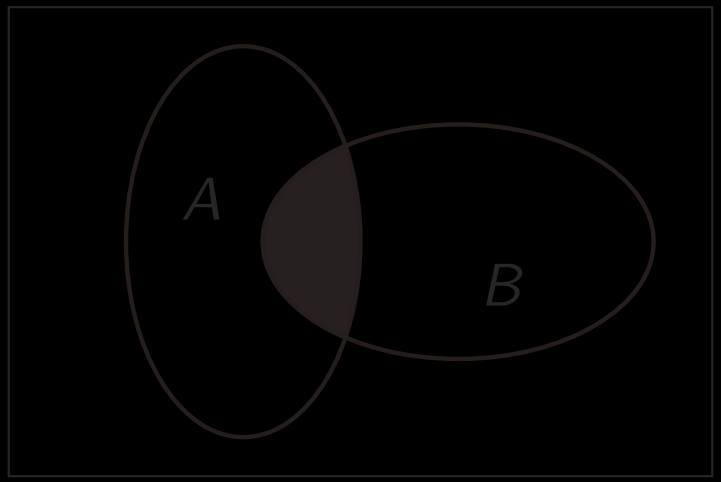
$A \cap B$

Définition 10

L'**intersection** des ensembles A et B est l'ensemble $A \cap B$ constitué des éléments communs à A et B.

$$A \cap B = \left\{ x \mid (x \in A) \wedge (x \in B) \right\}$$

C'est une partie de E. De plus $(x \in A \cap B) \leftrightarrow (x \in A \wedge x \in B)$.



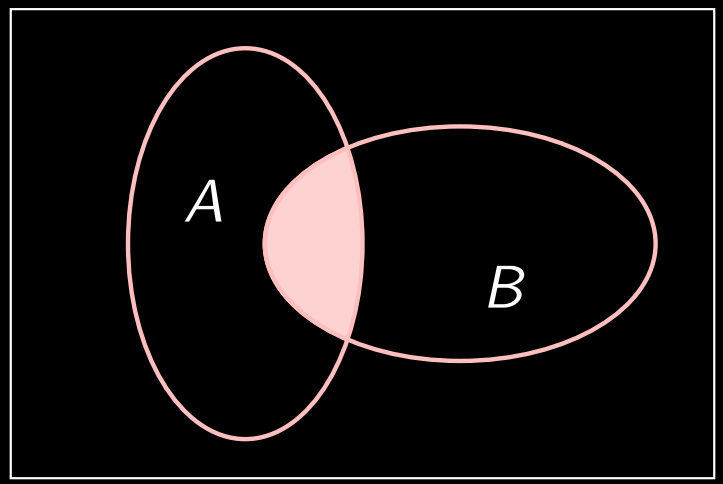
$A \cap B$

Définition 10

L'**intersection** des ensembles A et B est l'ensemble $A \cap B$ constitué des éléments communs à A et B.

$$A \cap B = \left\{ x \mid (x \in A) \wedge (x \in B) \right\}$$

C'est une partie de E. De plus $(x \in A \cap B) \leftrightarrow (x \in A \wedge x \in B)$.



$A \cap B$

Exercice 6

Démontrez que $(A \cap B = B) \leftrightarrow (B \subseteq A)$.

Définition 11

L'**union** ou la **réunion** des ensembles A et B est l'ensemble :

$$A \cup B = \left\{ x \mid (x \in A) \vee (x \in B) \right\}$$

C'est une partie de E. De plus $(x \in A \cup B) \leftrightarrow (x \in A \vee x \in B)$.
« ou non exclusif »

$A \cup B$

Définition 11

L'**union** ou la **réunion** des ensembles A et B est l'ensemble :

$$A \cup B = \left\{ x \mid (x \in A) \vee (x \in B) \right\}$$

C'est une partie de E. De plus $(x \in A \cup B) \leftrightarrow (x \in A \vee x \in B)$.
« ou non exclusif »

$A \cup B$

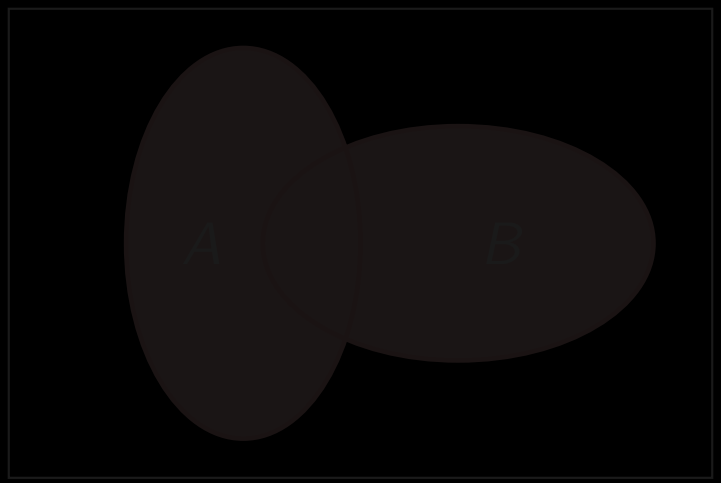
Définition 11

L'**union** ou la **réunion** des ensembles A et B est l'ensemble :

$$A \cup B = \left\{ x \mid (x \in A) \vee (x \in B) \right\}$$

C'est une partie de E. De plus $(x \in A \cup B) \leftrightarrow (x \in A \vee x \in B)$.

« ou non exclusif »



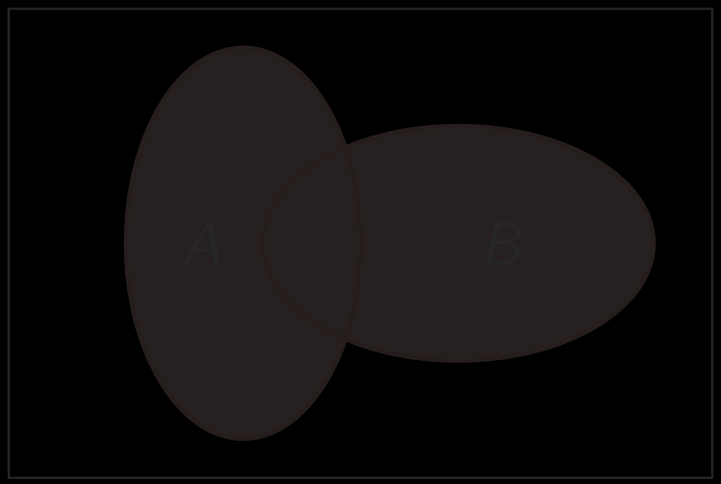
$A \cup B$

Définition 11

L'**union** ou la **réunion** des ensembles A et B est l'ensemble :

$$A \cup B = \left\{ x \mid (x \in A) \vee (x \in B) \right\}$$

C'est une partie de E. De plus $(x \in A \cup B) \leftrightarrow (x \in A \vee x \in B)$.
« ou non exclusif »



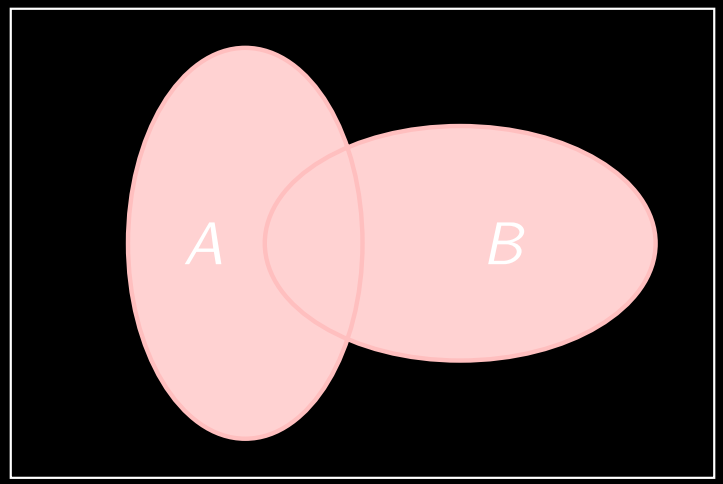
$A \cup B$

Définition 11

L'**union** ou la **réunion** des ensembles A et B est l'ensemble :

$$A \cup B = \left\{ x \mid (x \in A) \vee (x \in B) \right\}$$

C'est une partie de E. De plus $(x \in A \cup B) \leftrightarrow (x \in A \vee x \in B)$.
« ou non exclusif »



$A \cup B$

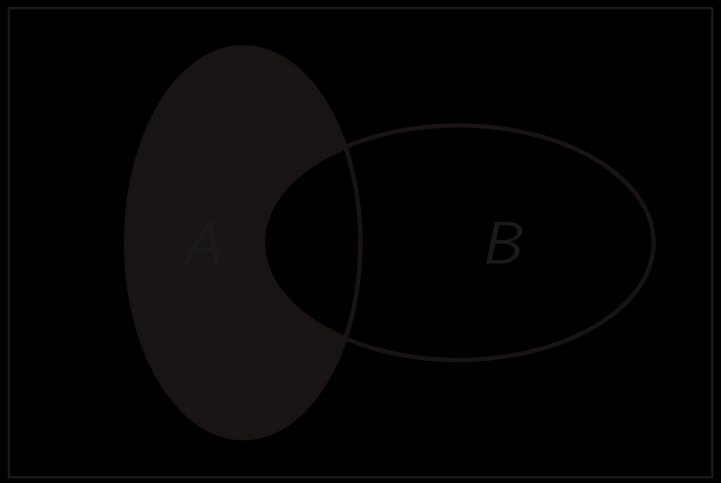
Exercice 7

Démontrez que $(A \cup B = A) \leftrightarrow (B \subseteq A)$.

Définition 12

La **différence** des ensembles A et B est l'ensemble

$$A \setminus B = \left\{ x \mid (x \in A) \wedge (x \notin B) \right\}$$

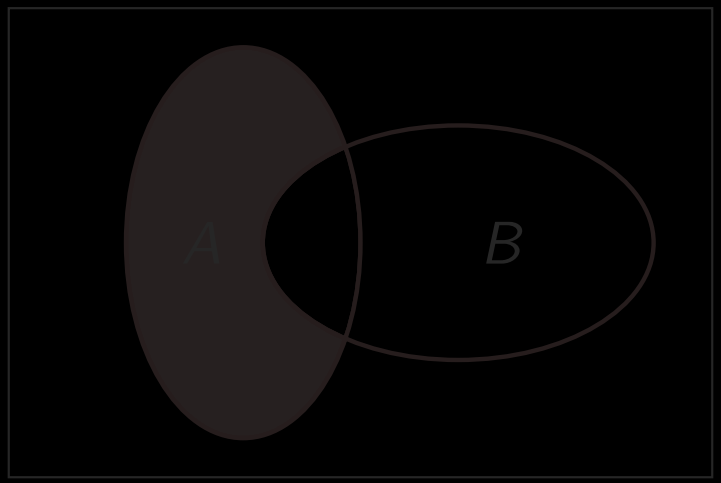


$A \setminus B$

Définition 12

La **différence** des ensembles A et B est l'ensemble

$$A \setminus B = \left\{ x \mid (x \in A) \wedge (x \notin B) \right\}$$

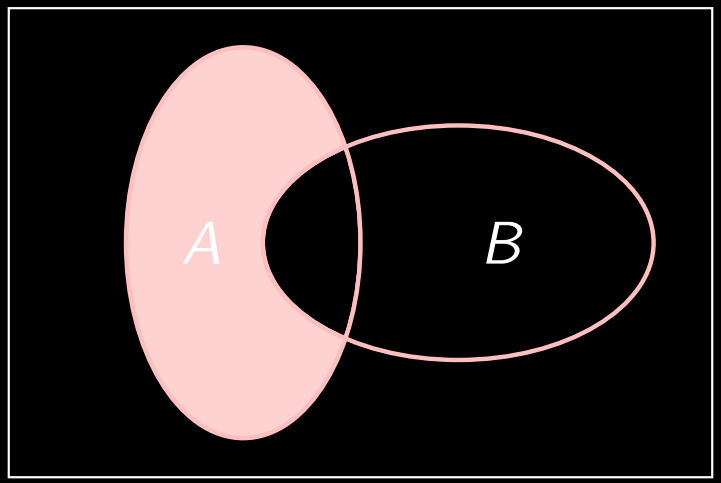


$A \setminus B$

Définition 12

La **différence** des ensembles A et B est l'ensemble

$$A \setminus B = \left\{ x \mid (x \in A) \wedge (x \notin B) \right\}$$



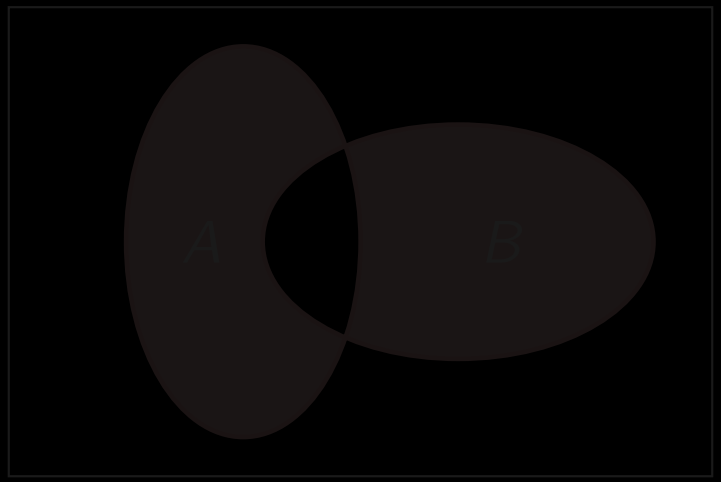
$A \setminus B$

Définition 13

La **différence symétrique** des ensembles A et B est l'ensemble :

$$A\Delta B = \left\{x \mid (x \in A \setminus B) \vee (x \in B \setminus A)\right\} = (A \setminus B) \cup (B \setminus A)$$

$$A\Delta B = \left\{x \mid (x \in A \cup B) \wedge (x \notin A \cap B)\right\} = (A \cup B) \setminus (A \cap B)$$



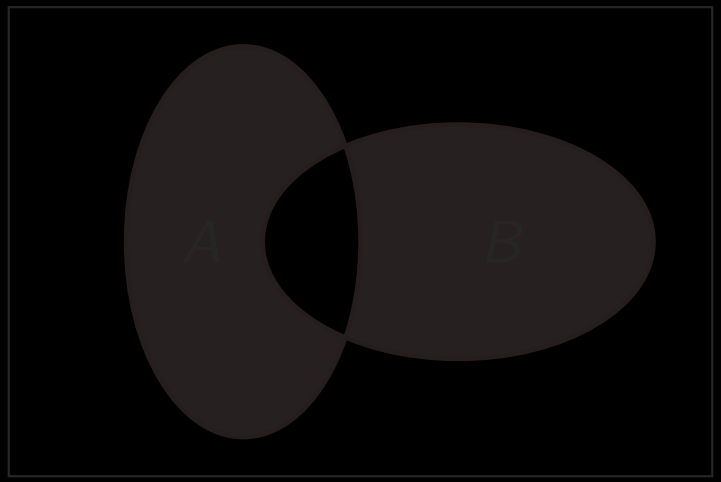
$A\Delta B$

Définition 13

La **différence symétrique** des ensembles A et B est l'ensemble :

$$A \Delta B = \left\{ x \mid (x \in A \setminus B) \vee (x \in B \setminus A) \right\} = (A \setminus B) \cup (B \setminus A)$$

$$A \Delta B = \left\{ x \mid (x \in A \cup B) \wedge (x \notin A \cap B) \right\} = (A \cup B) \setminus (A \cap B)$$



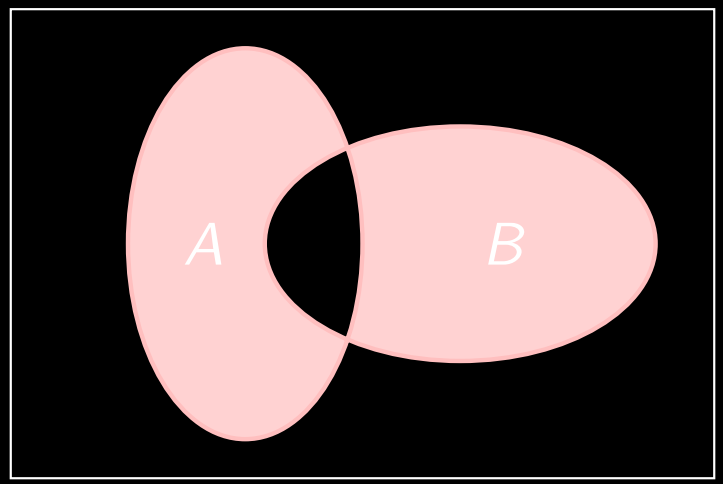
$A \Delta B$

Définition 13

La **différence symétrique** des ensembles A et B est l'ensemble :

$$A \Delta B = \left\{ x \mid (x \in A \setminus B) \vee (x \in B \setminus A) \right\} = (A \setminus B) \cup (B \setminus A)$$

$$A \Delta B = \left\{ x \mid (x \in A \cup B) \wedge (x \notin A \cap B) \right\} = (A \cup B) \setminus (A \cap B)$$



$A \Delta B$

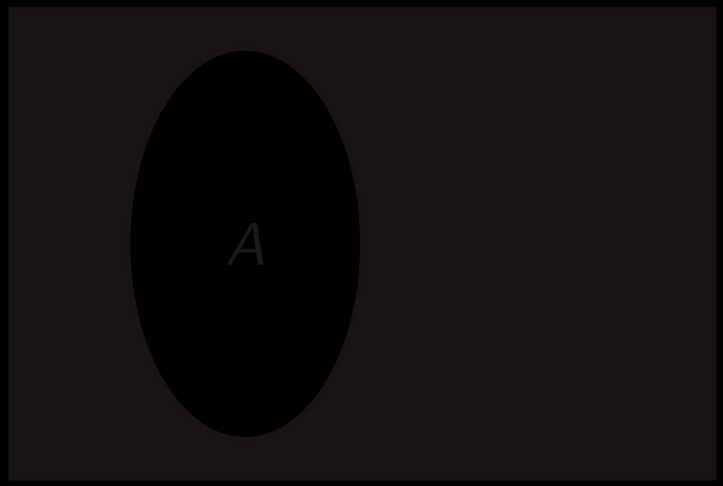
Remarque

La différence symétrique correspond au « ou exclusif » : fromage ou dessert...

Définition 14

A désignant une partie de E, le **complémentaire** de A par rapport à E ou le complémentaire de A dans E est l'ensemble noté $\complement_E A$ défini par

$$\complement_E A = E \setminus A = \left\{ x \mid (x \in E) \wedge (x \notin A) \right\}$$



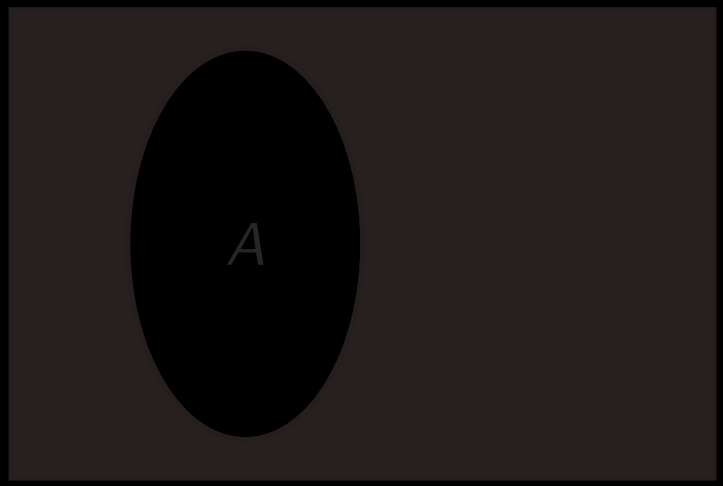
$\complement_E A$

S'il n'y a pas d'ambiguïté sur le référentiel E, $\complement_E A$ est noté $\overline{A}$ et $\overline{\overline{A}} = A$.

Définition 14

A désignant une partie de E, le **complémentaire** de A par rapport à E ou le complémentaire de A dans E est l'ensemble noté $\complement_E A$ défini par

$$\complement_E A = E \setminus A = \left\{ x \mid (x \in E) \wedge (x \notin A) \right\}$$



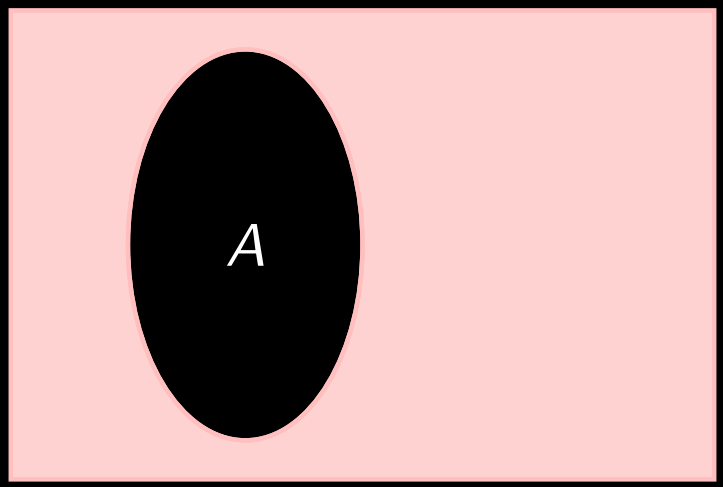
$\complement_E A$

S'il n'y a pas d'ambiguïté sur le référentiel E, $\complement_E A$ est noté $\overline{A}$ et $\overline{\overline{A}} = A$.

Définition 14

A désignant une partie de E, le **complémentaire** de A par rapport à E ou le complémentaire de A dans E est l'ensemble noté $\complement_E A$ défini par

$$\complement_E A = E \setminus A = \left\{ x \mid (x \in E) \wedge (x \notin A) \right\}$$

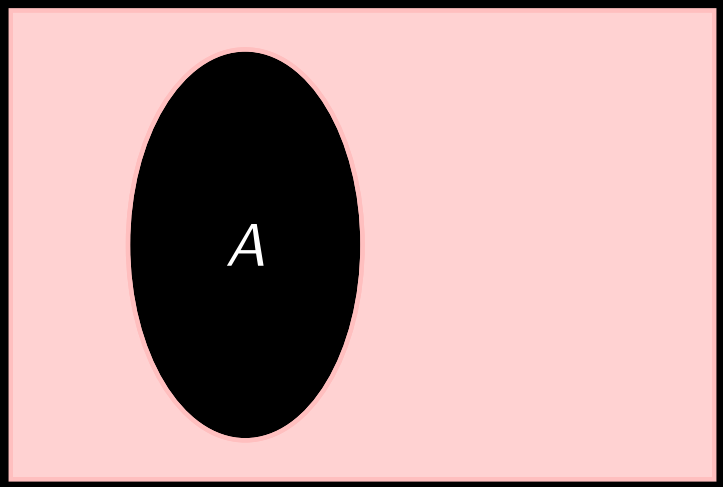


S'il n'y a pas d'ambiguïté sur le référentiel E, $\complement_E A$ est noté $\overline{A}$ et $\overline{\overline{A}} = A$.

Définition 14

A désignant une partie de E, le **complémentaire** de A par rapport à E ou le complémentaire de A dans E est l'ensemble noté $\complement_E A$ défini par

$$\complement_E A = E \setminus A = \left\{ x \mid (x \in E) \wedge (x \notin A) \right\}$$



$\complement_E A$

S'il n'y a pas d'ambiguïté sur le référentiel E, $\complement_E A$ est noté $\overline{A}$ et $\overline{\overline{A}} = A$.

Exercice 8

$A \setminus B = A \cap \overline{B}$ *et* $A \Delta B = \left(A \cap \overline{B}\right) \cup \left(\overline{A} \cap B\right)$.

Théorème 15 (Lois de De Morgan)

$$\overline{\left(\bigcup_{i=1}^{n} A_i\right)} = \bigcap_{i=1}^{n} \overline{A_i} \qquad et \qquad \overline{\left(\bigcap_{i=1}^{n} A_i\right)} = \bigcup_{i=1}^{n} \overline{A_i}$$

Ça se démontre...

Théorème 15 (Lois de De Morgan)

$$\overline{\left(\bigcup_{i=1}^{n} A_i\right)} = \bigcap_{i=1}^{n} \overline{A_i} \qquad et \qquad \overline{\left(\bigcap_{i=1}^{n} A_i\right)} = \bigcup_{i=1}^{n} \overline{A_i}$$

Ça se démontre...

Théorème 15 (Lois de De Morgan)

$$\overline{\left(\bigcup_{i=1}^{n} A_i\right)} = \bigcap_{i=1}^{n} \overline{A_i} \qquad et \qquad \overline{\left(\bigcap_{i=1}^{n} A_i\right)} = \bigcup_{i=1}^{n} \overline{A_i}$$

Ça se démontre...

Idempotence	$A \cup A = A$	$A \cap A = A$
Associativité	$(A \cup B) \cup C = A \cup (B \cup C)$	$(A \cap B) \cap C = A \cup (B \cap C)$
Commutativité	$A \cup B = B \cup A$	$A \cap B = B \cap A$
Distributivité	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$	$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
Identité	$A \cup \varnothing = A$	$A \cap E = A$
Involution	$\overline{\overline{A}} = A$	
Complémentaire	$A \cup \overline{A} = E$ $\overline{E} = \varnothing$	$A \cap \overline{A} = \varnothing$ $\overline{\varnothing} = E$
De Morgan	$\overline{A \cup B} = \overline{A} \cap \overline{B}$	$\overline{A \cap B} = \overline{A} \cup \overline{B}$

Exercice 9

Le principe de dualité affirme que si e est vérifiée pour tout élément de ¶(E), alors sa duale $e^\star$ aussi : ça se démontre...

Sommaire

Définition 16

E désignant ici un ensemble non vide. Soit P, une partie non vide de $P(E)$ (on a donc $P \subseteq P(E)$ et $P \neq \emptyset$) P est un ensemble non vide de parties de E. On dit que P est une **partition** de E si, et seulement si,

1. tout élément de P est non vide,
2. deux éléments distincts quelconques de P sont disjoints,
3. tout élément de E appartient à l'un des éléments de P.

Définition 16

E désignant ici un ensemble non vide. Soit P, une partie non vide de $P(E)$ (on a donc $P \subseteq P(E)$ et $P \neq \emptyset$) P est un ensemble non vide de parties de E. On dit que P est une **partition** de E si, et seulement si,

1. tout élément de P est non vide,
2. deux éléments distincts quelconques de P sont disjoints,
3. tout élément de E appartient à l'un des éléments de P.

Définition 16

E désignant ici un ensemble non vide. Soit P, une partie non vide de $P(E)$ (on a donc $P \subseteq P(E)$ et $P \neq \emptyset$) P est un ensemble non vide de parties de E. On dit que P est une **partition** de E si, et seulement si,

1. tout élément de P est non vide,
2. deux éléments distincts quelconques de P sont disjoints,
3. tout élément de E appartient à l'un des éléments de P.

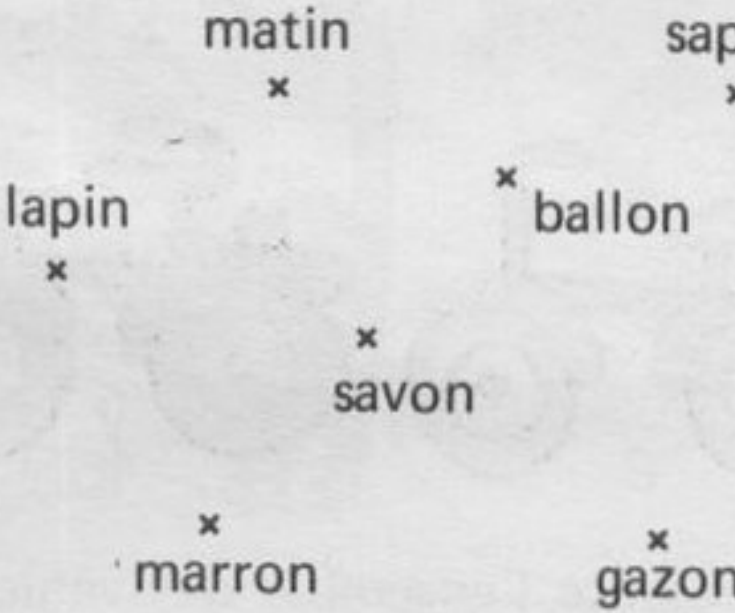
Définition 16

E désignant ici un ensemble non vide. Soit P, une partie non vide de $P(E)$ (on a donc $P \subseteq P(E)$ et $P \neq \emptyset$) P est un ensemble non vide de parties de E. On dit que P est une **partition** de E si, et seulement si,

1. tout élément de P est non vide,
2. deux éléments distincts quelconques de P sont disjoints,
3. tout élément de E appartient à l'un des éléments de P.

Recopie les mots.
Entoure en rouge l'ensemble C des mots où tu entends le son **in**.
Entoure en vert l'ensemble D des mots où tu entends le son **on**.
Entoure en bleu l'ensemble E des mots où tu entends le son **a**

Que peux-tu dire de l'ensemble E ?
(fais une phrase où tu utiliseras E, C, D).



Haskell

```
λ> Data.List.partition (\x→ x < 0) [-1,5,-65,8,5,-12]
([-1,-65,-12],[5,8,5])
```

Sommaire

- l'ensemble des entrées :

$$E = \{\text{Cuisses de sauterelles panées},\ \text{œuf mou},\ \text{huîtres de l'Erdre}\}$$

- l'ensemble des plats de résistance :

$$P = \{\text{Turbot à l'huile de ricin},\ \text{Chien à l'andalouse},\ \text{Soupe d'orties}\}$$

- l'ensemble des desserts :

$$D = \{\text{Pomme}, \text{Banane}, \text{Noix}\}$$

Haskell

```haskell
data Entree  = Cuisses | Oeufs | Huitres
             deriving (Enum, Show)
data Plat    = Turbot | Chien | Soupe
             deriving (Enum, Show)
data Dessert = Pomme | Banane | Noix
             deriving (Enum, Show)

lesEntrees  = [Cuisses .. ]
lesPlats    = [Turbot .. ]
lesDesserts = [Pomme .. ]

lesMenus = [(e,p,d) | e←lesEntrees, p←lesPlats,
    d←lesDesserts]
```

Haskell

```
λ> lesMenus
[(Cuisses,Turbot,Pomme),(Cuisses,Turbot,Banane),(Cuisses,Turbot,No
 (Cuisses,Chien,Pomme),(Cuisses,Chien,Banane),(Cuisses,Chien,Noix)
 (Cuisses,Soupe,Pomme),(Cuisses,Soupe,Banane),(Cuisses,Soupe,Noix)
 (Oeufs,Turbot,Pomme),(Oeufs,Turbot,Banane),(Oeufs,Turbot,Noix),
 (Oeufs,Chien,Pomme),(Oeufs,Chien,Banane),(Oeufs,Chien,Noix),
 (Oeufs,Soupe,Pomme),(Oeufs,Soupe,Banane),(Oeufs,Soupe,Noix),
 (Huitres,Turbot,Pomme),(Huitres,Turbot,Banane),(Huitres,Turbot,No
 (Huitres,Chien,Pomme),(Huitres,Chien,Banane),(Huitres,Chien,Noix)
 (Huitres,Soupe,Pomme),(Huitres,Soupe,Banane),(Huitres,Soupe,Noix)
    ]
λ> length lesMenus
27
λ> :t lesMenus
lesMenus :: [(Entree, Plat, Dessert)]
```

Définition 17 (Paire ordonnée)

On note $\langle a, b \rangle = \{\{a\}, \{a, b\}\}$

Théorème 18

$$(\{a, b\} =_{ens} \{x, y\}) = ((a = x \wedge b = y) \vee (a = y \wedge b = x))$$

Définition 17 (Paire ordonnée)

On note $\langle a, b \rangle = \{\{a\}, \{a, b\}\}$

Théorème 18

$$\left(\{a, b\} =_{ens} \{x, y\}\right) \equiv \left((a = x \wedge b = y) \vee (a = y \wedge b = x)\right)$$

Définition 17 (Paire ordonnée)

On note $\langle a, b\rangle = \{\{a\}, \{a, b\}\}$

Théorème 18

$$(\{a, b\} =_{ens} \{x, y\}) \equiv ((a = x \wedge b = y) \vee (a = y \wedge b = x))$$

Théorème 19

$$(\langle a, b\rangle =_{paire} \langle x, y\rangle) \equiv (a = x \wedge b = y)$$

Définition 17 (Paire ordonnée)

On note $\langle a, b\rangle = \{\{a\}, \{a, b\}\}$

Théorème 18

$$(\{a, b\} =_{ens} \{x, y\}) \equiv ((a = x \wedge b = y) \vee (a = y \wedge b = x))$$

Théorème 19

$$(\langle a, b\rangle =_{paire} \langle x, y\rangle) \equiv (a = x \wedge b = y)$$

Définition 17 (Paire ordonnée)

On note $\langle a, b\rangle = \{\{a\}, \{a, b\}\}$

Théorème 18

$$(\{a, b\} =_{ens} \{x, y\}) \equiv ((a = x \wedge b = y) \vee (a = y \wedge b = x))$$

Théorème 19

$$(\langle a, b\rangle =_{paire} \langle x, y\rangle) \equiv (a = x \wedge b = y)$$

Définition 20

$$A \times B = \{(a, b) \mid (a \in A) \wedge (b \in B)\}$$

Définition 17 (Paire ordonnée)

On note $\langle a, b\rangle = \{\{a\}, \{a, b\}\}$

Théorème 18

$$(\{a, b\} =_{ens} \{x, y\}) \equiv ((a = x \wedge b = y) \vee (a = y \wedge b = x))$$

Théorème 19

$$(\langle a, b\rangle =_{paire} \langle x, y\rangle) \equiv (a = x \wedge b = y)$$

Définition 20

$$A \times B = \{(a, b) \mid (a \in A) \wedge (b \in B)\}$$

Définition 17 (Paire ordonnée)

On note $\langle a, b\rangle = \{\{a\}, \{a, b\}\}$

Théorème 18

$$(\{a, b\} =_{ens} \{x, y\}) \equiv ((a = x \wedge b = y) \vee (a = y \wedge b = x))$$

Théorème 19

$$(\langle a, b\rangle =_{paire} \langle x, y\rangle) \equiv (a = x \wedge b = y)$$

Définition 20

$$A \times B = \{(a, b) \mid (a \in A) \wedge (b \in B)\}$$

$\otimes$	1	2	3	4	5
a	$(a,1)$	$(a,2)$	$(a,3)$	$(a,4)$	$(a,5)$
b	$(b,2)$	$(b,2)$	$(b,3)$	$(b,4)$	$(b,5)$
c	$(c,1)$	$(c,2)$	$(c,3)$	$(c,4)$	$(c,5)$

Haskell

```
λ> [(x,y) | x←['a','b','c'], y←[1,2,3,4,5]]
[('a',1),('a',2),('a',3),('a',4),('a',5),
 ('b',1),('b',2),('b',3),('b',4),('b',5),
 ('c',1),('c',2),('c',3),('c',4),('c',5)]

λ> [(x,y) | x←['a','b','c'], y←[1,2,3,4,5], mod y 2 == 0]
[('a',2),('a',4),('b',2),('b',4),('c',2),('c',4)]
```

Définissons temporairement un triplet ordonné $\langle a, b, c\rangle$ par :
$\langle a, b, c\rangle = \langle a, \langle b, c\rangle\rangle$.

Exercice 10

Est-ce que c'est licite ? C'est-à-dire est-ce que c'est une définition...qui définit vraiment un triplet ordonné ? Qu'est-ce que ça veut dire en fait « définit vraiment un triplet ordonné » ?

Comment va-t-on faire alors pour un quadruplet ? Un n-uplet ?

Qu'est-ce que c'est que ça : $E_1 \times E_2 \times \cdots \times E_n = \prod_{i=1}^{n} E_i$?

Définissons temporairement un triplet ordonné $\langle a, b, c\rangle$ par :
$\langle a, b, c\rangle = \langle a, \langle b, c\rangle\rangle$.

Exercice 10

Est-ce que c'est licite ? C'est-à-dire est-ce que c'est une définition...qui définit vraiment un triplet ordonné ? Qu'est-ce que ça veut dire en fait « définit vraiment un triplet ordonné » ?
Comment va-t-on faire alors pour un quadruplet ? Un n-uplet ?
Qu'est-ce que c'est que ça : $E_1 \times E_2 \times \cdots \times E_n = \prod_{i=1}^{n} E_i$ *?*

Sommaire

$\mathrm{Card}\,(E)$ ou $|E|$ ou $\#E$

Théorème 21 (Propriétés des cardinaux des ensembles finis)

- *Si $A \subseteq B$ alors $|A| \leqslant |B|$. Une conséquence intéressante est celle ci : si $A \subseteq B$ avec $|A| = |B|$ alors $A = B$.*
- $|A - B| \leqslant |A|$.
- $|A \cup B| = |A| + |B| - |A \cap B|$
- *Si on a $E_i \cap E_j = \varnothing$ lorsque $i \neq j$, alors* $\left|\bigcup_{i=1}^{n} E_i\right| = \sum_{i=1}^{n} |E_i|$.
- $|E_1 \times E_2| = |E_1| . |E_2|$ *et*

$$|E_1 \times E_2 \times \cdots \times E_n| = |E_1| . |E_2| \cdots |E_n|$$

$$\left|E^n\right| = (|E|)^n$$

- *La formule suivante est absolument à connaître :*

$$|P(E)| = 2^{|E|}$$

Nous la démontrerons en exercice.

Théorème 21 (Propriétés des cardinaux des ensembles finis)

- *Si $A \subseteq B$ alors $|A| \leqslant |B|$. Une conséquence intéressante est celle ci : si $A \subseteq B$ avec $|A| = |B|$ alors $A = B$.*
- $|A - B| \leqslant |A|$.
- $|A \cup B| = |A| + |B| - |A \cap B|$
- ***Si on a** $E_i \cap E_j = \emptyset$ **lorsque** $i \neq j$, alors* $\left|\bigcup_{i=1}^{n} E_i\right| = \sum_{i=1}^{n} |E_i|$.
- $|E_1 \times E_2| = |E_1| . |E_2|$ *et*

$$|E_1 \times E_2 \times \cdots \times E_n| = |E_1| . |E_2| \cdots |E_n|$$

$$|E^n| = (|E|)^n$$

- *La formule suivante est absolument à connaître :*

$$|P(E)| = 2^{|E|}$$

Nous la démontrerons en exercice.

Théorème 21 (Propriétés des cardinaux des ensembles finis)

- *Si* $A \subseteq B$ *alors* $|A| \leqslant |B|$. *Une conséquence intéressante est celle ci : si* $A \subseteq B$ *avec* $|A| = |B|$ *alors* $A = B$.
- $|A - B| \leqslant |A|$.
- $|A \cup B| = |A| + |B| - |A \cap B|$
- ***Si on a*** $E_i \cap E_j = \varnothing$ ***lorsque*** $i \neq j$, *alors* $\left|\bigcup_{i=1}^{n} E_i\right| = \sum_{i=1}^{n} |E_i|$.
- $|E_1 \times E_2| = |E_1|.|E_2|$ *et*

$$|E_1 \times E_2 \times \cdots \times E_n| = |E_1|.|E_2|.\cdots.|E_n|$$
$$|E^n| = (|E|)^n$$

- *La formule suivante est absolument à connaître :*

$$|P(E)| = 2^{|E|}$$

Nous la démontrerons en exercice.

Théorème 21 (Propriétés des cardinaux des ensembles finis)

- *Si* $A \subseteq B$ *alors* $|A| \leqslant |B|$. *Une conséquence intéressante est celle ci : si* $A \subseteq B$ *avec* $|A| = |B|$ *alors* $A = B$.
- $|A - B| \leqslant |A|$.
- $|A \cup B| = |A| + |B| - |A \cap B|$
- ***Si on a*** $E_i \cap E_j = \varnothing$ ***lorsque*** $i \neq j$, *alors* $\left|\bigcup_{i=1}^{n} E_i\right| = \sum_{i=1}^{n} |E_i|$.
- $|E_1 \times E_2| = |E_1|.|E_2|$ *et*

$$\begin{aligned} |E_1 \times E_2 \times \cdots \times E_n| &= |E_1|.|E_2|.\cdots.|E_n| \\ |E^n| &= (|E|)^n \end{aligned}$$

- *La formule suivante est absolument à connaître :*

$$|P(E)| = 2^{|E|}$$

Nous la démontrerons en exercice.

Théorème 21 (Propriétés des cardinaux des ensembles finis)

- *Si* $A \subseteq B$ *alors* $|A| \leqslant |B|$. *Une conséquence intéressante est celle ci : si* $A \subseteq B$ *avec* $|A| = |B|$ *alors* $A = B$.
- $|A - B| \leqslant |A|$.
- $|A \cup B| = |A| + |B| - |A \cap B|$
- ***Si on a*** $E_i \cap E_j = \varnothing$ ***lorsque*** $i \neq j$, *alors* $\left|\bigcup_{i=1}^{n} E_i\right| = \sum_{i=1}^{n} |E_i|$.
- $|E_1 \times E_2| = |E_1|.|E_2|$ *et*

$$\begin{aligned} |E_1 \times E_2 \times \cdots \times E_n| &= |E_1|.|E_2|.\cdots.|E_n| \\ |E^n| &= (|E|)^n \end{aligned}$$

- *La formule suivante est absolument à connaître :*

$$|P(E)| = 2^{|E|}$$

Nous la démontrerons en exercice.

Théorème 21 (Propriétés des cardinaux des ensembles finis)

- *Si* $A \subseteq B$ *alors* $|A| \leqslant |B|$. *Une conséquence intéressante est celle ci : si* $A \subseteq B$ *avec* $|A| = |B|$ *alors* $A = B$.
- $|A - B| \leqslant |A|$.
- $|A \cup B| = |A| + |B| - |A \cap B|$
- ***Si on a*** $E_i \cap E_j = \varnothing$ ***lorsque*** $i \neq j$, *alors* $\left|\bigcup_{i=1}^{n} E_i\right| = \sum_{i=1}^{n} |E_i|$.
- $|E_1 \times E_2| = |E_1|.|E_2|$ *et*

$$\begin{aligned} |E_1 \times E_2 \times \cdots \times E_n| &= |E_1|.|E_2|.\cdots.|E_n| \\ |E^n| &= \left(|E|\right)^n \end{aligned}$$

- *La formule suivante est absolument à connaître :*

$$|P(E)| = 2^{|E|}$$

Nous la démontrerons en exercice.

Théorème 21 (Propriétés des cardinaux des ensembles finis)

- *Si* $A \subseteq B$ *alors* $\left|A\right| \leqslant \left|B\right|$. *Une conséquence intéressante est celle ci : si* $A \subseteq B$ *avec* $\left|A\right| = \left|B\right|$ *alors* $A = B$.
- $\left|A - B\right| \leqslant \left|A\right|$.
- $\left|A \cup B\right| = \left|A\right| + \left|B\right| - \left|A \cap B\right|$
- ***Si on a*** $E_i \cap E_j = \varnothing$ ***lorsque*** $i \neq j$, *alors* $\left|\bigcup_{i=1}^{n} E_i\right| = \sum_{i=1}^{n} \left|E_i\right|$.
- $\left|E_1 \times E_2\right| = \left|E_1\right|.\left|E_2\right|$ *et*

$$\begin{aligned} \left|E_1 \times E_2 \times \cdots \times E_n\right| &= \left|E_1\right|.\left|E_2\right|.\cdots.\left|E_n\right| \\ \left|E^n\right| &= \left(\left|E\right|\right)^n \end{aligned}$$

- *La formule suivante est absolument à connaître :*

$$\left|\ P\left(E\right)\right| = 2^{|E|}$$

Nous la démontrerons en exercice.

Exercice 11

Une petite question d'anglais de spécialité... : nos amis anglo-saxons appellent ¶(E) power set of E. Pourquoi ?...

Exercice 12

Comment calculer récursivement le cardinal d'un ensemble ?

Haskell

```
cardinal :: (Eq a)⇒Ens a→ Int
cardinal Vide = 0
cardinal (Ens (t,q)) = 1 + (cardinal q)
```

Exercice 11

Une petite question d'anglais de spécialité... : nos amis anglo-saxons appellent ¶(E) power set of E. Pourquoi ?...

Exercice 12

Comment calculer récursivement le cardinal d'un ensemble ?

Haskell

```
cardinal :: (Eq a)⇒Ens a→ Int
cardinal Vide = 0
cardinal (Ens (t,q)) = 1 + (cardinal q)
```

Exercice 11

Une petite question d'anglais de spécialité... : nos amis anglo-saxons appellent ¶(E) power set of E. Pourquoi ?...

Exercice 12

Comment calculer récursivement le cardinal d'un ensemble ?

Haskell

```
cardinal :: (Eq a)⇒Ens a→ Int
cardinal Vide = 0
cardinal (Ens (t,q)) = 1 + (cardinal q)
```

Sommaire

Définition 22

Soit E un ensemble. Pour toute partie A de E on définit la fonction caractéristique $\mathbb{1}_A$ de A dans E par :

$$\mathbb{1}_A : \begin{array}{ccc} E & \to & \{0;1\} \\ x & \mapsto & \begin{cases} 1 & \text{si } x \in A \\ 0 & \text{si } x \notin A \end{cases} \end{array}$$

$$\mathbb{1}_A = \mathbb{1}_B \iff A = B$$

Définition 22

Soit E un ensemble. Pour toute partie A de E on définit la fonction caractéristique $\mathbb{1}_A$ de A dans E par :

$$\mathbb{1}_A : \begin{array}{ccc} E & \to & \{0;1\} \\ x & \mapsto & \begin{cases} 1 & \text{si } x \in A \\ 0 & \text{si } x \notin A \end{cases} \end{array}$$

$$\mathbb{1}_A = \mathbb{1}_B \iff A = B$$

Définition 22

Soit E un ensemble. Pour toute partie A de E on définit la fonction caractéristique $\mathbb{1}_A$ de A dans E par :

$$\mathbb{1}_A : \begin{array}{ccc} E & \to & \{0;1\} \\ x & \mapsto & \begin{cases} 1 & \text{si } x \in A \\ 0 & \text{si } x \notin A \end{cases} \end{array}$$

$$\mathbb{1}_A = \mathbb{1}_B \iff A = B$$

Haskell

```haskell
module EnsembleCompréhension where

type Element  = Int
type Predicat = Element→ Bool
data Ensemble = Ens (Predicat)

contient :: Ensemble→ Element→ Bool
contient    (Ens f)     e        =  f e

insere :: Element→ Ensemble→ Ensemble
insere    e          ens      =  Ens (\x→ (x == e) || (ens
    `contient` x))

union ::  Ensemble→ Ensemble→ Ensemble
union     ens1        ens2     =  Ens (\x→ (ens1 `contient` x)
    || (ens2 `contient` x))
```

Haskell

```
filtre :: Predicat→ Ensemble→ Ensemble
filtre    predic       ens       =  Ens (\x→ (ens `contient` x)
    && (predic x))

complementaire :: Ensemble→ Ensemble
complementaire    (Ens f)  =  (Ens (not . f))

difference :: Ensemble→ Ensemble→ Ensemble
difference    ens1         ens2     =  ens1 `inter`
    (complementaire ens2)

estInclus :: Ensemble→ Ensemble→ Bool
estInclus    ens1         ens2     =  estVide (Ens (\x→ (ens1
    `contient` x) &&  not (ens2 `contient` x)))

estEgal :: Ensemble→ Ensemble→ Bool
estEgal    ens1         ens2     =  (ens1 `estInclus` ens2) &&
    (ens2 `estInclus` ens1)
```

Haskell

```haskell
-- Pour un joli affichage et le test de vacuité

listeUnivers :: [Element]
listeUnivers =  [mini .. maxi]

listeVersEns :: Ensemble→[Element]
listeVersEns    (Ens f)  =  [x | x←listeUnivers, f x]

instance  Show Ensemble where
                    show = show . listeVersEns

estVide :: Ensemble→Bool
estVide    ens      =  listeVersEns ens == []
```

Haskell

```
mini,maxi  :: Element
(mini,maxi) = (-128,127)

univers :: Ensemble
univers =  Ens (λn→ n ≥ mini && n ≤ maxi)
```

Haskell

```
λ> let e' = Ens (λn→ univers `contient` (n^2) )
λ> e'
[-11,-10,-9,-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10,11]

λ> let naturels = filtre (> 0) univers

λ> ilExiste (λn→ 2^11 - 1 `mod` n == 0) naturels
False
```

Sommaire

Haskell

```haskell
data Booleen = Faux | Vrai

non :: Booleen → Booleen
non    Vrai      =  Faux
non    Faux      =  Vrai
```

Définition 23 (Graphe d'une fonction)

Le graphe d'une fonction est l'ensemble de tous les couples :
⟨ élément du départ, son image dans l'arrivée ⟩
ou si vous préférez :

$$\mathcal{G}(f) = \{\langle x, f(x)\rangle \mid x \in \text{départ}\}$$

$$f = \Big\langle Booleen, Booleen, \{\langle Vrai, Faux\rangle, \langle Faux, Vrai\rangle\}\Big\rangle$$

ATTENTION !

Une fonction est donc un triplet $\langle$ Départ, Arrivée, Graphe$\rangle$


T'AS COMPRIS OU FAUT
QUE JE CONTINUE ?

Haskell

```
et1 :: Booleen→ Booleen→ Booleen
et1    Vrai       Vrai    =  Vrai
et1    _          _       =  Faux
```

Haskell

```
et2 :: Booleen→ Booleen→ Booleen
et2    Vrai        x        =  x
et2    Faux        x        =  Faux
```

$$\begin{array}{rcl} & \mathit{Booleen} \times \mathit{Booleen} & \rightarrow \quad \mathit{Booleen} \\ \text{et:} & & \\ & (x, y) & \mapsto \quad \begin{cases} \text{si } x == \text{ Vrai alors } y \\ \text{sinon Faux} \end{cases} \end{array}$$

ATTENTION !

Sur Haskell , toutes les fonctions sont d'UNE SEULE VARIABLE !

Haskell

```
et3 :: Booleen→ (Booleen→ Booleen)
et3    Vrai        x        =  x
et3    Faux        x        =  Faux
```

Haskell

```
λ> :t et3 Vrai
et3 Vrai :: Booleen→ Booleen
```

Haskell

```
et3 :: Booleen→ (Booleen→ Booleen)
et3    Vrai        x        =  x
et3    Faux        x        =  Faux
```

Haskell

```
λ> :t et3 Vrai
et3 Vrai :: Booleen→ Booleen
```

Haskell

```
idem :: Booleen→ Booleen
idem    x        =  x
```

Haskell

```
tjsFaux :: Booleen→ Booleen
tjsFaux    _          = Faux
```

Haskell

```
et4 :: Booleen→ Booleen→ Booleen
et4    Vrai     =  idem
et4    Faux     =  tjsFaux
```

Haskell

```
idem :: Booleen→ Booleen
idem    x        =  x
```

Haskell

```
tjsFaux :: Booleen→ Booleen
tjsFaux    _         = Faux
```

Haskell

```
et4 :: Booleen→ Booleen→ Booleen
et4    Vrai     =  idem
et4    Faux     =  tjsFaux
```

Haskell

```
idem :: Booleen→ Booleen
idem    x        =  x
```

Haskell

```
tjsFaux :: Booleen→ Booleen
tjsFaux    _         = Faux
```

Haskell

```
et4 :: Booleen→ Booleen→ Booleen
et4    Vrai     =  idem
et4    Faux     =  tjsFaux
```

Haskell

```
et5 :: Booleen→ Booleen→ Booleen
et5    Vrai     =  λx→ x
et5    Faux     =  λx→ Faux
```

Haskell

```
et6 :: Booleen→ Booleen→ Booleen
et6 y x = if y == Vrai
          then x
          else Faux
```

Haskell

```haskell
placeDeCinema :: Int→ String
placeDeCinema age
    | age < 0    = "Euh, t'as bu quoi au petit déjeuner?"
    | age < 12   = "Pour toi c'est 2 euros et viens prendre un
      bonbon"
    | age < 18   = "Pour toi c'est 8 euros"
    | otherwise = "T'es pas un peu vieux pour aller voir Petit
      Poney?"
```

Haskell

```
et7 :: Booleen→ Booleen→ Booleen
et7 x y
    |x == Vrai = x
    |otherwise = Faux
```

Sommaire

Exemple 24

Décrire une fonction qui étant donnés quatre flottants leur associe la moyenne des deux nombres parmi les quatre qui ne sont ni le plus grand ni le plus petit. Par exemple, la moyenne olympique de 10, 8, 12 et 14 est 11 et celle de 12, 12, 12 et 12 est 12.

Spécification

Haskell

```
moyenne_olympique4 :: Float→ Float→ Float→ Float→ Float
-- renvoie la moyenne olympique de 4 flottants sous forme d'un
    flottant
```

Réalisation

Nous allons par exemple additionner les quatre nombres, retirer le plus grand et le plus petit puis diviser par 2.

Haskell

```
λ> :t min
min :: Ord a⇒a→ a→ a
```

Haskell

```
λ> min 'a' 'b'
'a'
λ> min 5 3
3
λ> min "Tralala" "Pouet Pouet"
"Pouet Pouet"
λ> min True False
False
```

Réalisation

Nous allons par exemple additionner les quatre nombres, retirer le plus grand et le plus petit puis diviser par 2.

Haskell

```
λ> :t min
min :: Ord a⇒a→ a→ a
```

Haskell

```
λ> min 'a' 'b'
'a'
λ> min 5 3
3
λ> min "Tralala" "Pouet Pouet"
"Pouet Pouet"
λ> min True False
False
```

Réalisation

Nous allons par exemple additionner les quatre nombres, retirer le plus grand et le plus petit puis diviser par 2.

Haskell

```
λ> :t min
min :: Ord a⇒a→ a→ a
```

Haskell

```
λ> min 'a' 'b'
'a'
λ> min 5 3
3
λ> min "Tralala" "Pouet Pouet"
"Pouet Pouet"
λ> min True False
False
```

Réalisation

Haskell

```
moyenne_olympique4 a b c d =
  (s - mini - maxi) / 2 -- l'expression correspondant à la
      méthode choisie
  where s = a + b + c + d -- la somme des 4 nombres
        mini = min a (min  b (min c d) ) -- le plus petit des 4
        maxi = max a (max  b (max c d) ) -- le plus grand des 4
```

Haskell

```
λ> moyenne_olympique4 10 8 12 14
11.0
```

Réalisation

Haskell

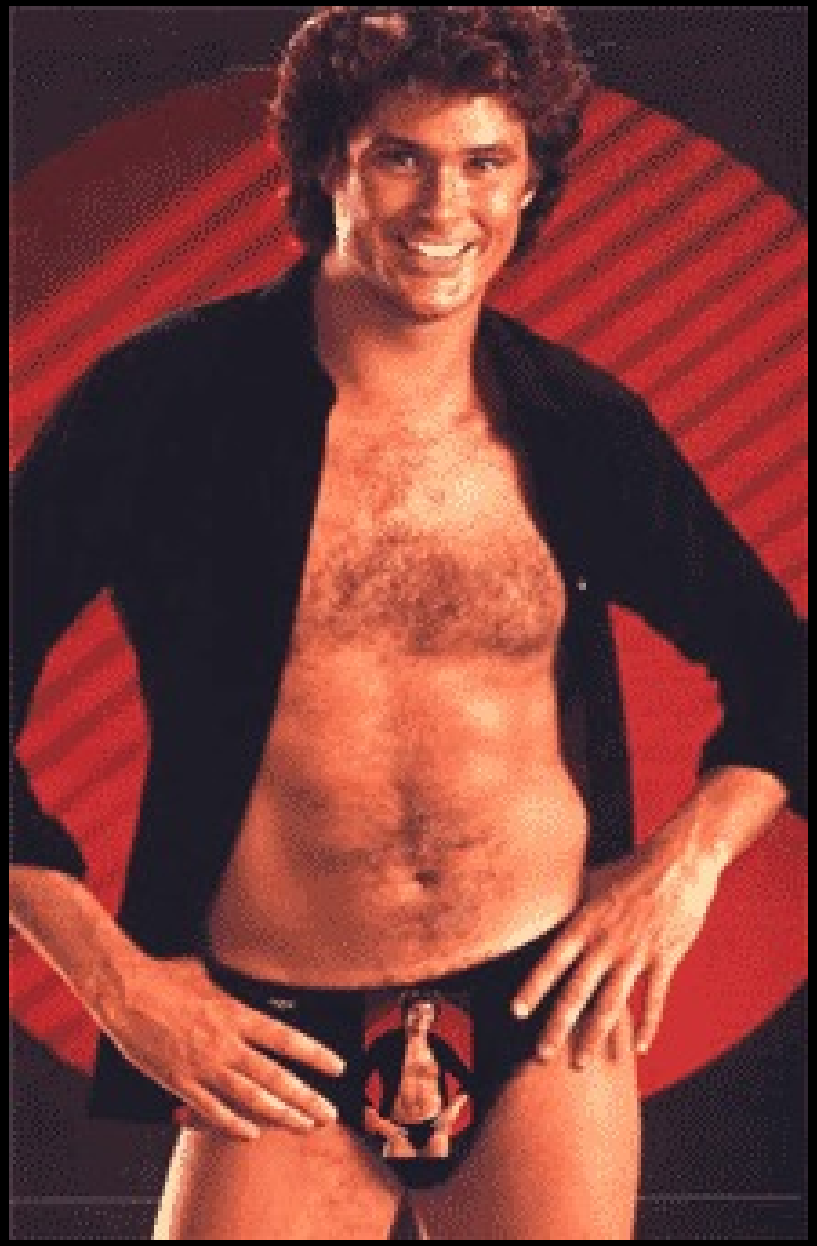
```
moyenne_olympique4 a b c d =
  (s - mini - maxi) / 2 -- l'expression correspondant à la
      méthode choisie
  where s = a + b + c + d -- la somme des 4 nombres
        mini = min a (min  b (min c d) ) -- le plus petit des 4
        maxi = max a (max  b (max c d) ) -- le plus grand des 4
```

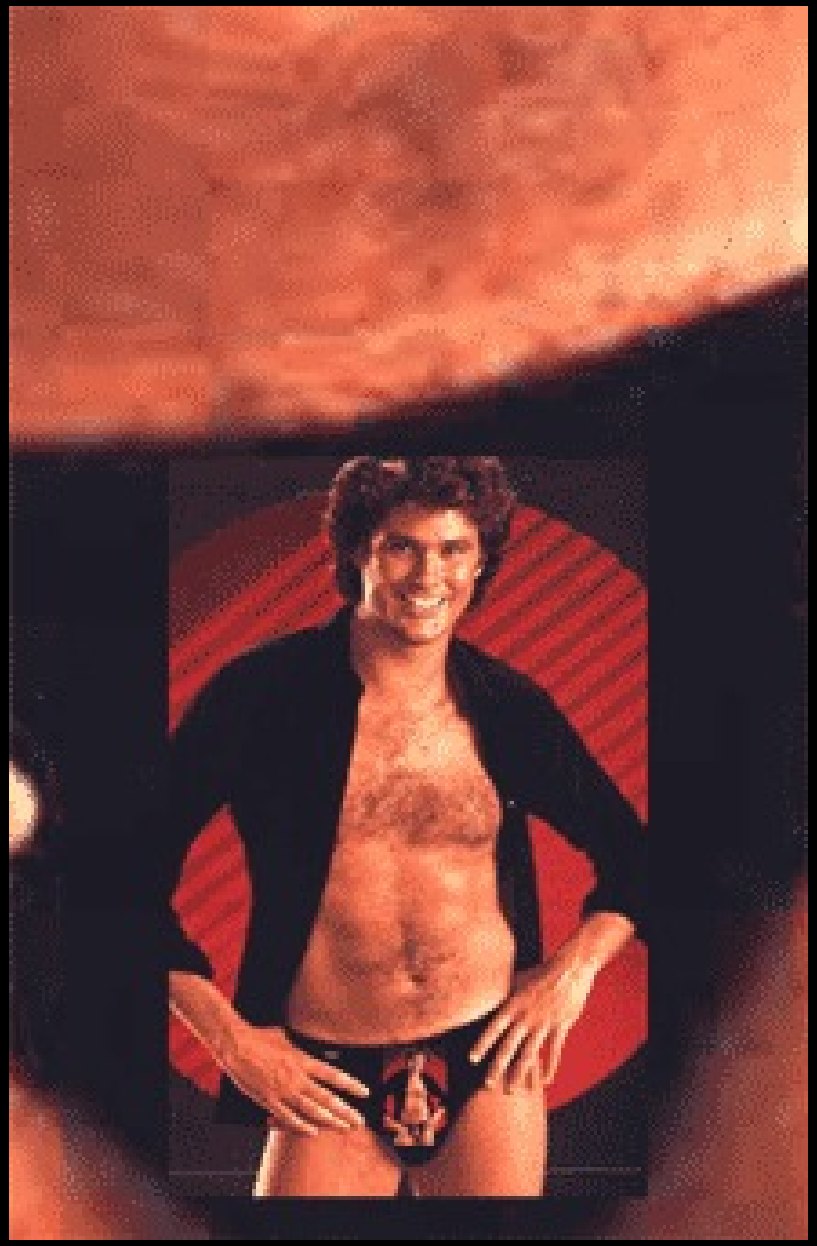
Haskell

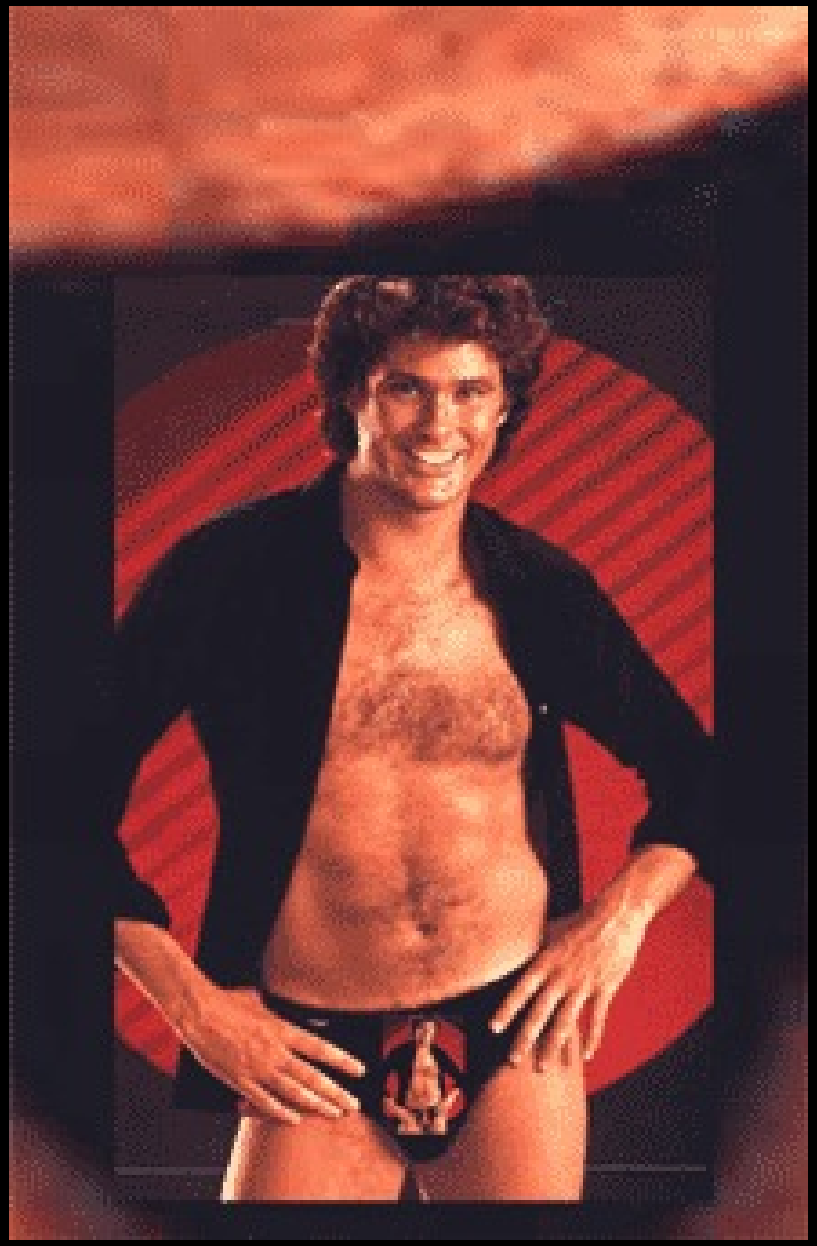
```
λ> moyenne_olympique4 10 8 12 14
11.0
```

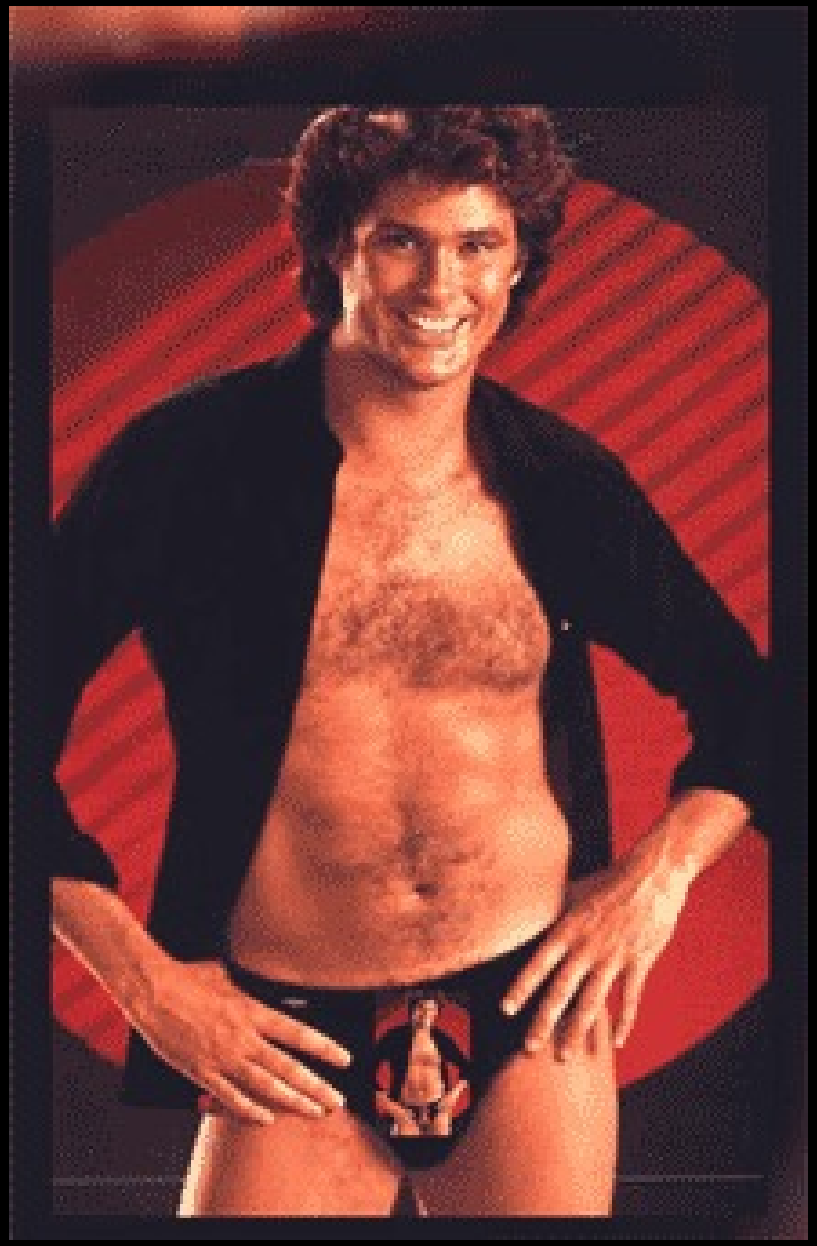
Sommaire

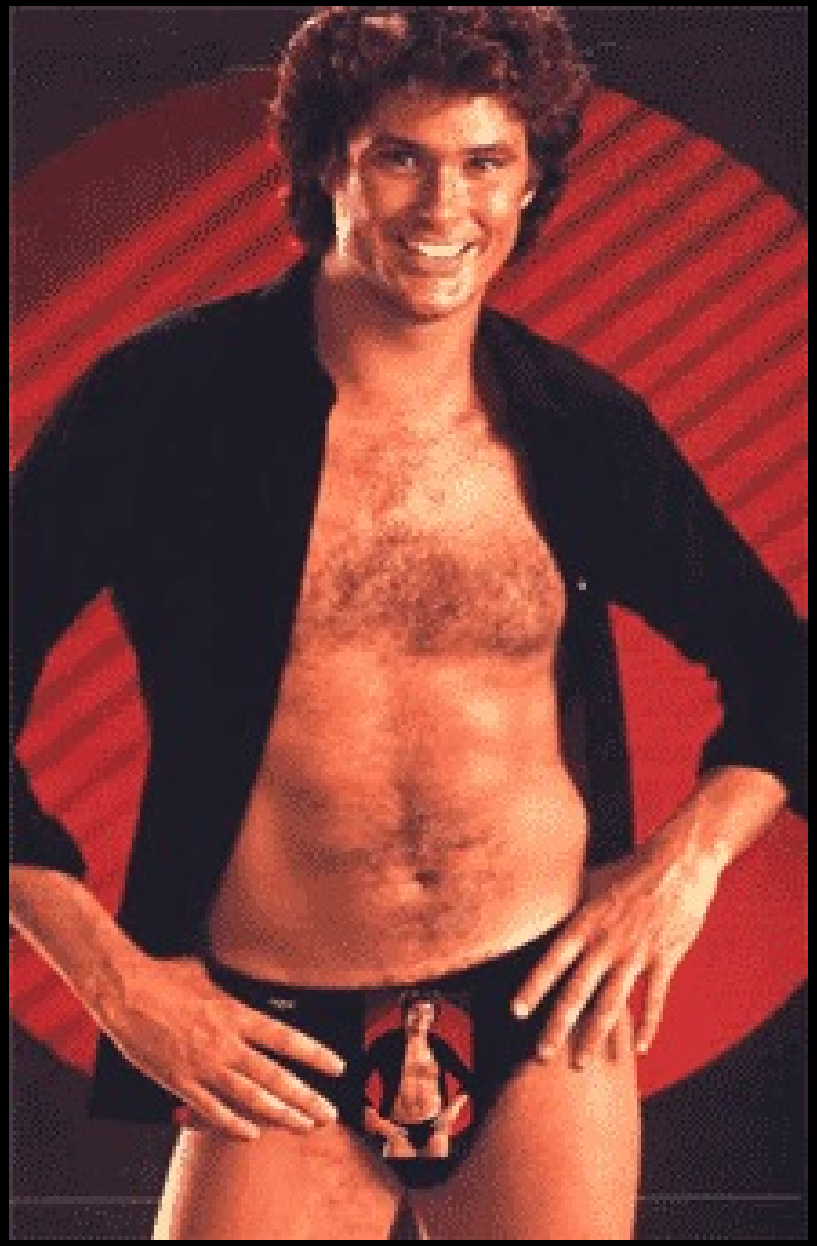

La Vache qui rit

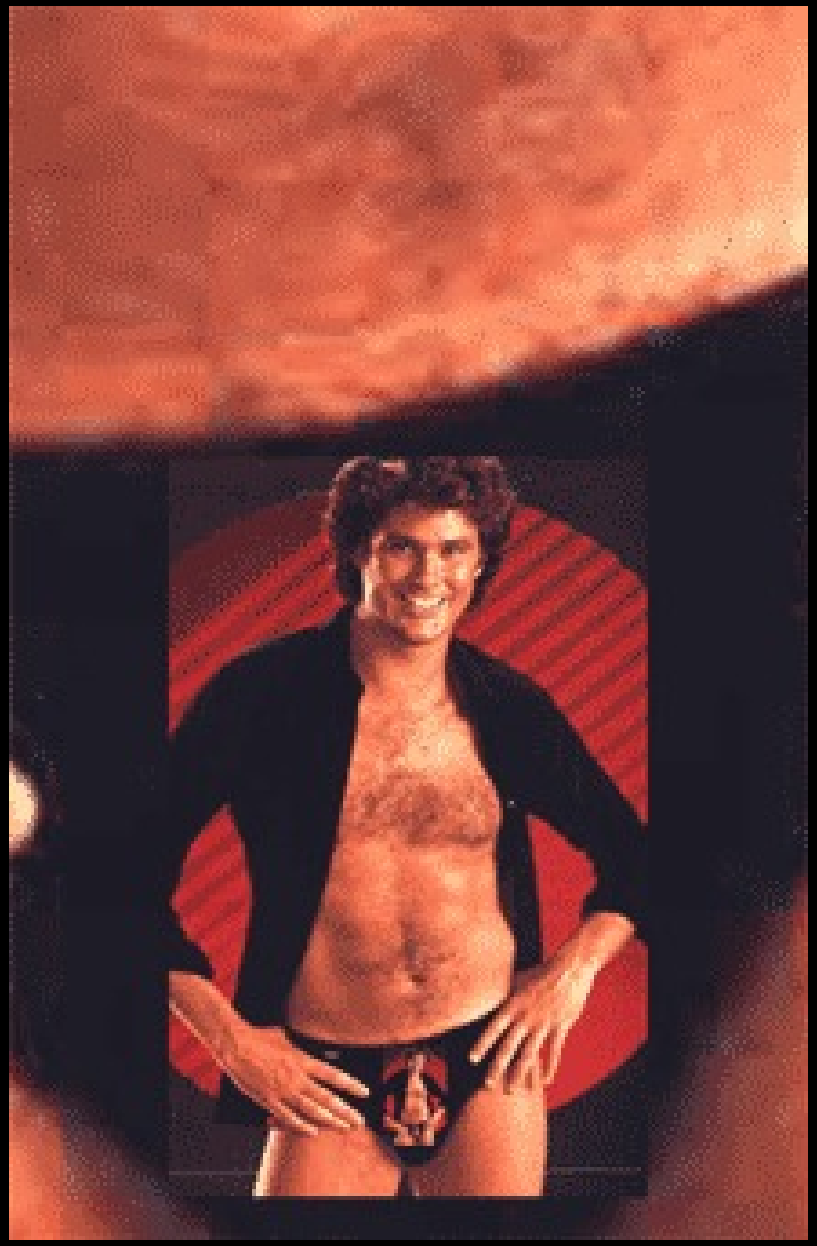


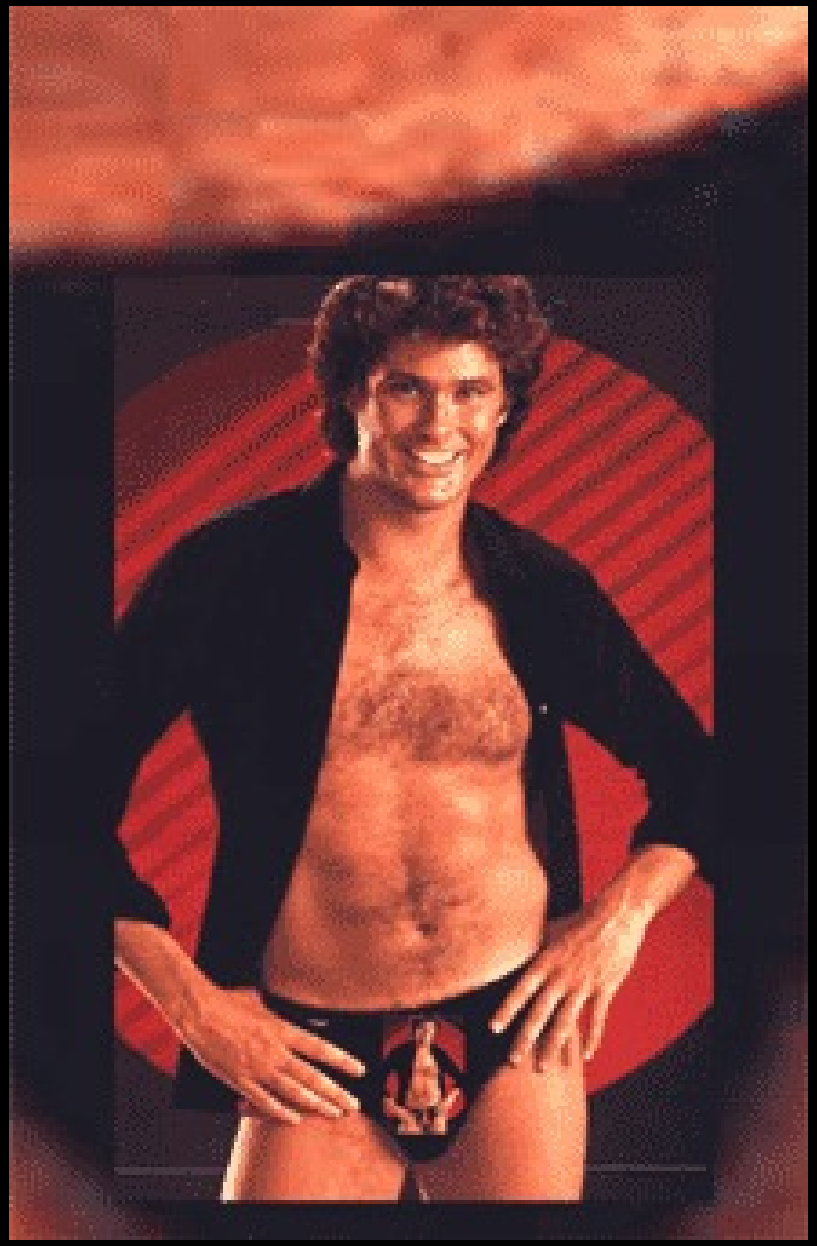


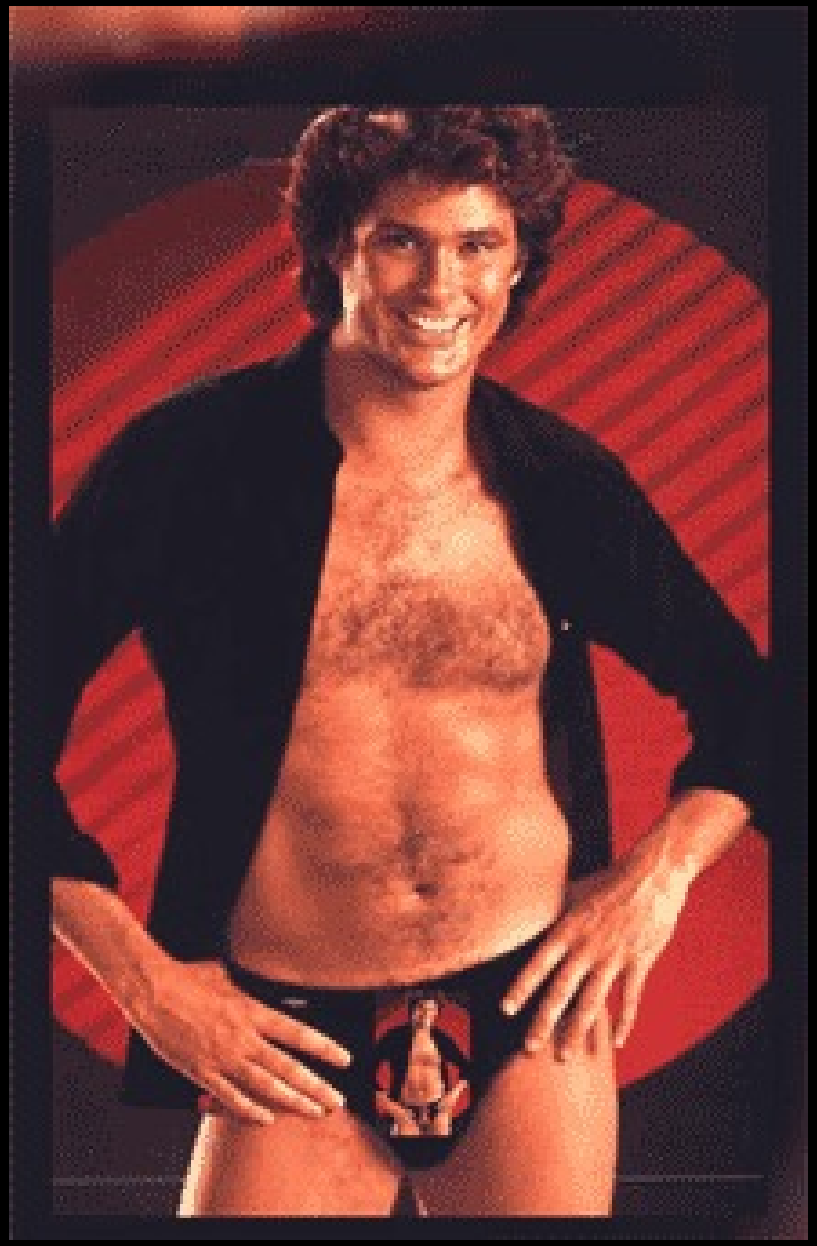


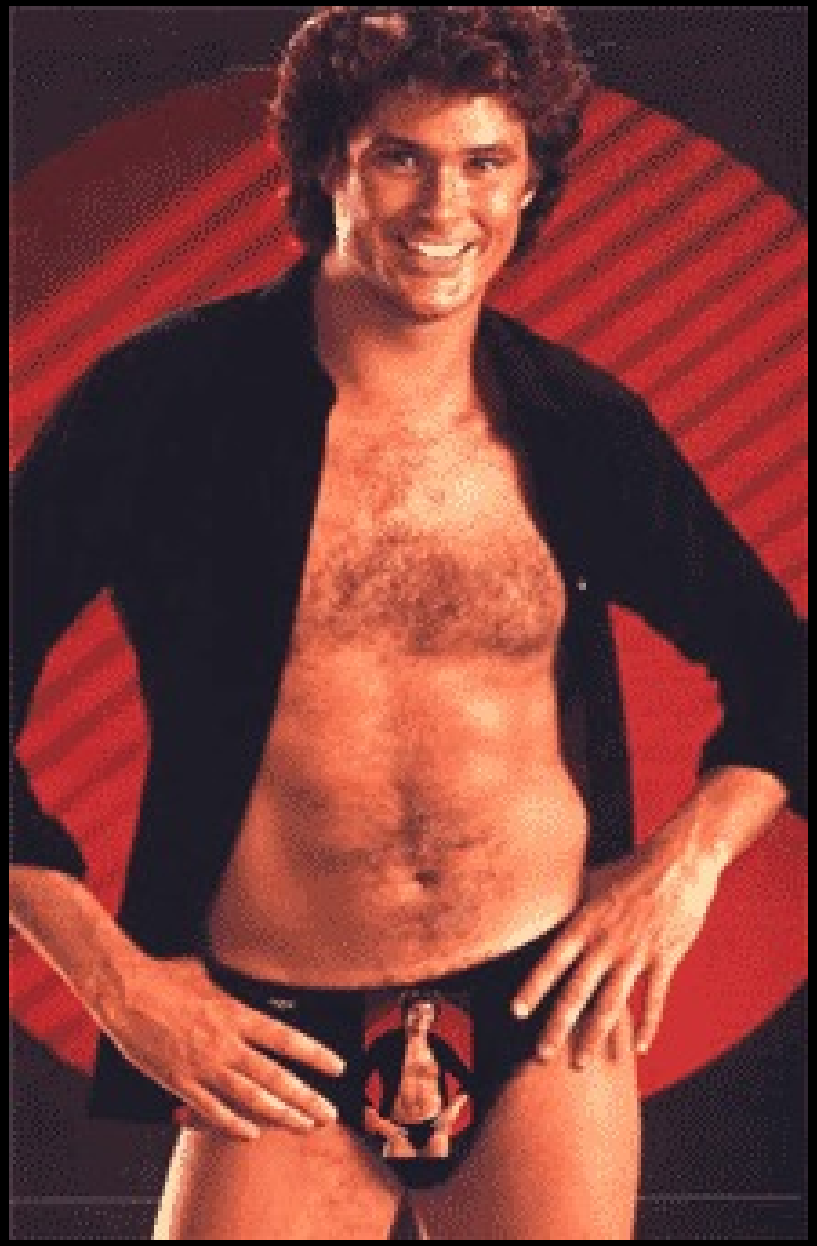












- induction
- induction mathématique (raisonnement par récurrence)
- fonction récurive et type récursif

- induction
- induction mathématique (raisonnement par récurrence)
- fonction récurive et type récursif

- induction
- induction mathématique (raisonnement par récurrence)
- fonction récurive et type récursif

$$S(n) = 0 + 1 + \cdots + (n-1) + n = S(n-1) + n$$

Haskell

```haskell
s :: Int → Int
s n
    | n <  0    = error "l'argument doit être un entier
      naturel"
    | n == 0    = 0
    | otherwise = n + s (n -1)
```

Haskell

```haskell
s' n = foldl (+) 0 [0 .. n]
```

$$S(n) = 0 + 1 + \cdots + (n-1) + n = S(n-1) + n$$

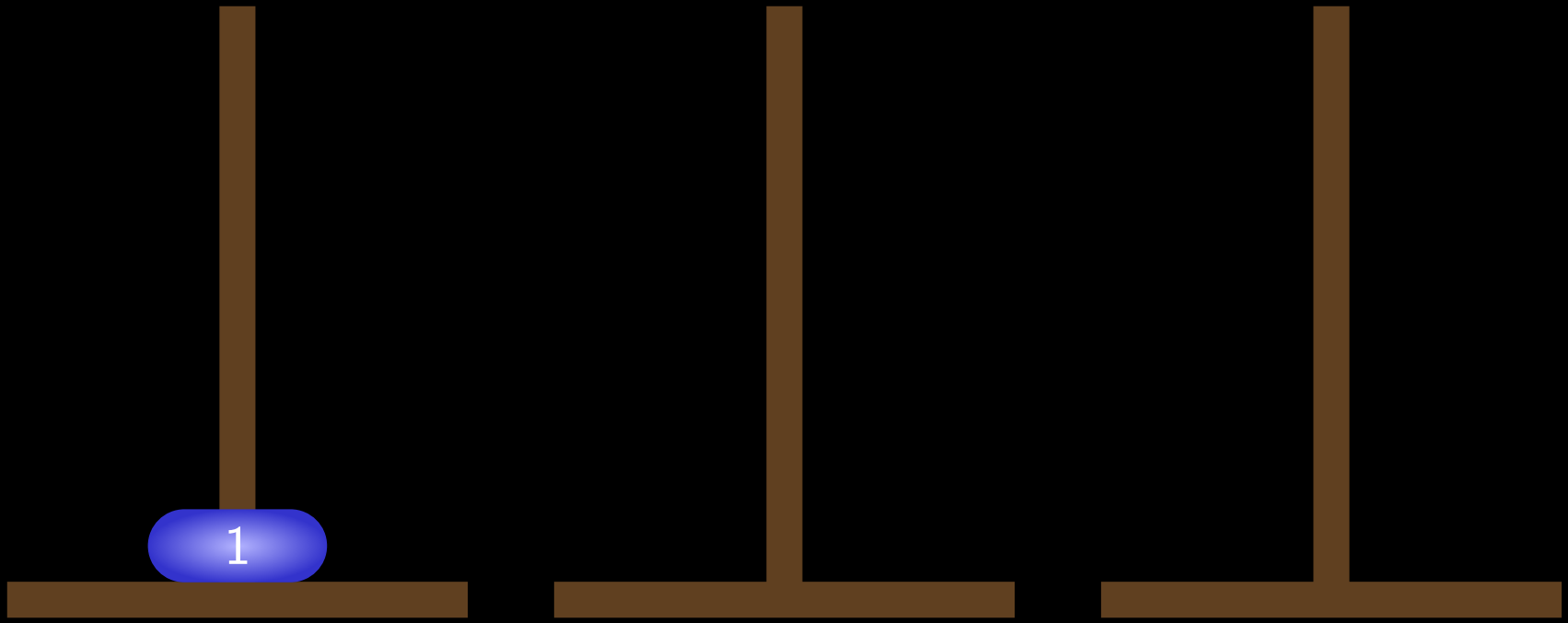
Haskell

```haskell
s :: Int → Int
s n
    | n <  0    = error "l'argument doit être un entier
      naturel"
    | n == 0    = 0
    | otherwise = n + s (n -1)
```

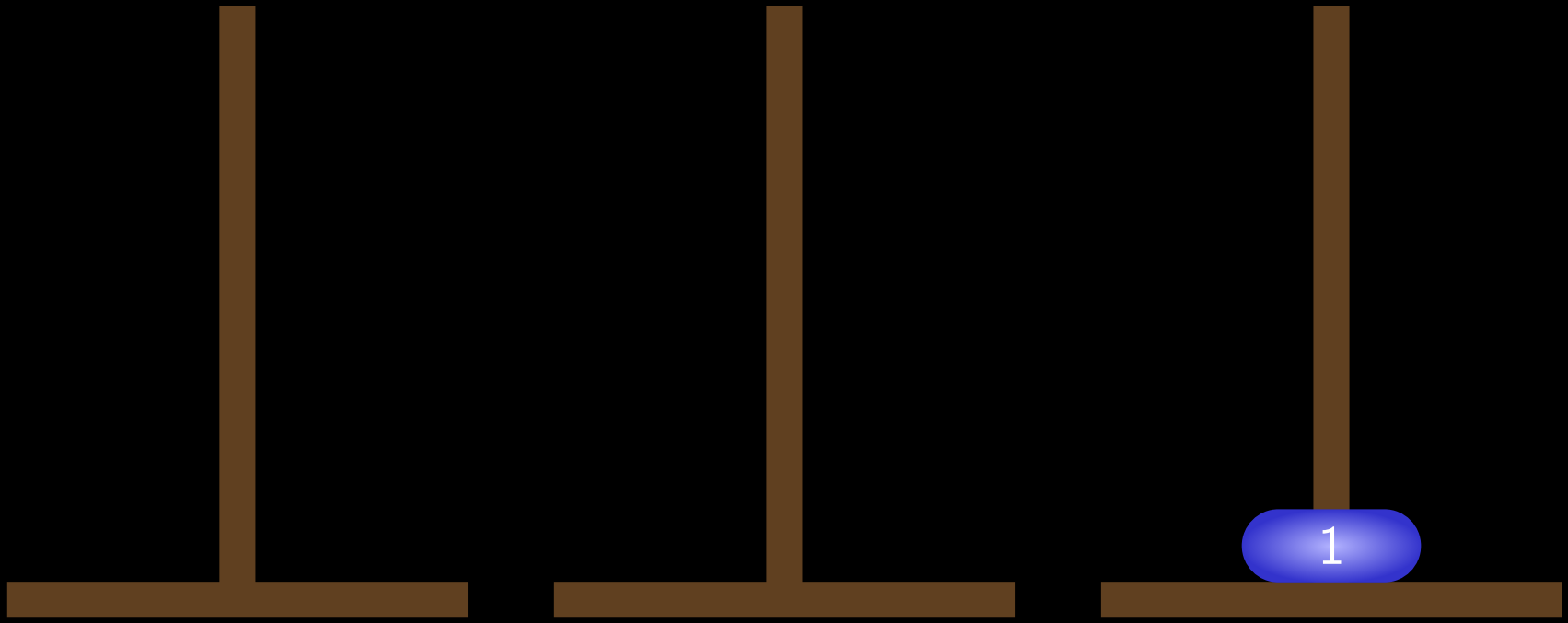
Haskell

```haskell
s' n = foldl (+) 0 [0 .. n]
```

Tour de Hanoi – 1 Disque

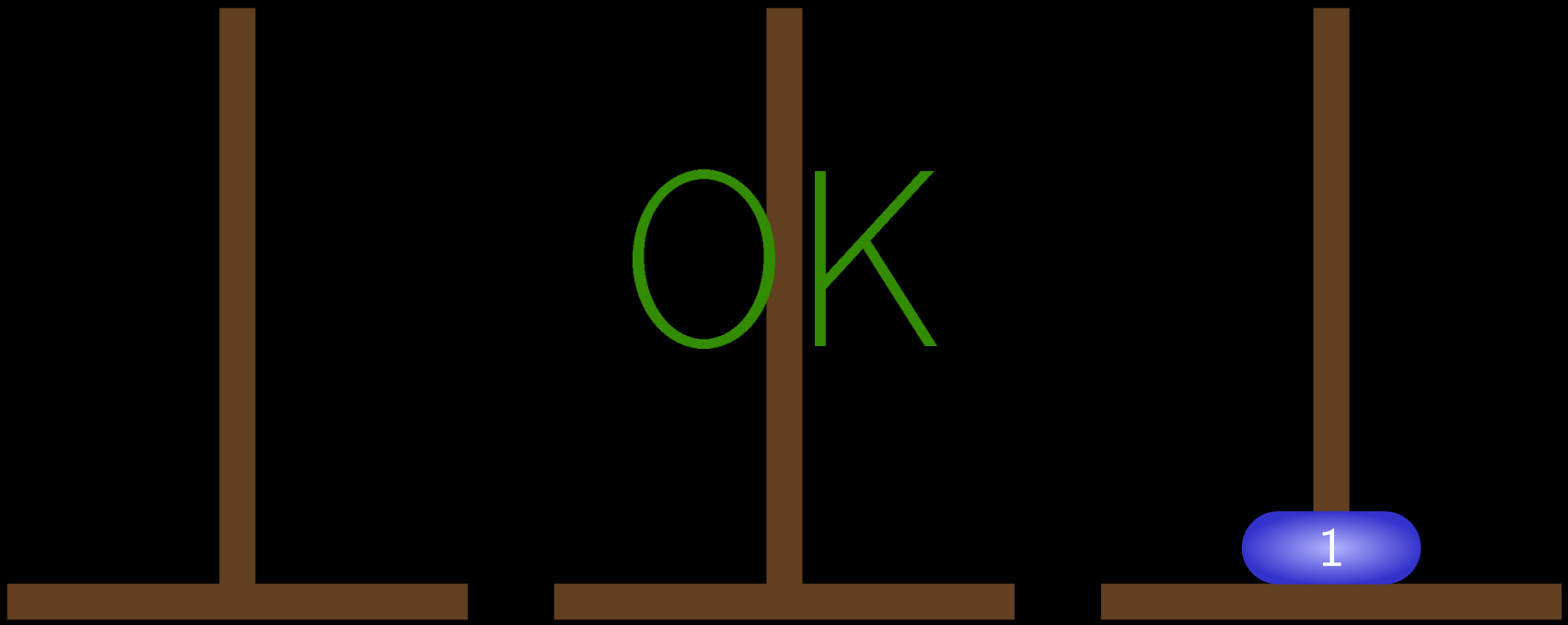


Tour de Hanoi – 1 Disque

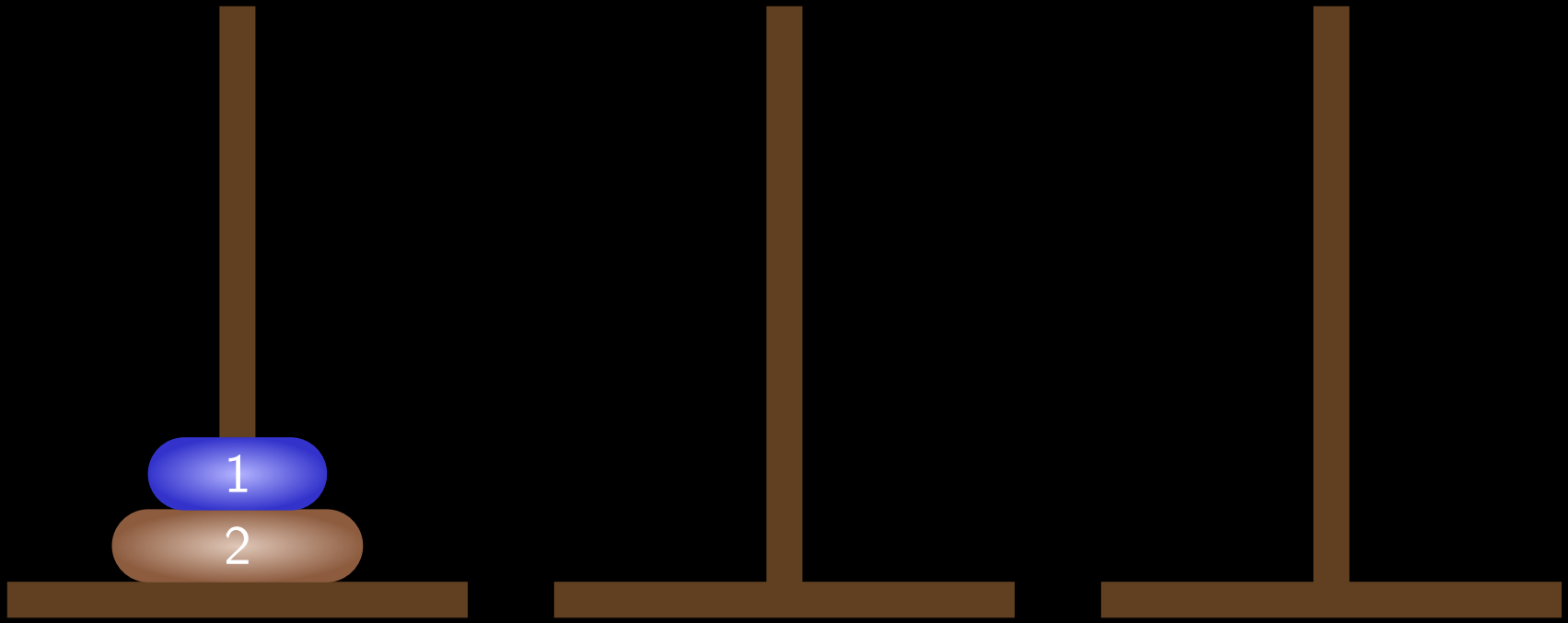


On déplace le disque de la tige 1 vers la tige 3.

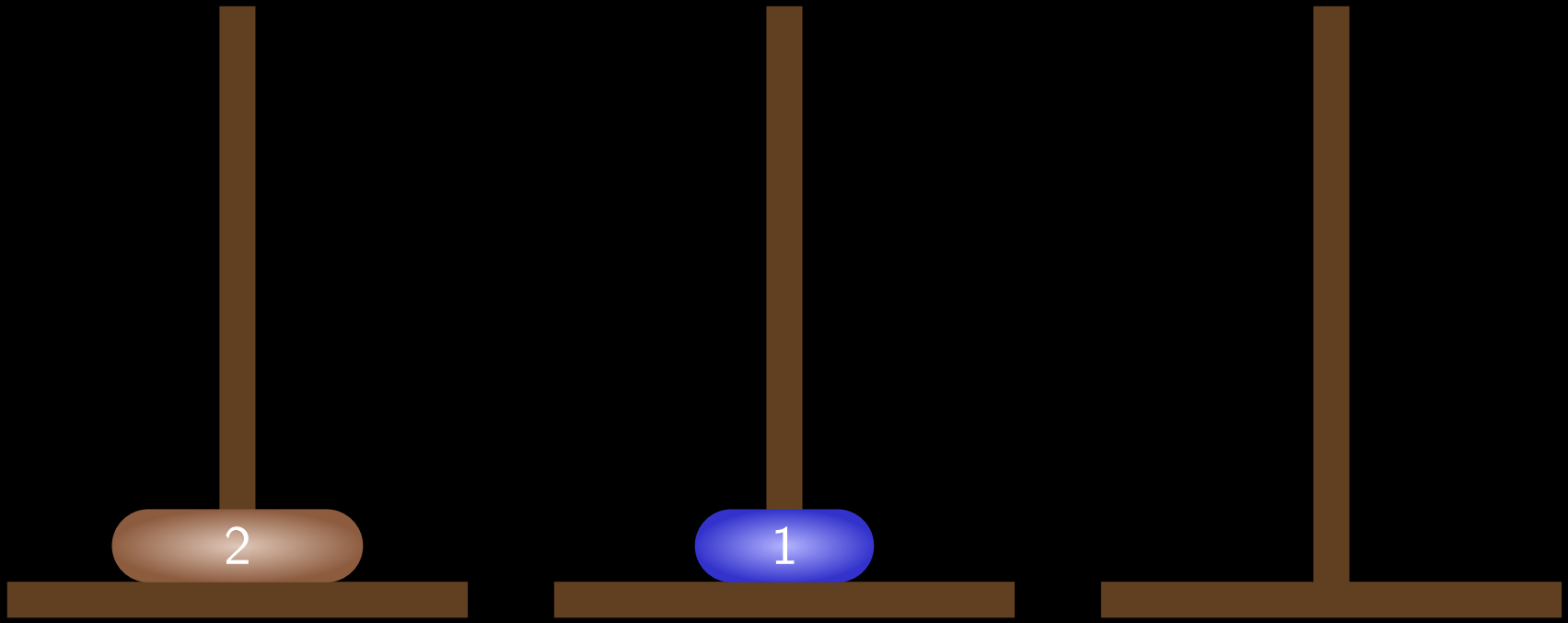
Tour de Hanoi – 1 Disque



Tour de Hanoi – 2 Disques

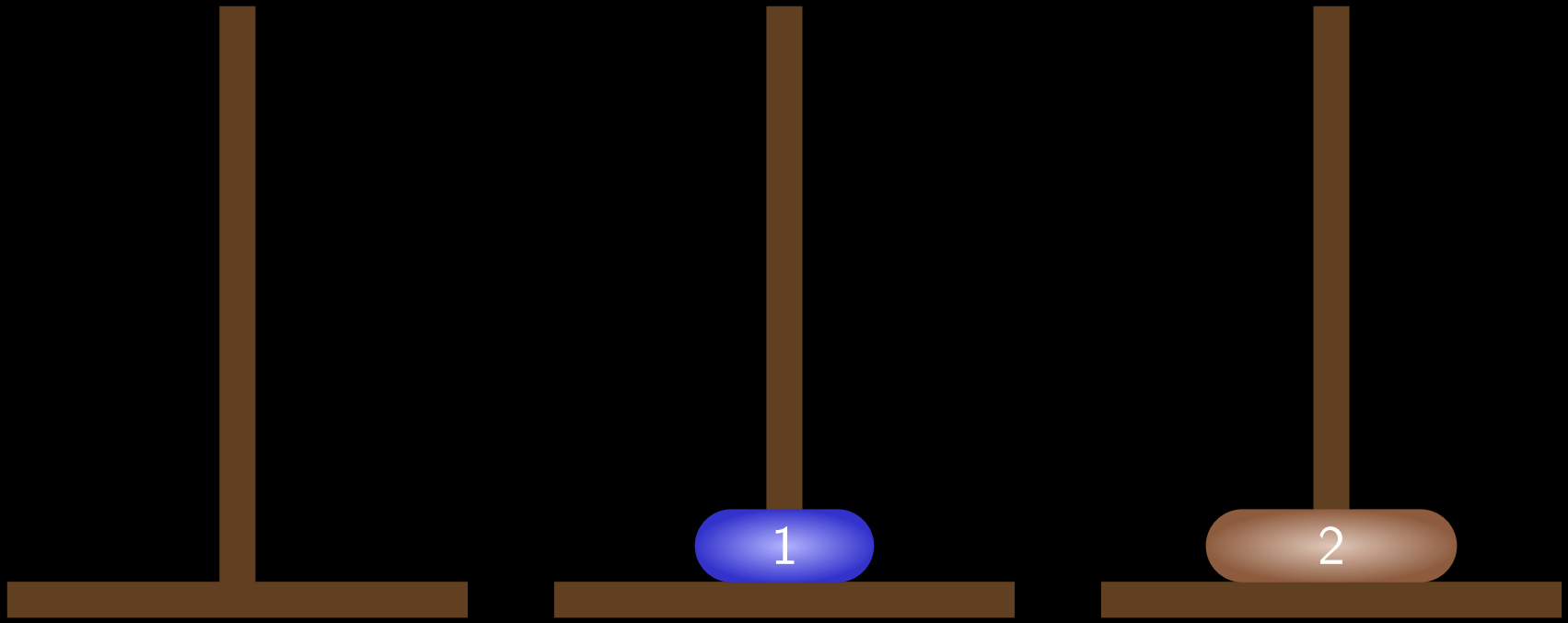


Tour de Hanoi – 2 Disques



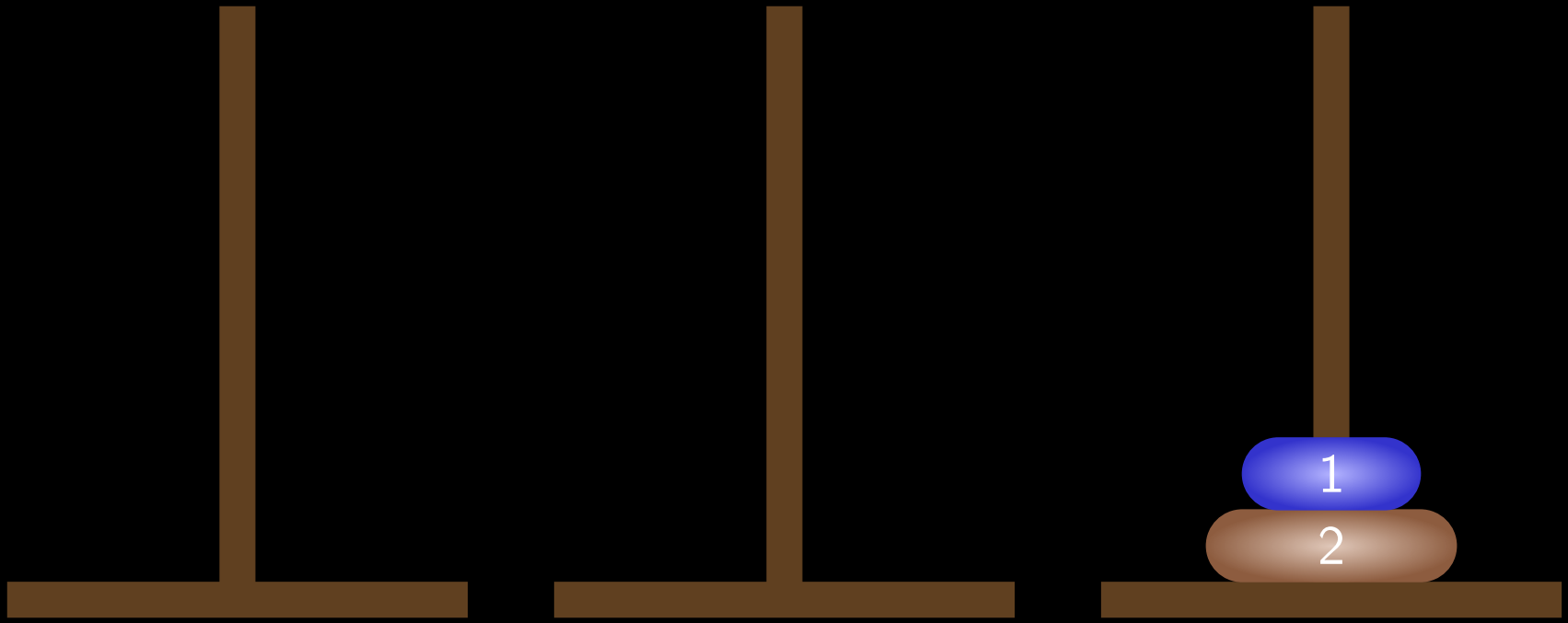
On déplace le disque de la tige 1 vers la tige 2.

Tour de Hanoi – 2 Disques



On déplace le disque de la tige 1 vers la tige 3.

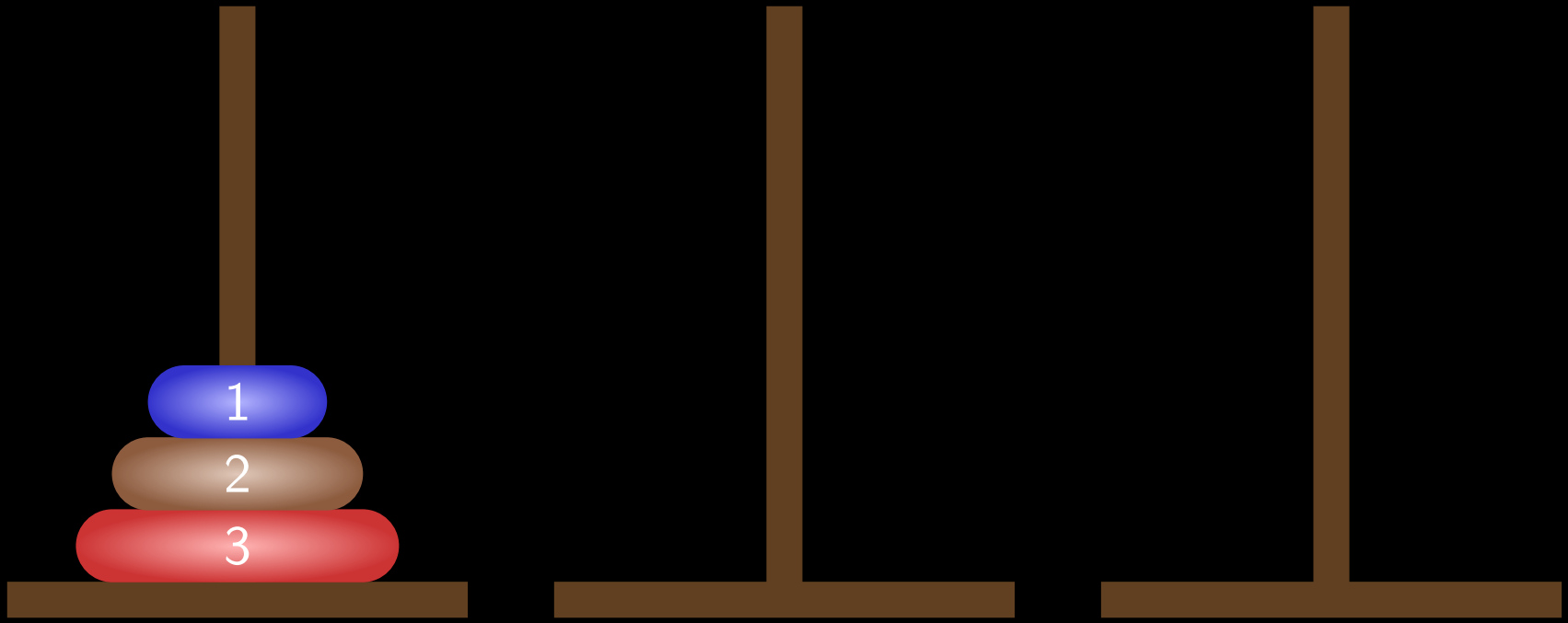
Tour de Hanoi – 2 Disques



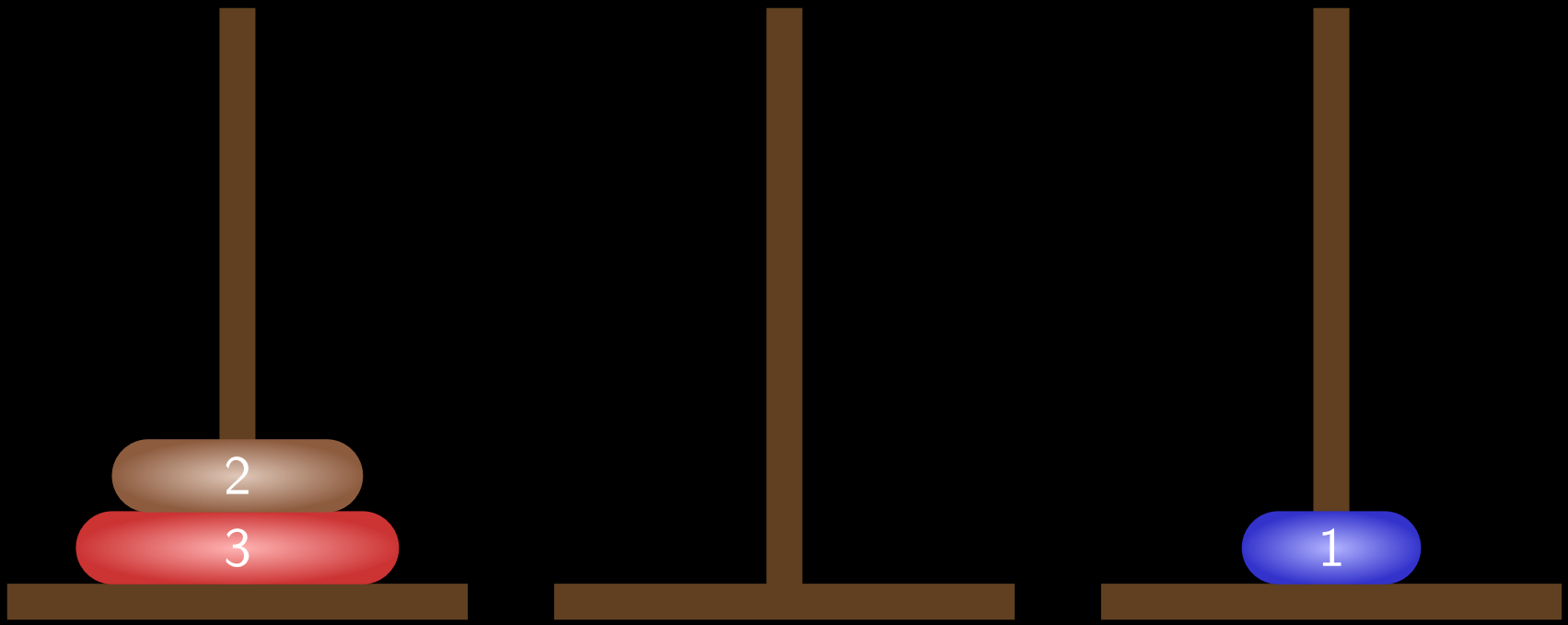
On déplace le disque de la tige 2 vers la tige 3.

Tour de Hanoi – 2 Disques

Tour de Hanoi – 3 Disques

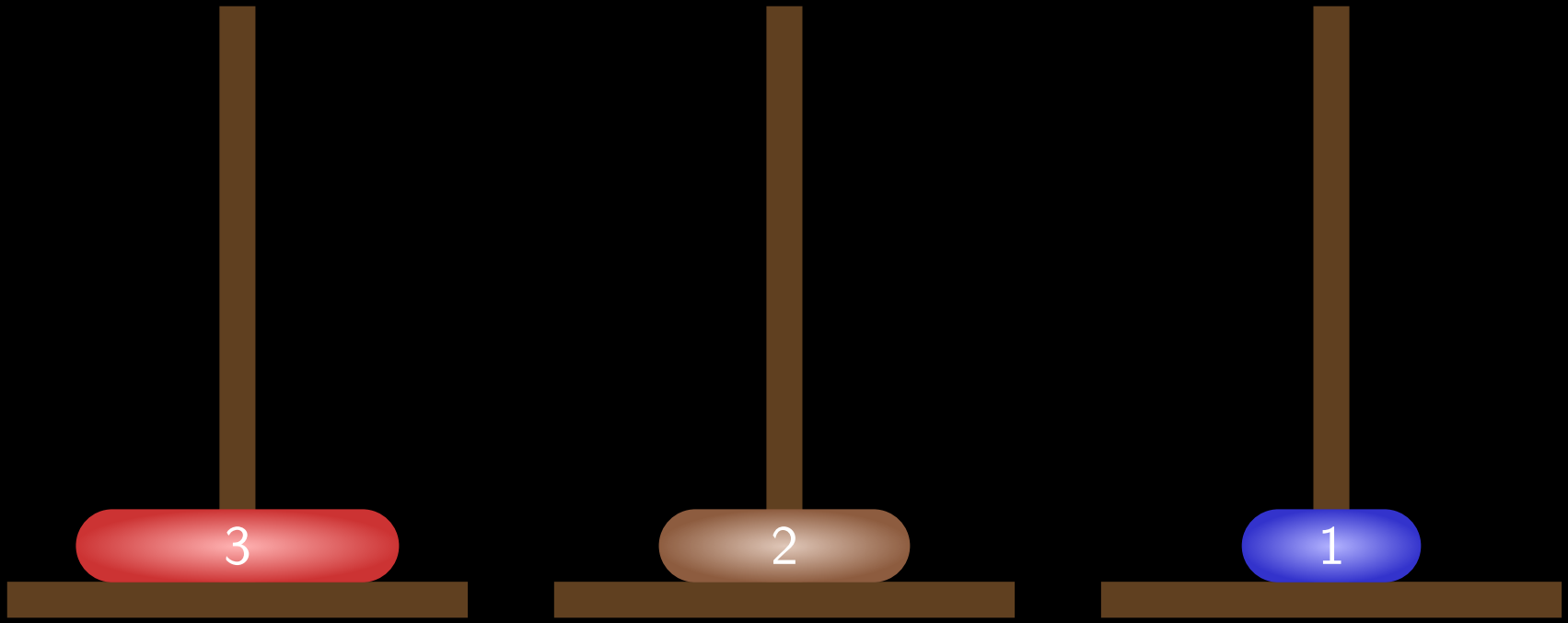


Tour de Hanoi – 3 Disques



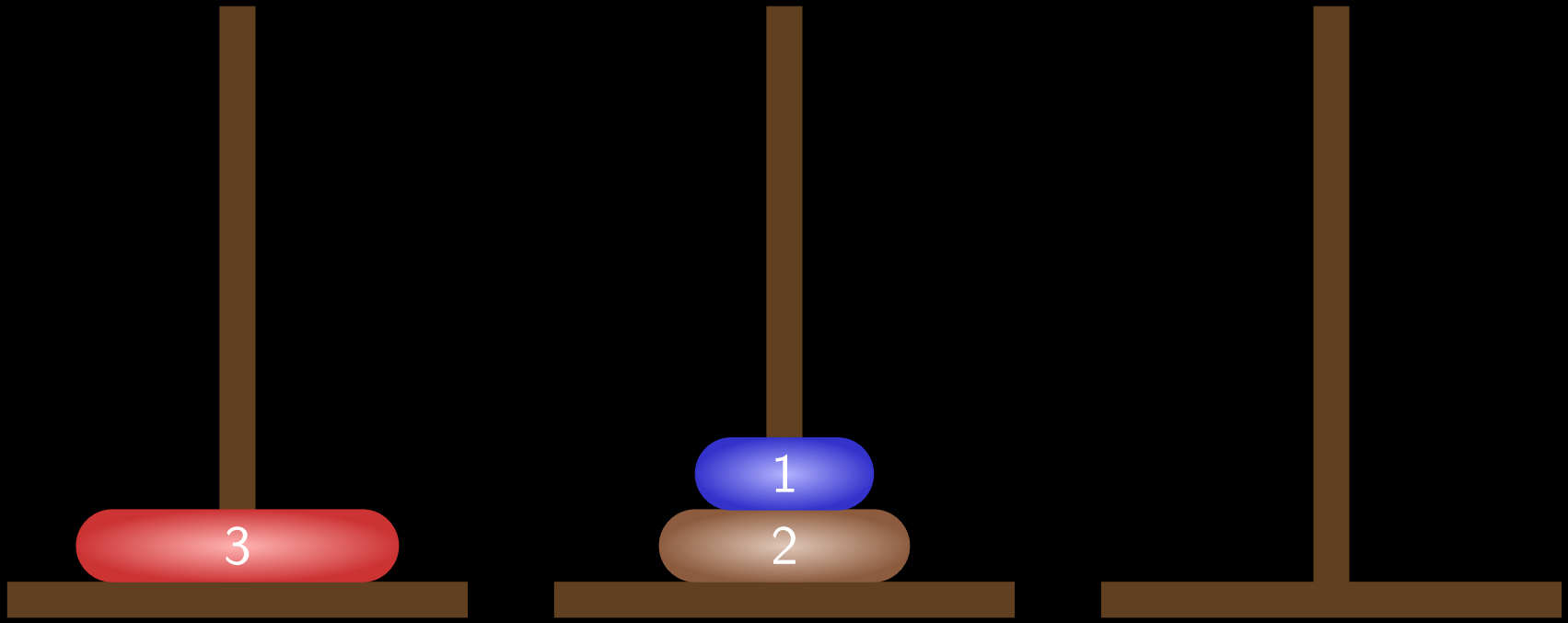
On déplace le disque de la tige 1 vers la tige 3.

Tour de Hanoi – 3 Disques



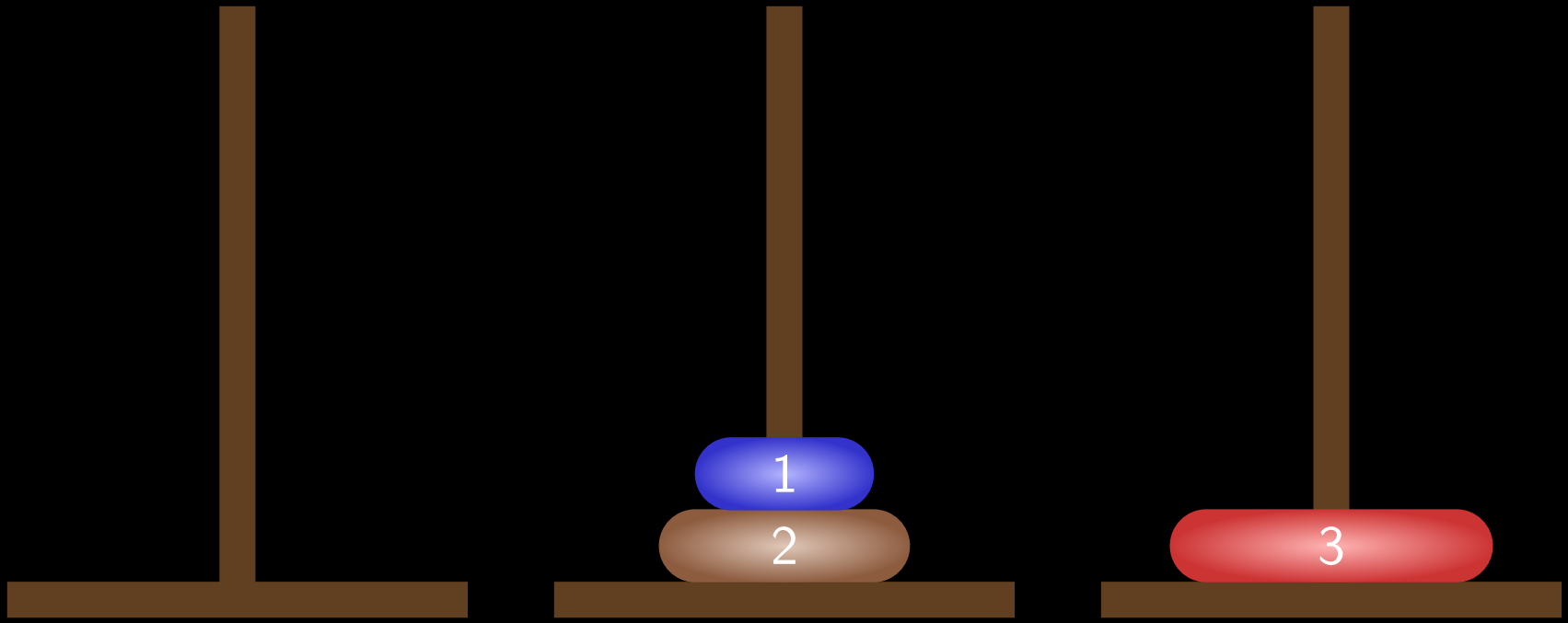
On déplace le disque de la tige 1 vers la tige 2.

Tour de Hanoi – 3 Disques



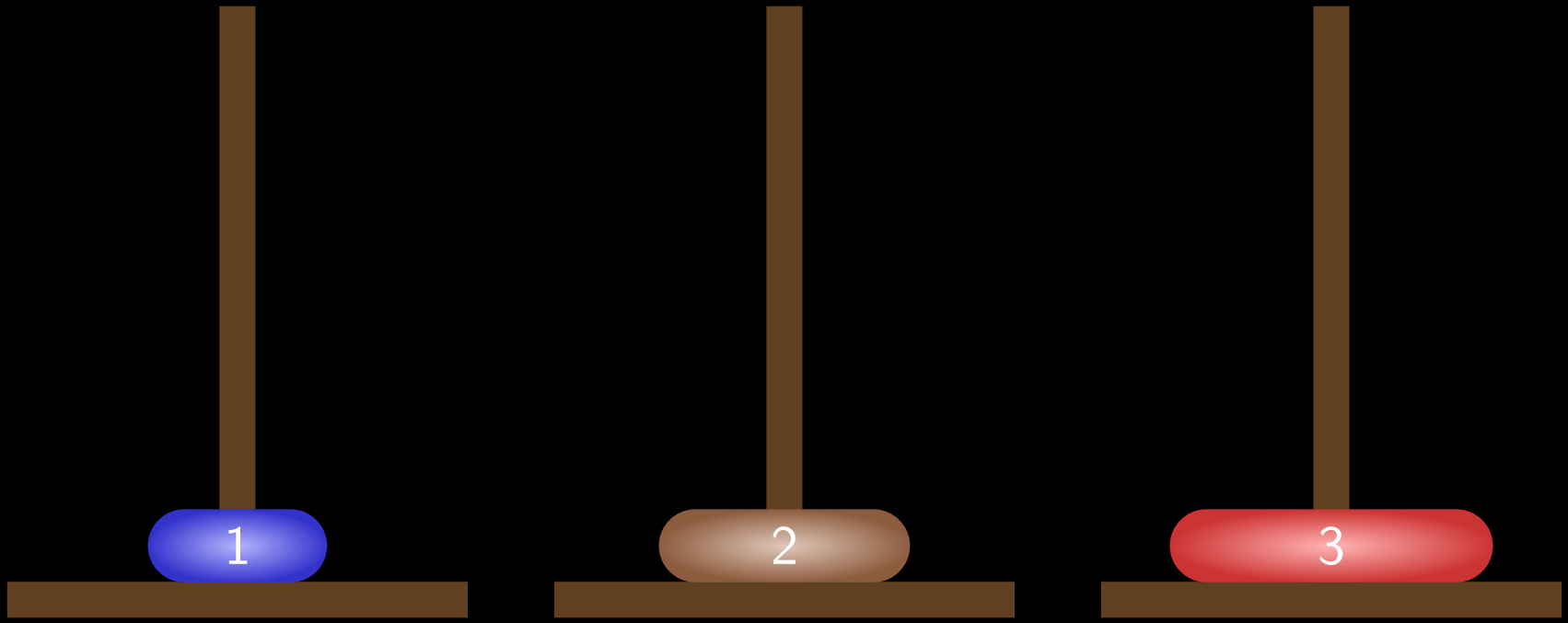
On déplace le disque de la tige 3 vers la tige 2.

Tour de Hanoi – 3 Disques



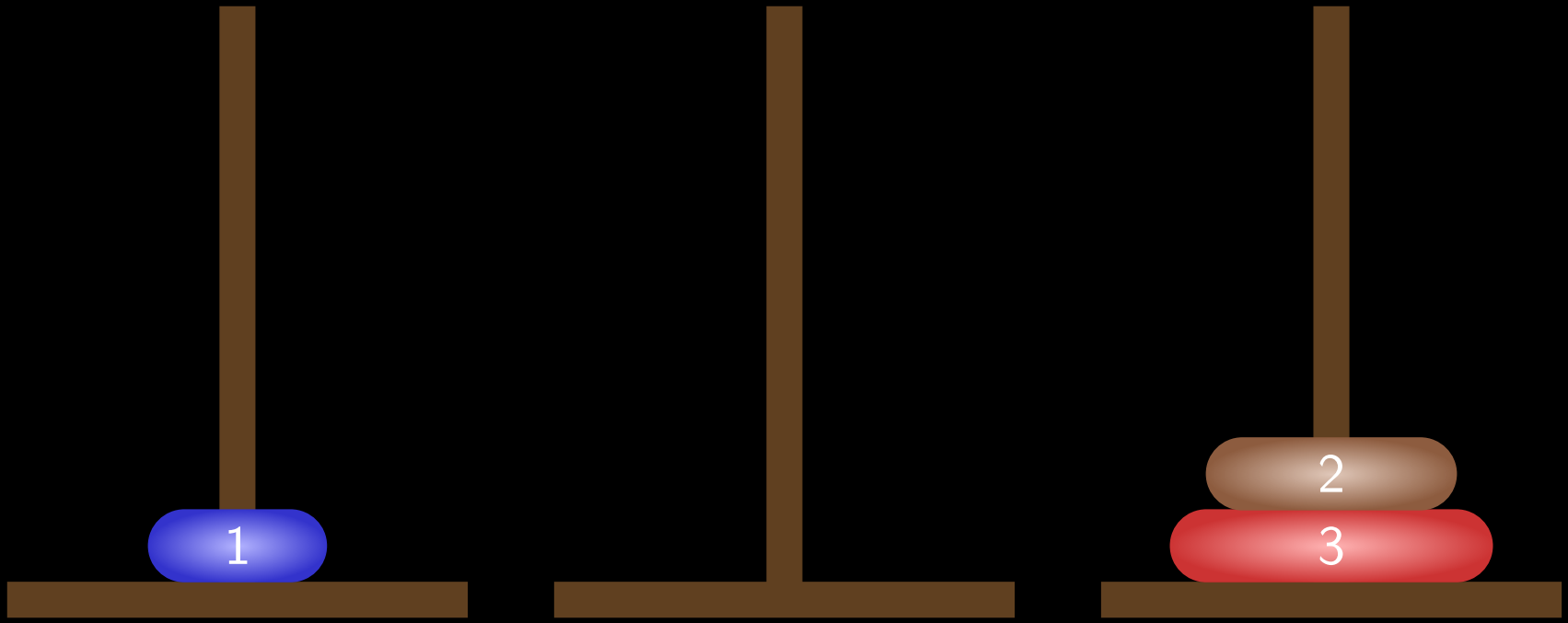
On déplace le disque de la tige 1 vers la tige 3.

Tour de Hanoi – 3 Disques



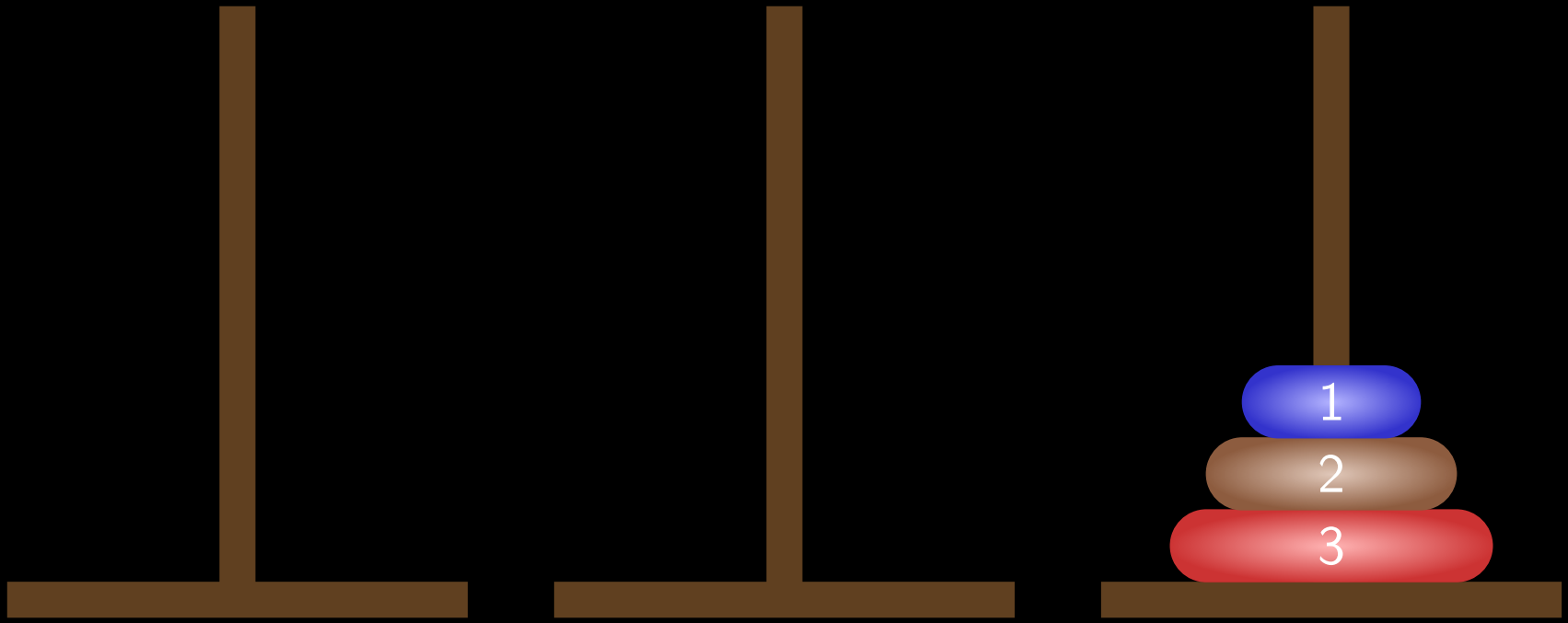
On déplace le disque de la tige 2 vers la tige 1.

Tour de Hanoi – 3 Disques



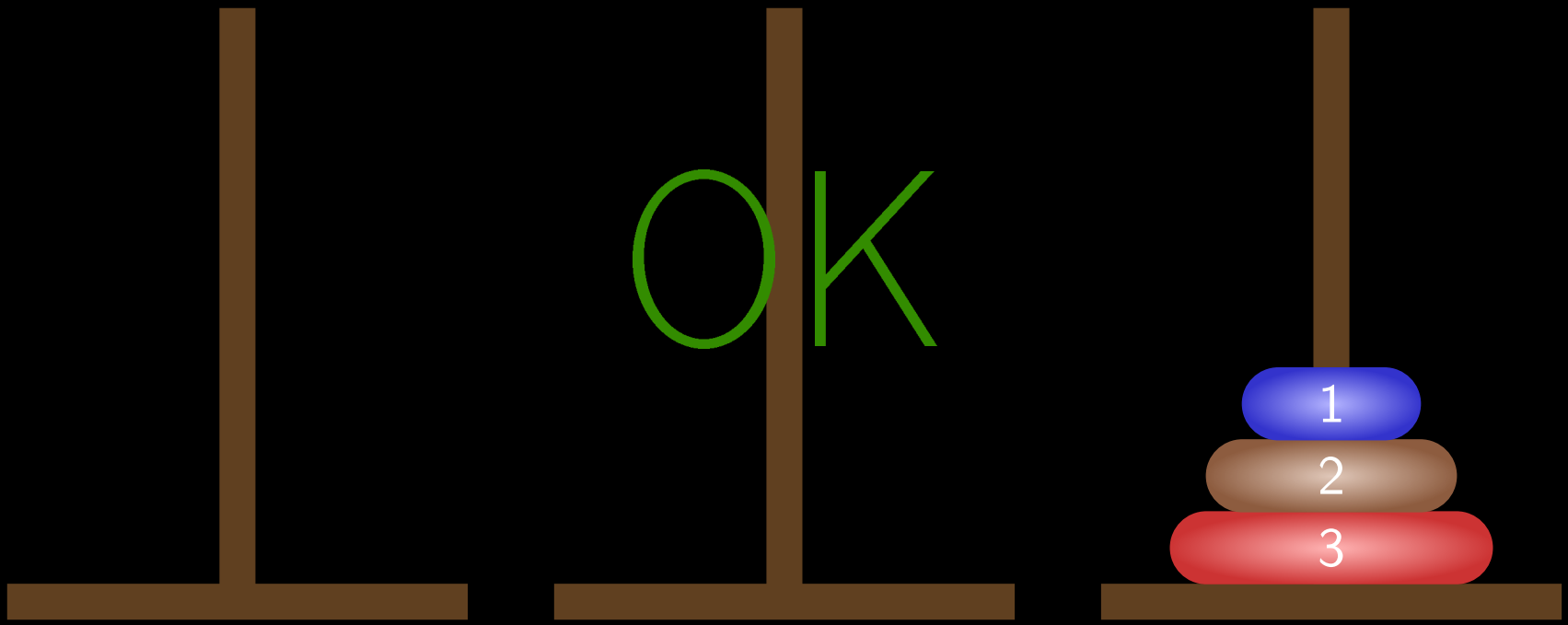
On déplace le disque de la tige 2 vers la tige 3.

Tour de Hanoi – 3 Disques

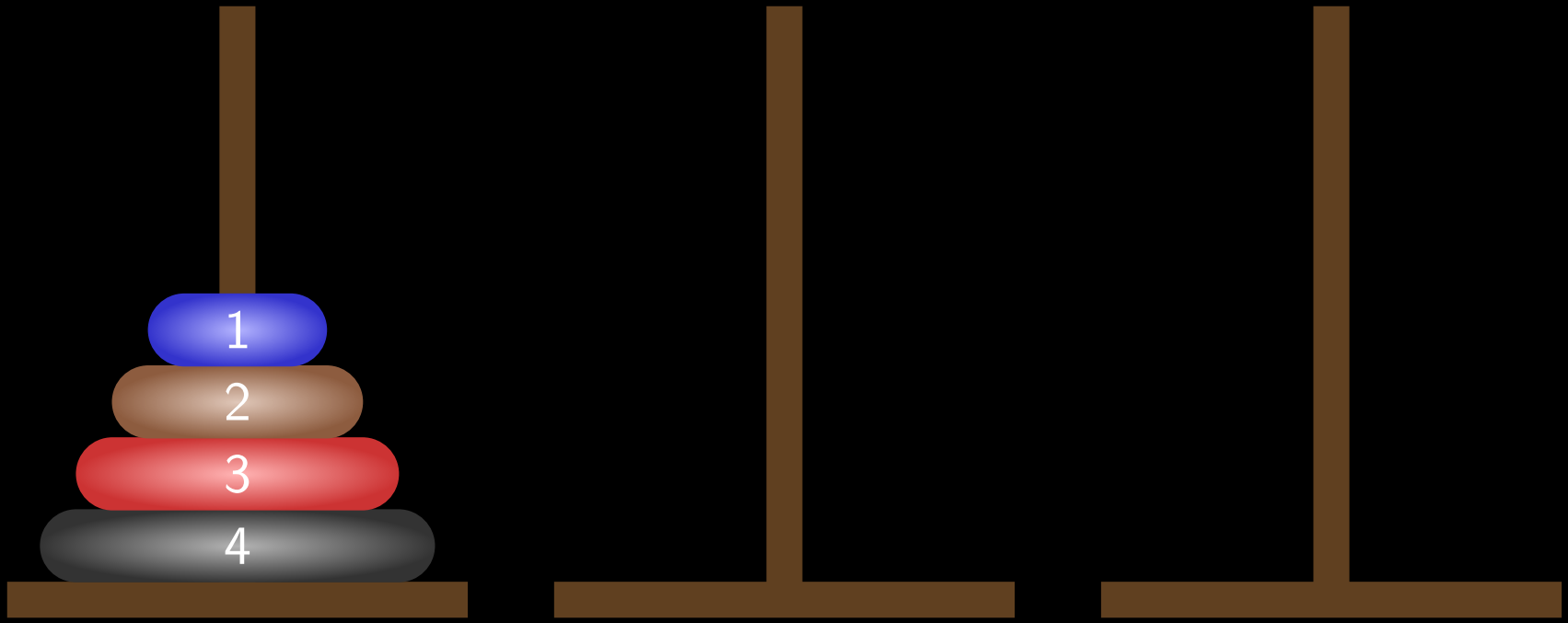


On déplace le disque de la tige 1 vers la tige 3.

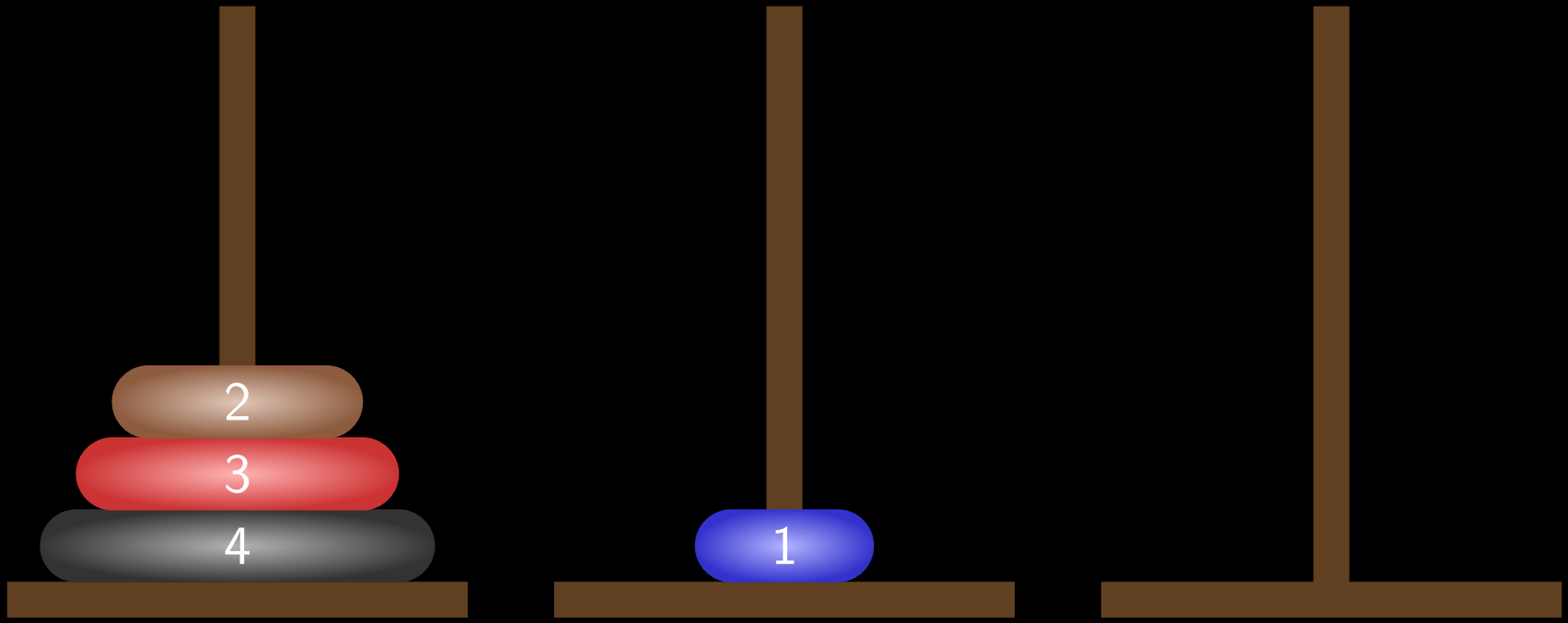
Tour de Hanoi – 3 Disques



Tour de Hanoi – 4 Disques

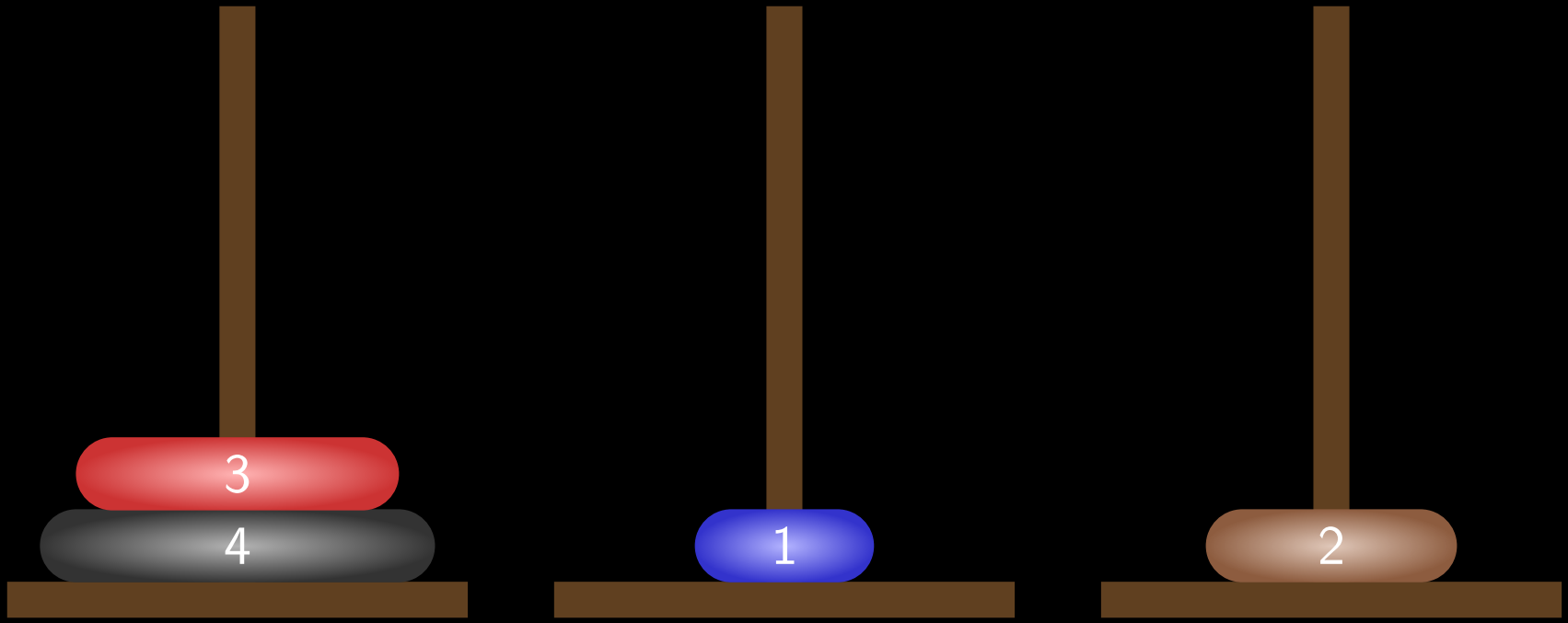


Tour de Hanoi – 4 Disques



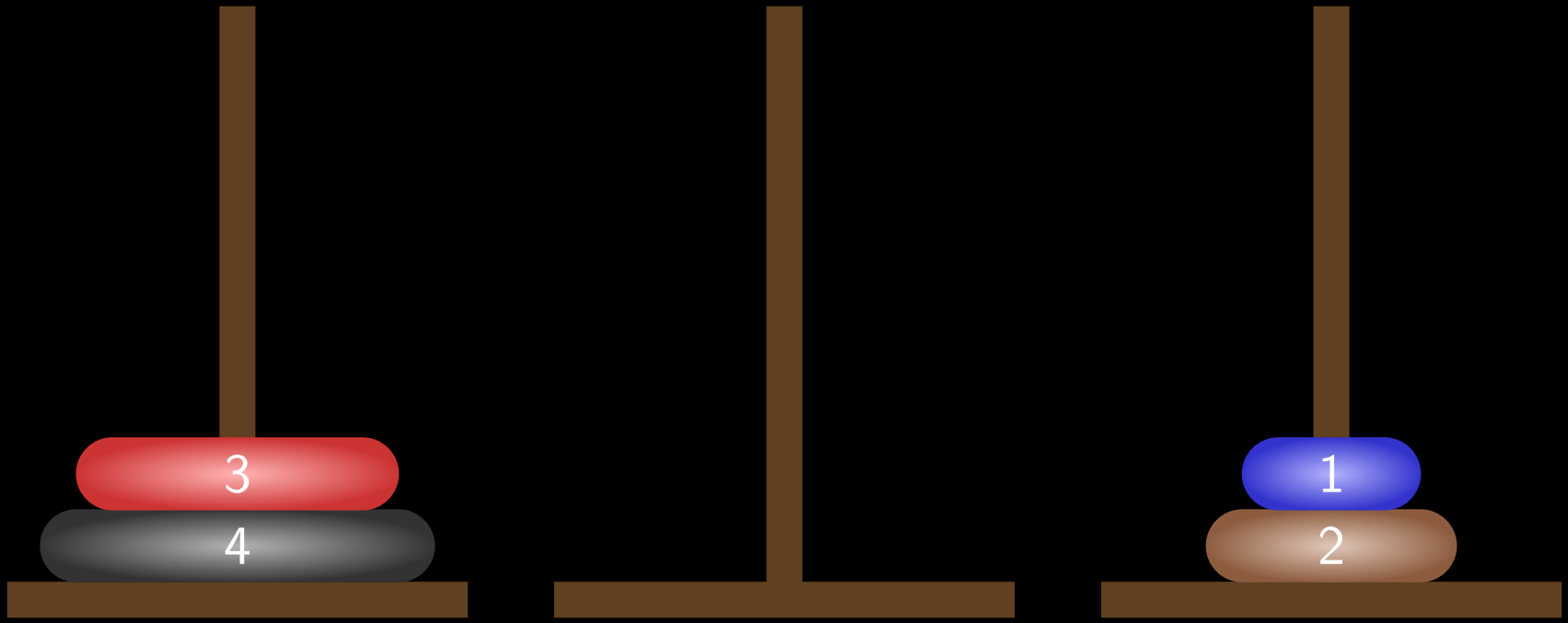
On déplace le disque de la tige 1 vers la tige 2.

Tour de Hanoi – 4 Disques



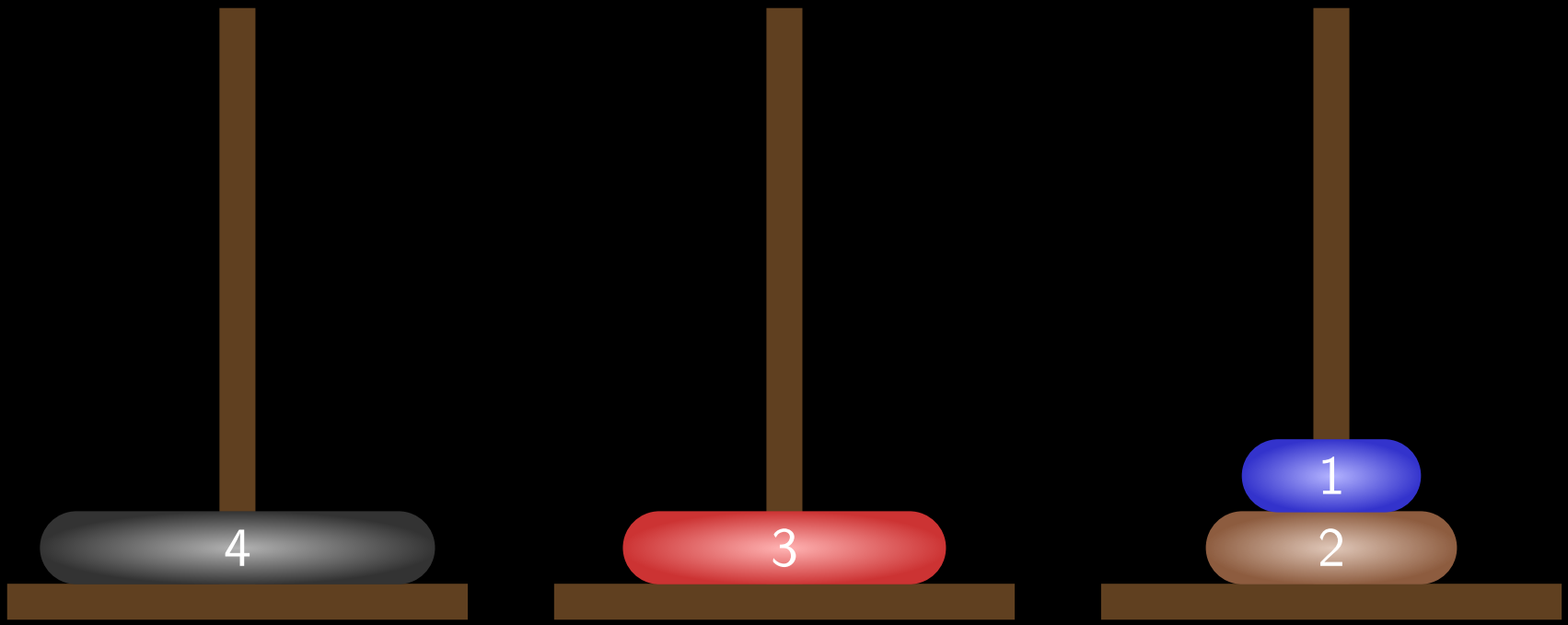
On déplace le disque de la tige 1 vers la tige 3.

Tour de Hanoi – 4 Disques



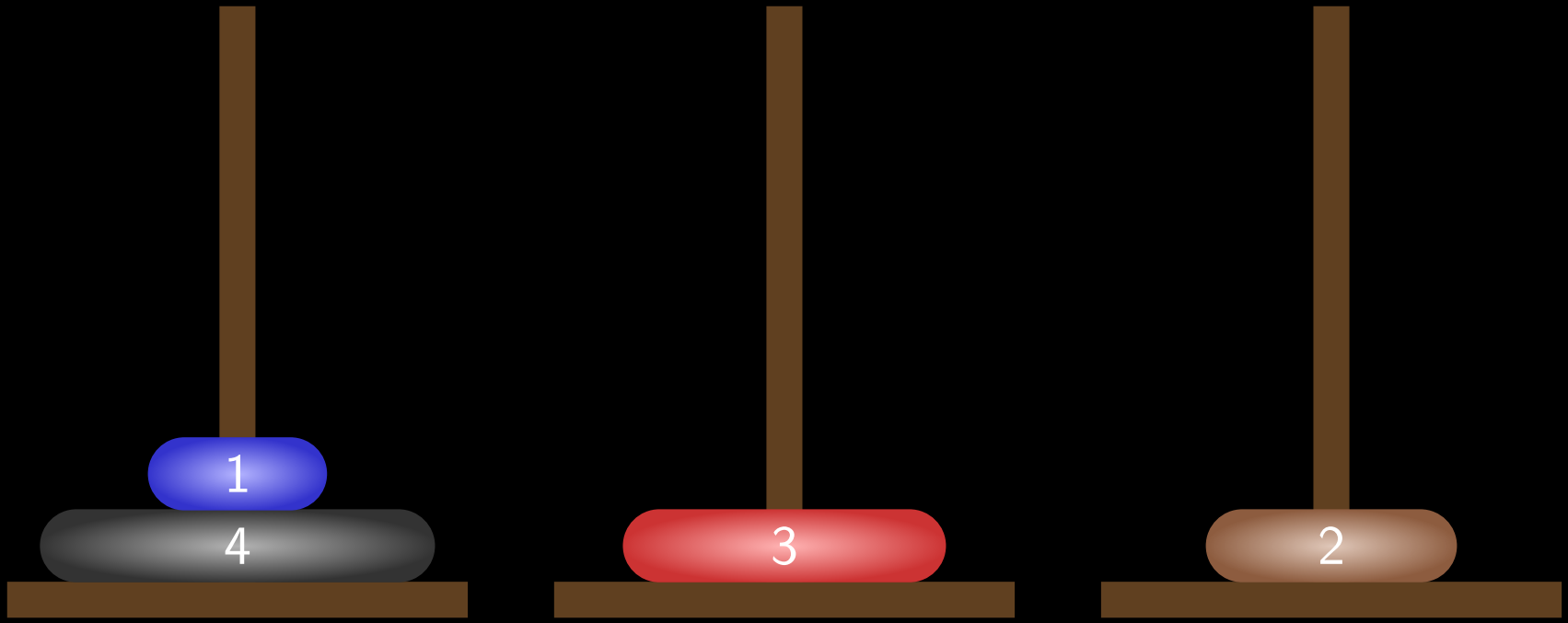
On déplace le disque de la tige 2 vers la tige 3.

Tour de Hanoi – 4 Disques



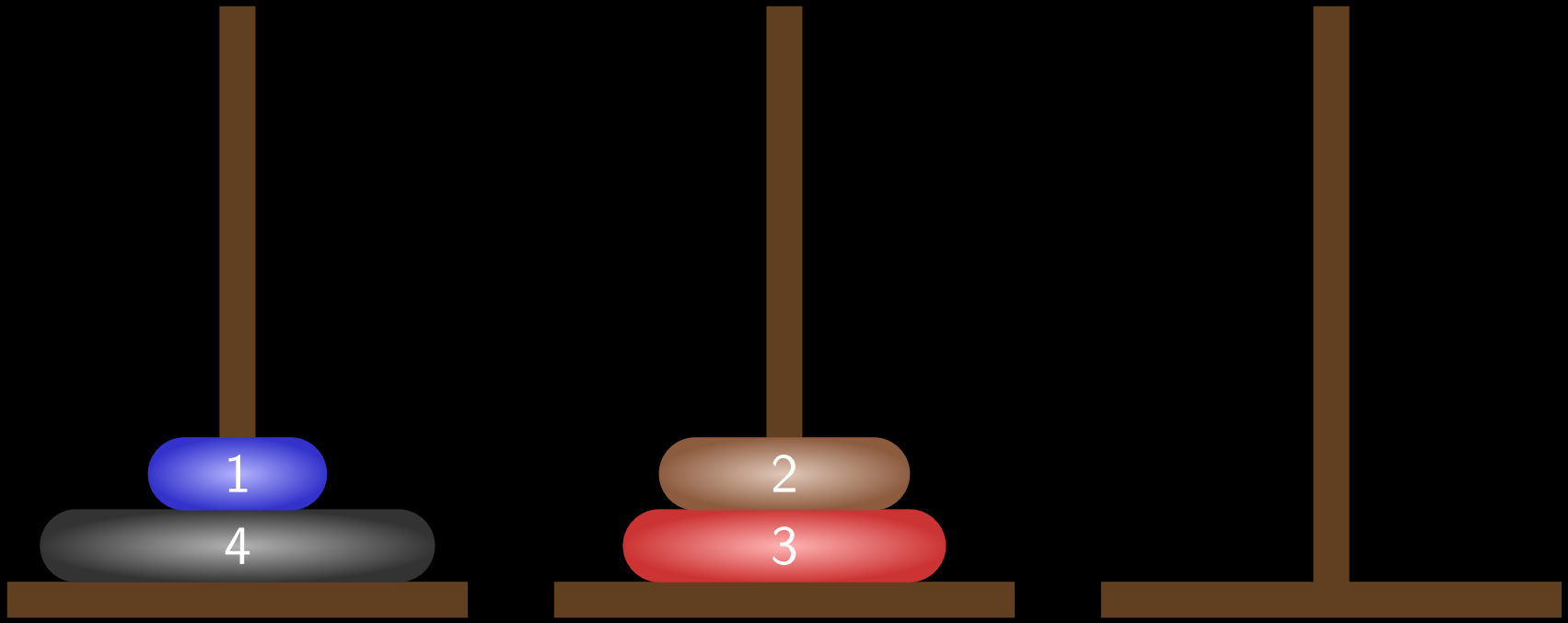
On déplace le disque de la tige 1 vers la tige 2.

Tour de Hanoi – 4 Disques



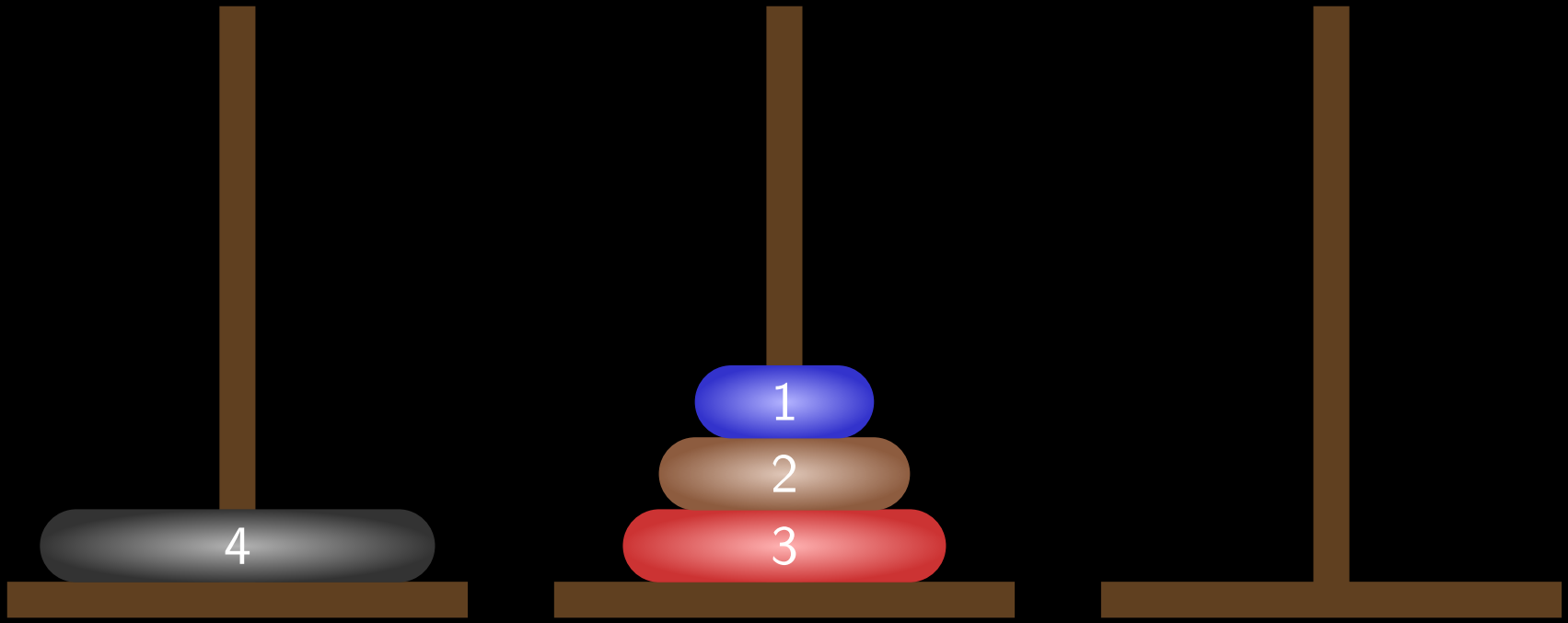
On déplace le disque de la tige 3 vers la tige 1.

Tour de Hanoi – 4 Disques



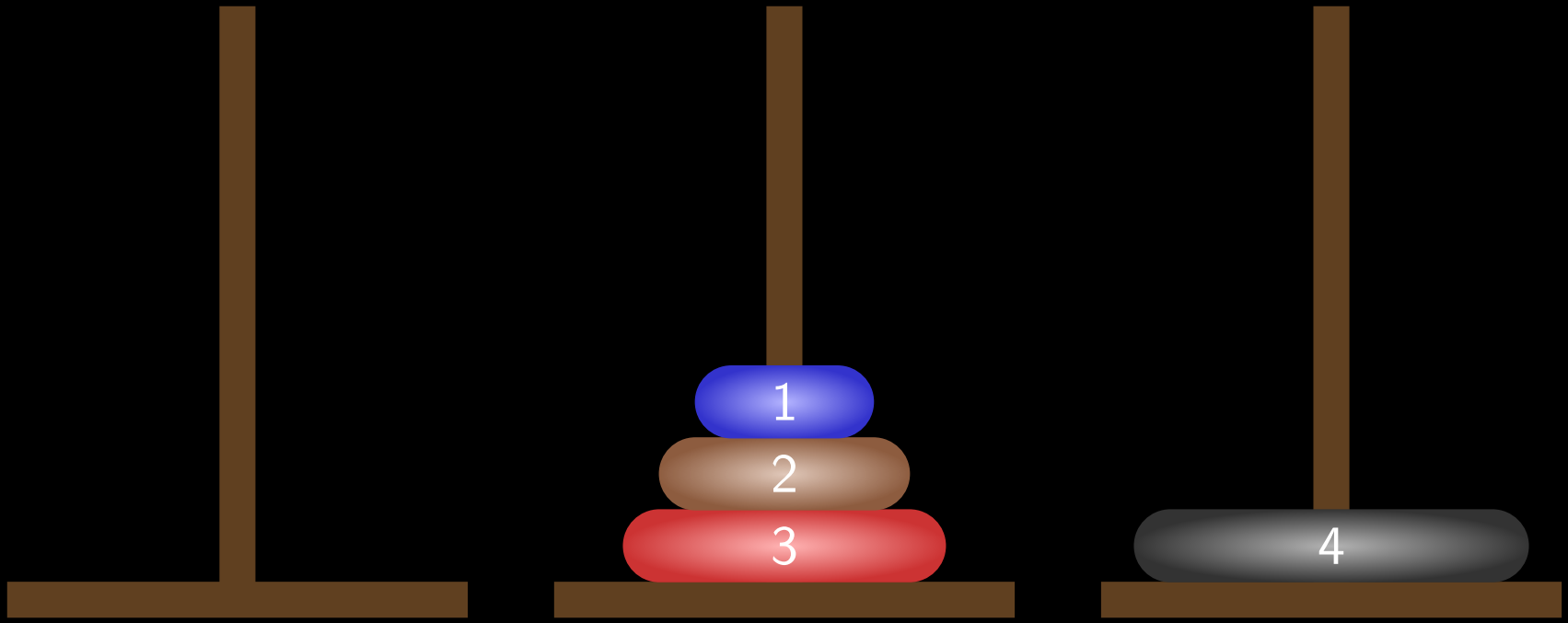
On déplace le disque de la tige 3 vers la tige 2.

Tour de Hanoi – 4 Disques



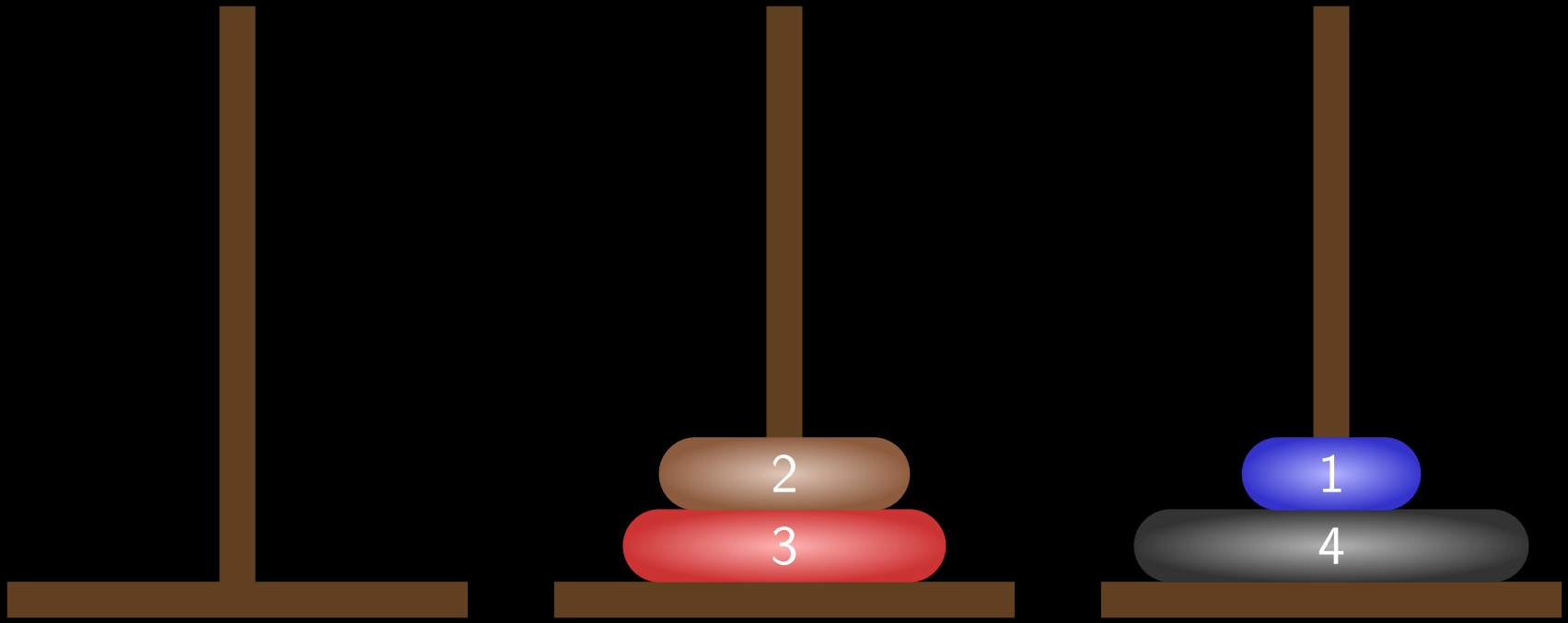
On déplace le disque de la tige 1 vers la tige 2.

Tour de Hanoi – 4 Disques



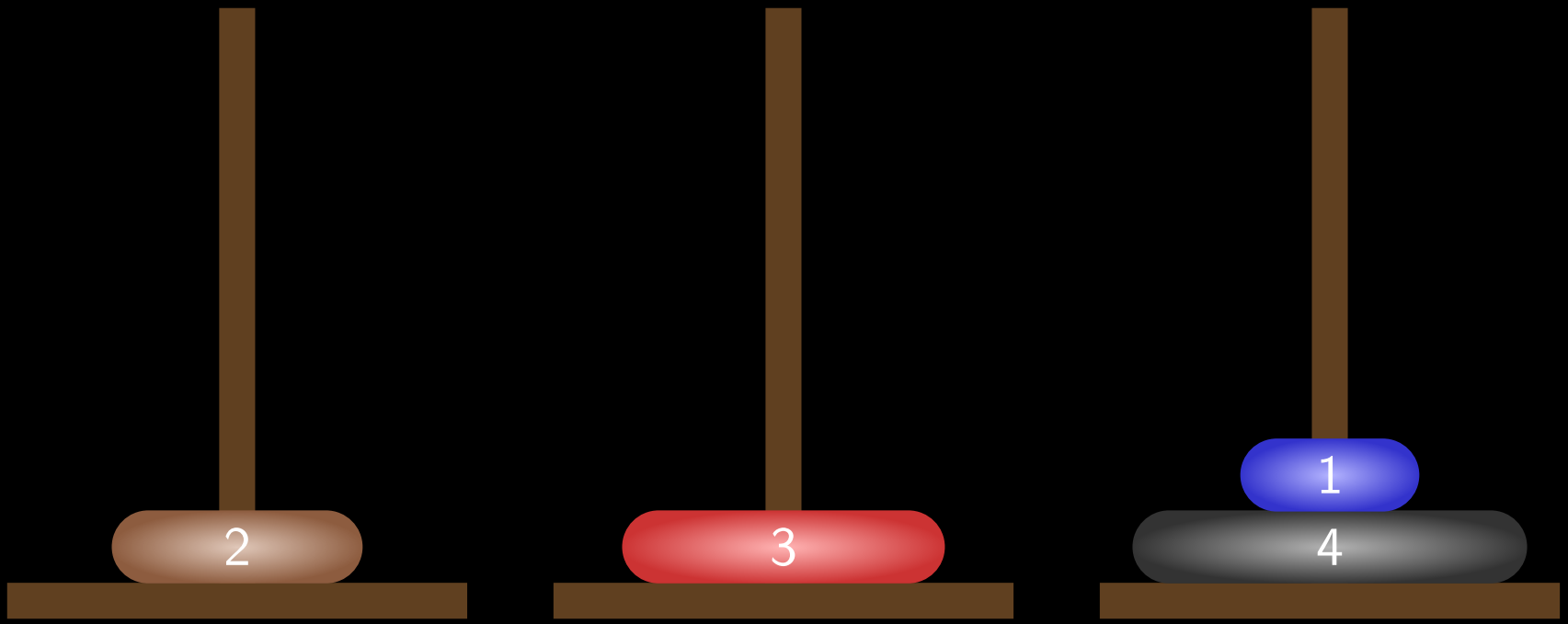
On déplace le disque de la tige 1 vers la tige 3.

Tour de Hanoi – 4 Disques



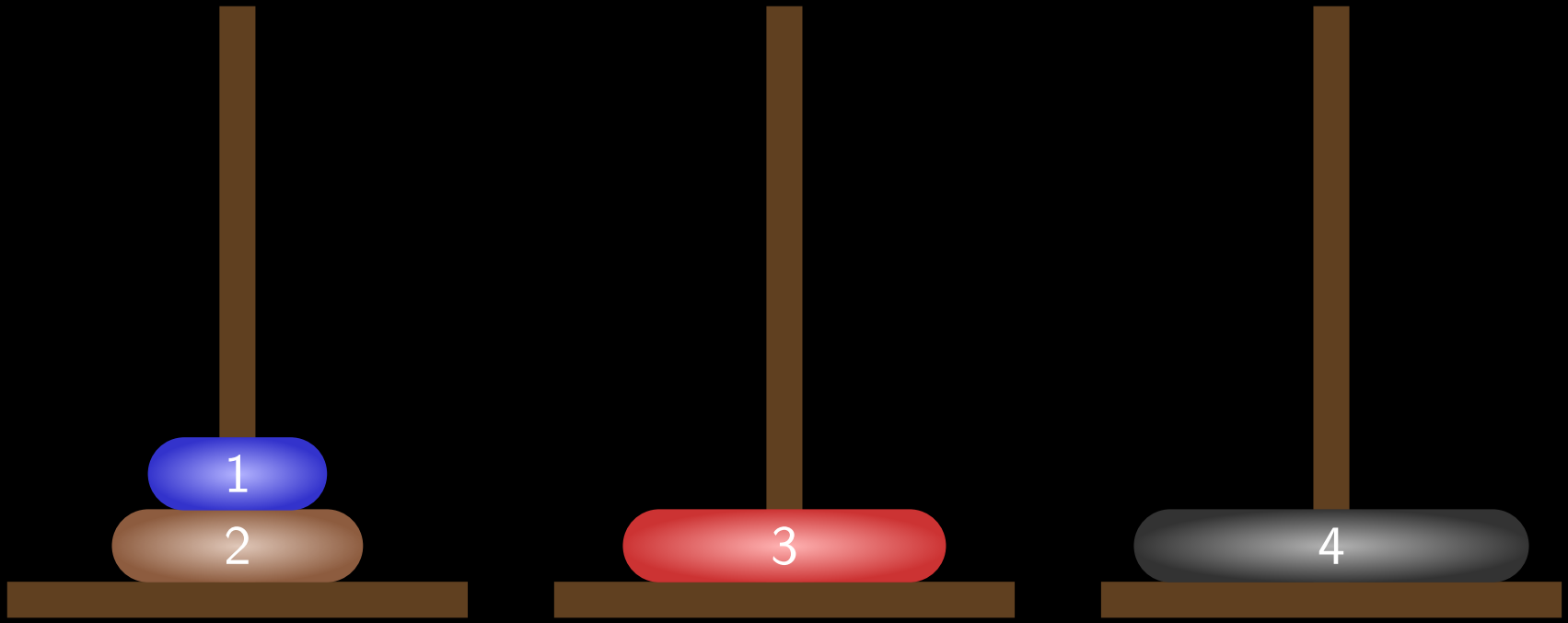
On déplace le disque de la tige 2 vers la tige 3.

Tour de Hanoi – 4 Disques



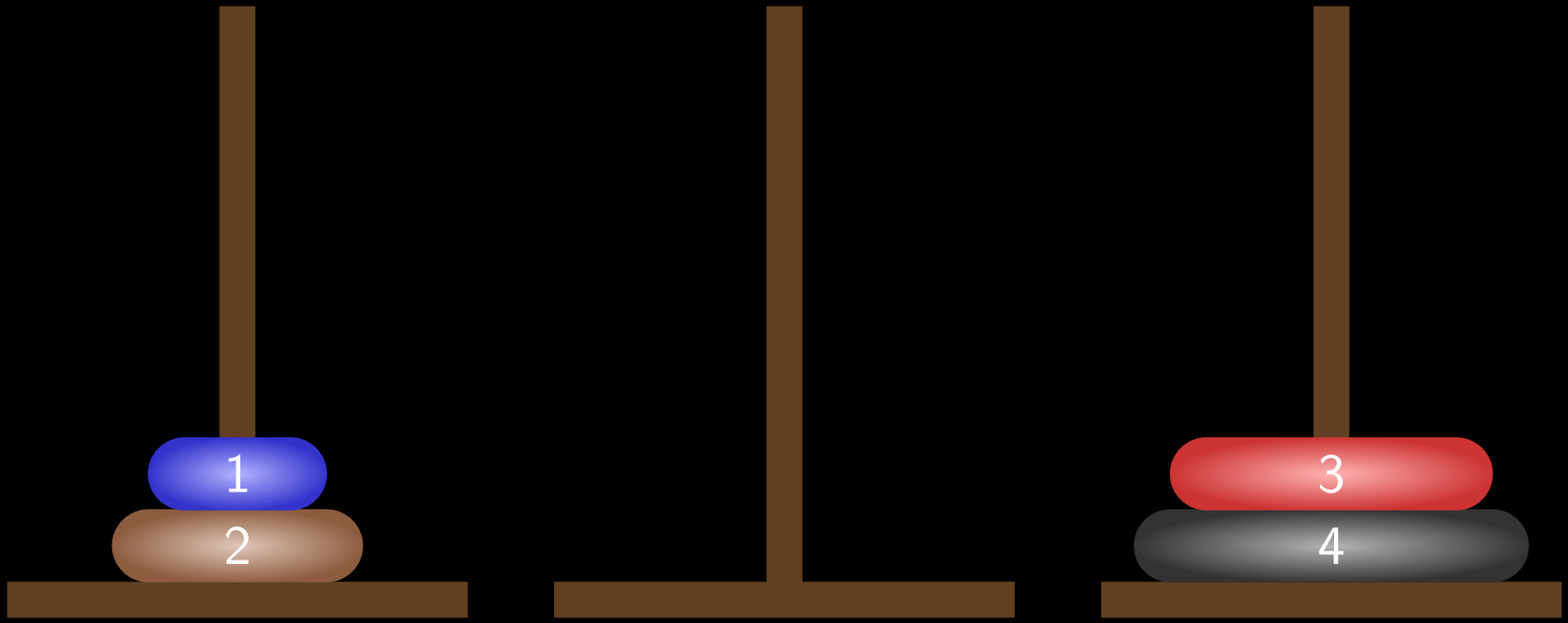
On déplace le disque de la tige 2 vers la tige 1.

Tour de Hanoi – 4 Disques



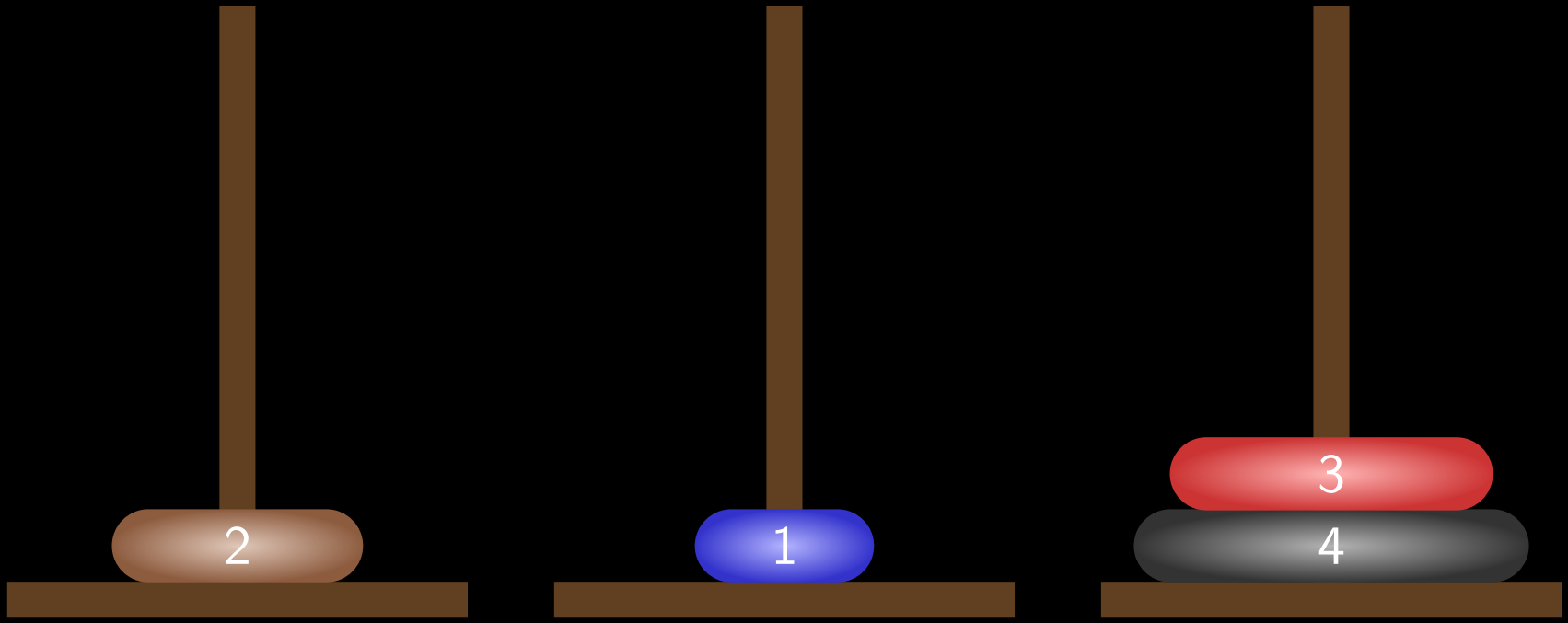
On déplace le disque de la tige 3 vers la tige 1.

Tour de Hanoi – 4 Disques



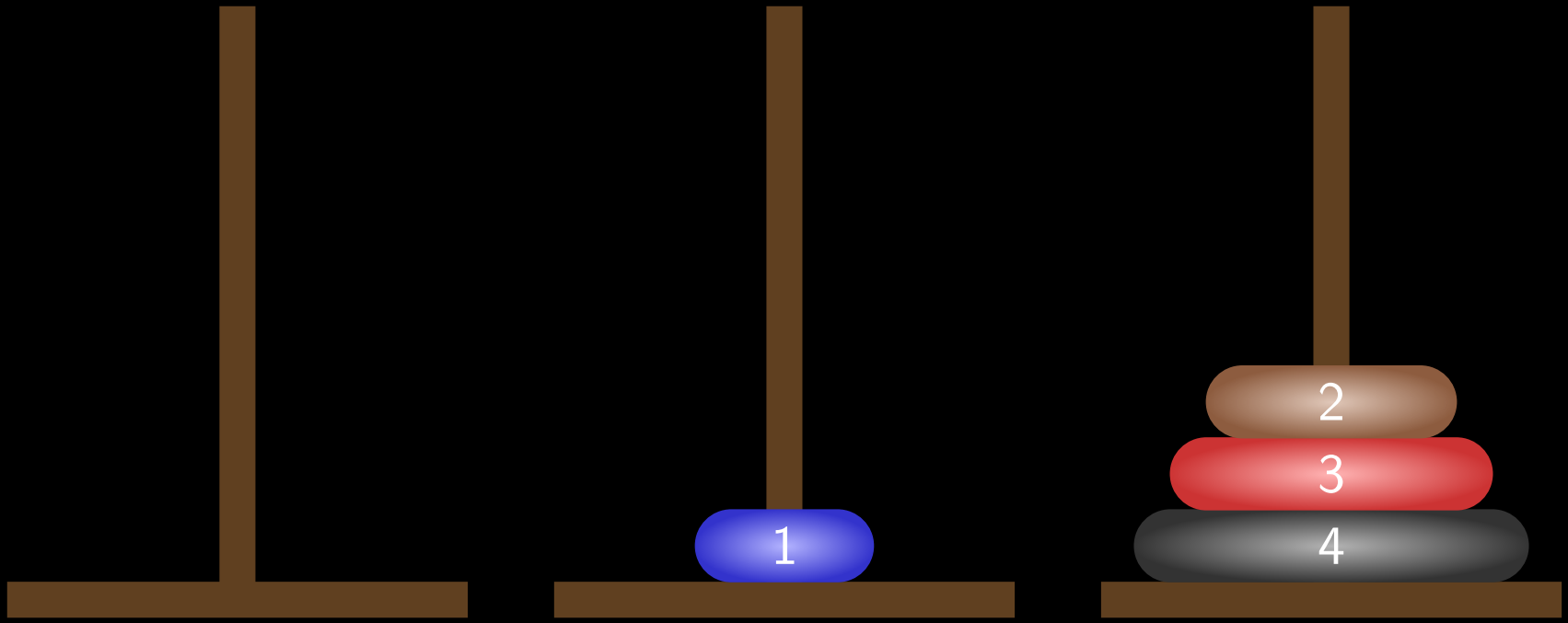
On déplace le disque de la tige 2 vers la tige 3.

Tour de Hanoi – 4 Disques



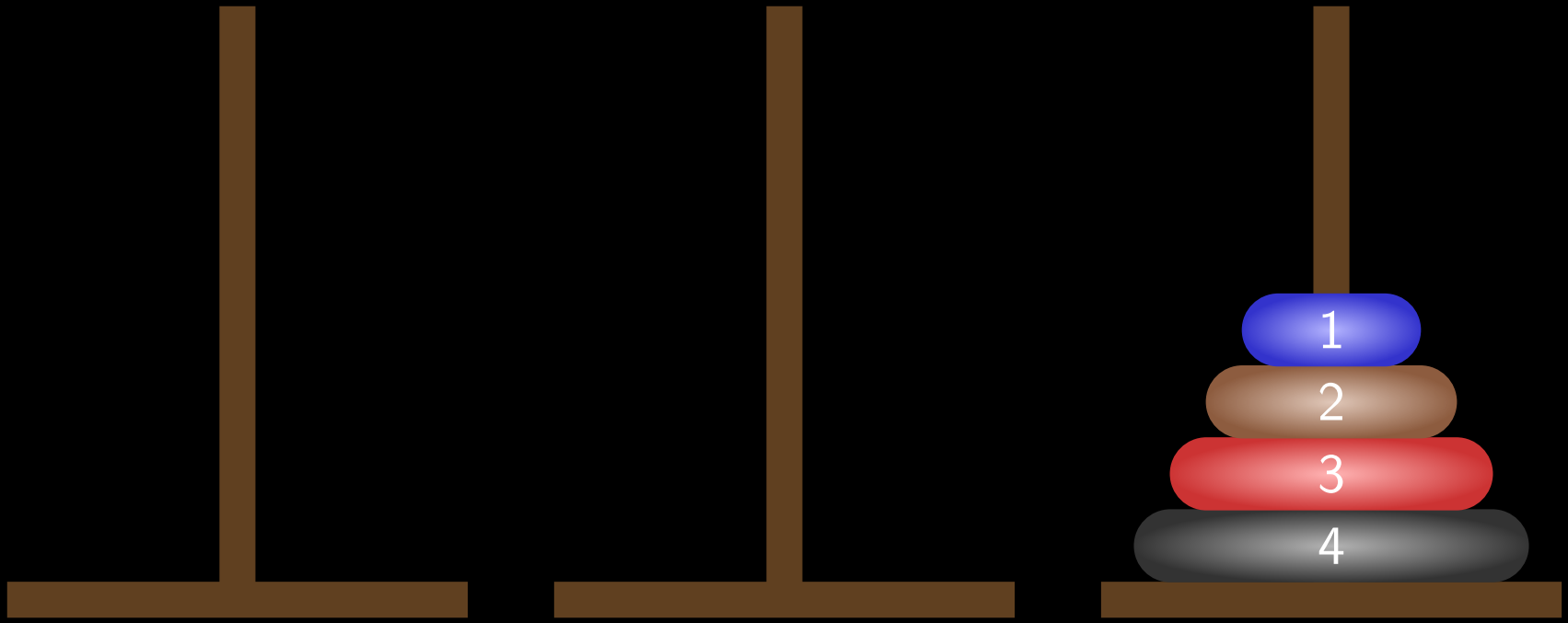
On déplace le disque de la tige 1 vers la tige 2.

Tour de Hanoi – 4 Disques



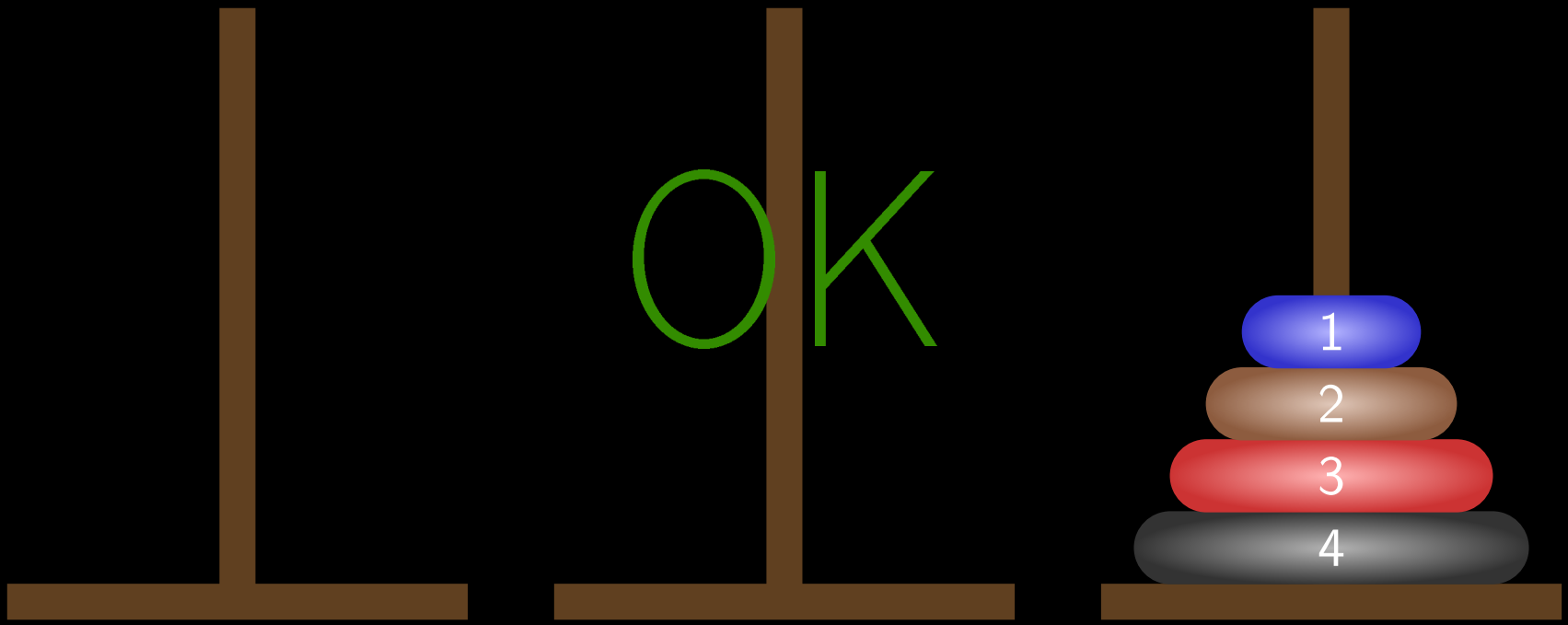
On déplace le disque de la tige 1 vers la tige 3.

Tour de Hanoi – 4 Disques

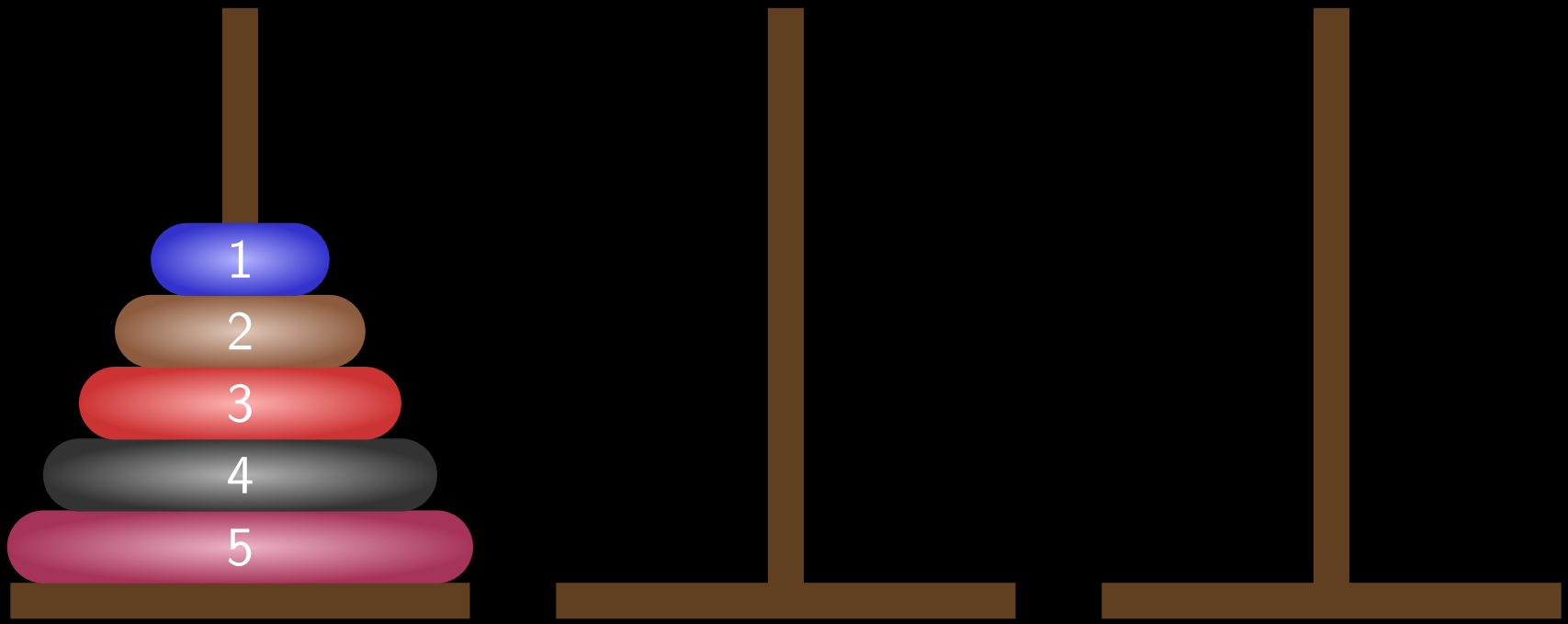


On déplace le disque de la tige 2 vers la tige 3.

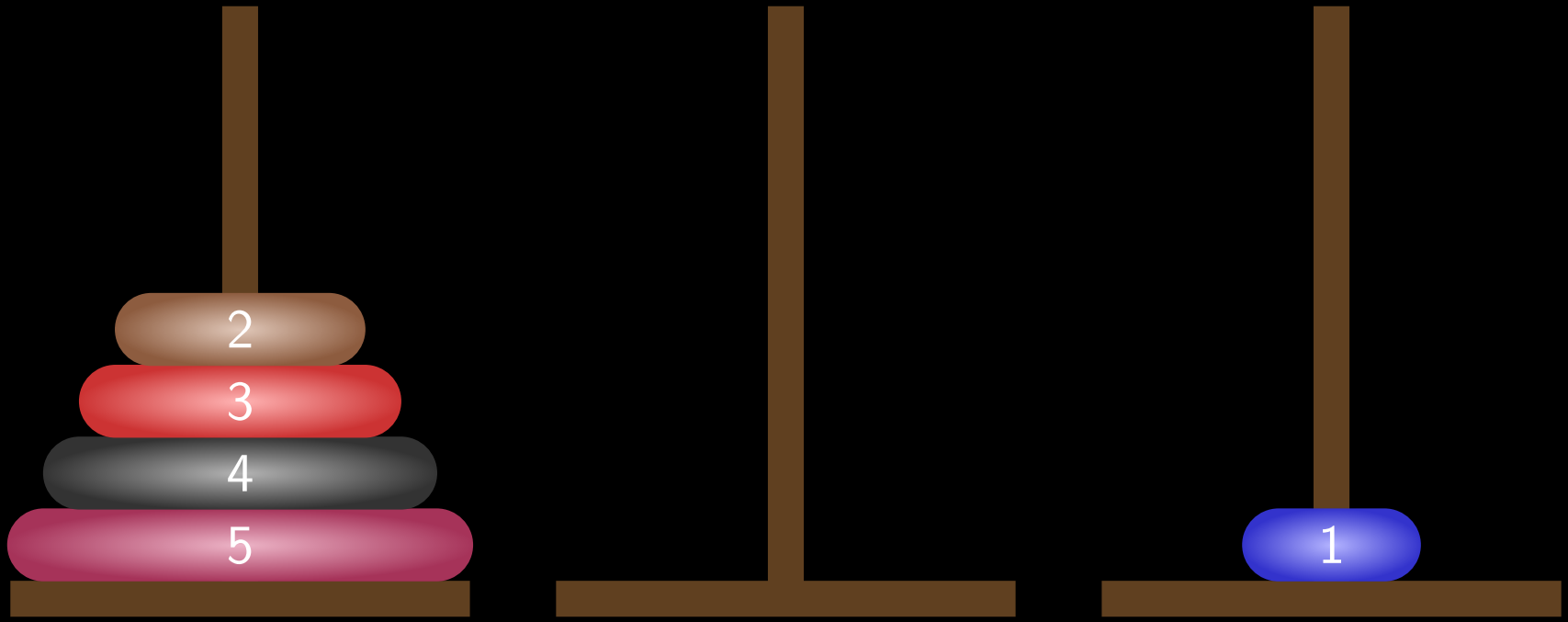
Tour de Hanoi – 4 Disques



Tour de Hanoi – 5 Disques

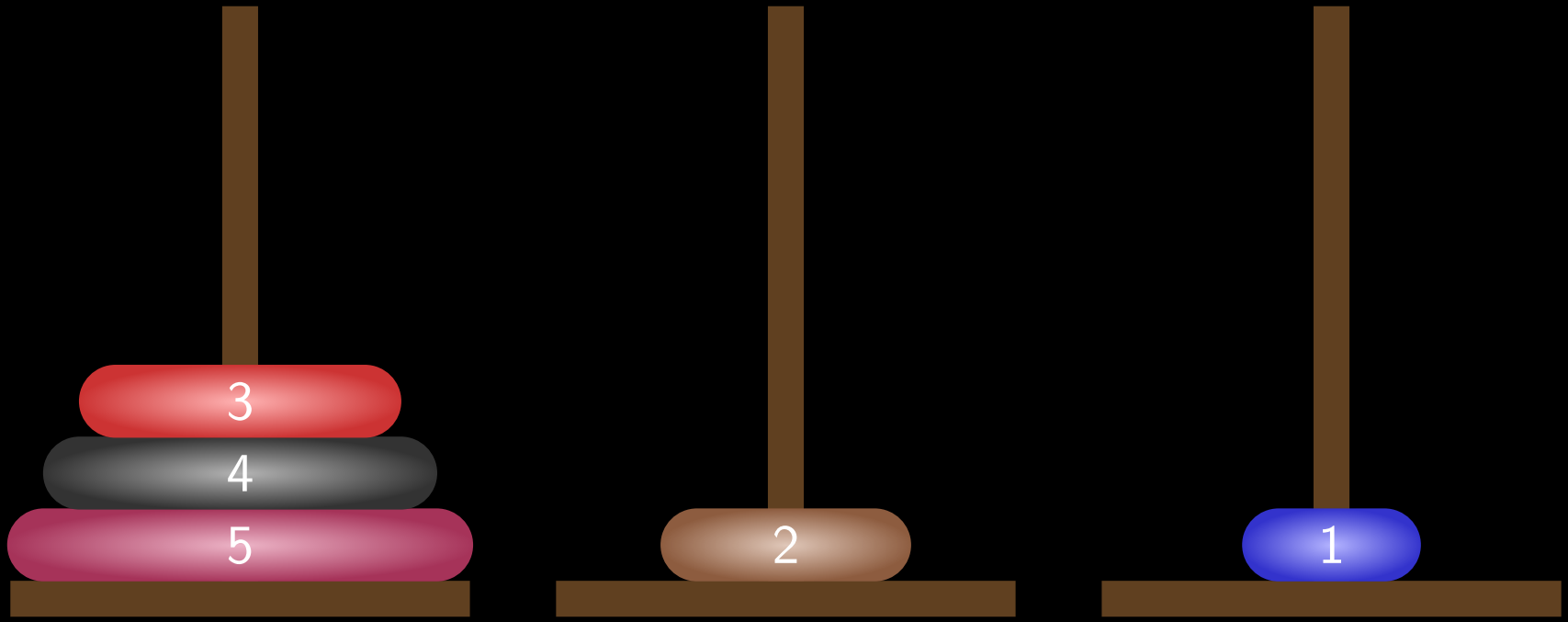


Tour de Hanoi – 5 Disques



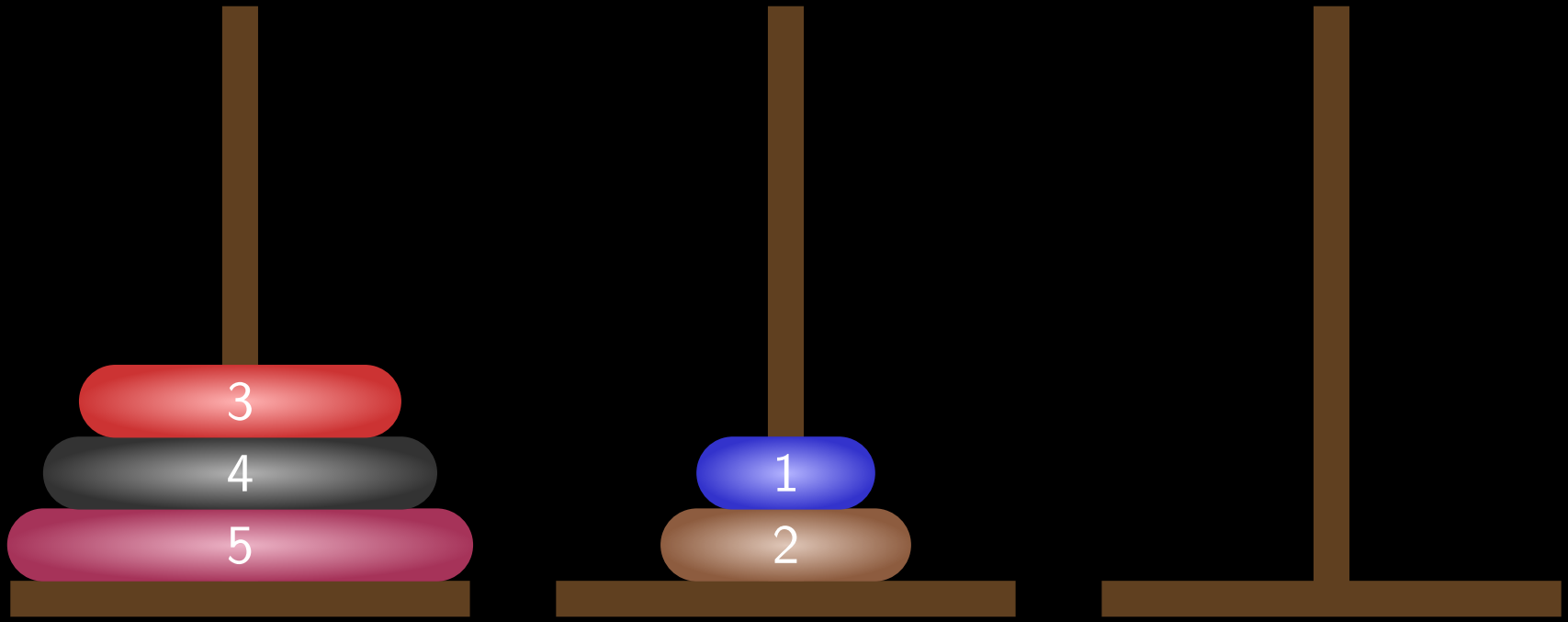
On déplace le disque de la tige 1 vers la tige 3.

Tour de Hanoi – 5 Disques



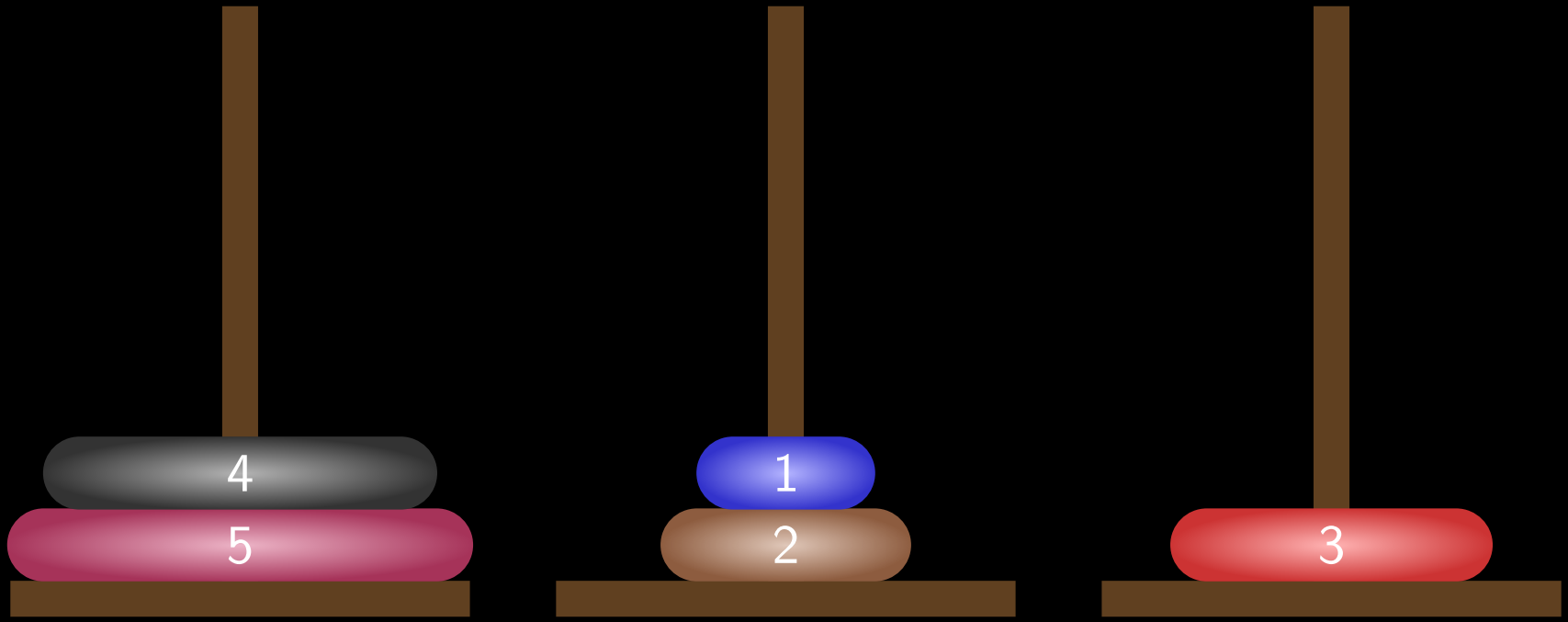
On déplace le disque de la tige 1 vers la tige 2.

Tour de Hanoi – 5 Disques



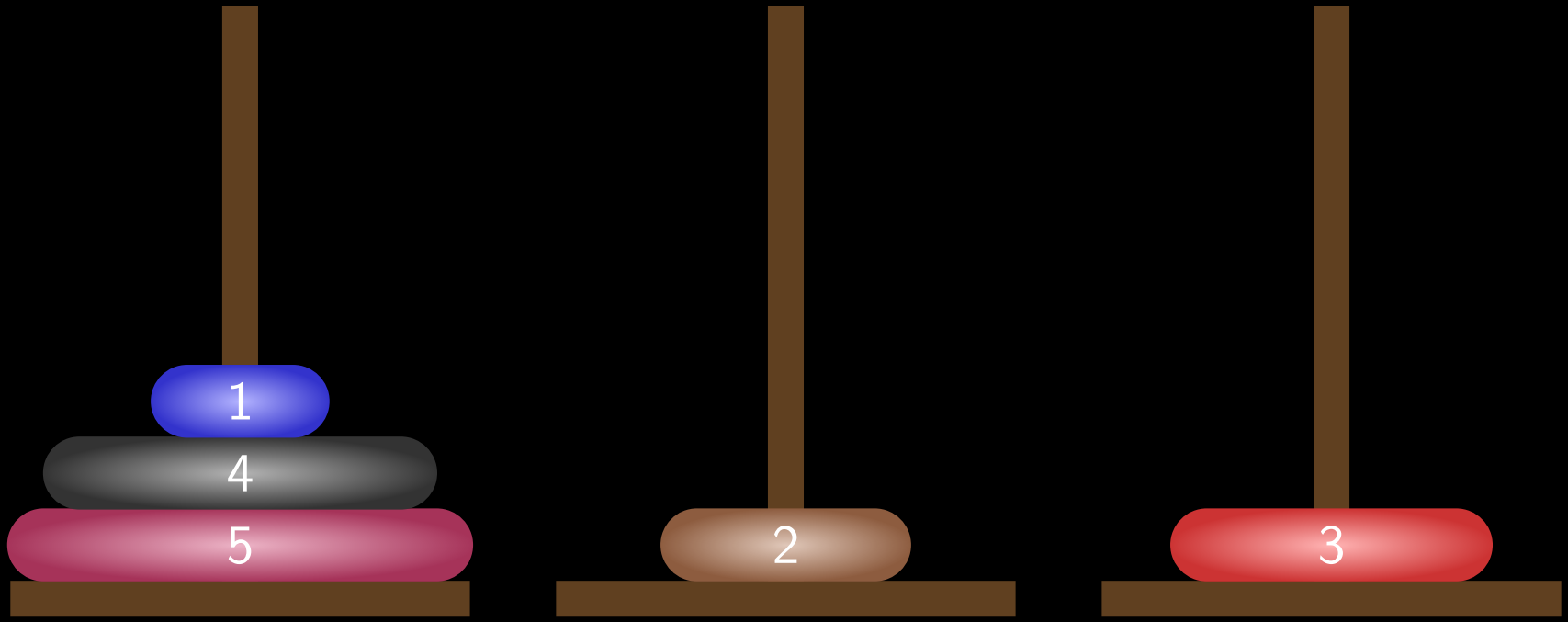
On déplace le disque de la tige 3 vers la tige 2.

Tour de Hanoi – 5 Disques



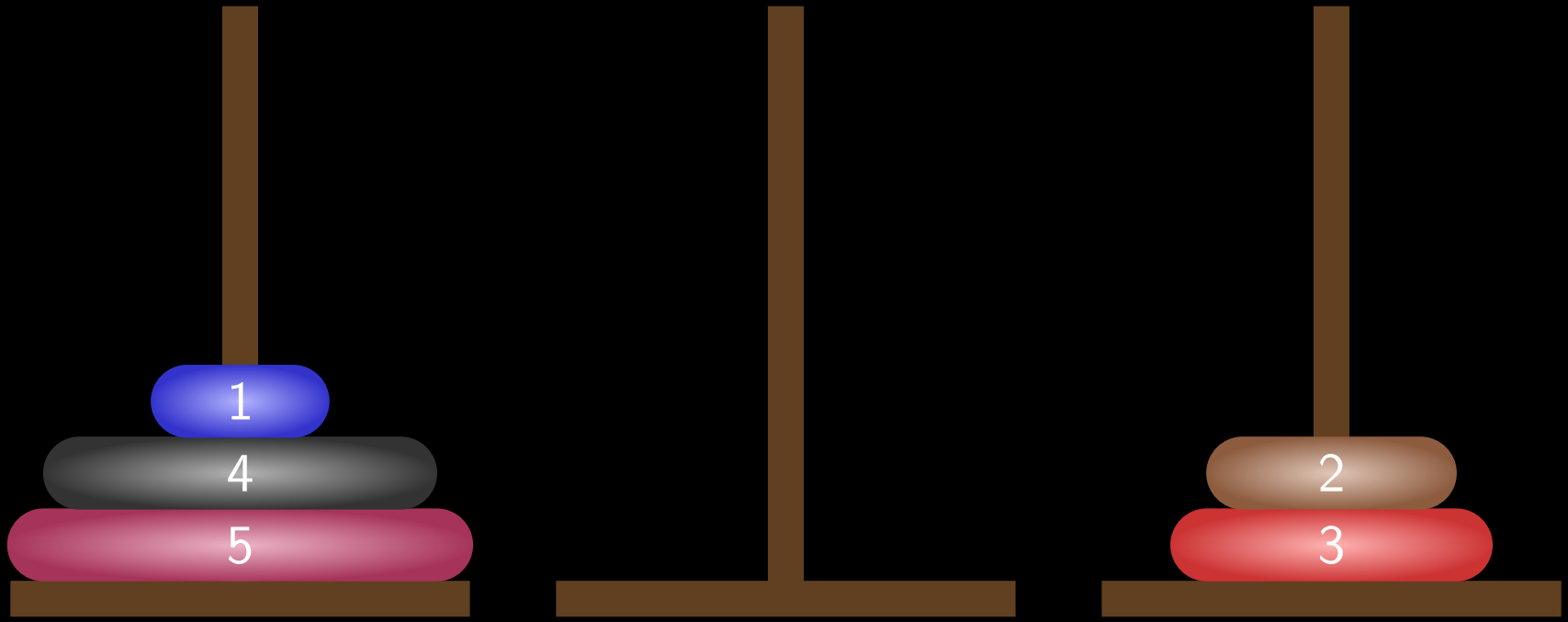
On déplace le disque de la tige 1 vers la tige 3.

Tour de Hanoi – 5 Disques



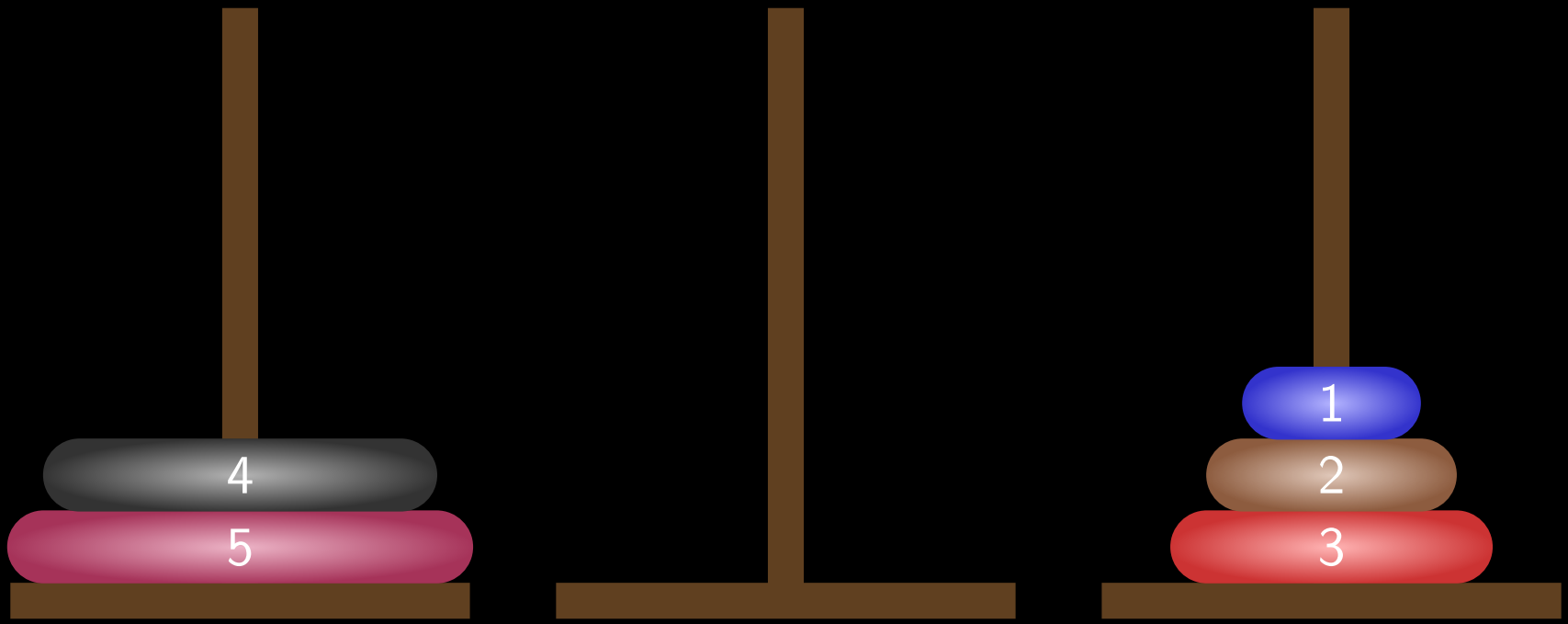
On déplace le disque de la tige 2 vers la tige 1.

Tour de Hanoi – 5 Disques



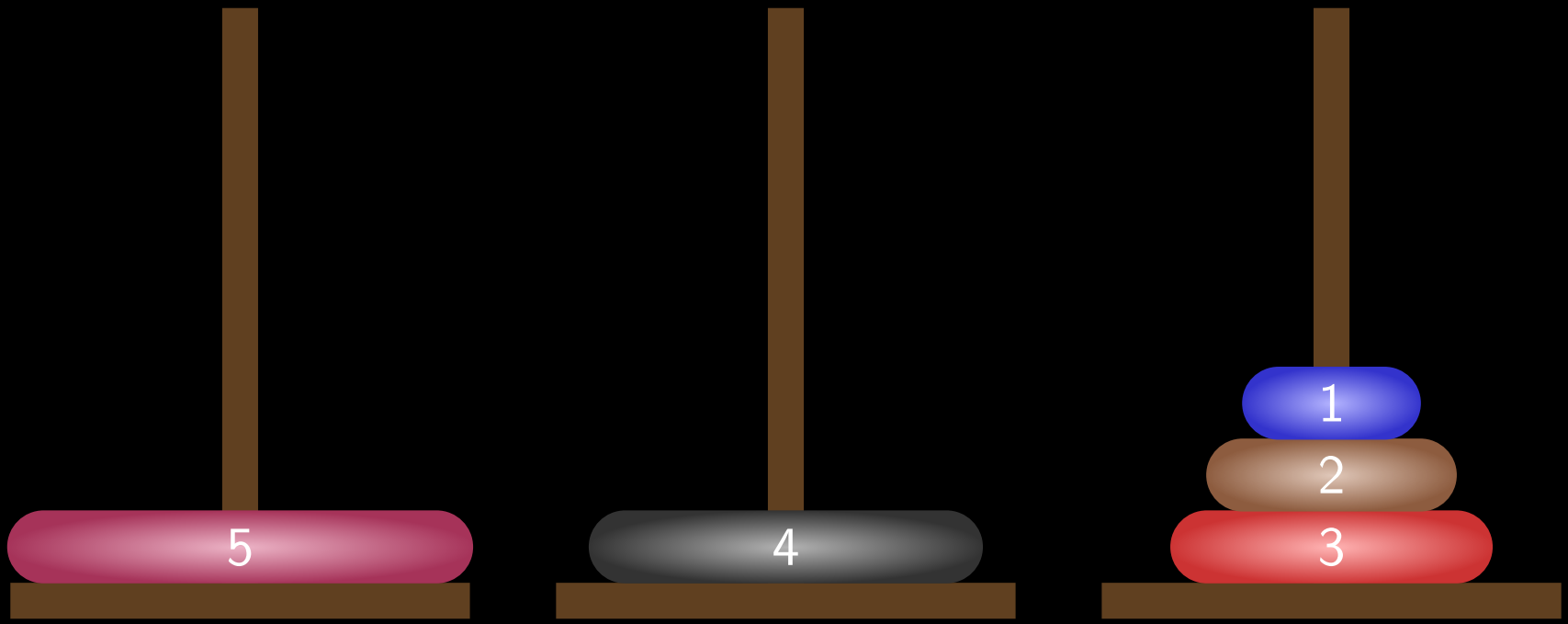
On déplace le disque de la tige 2 vers la tige 3.

Tour de Hanoi – 5 Disques



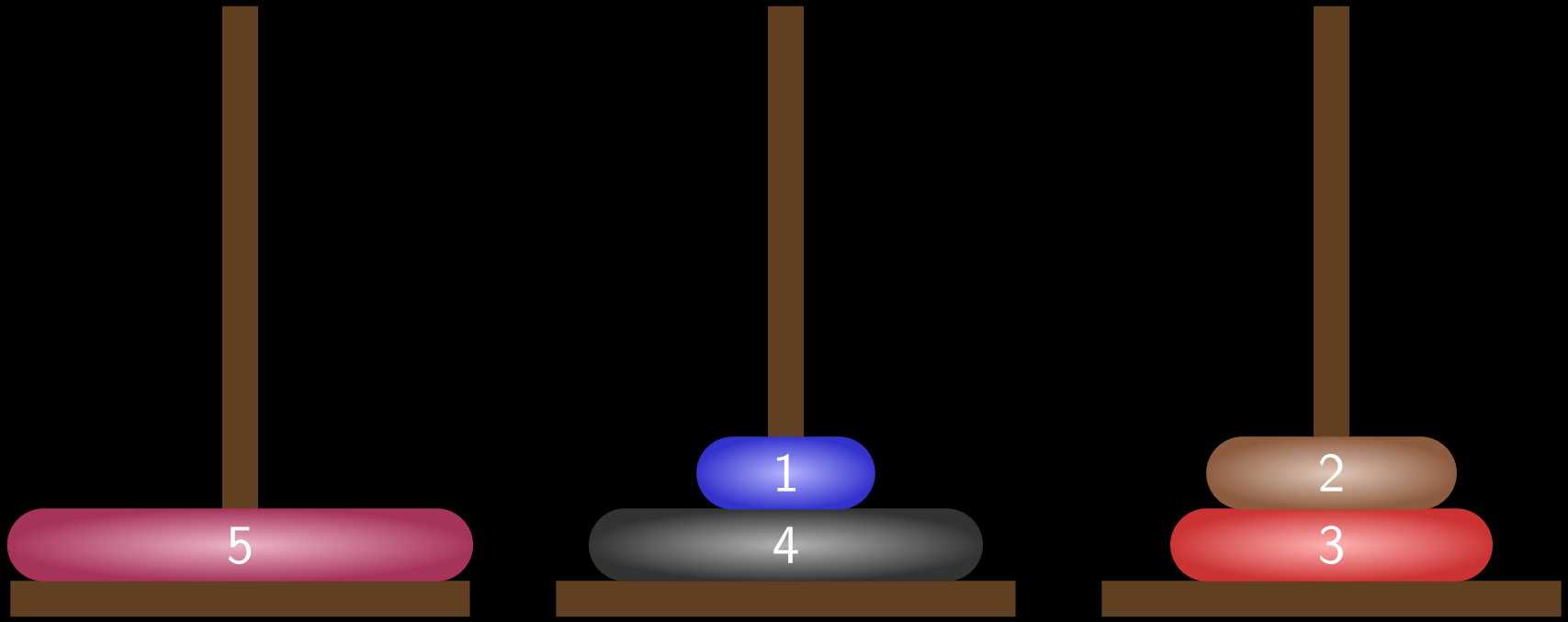
On déplace le disque de la tige 1 vers la tige 3.

Tour de Hanoi – 5 Disques



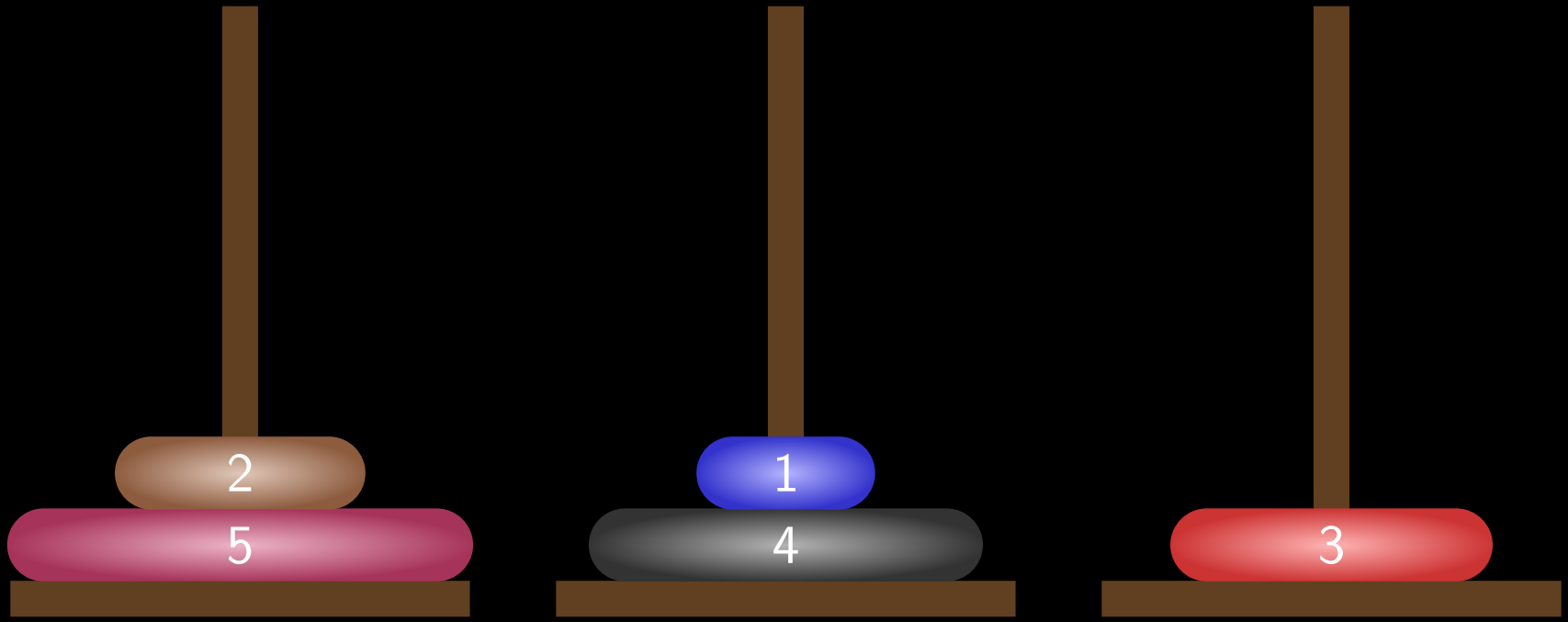
On déplace le disque de la tige 1 vers la tige 2.

Tour de Hanoi – 5 Disques



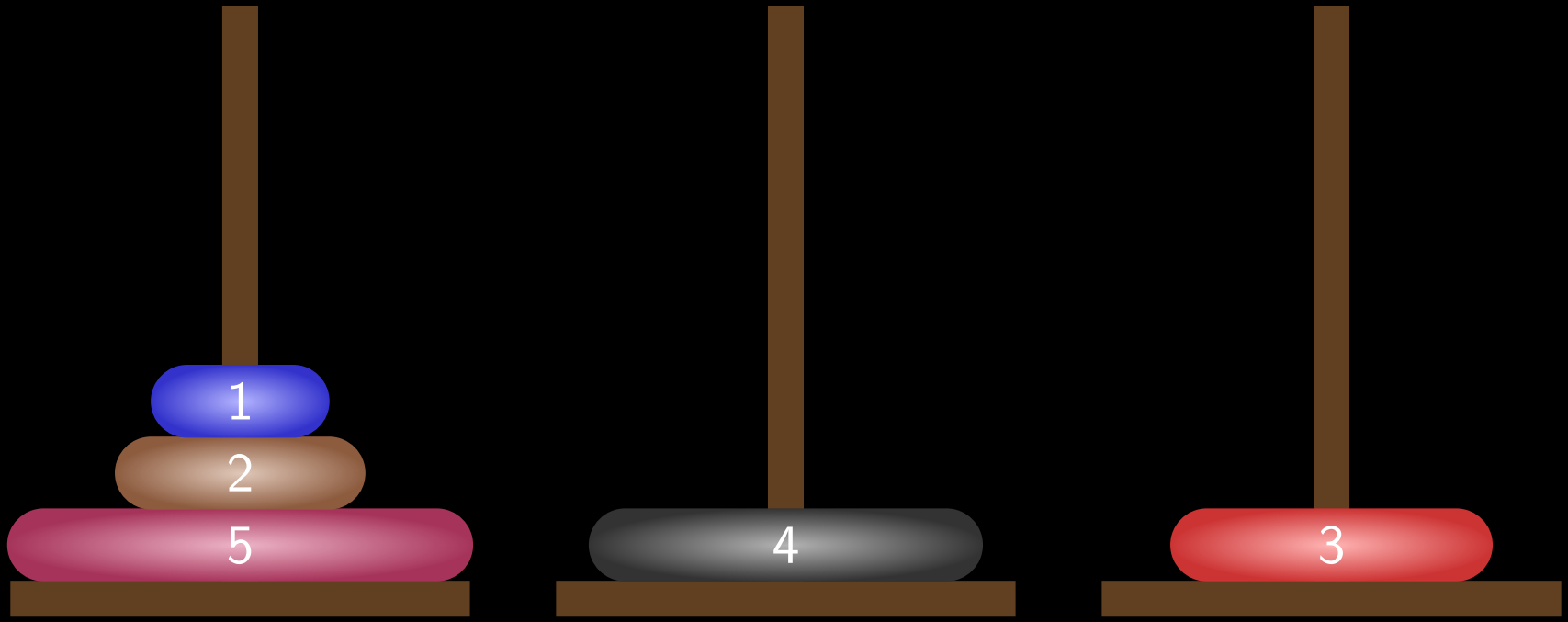
On déplace le disque de la tige 3 vers la tige 2.

Tour de Hanoi – 5 Disques



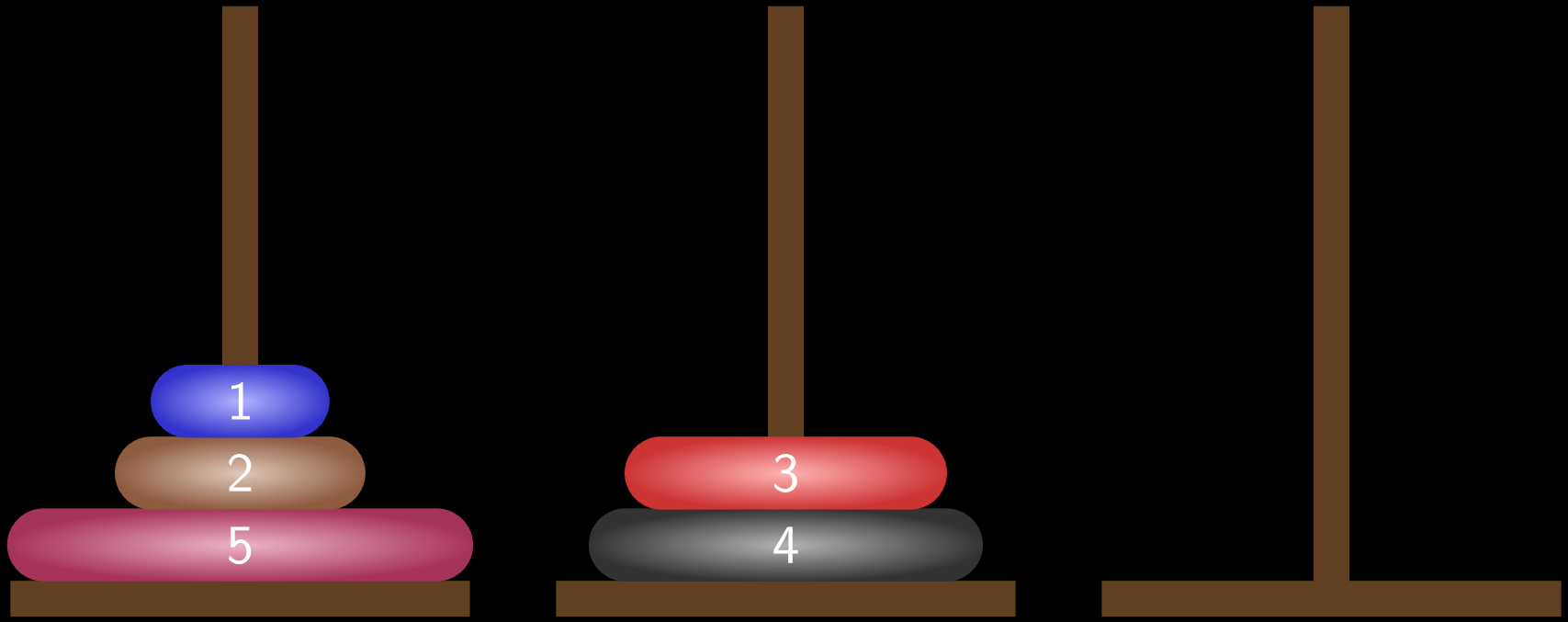
On déplace le disque de la tige 3 vers la tige 1.

Tour de Hanoi – 5 Disques



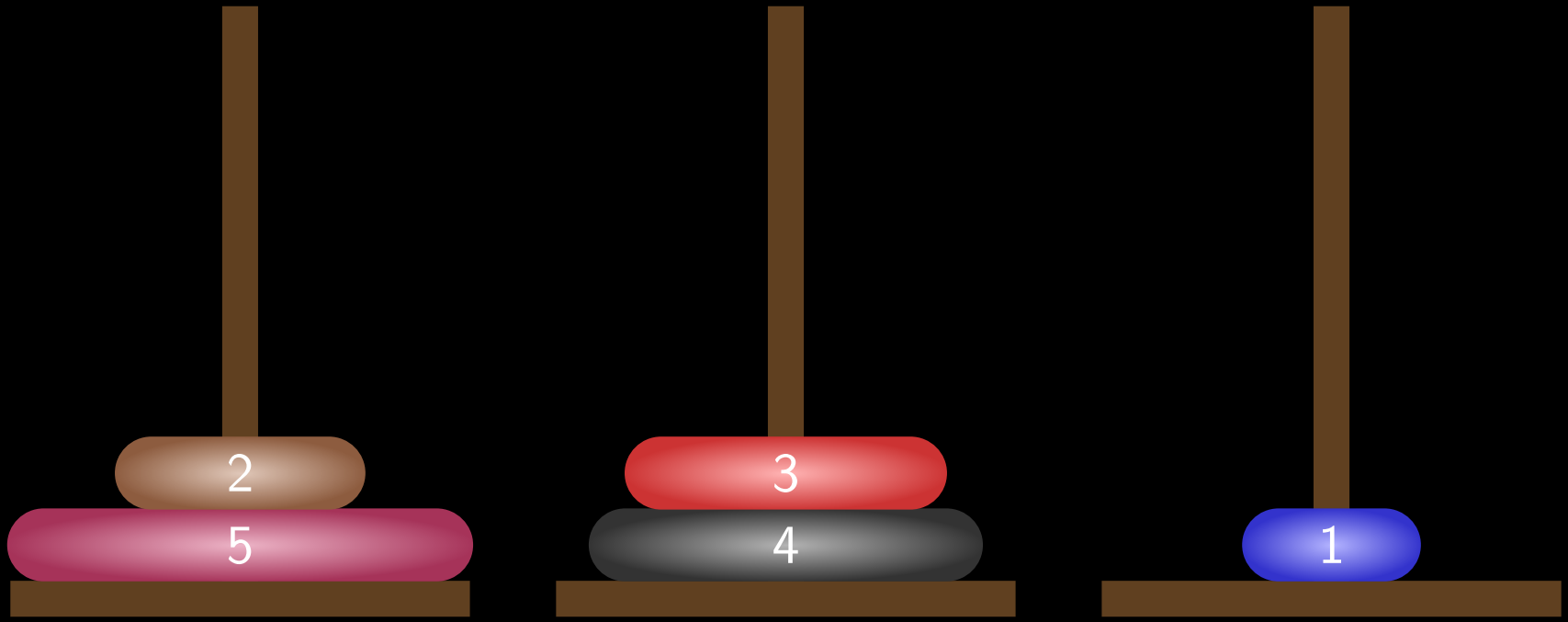
On déplace le disque de la tige 2 vers la tige 1.

Tour de Hanoi – 5 Disques



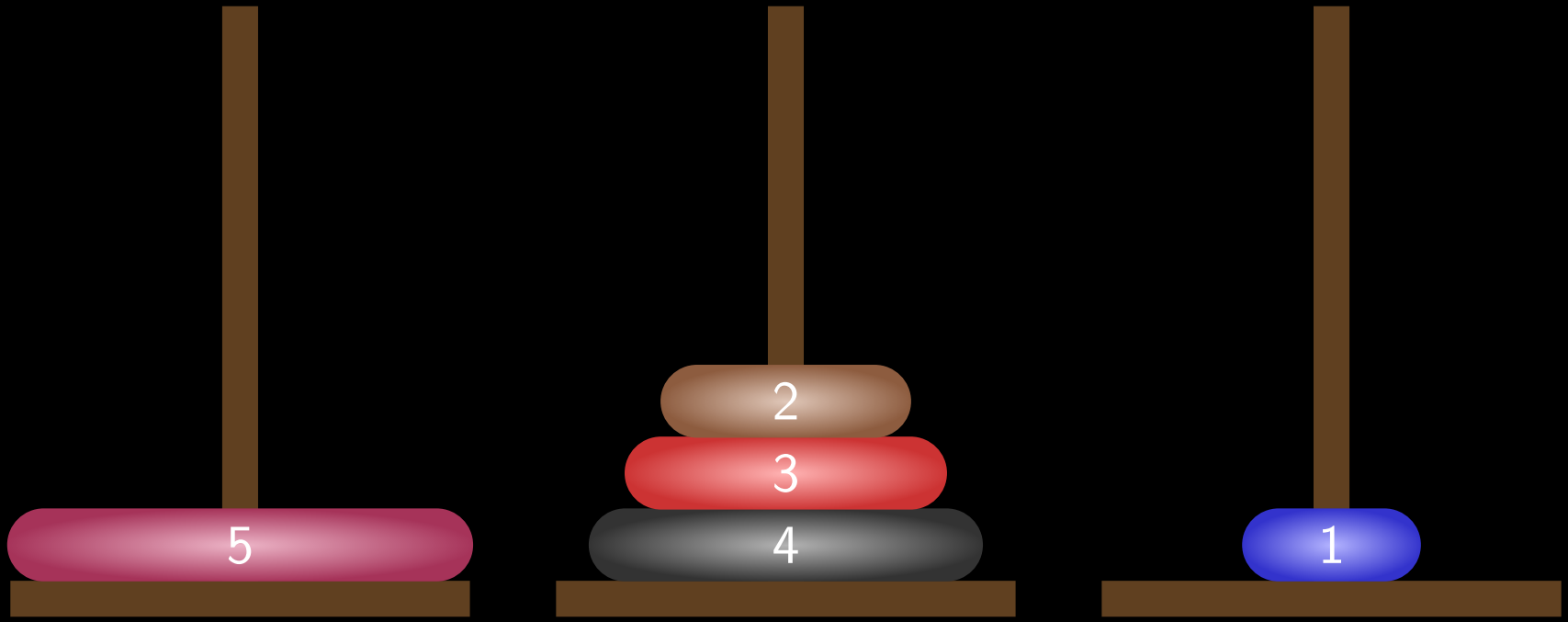
On déplace le disque de la tige 3 vers la tige 2.

Tour de Hanoi – 5 Disques



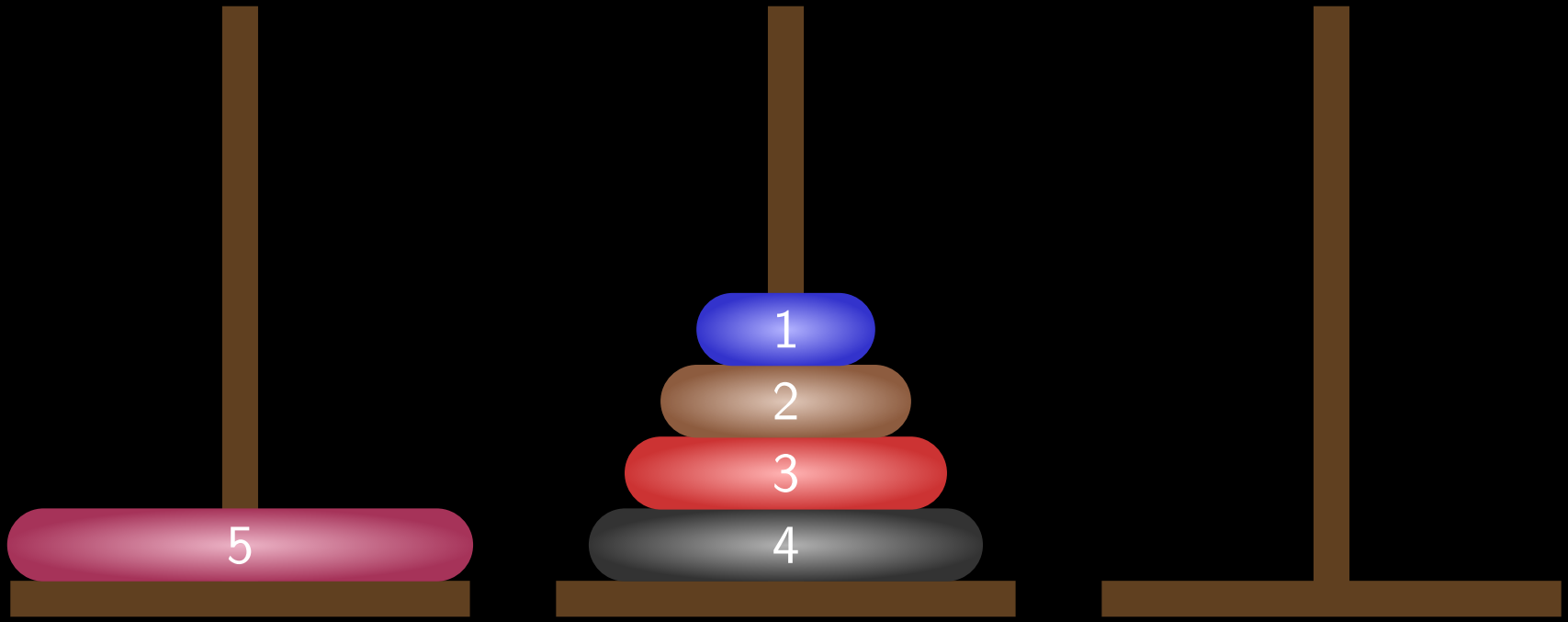
On déplace le disque de la tige 1 vers la tige 3.

Tour de Hanoi – 5 Disques



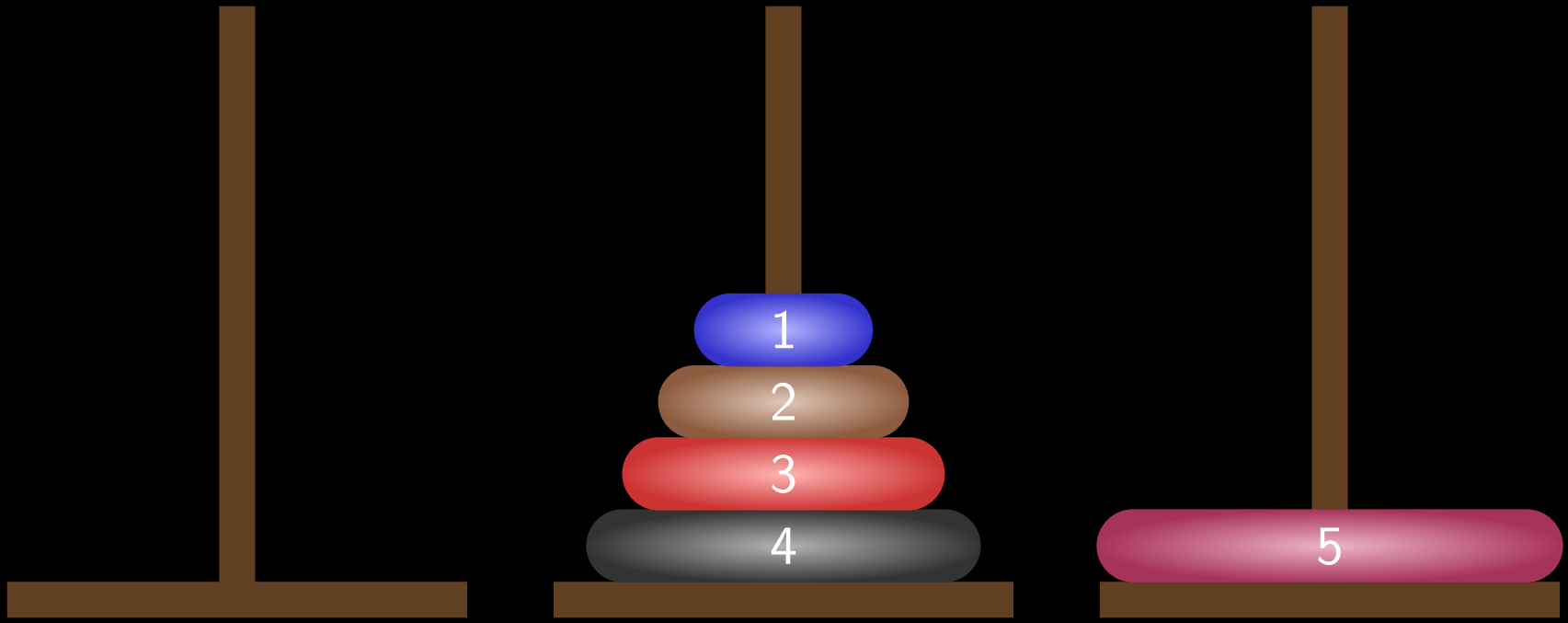
On déplace le disque de la tige 1 vers la tige 2.

Tour de Hanoi – 5 Disques



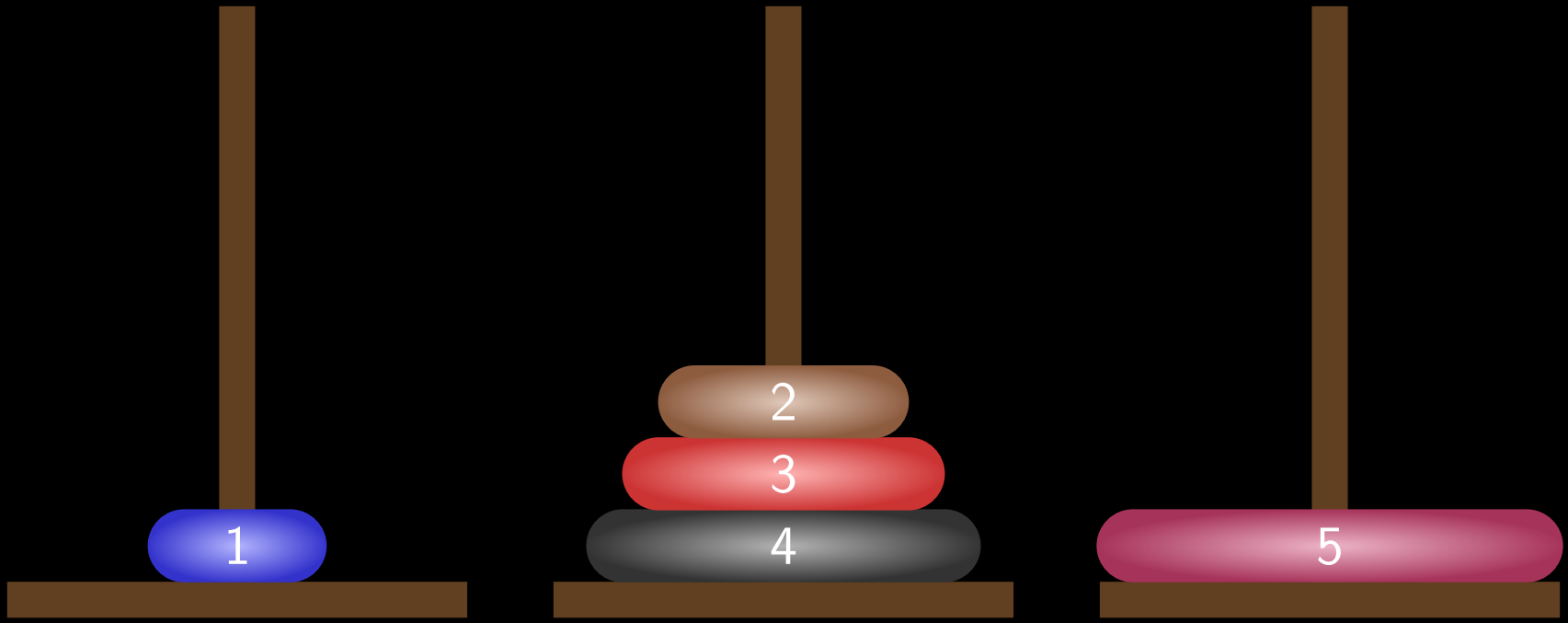
On déplace le disque de la tige 3 vers la tige 2.

Tour de Hanoi – 5 Disques



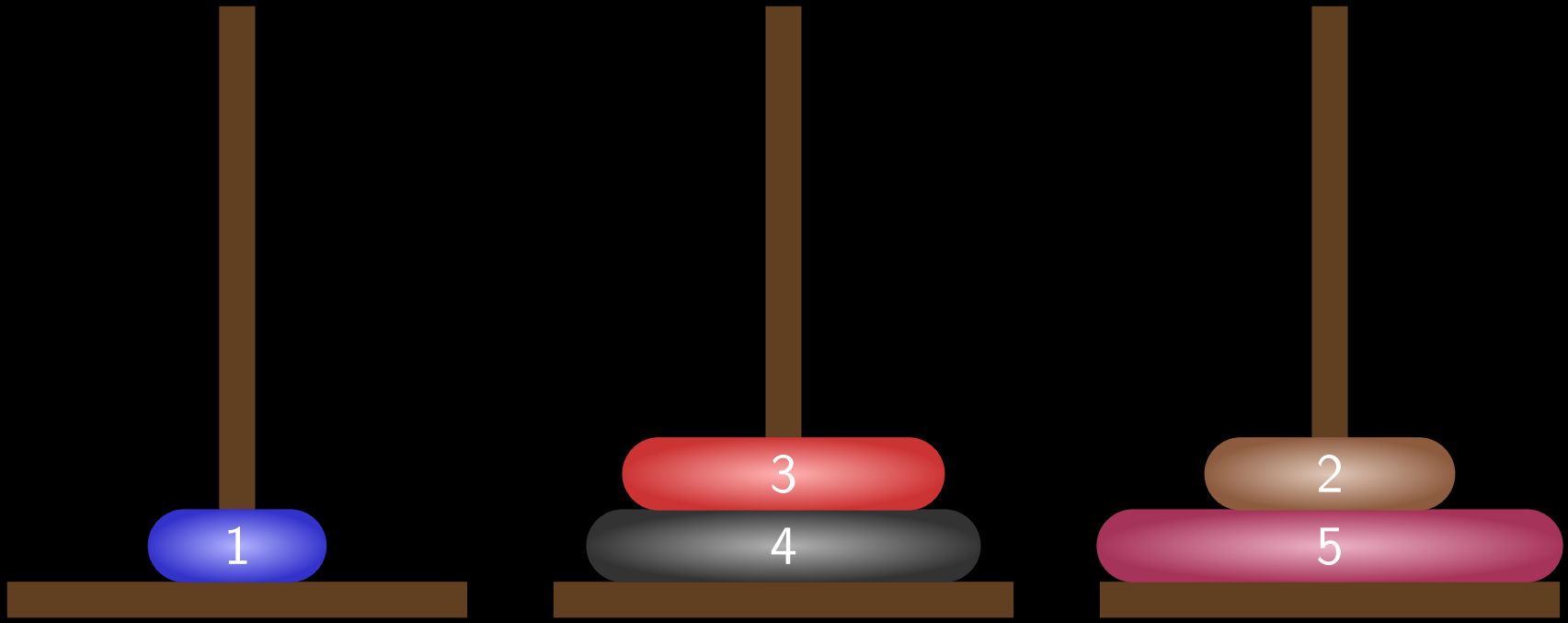
On déplace le disque de la tige 1 vers la tige 3.

Tour de Hanoi – 5 Disques



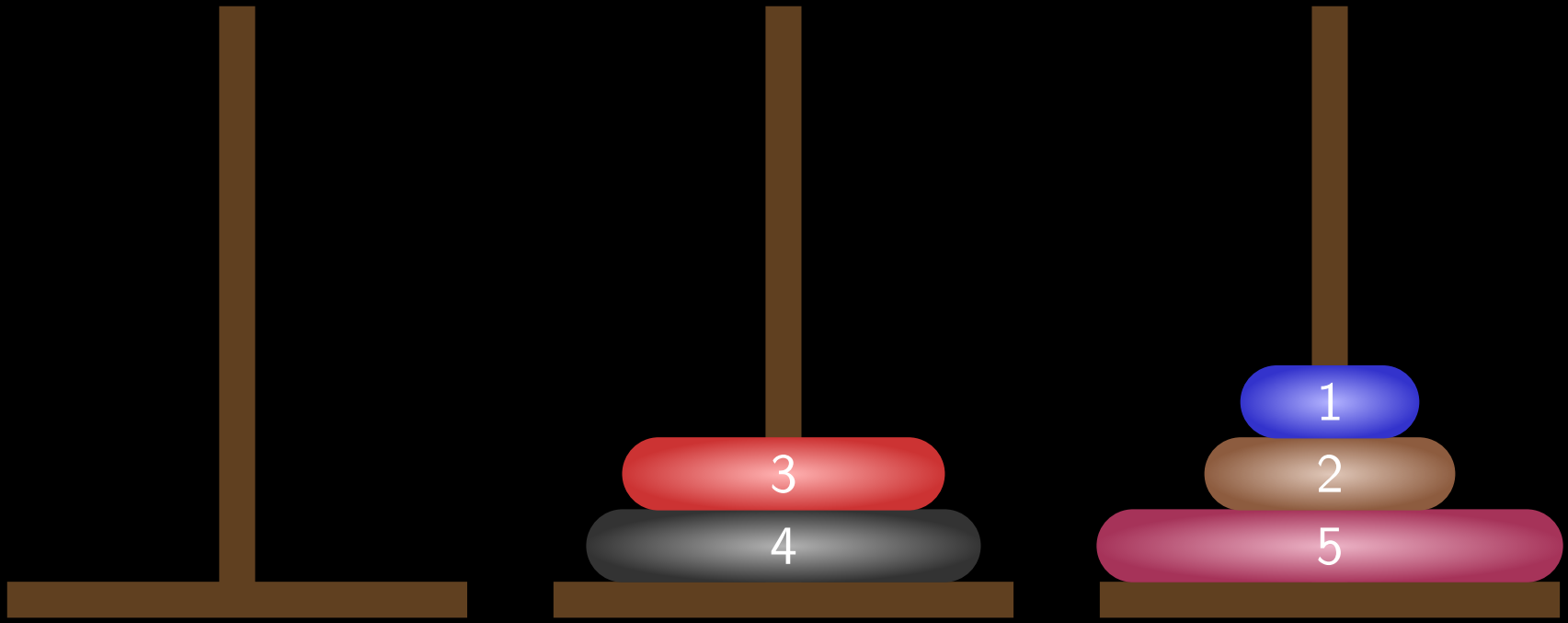
On déplace le disque de la tige 2 vers la tige 1.

Tour de Hanoi – 5 Disques



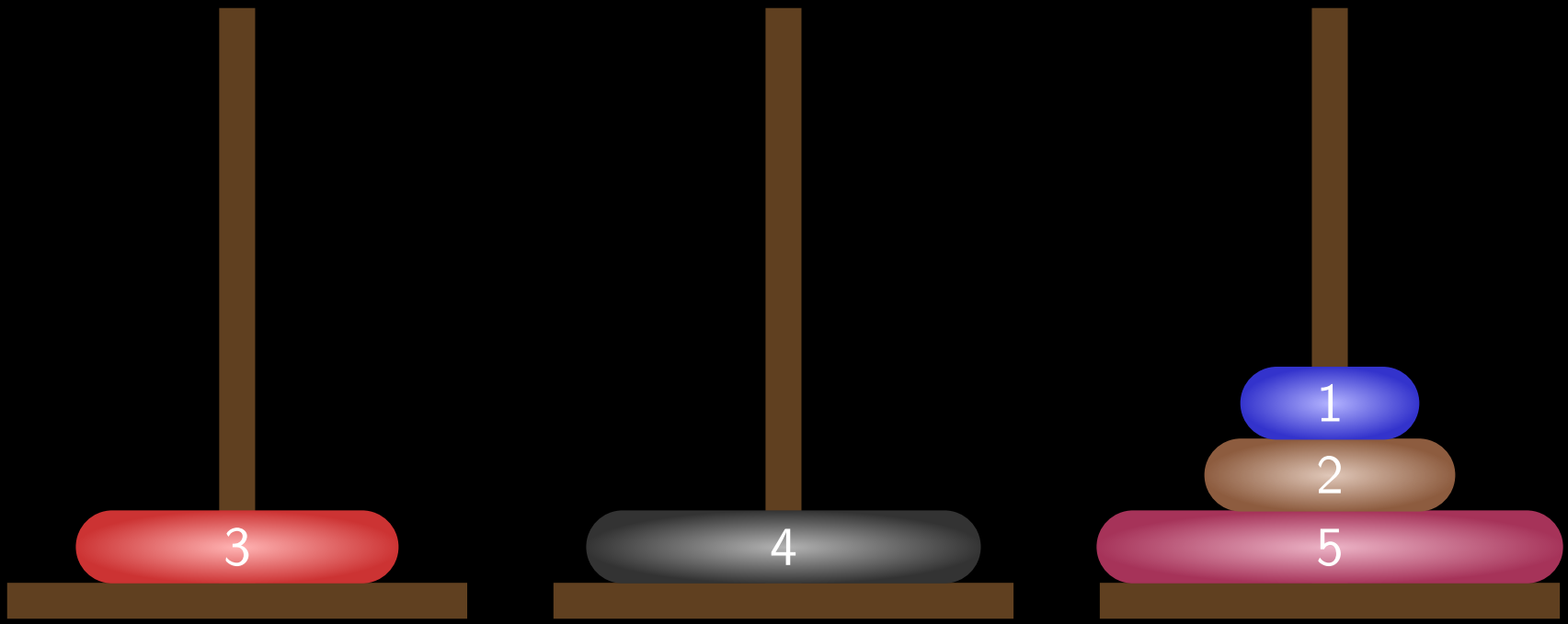
On déplace le disque de la tige 2 vers la tige 3.

Tour de Hanoi – 5 Disques



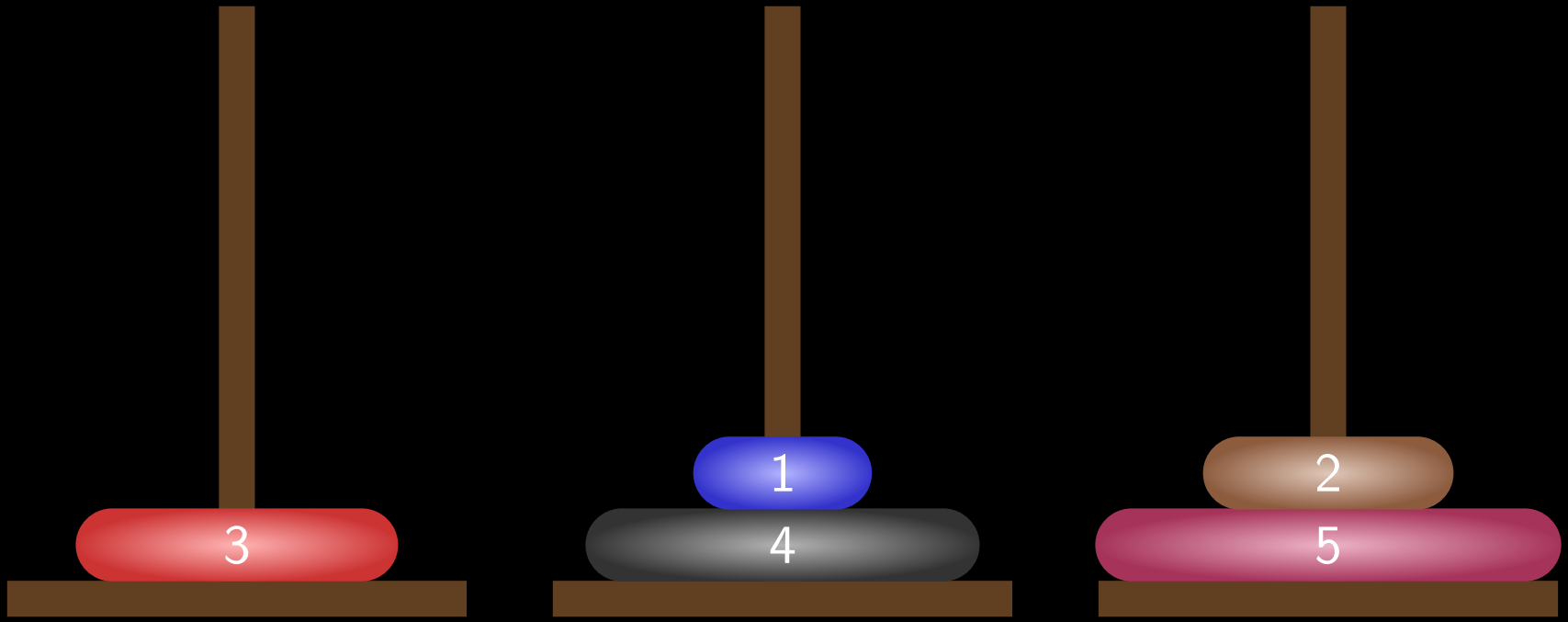
On déplace le disque de la tige 1 vers la tige 3.

Tour de Hanoi – 5 Disques



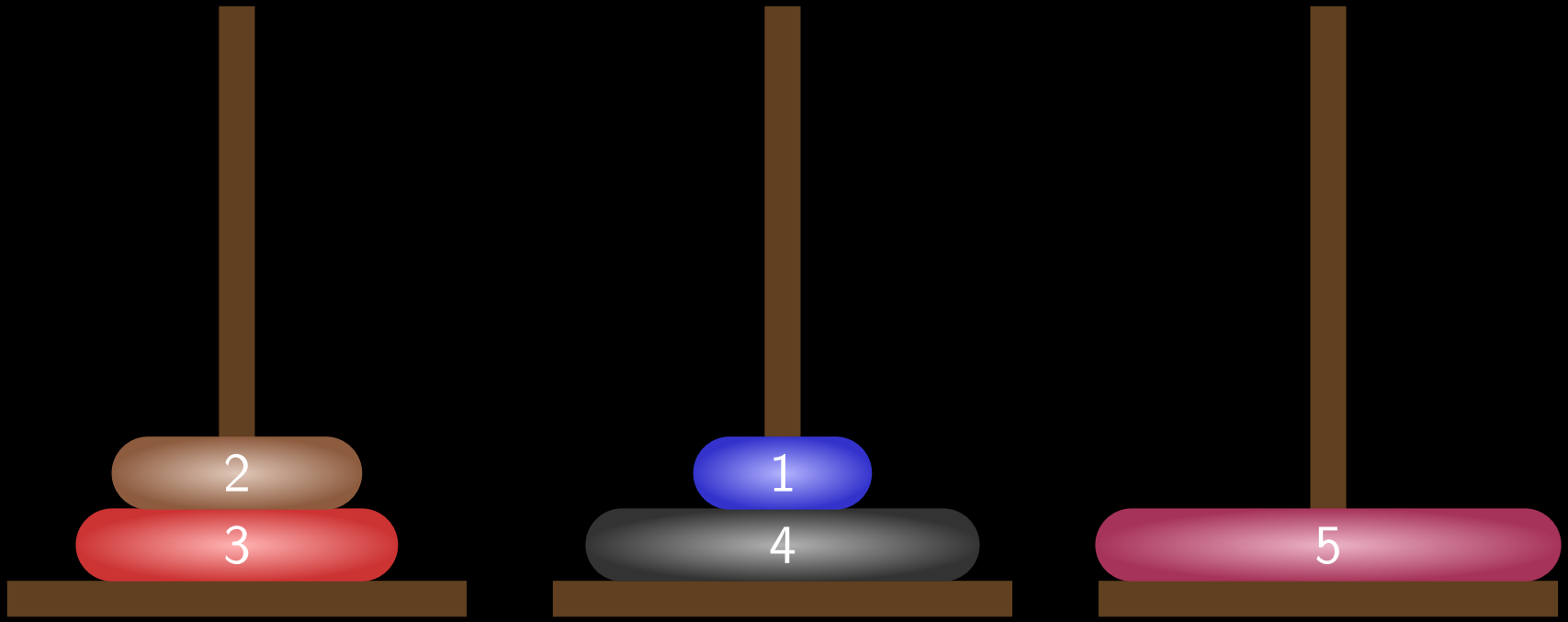
On déplace le disque de la tige 2 vers la tige 1.

Tour de Hanoi – 5 Disques



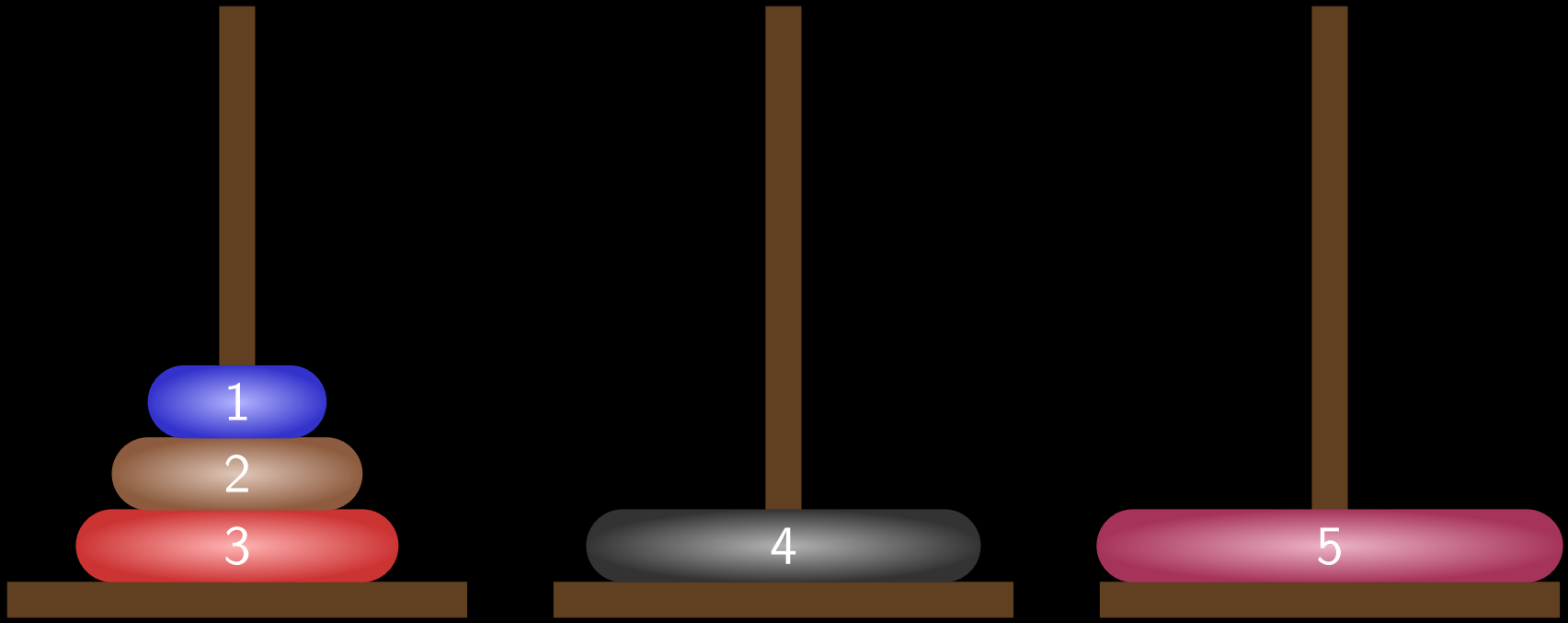
On déplace le disque de la tige 3 vers la tige 2.

Tour de Hanoi – 5 Disques



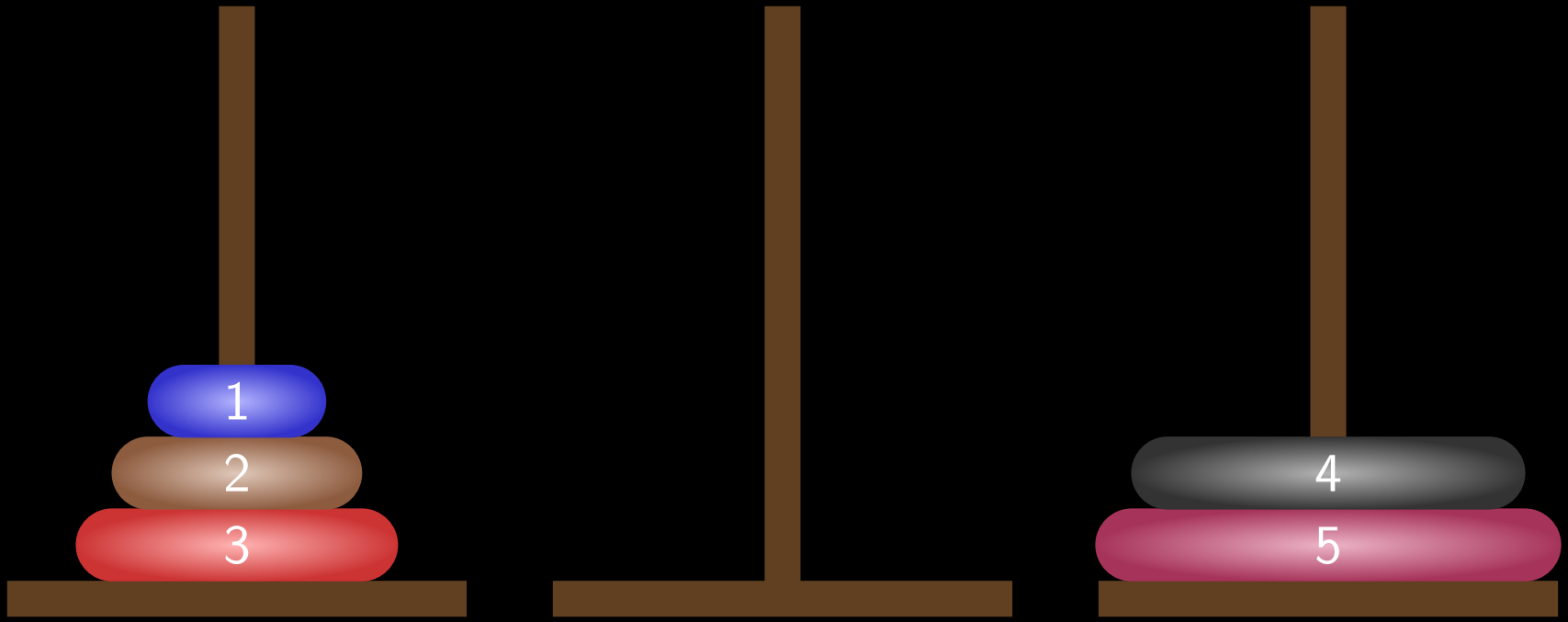
On déplace le disque de la tige 3 vers la tige 1.

Tour de Hanoi – 5 Disques



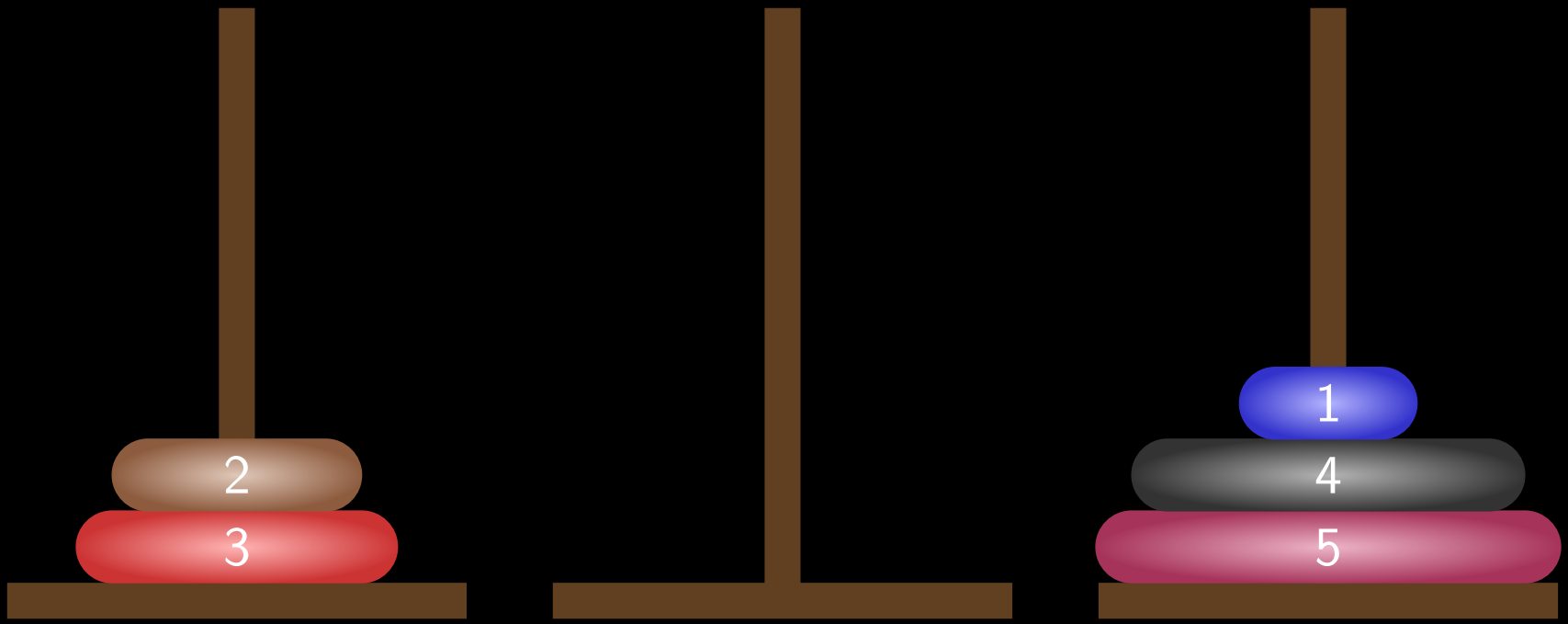
On déplace le disque de la tige 2 vers la tige 1.

Tour de Hanoi – 5 Disques



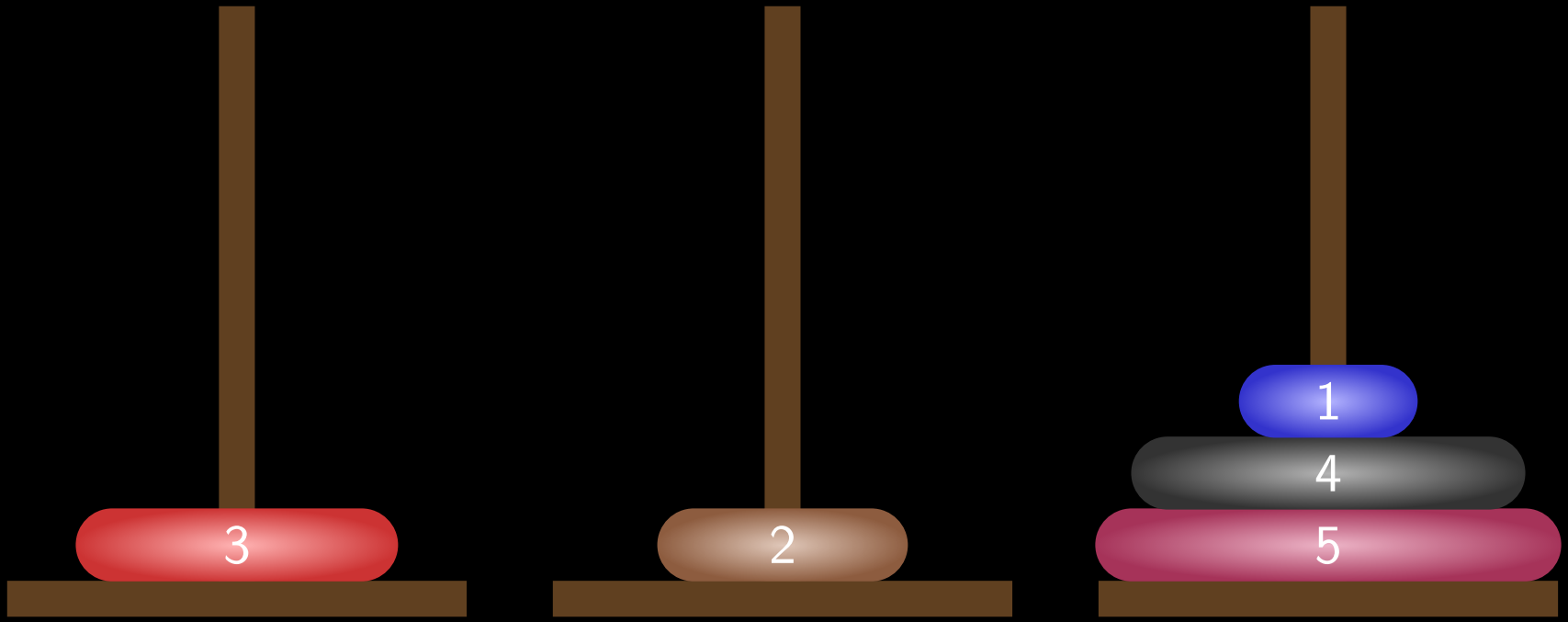
On déplace le disque de la tige 2 vers la tige 3.

Tour de Hanoi – 5 Disques



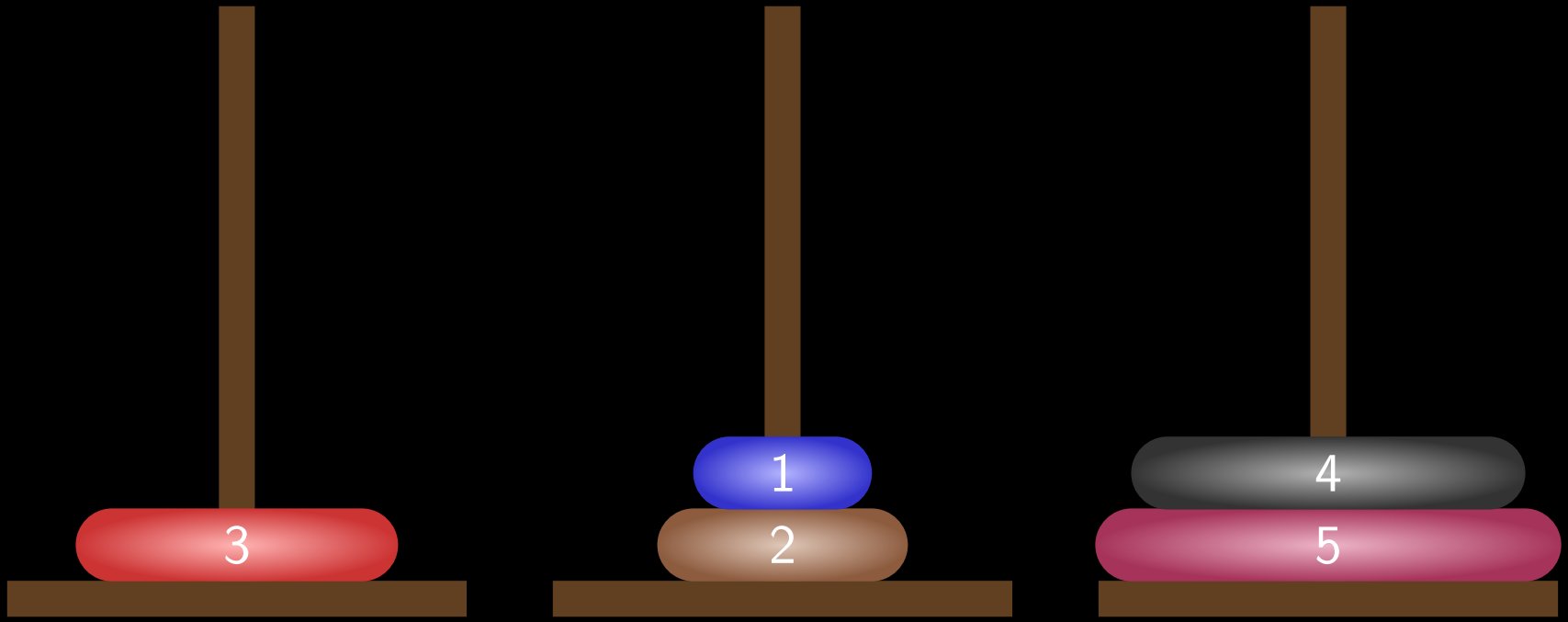
On déplace le disque de la tige 1 vers la tige 3.

Tour de Hanoi – 5 Disques



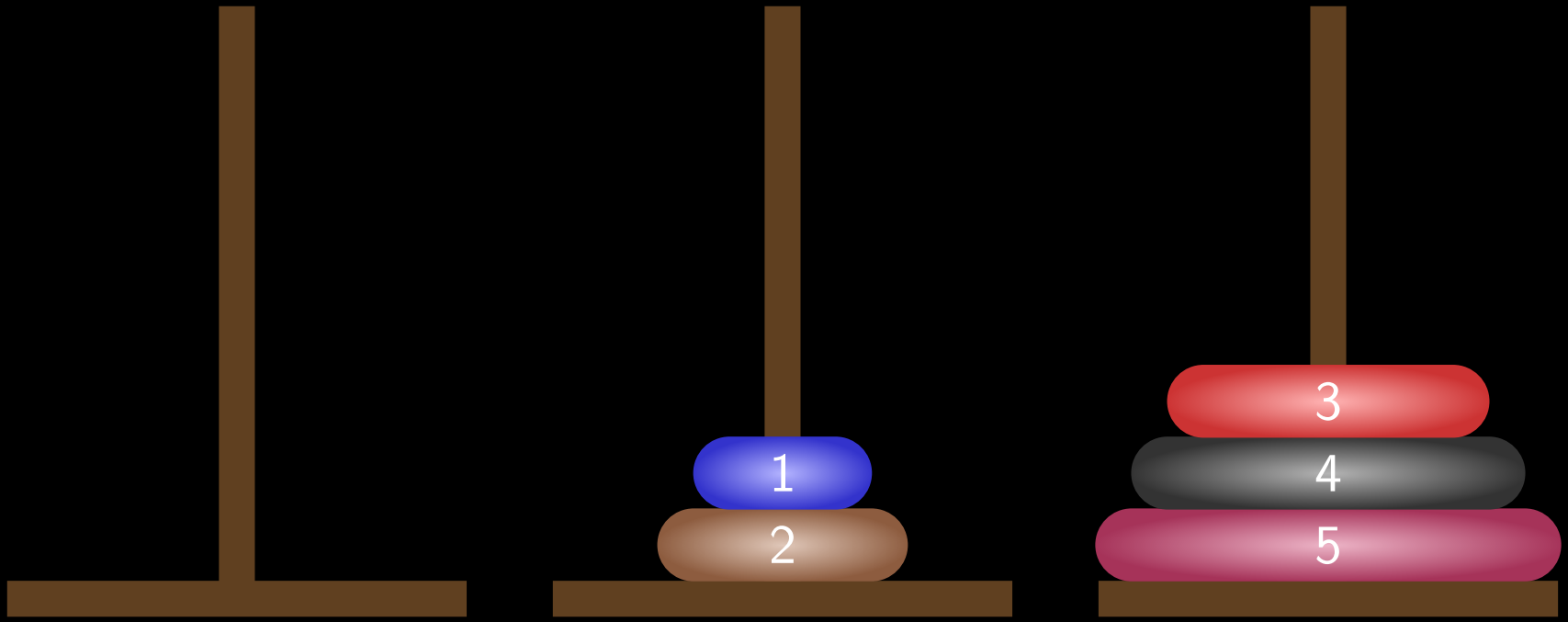
On déplace le disque de la tige 1 vers la tige 2.

Tour de Hanoi – 5 Disques



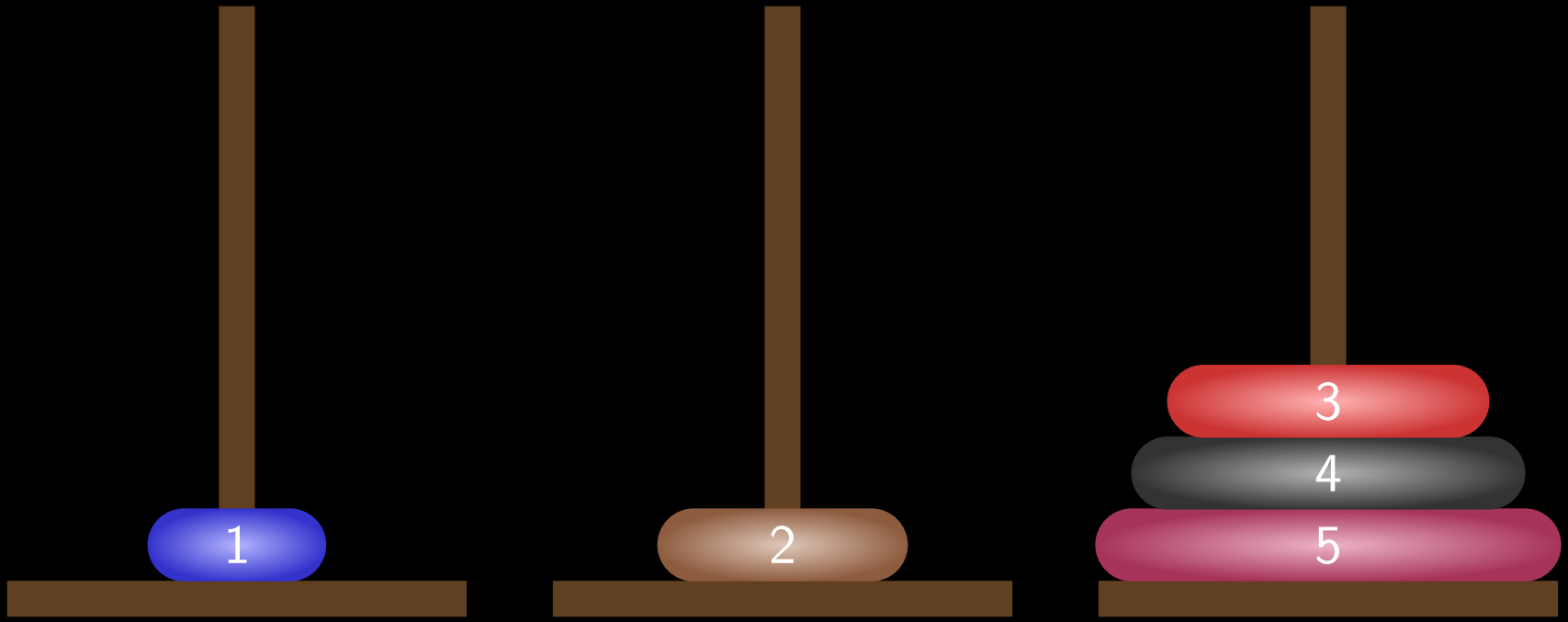
On déplace le disque de la tige 3 vers la tige 2.

Tour de Hanoi – 5 Disques



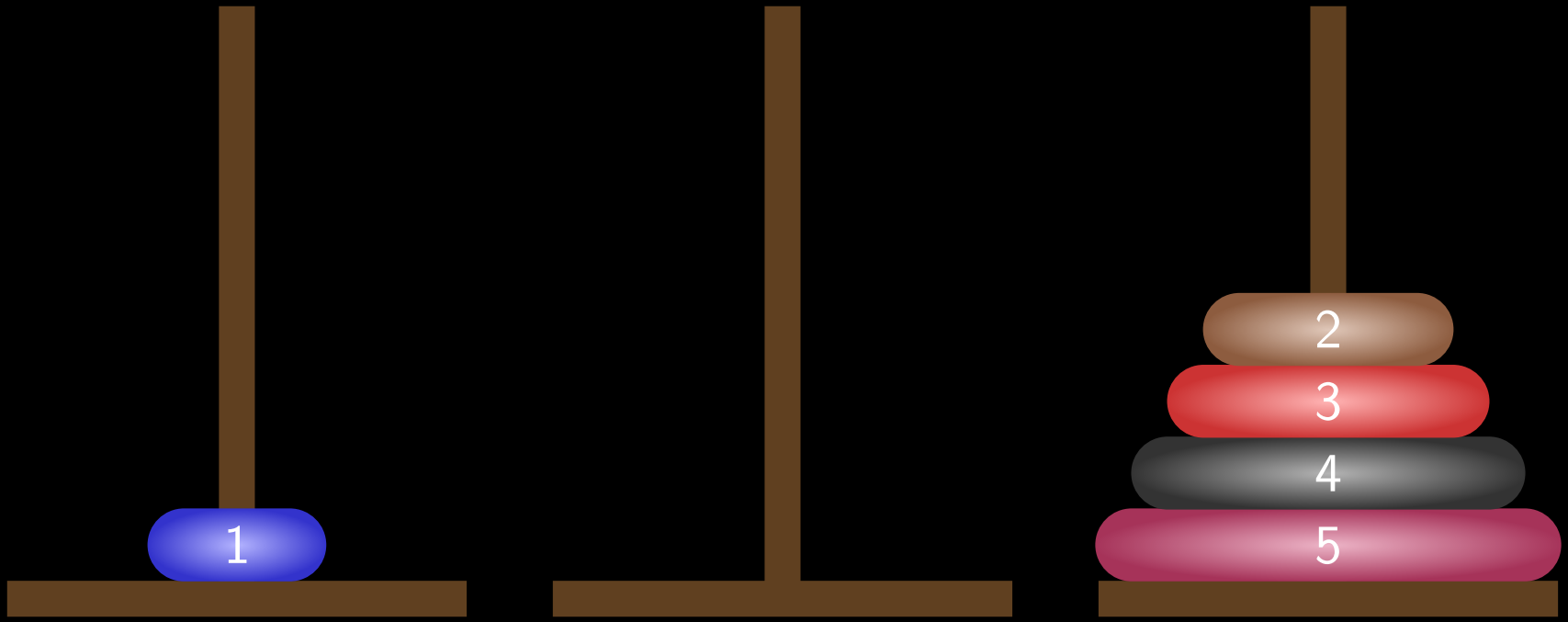
On déplace le disque de la tige 1 vers la tige 3.

Tour de Hanoi – 5 Disques



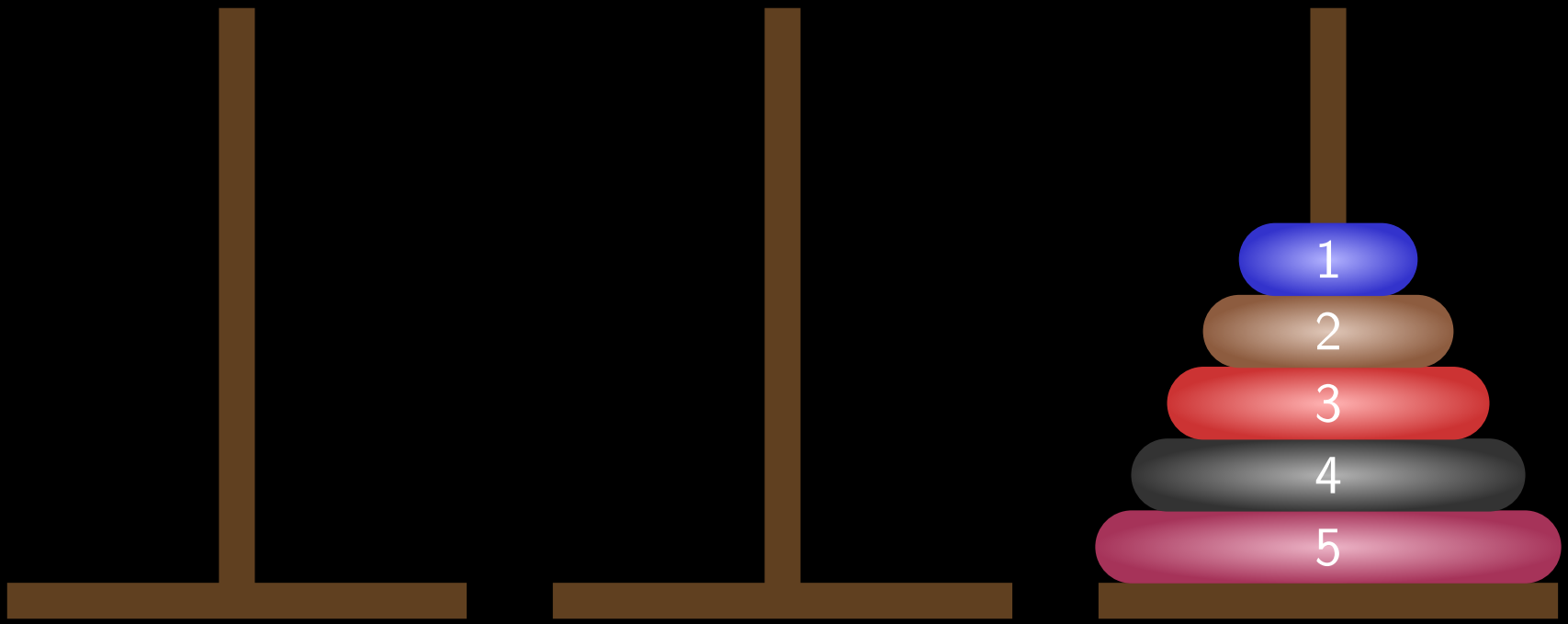
On déplace le disque de la tige 2 vers la tige 1.

Tour de Hanoi – 5 Disques



On déplace le disque de la tige 2 vers la tige 3.

Tour de Hanoi – 5 Disques



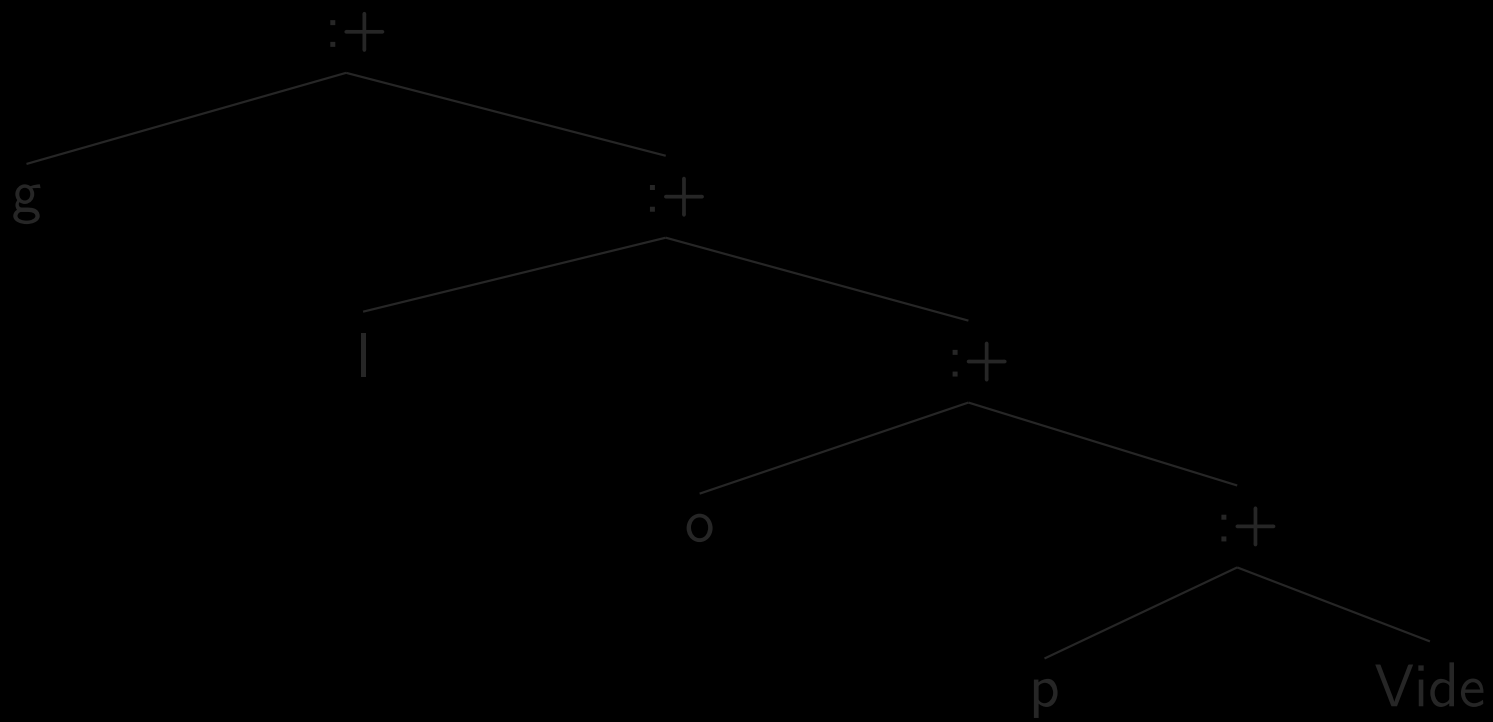
On déplace le disque de la tige 1 vers la tige 3.

Tour de Hanoi – 5 Disques

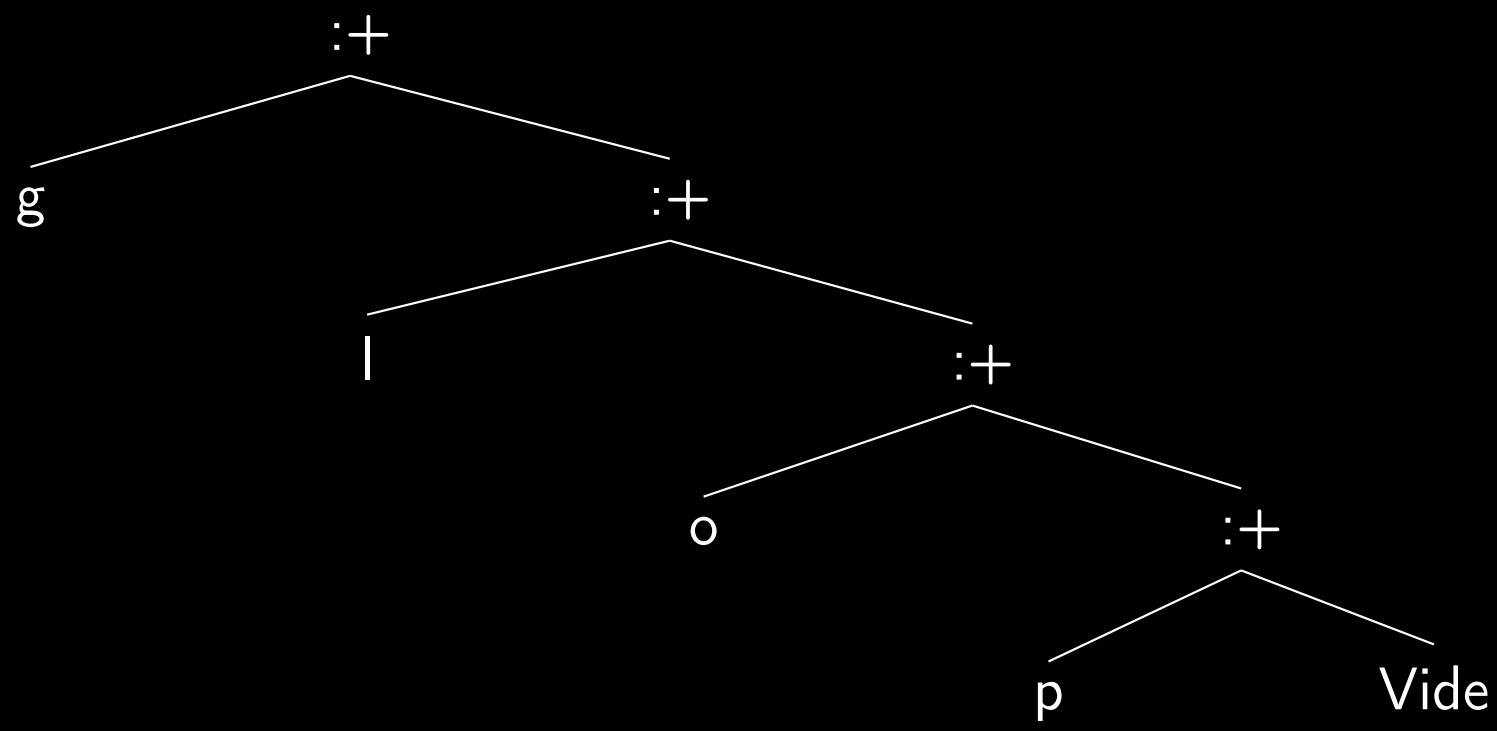
Sommaire

Qu'est-ce qu'un mot ?

« glop »



« glop »



Haskell

```haskell
infixr :+

data Mot =
  Vide
  | Char :+ Mot
```

Haskell

```haskell
infixr :+

data Mot =
  Vide
  | Char :+ Mot
  deriving (Show)
```

Haskell

```haskell
infixr :+

data Mot =
  Vide
  | Char :+ Mot
  deriving (Show, Eq)
```

Haskell

```
λ> let m = 'g' :+ 'l'  :+ 'o' :+ 'p' :+ Vide
λ> m
'g' :+ ('l' :+ ('o' :+ ('p' :+ Vide)))
```

Haskell

```
λ> :t m
m :: Mot
```

Haskell

```
λ> let m = 'g' :+ 'l'  :+ 'o' :+ 'p' :+ Vide
λ> m
'g' :+ ('l' :+ ('o' :+ ('p' :+ Vide)))
```

Haskell

```
λ> :t m
m :: Mot
```

Sélecteurs

Haskell

```haskell
tete :: Mot  → Char
-- renvoie le premier caractère d'un mot avec un filtrage par
    motif
tete    Vide  =  error "Mot vide !"
tete (t :+ q) =  t
```

Haskell

```
λ> tete m
'g'
```

Et la queue ?

Sélecteurs

Haskell

```haskell
tete :: Mot  → Char
-- renvoie le premier caractère d'un mot avec un filtrage par
    motif
tete    Vide  =  error "Mot vide !"
tete (t :+ q) =  t
```

Haskell

```
λ> tete m
'g'
```

Et la queue ?

Sélecteurs

Haskell

```
tete :: Mot  → Char
-- renvoie le premier caractère d'un mot avec un filtrage par
    motif
tete    Vide  =  error "Mot vide !"
tete (t :+ q) =  t
```

Haskell

```
λ> tete m
'g'
```

Et la queue ?

Testeurs

Haskell

```
estVide :: Mot→ Bool
estVide m = m == Vide
```

$$\neg(\text{estVide } m) \leftrightarrow \big(m = (\text{tete } m) :+ (\text{queue } m)\big)$$

Testeurs

Haskell

```
estVide :: Mot→ Bool
estVide m = m == Vide
```

$$\neg(\text{estVide } m) \leftrightarrow \big(m = (\text{tete } m) :+ (\text{queue } m)\big)$$

Construire une fonction définie sur un type récursif

On voudrait compter le nombre de « a » dans un mot.

Haskell

```haskell
nba :: Mot→ Int
-- calcule le nombre de 'a' dans un mot
```

Construire une fonction définie sur un type récursif

On voudrait compter le nombre de « a » dans un mot.

Haskell

```
nba :: Mot→ Int
-- calcule le nombre de 'a' dans un mot
```

- Si le mot est vide, alors son nombre de 'a' est 0.
- Sinon, le mot est construit par `t :+ q`.
  Si `t` = 'a', alors `nba mot` = 1 + `nba q`, sinon, `nba mot` = `nba q`.

Haskell

```haskell
nba Vide      = 0
nba (t :+ q) = (nba q) + (if t == 'a' then 1 else 0)
```

- Si le mot est vide, alors son nombre de 'a' est 0.
- Sinon, le mot est construit par `t :+ q`.
  Si `t` = 'a', alors `nba mot` = 1 + `nba q`, sinon, `nba mot` = `nba q`.

Haskell

```
nba Vide      = 0
nba (t :+ q) = (nba q) + (if t == 'a' then 1 else 0)
```

Filtrage par motif

Haskell

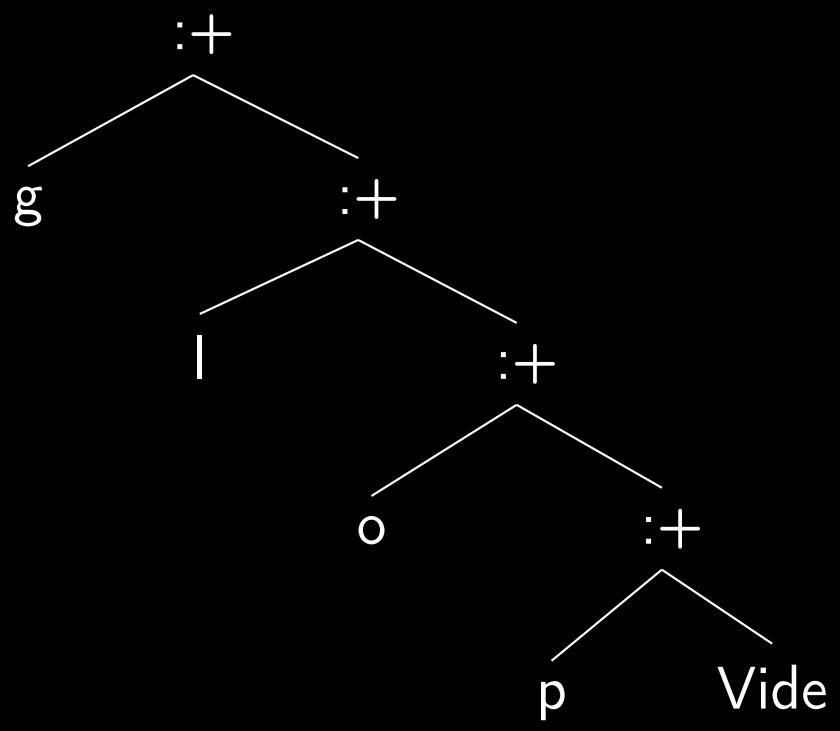
```
nba2 Vide = 0
nba2 (t :+ q)
   | t == 'a' = (nba2 q) + 1
   |otherwise =  nba2 q
```

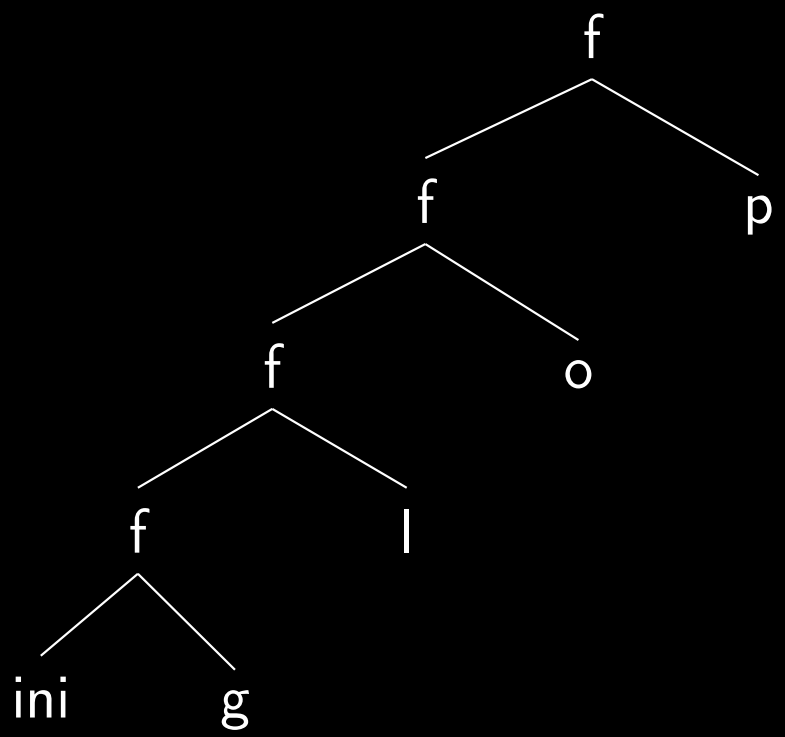
Case

Haskell

```
nba3 mot = case mot of Vide     → 0
                       (t :+ q)→ (nba3 q) + (if t == 'a' then
                          1 else 0)
```

Pliage





Pliage

Haskell

```haskell
nba4 :: Mot→ Int
-- calcule le nombre de 'a' dans un mot par pliage
nba4 mot =
  pliage (\accu t→ accu + (if t == 'a' then 1 else 0)) 0 mot
```

Pliage

Haskell

```
pliage :: (a→ Char→ a)→ a→ Mot→ a
-- adapte foldl aux mots
pliage _    accu Vide      = accu
pliage fonc accu (t :+ q) = pliage fonc (fonc accu t) q
```

Applique

Mettre en majuscule.

Haskell

```
λ> import Data.Char
λ> toUpper 'g'
'G'
```

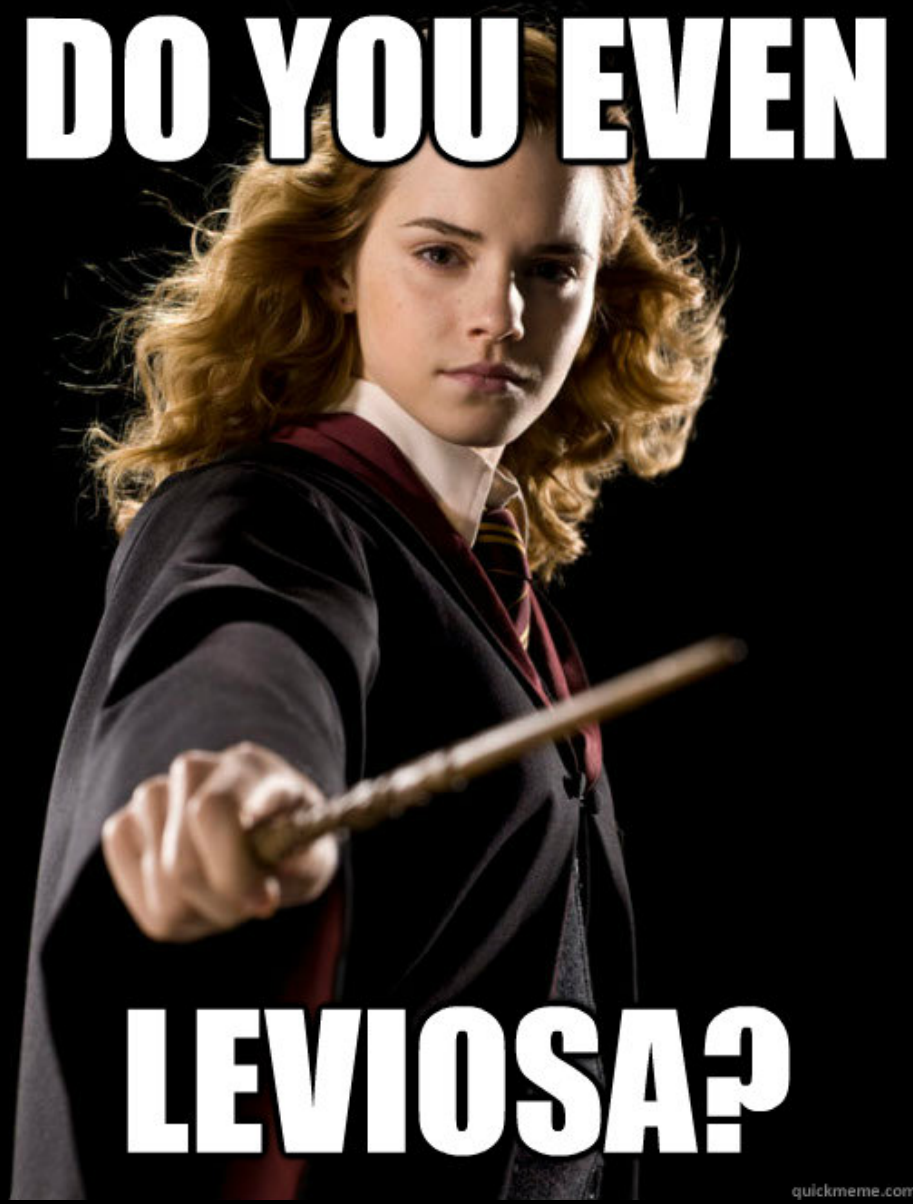
Applique

Mettre en majuscule.

Haskell

```
λ> import Data.Char
λ> toUpper 'g'
'G'
```

glop -> GLOP ?


DO YOU EVEN
LEVIOSA?
quickmeme.com

Maporum Leviosa

Haskell

```haskell
applique :: (Char→ Char) → Mot    → Mot
applique    _                 Vide    = Vide
applique    f                (t :+ q) = (f t) :+ (applique f q)
```

Haskell

```haskell
toMaj :: Mot→ Mot
toMaj    m   =  applique toUpper m
```

Haskell

```haskell
λ> let m = 'g' :+ 'l'  :+ 'o' :+ 'p' :+ Vide
λ> toMaj m
'G' :+ ('L' :+ ('O' :+ ('P' :+ Vide)))
```

Maporum Leviosa

Haskell

```haskell
applique :: (Char→ Char) → Mot    → Mot
applique    _                Vide    = Vide
applique    f               (t :+ q) = (f t) :+ (applique f q)
```

Haskell

```haskell
toMaj :: Mot→ Mot
toMaj    m   =  applique toUpper m
```

Haskell

```
λ> let m = 'g' :+ 'l'  :+ 'o' :+ 'p' :+ Vide
λ> toMaj m
'G' :+ ('L' :+ ('O' :+ ('P' :+ Vide)))
```

Maporum Leviosa

Haskell

```haskell
applique :: (Char→ Char) → Mot    → Mot
applique    _                Vide    = Vide
applique    f               (t :+ q) = (f t) :+ (applique f q)
```

Haskell

```haskell
toMaj :: Mot→ Mot
toMaj    m   =  applique toUpper m
```

Haskell

```haskell
λ> let m = 'g' :+ 'l'  :+ 'o' :+ 'p' :+ Vide
λ> toMaj m
'G' :+ ('L' :+ ('O' :+ ('P' :+ Vide)))
```

Sommaire

Haskell

```
infixr 9 .
(.) :: (b→ c)→ (a→ b)→ a→ c
f . g = \x→ f (g x)
```

DANGER

Il faut faire très attention à la signature de type des fonctions à composer !

Haskell

```
infixr 9 .
(.) :: (b→ c)→ (a→ b)→ a→ c
f . g = \x→ f (g x)
```

DANGER

Il faut faire très attention à la signature de type des fonctions à composer !

Haskell

```
λ> longMotPlusLong "Vos beaux yeux d'amour mourir me font
    Belle Marquise"
8
```

Haskell

```
λ> words "vos beaux yeux d'amour belle mourir me font Marquise"
["vos","beaux","yeux","d'amour","belle","mourir","me","font","Marq
```

Haskell

```
λ> :t words
words :: String→[String]
λ> :t length
length :: [a]→Int
λ> :t maximum
maximum :: Ord a⇒[a]→a
```

Haskell

```
λ> words "vos beaux yeux d'amour belle mourir me font Marquise"
["vos","beaux","yeux","d'amour","belle","mourir","me","font","Marq
```

Haskell

```
λ> :t words
words :: String→[String]
λ> :t length
length :: [a]→Int
λ> :t maximum
maximum :: Ord a⇒[a]→a
```

$$\texttt{String} \xrightarrow{\texttt{words}} [\texttt{String}] \xrightarrow{\texttt{map length}} [\texttt{Int}] \xrightarrow{\texttt{maximum}} \texttt{Int}$$

$$\texttt{s} \xrightarrow{\texttt{words}} \texttt{words s} \xrightarrow{\texttt{map length}} \texttt{map length (words s)} \xrightarrow{\texttt{maximum}} \texttt{maximum (map}$$

$$\texttt{String} \xrightarrow{\texttt{words . (map length . maximum)}} \texttt{Int}$$

$$\texttt{String} \xrightarrow{\texttt{words . map length . maximum}} \texttt{Int}$$

$$\texttt{String} \xrightarrow{\texttt{words}} [\texttt{String}] \xrightarrow{\texttt{map length}} [\texttt{Int}] \xrightarrow{\texttt{maximum}} \texttt{Int}$$

$$\texttt{s} \xrightarrow{\texttt{words}} \texttt{words s} \xrightarrow{\texttt{map length}} \texttt{map length (words s)} \xrightarrow{\texttt{maximum}} \texttt{maximum (map}$$

$$\texttt{String} \xrightarrow{\texttt{words . (map length . maximum)}} \texttt{Int}$$

$$\texttt{String} \xrightarrow{\texttt{words . map length . maximum}} \texttt{Int}$$

$$\texttt{String} \xrightarrow{\texttt{words}} \texttt{[String]} \xrightarrow{\texttt{map length}} \texttt{[Int]} \xrightarrow{\texttt{maximum}} \texttt{Int}$$

$$\texttt{s} \xrightarrow{\texttt{words}} \texttt{words s} \xrightarrow{\texttt{map length}} \texttt{map length (words s)} \xrightarrow{\texttt{maximum}} \texttt{maximum (map}$$

$$\texttt{String} \xrightarrow{\texttt{words . (map length . maximum)}} \texttt{Int}$$

$$\texttt{String} \xrightarrow{\texttt{words . map length . maximum}} \texttt{Int}$$

$$\texttt{String} \xrightarrow{\texttt{words}} \texttt{[String]} \xrightarrow{\texttt{map length}} \texttt{[Int]} \xrightarrow{\texttt{maximum}} \texttt{Int}$$

$$\texttt{s} \xrightarrow{\texttt{words}} \texttt{words s} \xrightarrow{\texttt{map length}} \texttt{map length (words s)} \xrightarrow{\texttt{maximum}} \texttt{maximum (map}$$

$$\texttt{String} \xrightarrow{\texttt{words . (map length . maximum)}} \texttt{Int}$$

$$\texttt{String} \xrightarrow{\texttt{words . map length . maximum}} \texttt{Int}$$

Haskell

```
λ> let longMotPlusLong = maximum . map length . words
λ> :t longMotPlusLong
longMotPlusLong :: String → Int
λ> longMotPlusLong "Vos beaux yeux d'amour mourir me font"
7
```

- `maximum . map length . words`
- `\s→ maximum (map length (words s))`
- `\s→ maximum $ map length $ words s`

Haskell

```
λ> let longMotPlusLong = maximum . map length . words
λ> :t longMotPlusLong
longMotPlusLong :: String→ Int
λ> longMotPlusLong "Vos beaux yeux d'amour mourir me font"
7
```

- `maximum . map length . words`
- `\s→ maximum (map length (words s))`
- `\s→ maximum $ map length $ words s`

Sommaire

Pour prouver qu'une propriété $\mathcal{P}_n$ dépendant uniquement d'un paramètre n est vraie pour tout $n \geqslant n_0$, il faut vérifier que :

- $\mathcal{P}_{n_0}$ est vraie (on parle parfois d'initialisation) ;
- pour tout $n \geqslant n_0$, $\mathcal{P}_n \to \mathcal{P}_{n+1}$ (on parle parfois d'hérédité).

Pour prouver qu'une propriété $\mathcal{P}_n$ dépendant uniquement d'un paramètre n est vraie pour tout $n \geqslant n_0$, il faut vérifier que :

- $\mathcal{P}_{n_0}$ est vraie (on parle parfois d'initialisation) ;
- pour tout $n \geqslant n_0$, $\mathcal{P}_n \to \mathcal{P}_{n+1}$ (on parle parfois d'hérédité).

Sommaire

Haskell

```haskell
f1 :: [Int]→[Int]
f1    []     = []
f1    (k:ks) = abs k : f1 ks

f2 :: [Int]→[Int]
f2 = map abs
```

Extrait de Data.List

```haskell
-- map f xs is the list obtained by applying f to each element
    of xs, i.e.,
-- map f [x1, x2, ..., xn] == [f x1, f x2, ..., f xn]
-- map f [x1, x2, ...] == [f x1, f x2, ...]
map :: (a→ b)→ [a]→ [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

- f1 []
- => [] définition de f1
- => map abs [] définition de map
- => f2 [] défintion de f2

- f1 []
- => [] définition de f1
- => map abs [] définition de map
- => f2 [] défintion de f2

- f1 []
- => [] définition de f1
- => map abs [] définition de map
- => f2 [] défintion de f2

- f1 []
- => [] définition de f1
- => map abs [] définition de map
- => f2 [] défintion de f2

- f1 []
- => [] définition de f1
- => map abs [] définition de map
- => f2 [] défintion de f2

- f1 (k :ks)
- => abs k : f1 ks définition de f1
- => abs k : f2 ks hypothèse de récurrence

- f1 []
- => [] définition de f1
- => map abs [] définition de map
- => f2 [] défintion de f2

- f1 (k :ks)
- => abs k : f1 ks définition de f1
- => abs k : f2 ks hypothèse de récurrence
- => abs k : map abs ks définition de f2
- => map abs (k :ks) définition de map
- => f2 (k :ks) définition de f2

- f1 []
- => [] définition de f1
- => map abs [] définition de map
- => f2 [] défintion de f2

- f1 (k :ks)
- => abs k : f1 ks définition de f1
- => abs k : f2 ks hypothèse de récurrence
- => abs k : map abs ks définition de f2
- => map abs (k :ks) définition de map
- => f2 (k :ks) définition de f2

- f1 []
- => [] définition de f1
- => map abs [] définition de map
- => f2 [] défintion de f2

- f1 (k :ks)
- => abs k : f1 ks définition de f1
- => abs k : f2 ks hypothèse de récurrence
- => abs k : map abs ks définition de f2
- => map abs (k :ks) définition de map
- => f2 (k :ks) définition de f2

- f1 []
- => [] définition de f1
- => map abs [] définition de map
- => f2 [] défintion de f2

- f1 (k :ks)
- => abs k : f1 ks définition de f1
- => abs k : f2 ks hypothèse de récurrence
- => abs k : map abs ks définition de f2
- => map abs (k :ks) définition de map
- => f2 (k :ks) définition de f2

- f1 []
- => [] définition de f1
- => map abs [] définition de map
- => f2 [] défintion de f2

- f1 (k :ks)
- => abs k : f1 ks définition de f1
- => abs k : f2 ks hypothèse de récurrence
- => abs k : map abs ks définition de f2
- => map abs (k :ks) définition de map
- => f2 (k :ks) définition de f2

- f1 []
- => [] définition de f1
- => map abs [] définition de map
- => f2 [] défintion de f2

- f1 (k :ks)
- => abs k : f1 ks définition de f1
- => abs k : f2 ks hypothèse de récurrence
- => abs k : map abs ks définition de f2
- => map abs (k :ks) définition de map
- => f2 (k :ks) définition de f2