A Formalisation of the Generalised Towers of Hanoi

Laurent Théry

To cite this version:

Laurent Théry. A Formalisation of the Generalised Towers of Hanoi. 2017. hal-01446070v2

HAL Id: hal-01446070
https://hal.science/hal-01446070v2

Preprint submitted on 28 Jan 2017

A Formalisation of the Generalised Towers of Hanoi

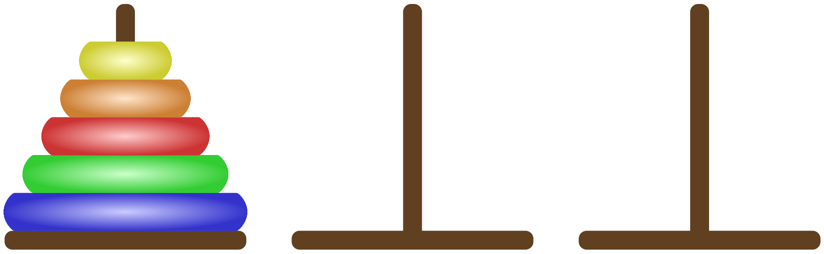
Laurent Théry
`Laurent.Thery@sophia.inria.fr`

Abstract

This notes explains how the optimal algorithm for the generalised towers of Hanoi has been formalised in the Coq proof assistant using the SSReflect extension.

1 Introduction

The famous problem of the towers of Hanoi was proposed by the french mathematician Édouard Lucas. It is composed of three pegs and some disks of different size. Here is a drawing of the initial configuration for 5 disks[1]:

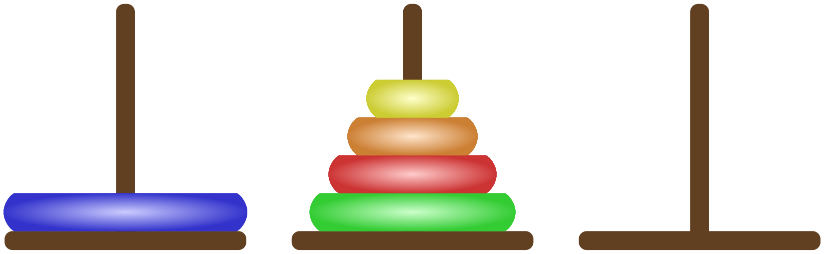


Initially, all the disks are pilled-up in decreasing order of size on the first peg. The goal is to move them all to another peg. There are two rules. First, only one disk can be moved at a time. Second, a larger disk can never be put on top of a smaller one.

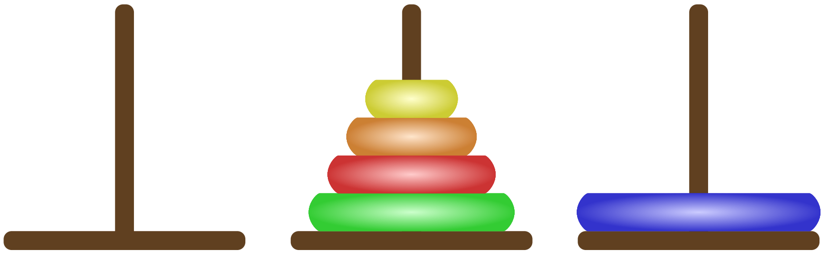
---

[1] We use macros designed by Martin Hofmann and Berteun Damman for our drawings.

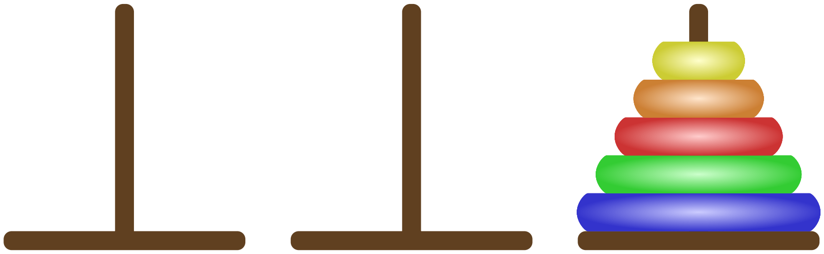
The towers of Hanoi one of the classical example that illustrates all the power of recursion. If we know how to solve the problem for n disks, then the problem for $n+1$ disks can be solved in 3 steps. Let us suppose we want to transfer all the disks to the last peg. The first step uses recursion and moves the n-top disks to the intermediate peg.



The second step moves the largest disk to its destination

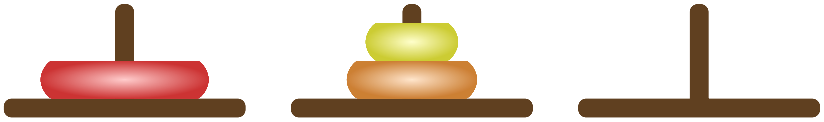


The last step uses recursion and moves the n disks on the intermediate peg to their destination.

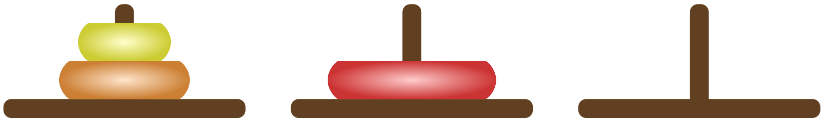


This simple recursive algorithm is also optiomal: it produces the minimal numbers of moves. In particular, if we look at each recursion depth, the key idea is that largest disk always moves once from its current peg to its destination.

The generalised version of the towers of Hanoi considers an arbitrary initial configuration and an arbitrary final configuration. These two configurations must be valid : there is no larger disk on top of a smaller disk. The problem is to find an algorithm that generates the minimal nunber of moves that connects the two configurations. Here, the naive recursive algorithm is still applicable to solve the problem but does not lead to an optimal algorithm. This can be illustrated by 3 disks when trying to go from the initial configuration:



to the final position



If the naive recursive approach, that tries to move the largest disk only once, leads to a 7-move long solution as depicted in Figure 1 at page 12. The optimal solution requires instead to move the largest disk twice and is 5-move long as depicted in Figure 2 at page 2. In the following, we explain how the generalised towers of Hanoi has been formalised in the Coq proof assistant and how an algorithm that solves this problem has been proved correct.

2 The formalisation

In this section, we present the different elements our formalisation, starting with pegs, disks, configurations and moves, then we describe the naive recursive algorithm and finally the optimal one.

2.1 Pegs

The set of natural numbers strictly smaller than three $\mathtt{I}_3$ is used to represent the three pegs.

```
Definition peg := I_3.
```

An operation that is frequently used in the algorithm is, having two arbitrary peg p_1 and p_2 to get the third one, This is done by the function *opeg* using some arithmetic:

```
Definition opeg p1 p2 : peg := inord (3 - (p1 + p2)).
```

where `inord` is the function that injects a natural number into the type $\mathtt{I}_n$.

2.2 Disks

A parameter n is used. A disk is then an element of $\mathtt{I}_n$.

```
Definition disk := I_n.
```

The comparison of the respective size of two disks is simply performed by comparing their natural number.

2.3 Configurations

Disks are ordered from the largest to the smallest on a peg. This means that a configuration just needs to record which disk is on which peg. It is then defined as a finitite function from disks to pegs.

```
Definition configuration := {ffun disk -> peg}.
```

Note that in this encoding, we do not have invalid configurations.

A configuration is called *perfect* if all its disks are on a single peg. It is the case for the initial and final configurations in the standard towers of Hanoi. The constant function that always returns p is then the perfect configuration where all the disks are on the peg p.

$$\texttt{Definition}\ \mathit{perfect}\ p := \texttt{[ffun}\ d \Rightarrow p\texttt{].}$$

We also need two functions to build configurations. The first one builds a new configuration from a configuration c by putting all the disks of size strictly smaller than m on the peg p: this new configuration is perfect at depth m.

$$\texttt{Definition}\ \mathit{mk\_perfect}\ m\ p\ c := \texttt{[ffun}\ d \Rightarrow \texttt{if}\ d < m\ \texttt{then}\ p\ \texttt{else}\ c\ d\texttt{].}$$

The second function performs a single change on the configuration c : the disk d is moved to the peg p.

$$\texttt{Definition}\ \mathit{setd}\ d\ p\ c_2 := \texttt{[ffun}\ d_1 \Rightarrow \texttt{if}\ d_1 = d\ \texttt{then}\ p\ \texttt{else}\ c\ d_1\texttt{].}$$

2.4 Moves

A move is defined as a relation between configuration. A parameter m is introduced. It represents a bound on the size of the disk that has been moved. Its purpose is to let us perform proof by induction: the induction is then performed on the depth of the moves m rather than on the number of disks n.

$$
\begin{array}{l}
\texttt{Definition}\ \mathit{move}_m : \texttt{rel}\ \mathit{configuration} := \\
\quad \texttt{[rel}\ c_1\ c_2\ \texttt{|}\ \texttt{[}\exists d_1 : \texttt{I}_n, \\
\qquad \texttt{[\&\&} \\
\qquad\quad d_1 < m, \\
\qquad\quad c_1\ d_1 \neq c_2\ d_1, \\
\qquad\quad \texttt{[}\forall d_2,\ d_1 \neq d_2 \Rightarrow c_1\ d_2 = c_2\ d_2\ \texttt{],} \\
\qquad\quad \texttt{[}\forall d_2,\ c_1\ d_1 = c_1\ d_2 \Rightarrow d_1 \leq d_2\ \texttt{]}\ \texttt{\&} \\
\qquad\quad \texttt{[}\forall d_2,\ c_2\ d_1 = c_2\ d_2 \Rightarrow d_1 \leq d_2\texttt{]]]].}
\end{array}
$$

The definition simply states that there is a disk d_1 htat fulfills five conditions. Its size is smaller than m. It is has moved. It is the unique disk that has moved. No disk is in on top of d_1 in c_1. No disk is on top of d_1 in c_2.
Trivial facts first need to be derived. For example, the *move* relation is cumulative and symmetrical.

$$\begin{array}{l}
\texttt{Lemma}\ \mathit{moveW}\ m_1\ m_2 : m_1 \leq m_2 \rightarrow \texttt{subrel}\ \mathit{move}_{m_1}\ \mathit{move}_{m_2}. \\
\texttt{Lemma}\ \mathit{move\_sym}\ m\ c_1\ c_2 : \mathit{move}_m\ c_1\ c_2 = \mathit{move}_m\ c_2\ c_1.
\end{array}$$

A first interesting is then that before and after a move, all the disks smaller than the disk that has moved are all pilled up on the same peg:

$$\begin{array}{l}
\texttt{Lemma}\ \mathit{move\_perfectl}\ m\ d\ c_1\ c_2 : \\
\quad \mathit{move}_m\ c_1\ c_2 \rightarrow c_1\ d \neq c_2\ d \rightarrow c_1 = \mathit{mk\_perfect}\ d\ (\mathit{opeg}\ (c_1\ d)\ (c_2\ d))\ c_1. \\
\texttt{Lemma}\ \mathit{move\_perfectr}\ m\ d\ c_1\ c_2 : \\
\quad \mathit{move}_m\ c_1\ c_2 \rightarrow c_1\ d \neq c_2\ d \rightarrow c_2 = \mathit{mk\_perfect}\ d\ (\mathit{opeg}\ (c_1\ d)\ (c_2\ d))\ c_2.
\end{array}$$

An important corollary is that if a disk d moves twice on different pegs then the two configurations after the first move and before the second one are perfect configurations at depth d.

$$\begin{array}{l}
\texttt{Lemma}\ \mathit{move\_twice}\ d\ c_1\ c_2\ c_3\ c_4 : \\
\qquad c_1 \neq c_4 \rightarrow c_1\ d \neq c_2\ d \rightarrow c_3\ d \neq c_4\ d \rightarrow \\
\qquad \mathit{move}_{d.+1}\ c_1\ c_2 \rightarrow \texttt{connect}\ \mathit{move}_d\ c_2\ c_3 \rightarrow \mathit{move}_{d.+1}\ c_3\ c_4 \rightarrow \\
\qquad c_2 = \mathit{mk\_perfect}\ d\ (\mathit{opeg}\ (c_1\ d)\ (c_2\ d))\ c_2 \wedge \\
\qquad c_3 = \mathit{mk\_perfect}\ d\ (c_1\ d)\ c_2.
\end{array}$$

This is a key lemma that is used to get a direct lower bound ($2^d - 1$) on the number of moves that are needed for connecting c_2 are c_3. We will explain this later.

In order to be able to decompose paths under the $\mathit{move}_{m.+1}$ relation, two inversion lemmas are needed. The first one checks if the disks m moves. If it is the case, it singles out its first move.

$$\begin{array}{l}
\texttt{Inductive}\ \mathit{pathS\_spec}\ (d : \mathtt{I}_n)\ c : \texttt{seq}\ \mathit{configuration} \rightarrow \texttt{bool} \rightarrow \texttt{Type} := \\
\qquad \mathit{pathS\_specW} : \forall cs,\ \texttt{path}\ \mathit{move}_d\ c\ cs \rightarrow \mathit{pathS\_spec}\ d\ c\ cs\ \texttt{true}\ \texttt{|} \\
\qquad \mathit{pathS\_spec\_move} : \forall c_1\ cs_1\ cs_2, \\
\qquad\qquad \texttt{path}\ \mathit{move}_d\ c\ cs_1 \rightarrow c\ d \neq c_1\ d \rightarrow \mathit{move}_{d.+1}\ (\texttt{last}\ c\ cs_1)\ c_1 \rightarrow \\
\qquad\qquad \texttt{path}\ \mathit{move}_{d.+1}\ c_1\ cs_2 \rightarrow \mathit{pathS\_spec}\ d\ c\ (cs_1\ \texttt{++}\ c_1\ \texttt{::}\ cs_2)\ \texttt{true}\ \texttt{|} \\
\qquad \mathit{pathS\_spec\_false} : \forall cs,\ \mathit{pathS\_spec}\ d\ c\ cs\ \texttt{false}. \\
\\
\texttt{Lemma}\ \mathit{pathSP}\ d\ c\ cs : \mathit{pathS\_spec}\ d\ c\ cs\ (\texttt{path}\ \mathit{move}_{d.+1}\ c\ cs).
\end{array}$$

The decomposition is presented as an inductive predicate in order to get the decomposition by the direct application `case:` *pathSP* tactic on a path. The second inversion lemma considers a path at depth $d+1$ for which the disk d may have moved but at the end it remains on the same peg. It builds the ”restricted” path for which the disk does not move.

```
Lemma pathS_restrictE d c cs :
path move_d.+1 c cs → last c cs d = c d →
{cs1 |
   [∧
     path move_d c cs1,
     last c cs1 = last c cs &
     size cs1 ≤ size cs ?= iff cs1 = cs ]}.
```

The number of moves of the restricted path gets strictly smaller only if there is a move of the disk d in the path cs.

2.5 Naive algorithm

Our definition of the naive algorithm works at depth m, starts with a configuration c and tries to move all the disks of size less than m to the peg p. The strategy works as follows. If the disk m is already on the peg p, there is nothing to do at depth m: the algorithm is recursively called at depth $m-1$. Otherwise, the disk m needs to be moved. The algorithm is first called at depth $m-1$ to move all the disks of size smaller than $m-1$ to the intermediate peg p_1. Then, the disk of size m is moved to the peg p and finally the algorithm is called a second time to move the disk of size smaller than $m-1$ to the peg p. Formally, this gives[2]:

```
Fixpoint rpeg_path_rec m c p :=
   if m is m1.+1 then
     if c m = p then rpeg_path_rec m1 c p else
       let p1 := opeg (c m) p in
       let c1 := setd m p (mk_perfect m1 p1 c) in
       rpeg_path_rec m1 c p1 ++ c1 :: rpeg_path_rec m1 c1 p
   else [::]

Definition rpeg_path c p := rpeg_path_rec n c p.
```

---

[2] This is a simplified version. The real code is less readable because m is an ordinal.

Note that c_1 is configuration after the disk m has been moved to the peg p: all the disks smaller than m are on the intermediate peg p_1.

The first basic property that needs to be proved is that this algorithm is correct: what is build is a path that goes from the configuration c to the perfect configuration on peg p

```
Lemma rpeg_path_correct c p (cs := rpeg_path c p) :
    path (move n) c cs ∧ last c cs = perfect p.
```

This directly gives the fact that any configuration is connected to any perfect configuration

```
Lemma move_connect_rpeg c p : connect (move n) c (perfect p).
```

Since the relation is symmetric, this gives that any two configurations are connected.

```
Lemma move_connect $c_1$ $c_2$ : connect (move n) $c_1$ $c_2$.
```

There is always a solution to the generalized tower of Hanoi.

If we are only interested by the size of the solution, it is possible to give an algorithm that computes the size of the connection given by the naive algorithm.

```
Fixpoint size_rpeg_path_rec m c p :=
    if m is $m_1$.+1 then
      if c m = p then size_rpeg_path_rec $m_1$ c p else
        let $p_1$ := opeg (c m) p in
        size_rpeg_path_rec $m_1$ c $p_1$ + $2^{m_1}$
    else 0.
```

Note that in this version, there is only one recursive call. The justification for this comes from the fact that this algorithm returns $2^m - 1$ when called on a perfect configuration c that is different from p.

$\texttt{Lemma}\ \mathit{size\_rpeg\_path\_rec\_2p}\ m\ p_1\ p_2\ c\ (c_1 := \mathit{mk\_perfect}\ m\ p_1\ c) :$
$\quad \mathit{size\_rpeg\_path\_rec}\ m\ c_1\ p_2 = (2^m - 1)(p_1 \neq p_2).$

This gives us directly that it computes the actual size of the naive algorithm.

$\texttt{Lemma}\ \mathit{size\_rpeg\_path\_rec\_pr}\ m\ c\ p :$
$\quad \texttt{size}\ (\mathit{rpeg\_path\_rec}\ m\ c\ p) = \mathit{size\_rpeg\_path\_rec}\ m\ c\ p.$

As a matter of fact $2^m - 1$ is the maximum a naive solution can get

$\texttt{Lemma}\ \mathit{size\_rpeg\_path\_rec\_pr}\ m\ c\ p :\ \mathit{size\_rpeg\_path\_rec}\ m\ c\ p \leq 2^m - 1.$

With these results, it is possible to prove the optimality of the naive algorithm for the special case where the initial configuration is perfect.

$\texttt{Lemma}\ \mathit{rpeg\_path\_rec\_min}\ m\ c_1\ p\ \mathit{cs}\ (c_2 := \mathit{mk\_perfect}\ m\ p\ c_1) :$
$\quad \mathit{path}\ (\mathit{move}\ m)\ c_1\ \mathit{cs} \to \texttt{last}\ c_1\ \mathit{cs} = c_2 \to$
$\quad \mathit{size\_rpeg\_path\_rec}\ m\ c_1\ p \leq \texttt{size}\ \mathit{cs}\ \texttt{?= iff}\ (\mathit{cs} = \mathit{rpeg\_path\_rec}\ m\ c_1\ p).$

Note that we also prove that the optimal solution is unique. The proof works by a double induction: one induction on the depth m and another strong induction on the size of cs. So, when proving the step case at depth m, the property is known to hold for all paths at depth $m-1$ and for paths at depth m which size is strictly smaller than cs. We then simply do a discussion on the number of moves the disk m does in the path cs:

- If it does not move, the inductive hypothesis for $m-1$ gives directly the result.
- If it moves once, the path cs mimics the strategy of the naive algorithm at depth m. So, combining the two applications of the inductive hypothesis for $m-1$ (one before and one after the move) gives the result.
- If it moves more than once, there are two possibilities. Either the disk visits a peg more than once (this is always the case if the disk moves

more than two times). In this case, the lemma *pathS restrictE* gives us a stricly smaller path on which we can apply the inductive hypothesis on the size. Either the disk has moved twice on different pegs. The lemma *move twice* tells us that if we consider the path the first move and before the second, it connects two perfect configurations at depth $m - 1$. So the inductive hypothesis for $m-1$ and the lemma *size_rpeg_path_rec_2p* tell us that its size is greater than $2^{m-1} - 1$. The same holds for the path after the second move and the final configuration. Altogether, this gives a size for cs (not considering the path before the first move of the disk m) that is larger than $1 + (2^{m-1} - 1) + 1 + (2^{m-1} - 1) = 2^m$. This is strictly more than the bound $2^m - 1$ for the naive algorithm given by the lemma *size rpeg path rec pr*.

This ends the proof.

2.6 Optimal algorithm

In order to define the optimal algorithm, we first define the symmetric of the naive algorithm that goes from a perfect configuration to any configuration by simply reversing the path.

```
Definition lpeg_path_rec m p c := rev (belast c (rpeg_path_rec m c p)).
Definition lpeg_path p c := lpeg_path_rec n p c.
```

This algorithm is clearly correct and optimal.

The optimal algorithm is defined recursively in order to find the first disk that has to be moved. When this disk is found, it simply chooses the best solution between moving it directly to where it has to go (going from c_1 to c_3 then c_2) and moving it twice (going from c_1 to c_3 then c_4 and fimally c_2) using the intermediate peg p. The computation of these two solutions can use the naive algorithm since one of the two configurations that are connected is perfect, so we know it is optimal.

```
Fixpoint hanoi_path_rec $m$ $c_1$ $c_2$ :=
  if $m$ is $m_1$.+1 then
    if $c_1$ $m$ = $c_2$ $m$ then hanoi_path_rec $m_1$ $c_1$ $c_2$ else
    let $p$ := opeg ($c_1$ $m$) ($c_2$ $m$) in
    let $n_1$ := size_rpeg_path_rec $m_1$ $c_1$ $p$ + size_rpeg_path_rec $m_1$ $c_2$ $p$ in
    let $n_2$ := size_rpeg_path_rec $m_1$ $c_1$ ($c_2$ $m$) +
                 $2^{m_1}$ + size_rpeg_path_rec $m_1$ $c_2$ ($c_1$ $m$) in
    if $n_1 \le n_2$ then
      let $c_3$ := setd $m$ ($c_2$ $m$) (mk_perfect $m_1$ $p$ $c_1$) in
      rpeg_path_rec $m_1$ $c_1$ $p$ ++ $c_3$ :: lpeg_path_rec $m_1$ $p$ $c_2$
    else
      let $c_3$ := setd $m$ $p$ (mk_perfect $m_1$ ($c_2$ $m$) $c_1$) in
      let $c_4$ := setd $m$ ($c_2$ $m$) (mk_perfect $m_1$ ($c_1$ $m$) $c_1$) in
      rpeg_path_rec $m_1$ $c_1$ ($c_2$ $m$) ++ $c_3$ :: rpeg_path_rec $m_1$ $c_3$ ($c_1$ $m$)
                                  ++ $c_4$ :: lpeg_path_rec $m_1$ ($c_1$ $m$) $c_2$
  else [::].
Definition hanoi_path $c_1$ $c_2$ := hanoi_path_rec $n$ $c_1$ $c_2$.
```

It is then easy to derive that this algorithm is correct. The proof for optimality is similar to the one for the naive algorithm : we simply show that the largest disk cannot move three times in the optimal solution.

```
Lemma hanoi_path_correct $c_1$ $c_2$ ($cs$ := hanoi_path $c_1$ $c_2$) :
  path (move $n$) $c_1$ $cs$ $\wedge$ last $c_1$ $cs$ = $c_2$.
Lemma hanoi_rec_min $m$ $c_1$ $c_2$ $cs$ :
  path (move $m$) $c_1$ $cs$ $\rightarrow$ last $c_1$ $cs$ = $c_2$ $\rightarrow$
  size (hanoi_path_rec $m$ $c_1$ $c_2$) $\le$ size $cs$.
```

3 Conclusion

We have presented a formalisation of the generalised towers of Hanoi. The formalisation clearly benefits from the SSReflect library. In particular, finite function have been a convenient tool to encode configuration. Most of the proofs are elementary. Without surprise, the difficult part is to get the optimality results. We had to device two dedicated inversion principles in order to mechanise the case distinctions that were needed. The complete proof is available at `http://www-sop.inria.fr/marelle/Laurent.Thery/Hanoi`.

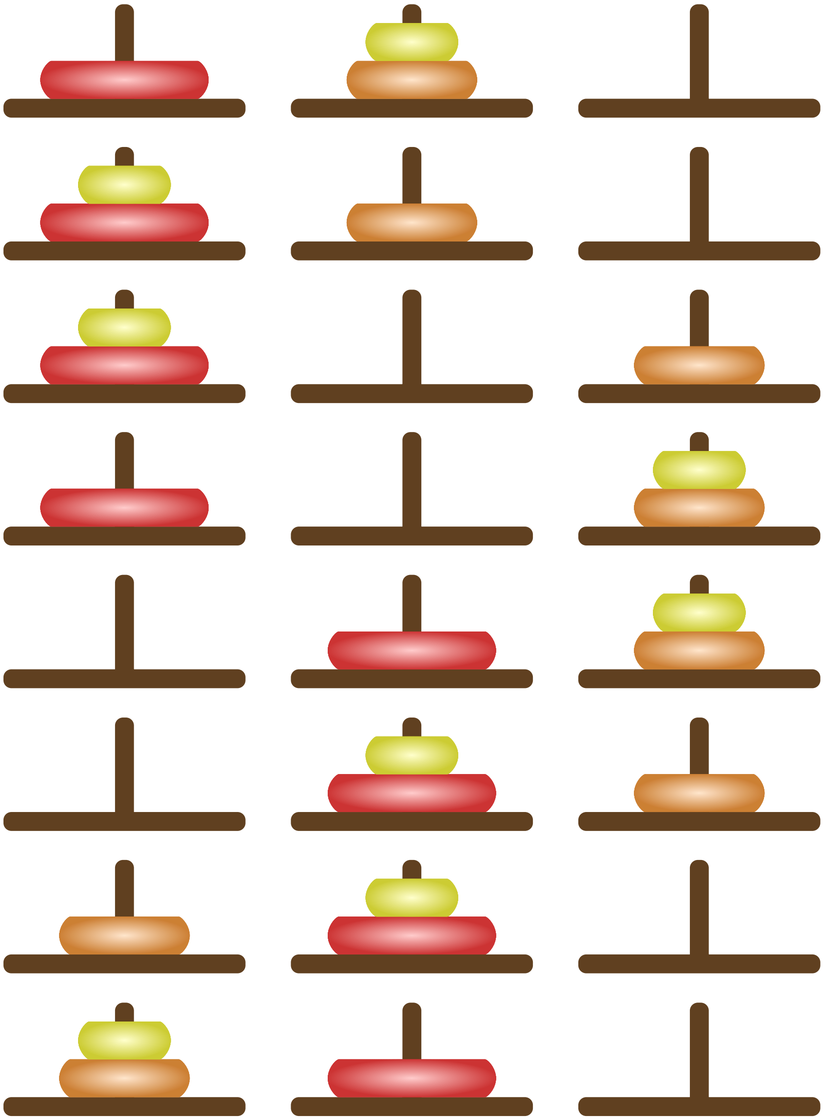


Figure 1: A non-optimal solution for the generalised towers of Hanoi

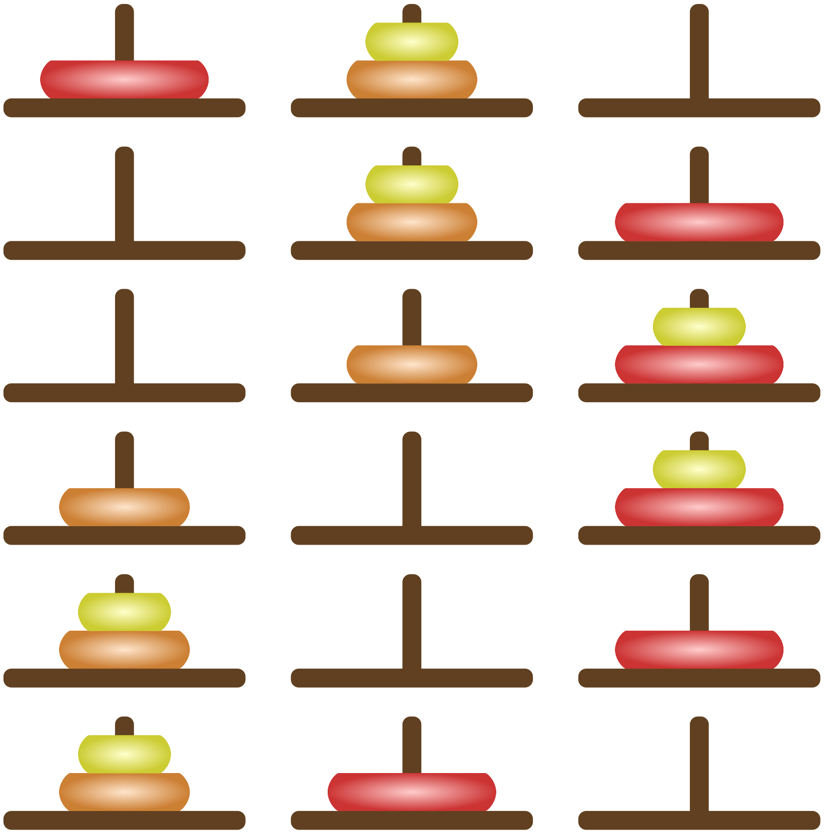


Figure 2: An optimal solution for the generalised towers of Hanoi