CSCI 3230 Data Structures

Recursion

Weitian Tong, Ph.D.

Department of Computer Science
Georgia Southern University
Website: www.weitiantong.com
Email:wtong@georgiasouthern.edu

Table of contents

Divide and Conquer

Divide and Conquer

When faced with a difficult problem, a classic technique is to break it down into smaller parts that can be solved more easily.

- How about keep partitioning recursively?
- How to guarantee the correctness?

Divide and Conquer

When faced with a difficult problem, a classic technique is to break it down into smaller parts that can be solved more easily.

- How about keep partitioning recursively?
- How to guarantee the correctness?

Recursion uses **induction** to do this.

Divide and Conquer

When faced with a difficult problem, a classic technique is to break it down into smaller parts that can be solved more easily.

- How about keep partitioning recursively?
- How to guarantee the correctness?

Recursion uses **induction** to do this.

- **Linear recursion**: one recursive call for each non-base case.
- **Binary recursion**: two recursive calls for each non-base case.

Induction (Optional)

Induction is a mathematical method for proving that a statement is true for a (possibly infinite) sequence of objects.

There are two things that must be proved:

- **The Base Case**: The statement is true for the first object
- **The Inductive Step**: If the statement is true for a given object, it is also true for the next object.

If these two statements hold, then the statement holds for all objects.

Why?

Linear recursion

Example 1: Factorials

The factorial function:

$$n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$$

Recursive definition:

$$f(n) = \begin{cases} 1, & n = 0; \\ n \cdot f(n-1), & n \geq 1. \end{cases}$$

As a Java method:

```
1  // recursive factorial function
2  public static int recursiveFactorial(int n) {
3      if (n == 0) return  1; // base case
4      else return n * recursiveFactorial(n - 1); // recursive case
5  }
```

Why is it linear recursion? What is the running time?

Example 1: Factorials

The factorial function:

$$n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$$

Recursive definition:

$$f(n) = \begin{cases} 1, & n = 0; \\ n \cdot f(n-1), & n \geq 1. \end{cases}$$

As a Java method:

```
1  // recursive factorial function
2  public static int recursiveFactorial(int n) {
3      if (n == 0) return  1; // base case
4      else return n * recursiveFactorial(n - 1); // recursive case
5  }
```

Why is it linear recursion? What is the running time? $O(n)$

Example 2: Powers

The power function:

$$p(x, n) = x^n$$

Example 2: Powers

The power function:

$$p(x, n) = x^n$$

Recursive definition:

$$p(x, n) = \begin{cases} 1, & n = 0; \\ x \cdot p(x, n-1), & n \geq 1. \end{cases}$$

As a Java method:

```
1  // recursive power function
2  public static double recursivePower(double x, int n) {
3      if (n == 0) return  1; // base case
4      else return x * recursivePower(x, n - 1); // recursive case
5  }
```

What is the running time? Can we do better?

Example 2: Powers

The power function:

$$p(x, n) = x^n$$

Recursive definition:

$$p(x, n) = \begin{cases} 1, & n = 0; \\ x \cdot p(x, n-1), & n \geq 1. \end{cases}$$

As a Java method:

```
1  // recursive power function
2  public static double recursivePower(double x, int n) {
3      if (n == 0) return  1; // base case
4      else return x * recursivePower(x, n - 1); // recursive case
5  }
```

What is the running time? Can we do better? $O(n)$

Example 2: Powers (Continue...)

Anather recursive definition:

$$p(x,n) = \begin{cases} 1, & n = 0; \\ x \cdot p(x,(n-1)/2)^2, & n \geq 1 \text{ is odd}; \\ p(x,n/2)^2, & n \geq 1 \text{ is even}. \end{cases}$$

As a Java method:

```
1  // faster recursive power function
2  public static double fasterRecursivePower(double x, int n) {
3      if (n == 0) return  1; // base case
4      elseif (n % 2 = 1) return ...; // recursive case 1
5      else return ...; // recursive case 2
6  }
```

What is the running time?

Example 2: Powers (Continue...)

Anather recursive definition:

$$p(x,n) = \begin{cases} 1, & n = 0; \\ x \cdot p(x,(n-1)/2)^2, & n \geq 1 \text{ is odd}; \\ p(x,n/2)^2, & n \geq 1 \text{ is even}. \end{cases}$$

As a Java method:

```
1  // faster recursive power function
2  public static double fasterRecursivePower(double x, int n) {
3      if (n == 0) return  1; // base case
4      elseif (n % 2 = 1) return ...; // recursive case 1
5      else return ...; // recursive case 2
6  }
```

What is the running time? $O(\log n)$

Binary recursion

Example 1: The Fibonacci sequence

Fibonacci numbers are defined recursively:

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_i &= F_{i-1} + F_{i-2}, \text{ for } i > 1. \end{aligned}$$

The ratio $F_i/F_{i-1} \to \frac{1+\sqrt{5}}{2} \approx 1.61803\ldots$ (the golden ratio)

Example 1: The Fibonacci sequence

Fibonacci numbers are defined recursively:

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_i &= F_{i-1} + F_{i-2}, \text{ for } i > 1. \end{aligned}$$

The ratio $F_i/F_{i-1} \to \frac{1+\sqrt{5}}{2} \approx 1.61803\ldots$ (the golden ratio)

A recursive algorithm to calculate the k-th Fibonacci number F_k

```
1 Algorithm BinaryFib(k):
2 if k < 2 then
3     return k
4 else
5     return BinaryFib(k - 1) + BinaryFib(k - 2)
```

What is the running time?

Example 1: The Fibonacci sequence

```
1  Algorithm BinaryFib(k):
2  if k < 2 then
3      return k
4  else
5      return BinaryFib(k - 1) + BinaryFib(k - 2)
```

Let n_k denote number of recursive calls made by **BinaryFib(k)**.

$$
\begin{aligned}
n_0 &= 1 \\
n_1 &= 1 \\
n_2 &= n_1 + n_0 + 1 = 1 + 1 + 1 = 3 \\
n_3 &= n_2 + n_1 + 1 = 3 + 1 + 1 = 5 \\
n_4 &= n_3 + n_2 + 1 = 5 + 3 + 1 = 9 \\
n_5 &= n_4 + n_3 + 1 = 9 + 5 + 1 = 15
\end{aligned}
$$

$n_k > 2^{k/2}$. **It increases exponentially!**

Example 1: The Fibonacci sequence

Modify the output as the pair of Fibonacci numbers (F_k, F_{k-1})

```
1  Algorithm LinearFibonacci(k):
2  if k = 1 then
3      return (k, 0)
4  else
5      (i, j) = LinearFibonacci(k - 1)
6  return (i + j, i)
```

What is the running time?

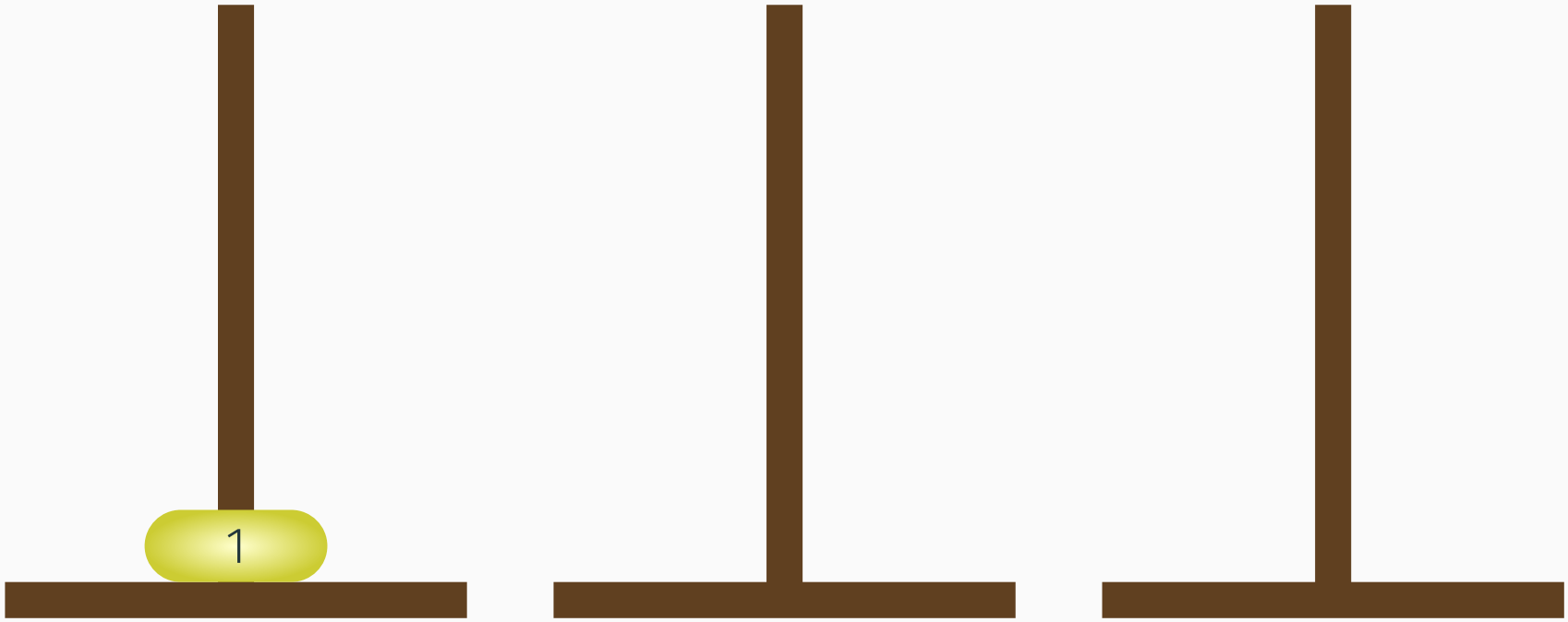
Example 1: The Fibonacci sequence

Modify the output as the pair of Fibonacci numbers (F_k, F_{k-1})

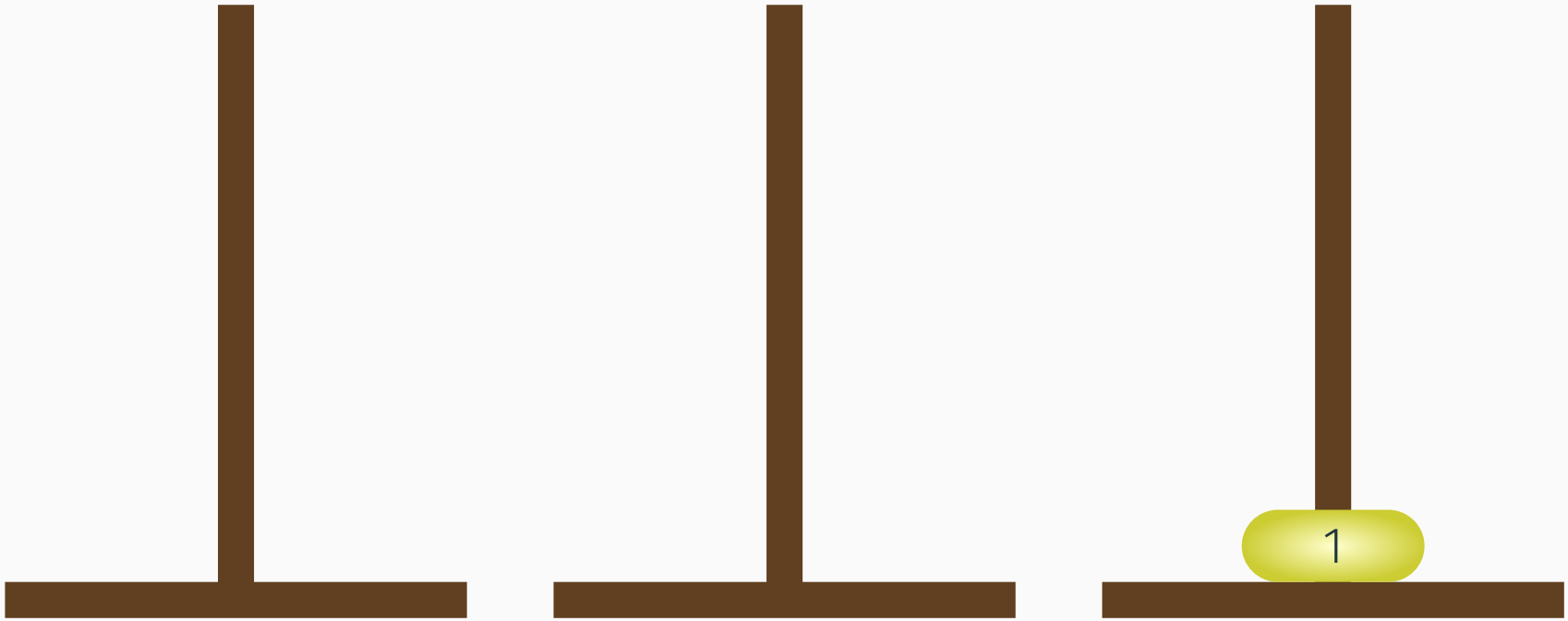
```
1  Algorithm LinearFibonacci(k):
2  if k = 1 then
3      return (k, 0)
4  else
5      (i, j) = LinearFibonacci(k - 1)
6  return (i + j, i)
```

What is the running time? $O(k)$

Tower of Hanoi – 1 Disc

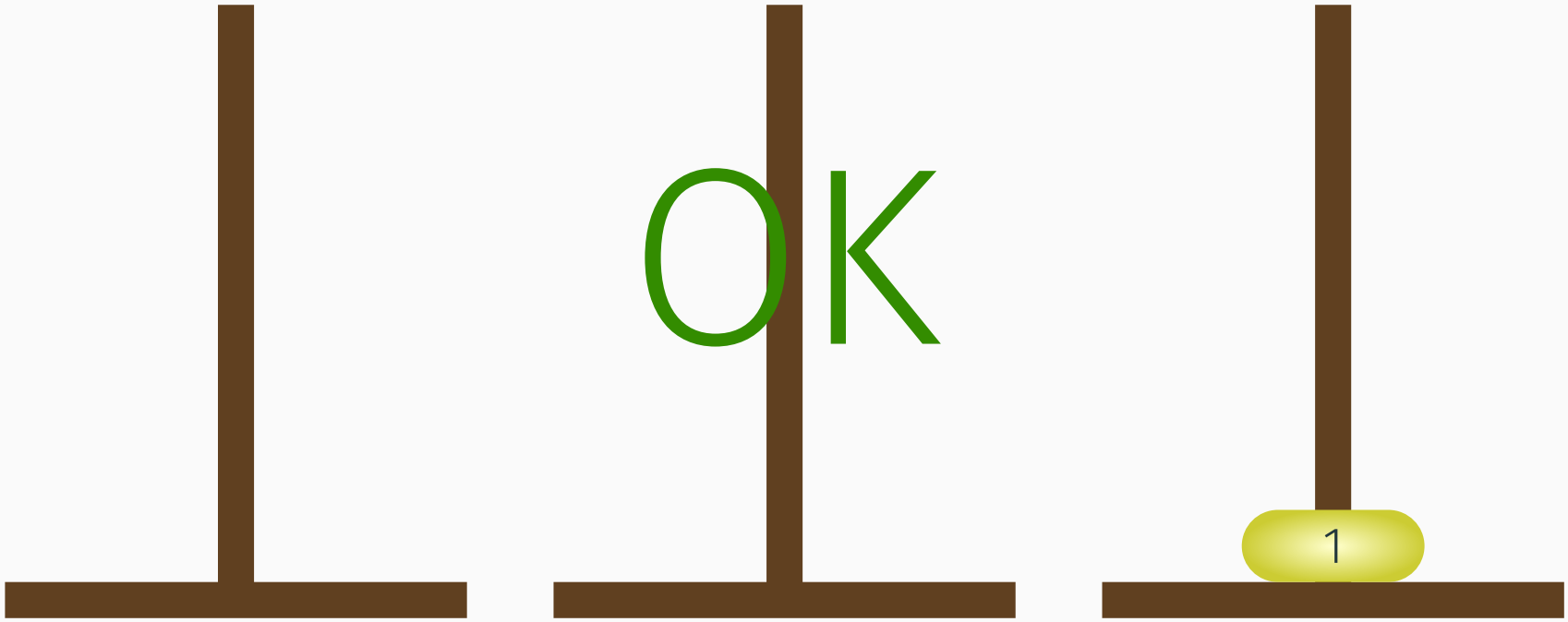


Tower of Hanoi – 1 Disc

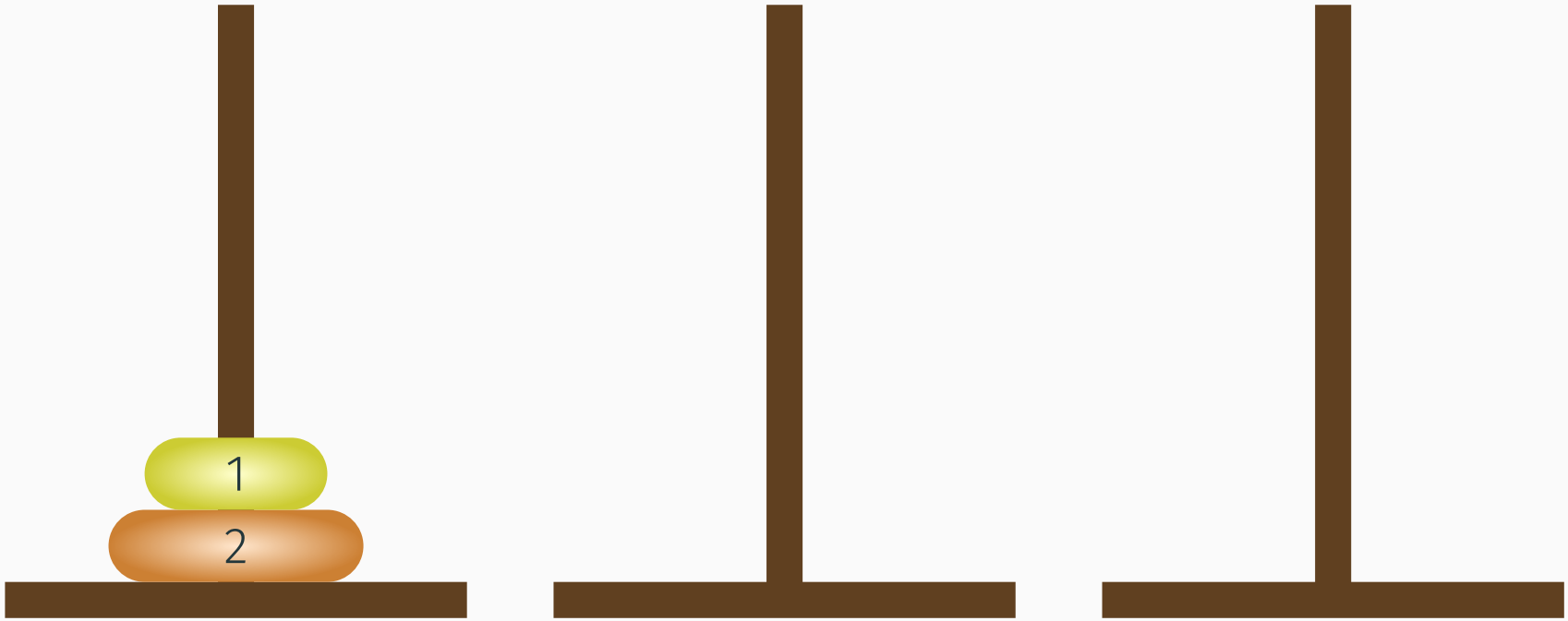


Moved disc from pole 1 to pole 3.

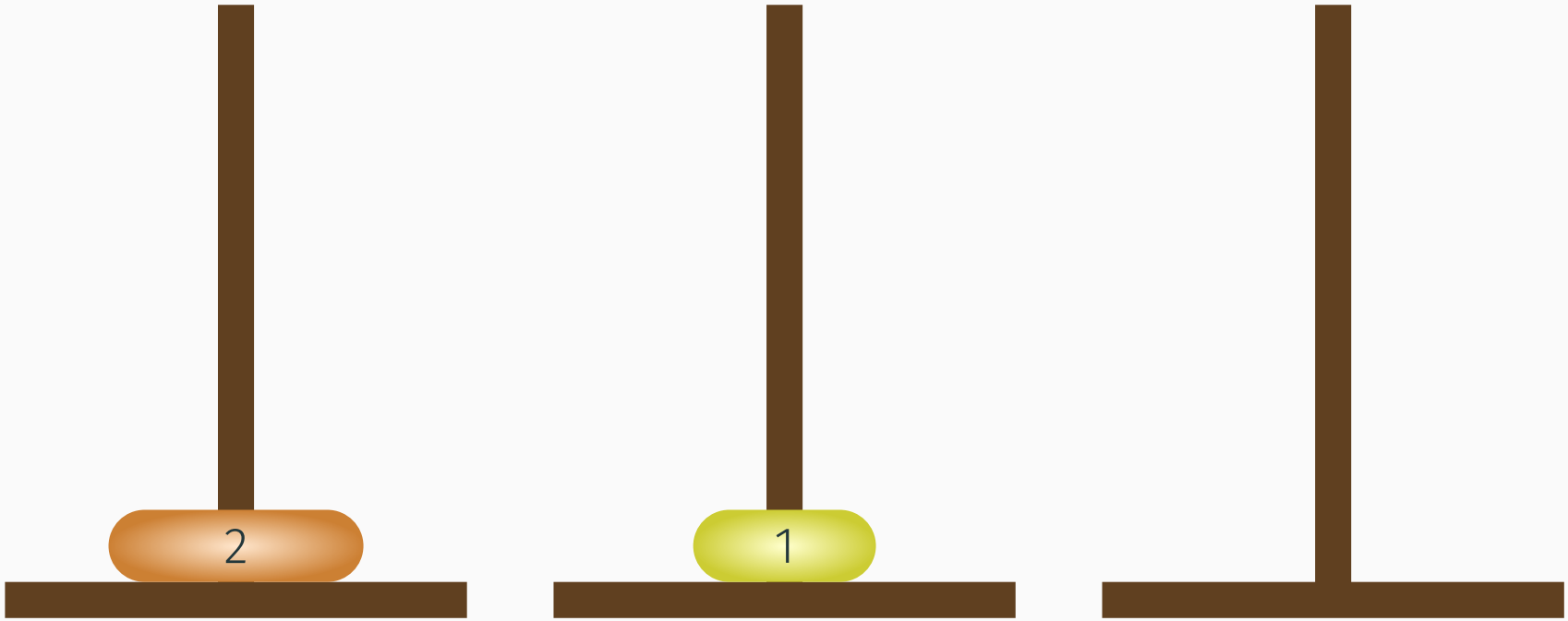
Tower of Hanoi – 1 Disc



Tower of Hanoi – 2 Discs

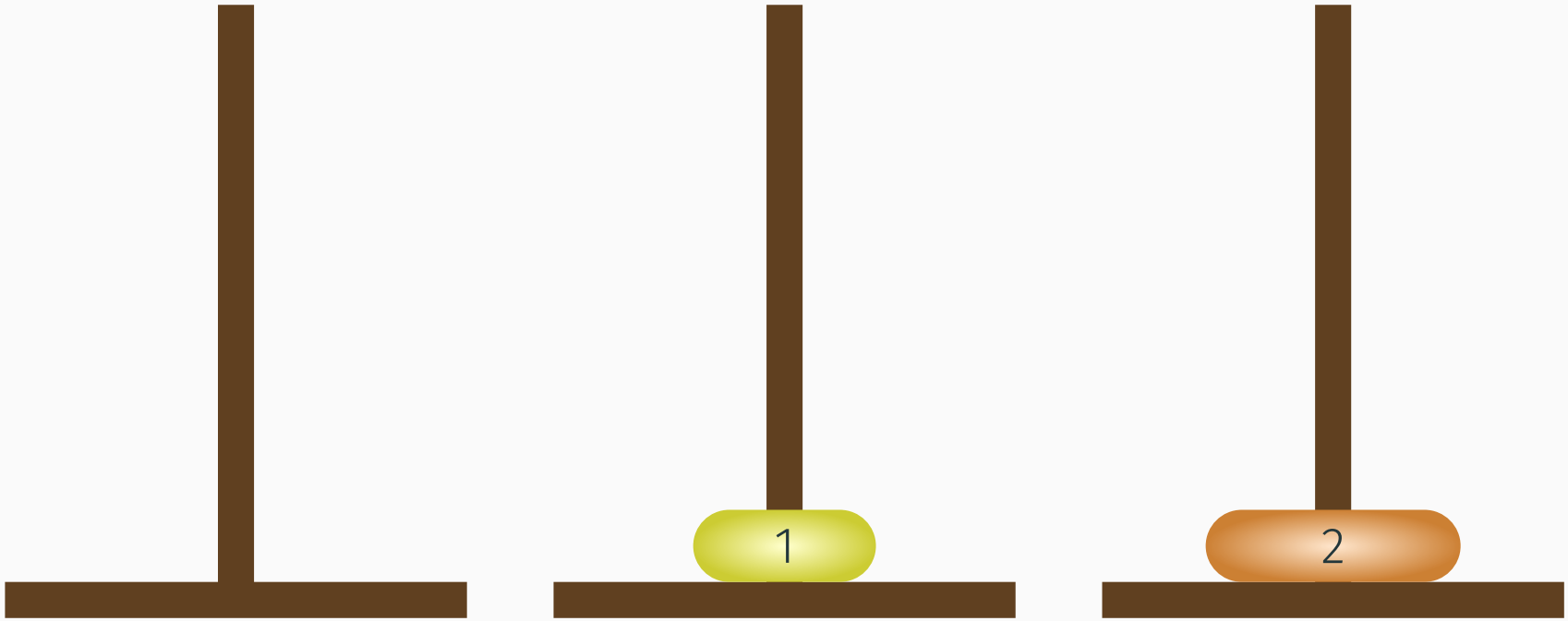


Tower of Hanoi – 2 Discs



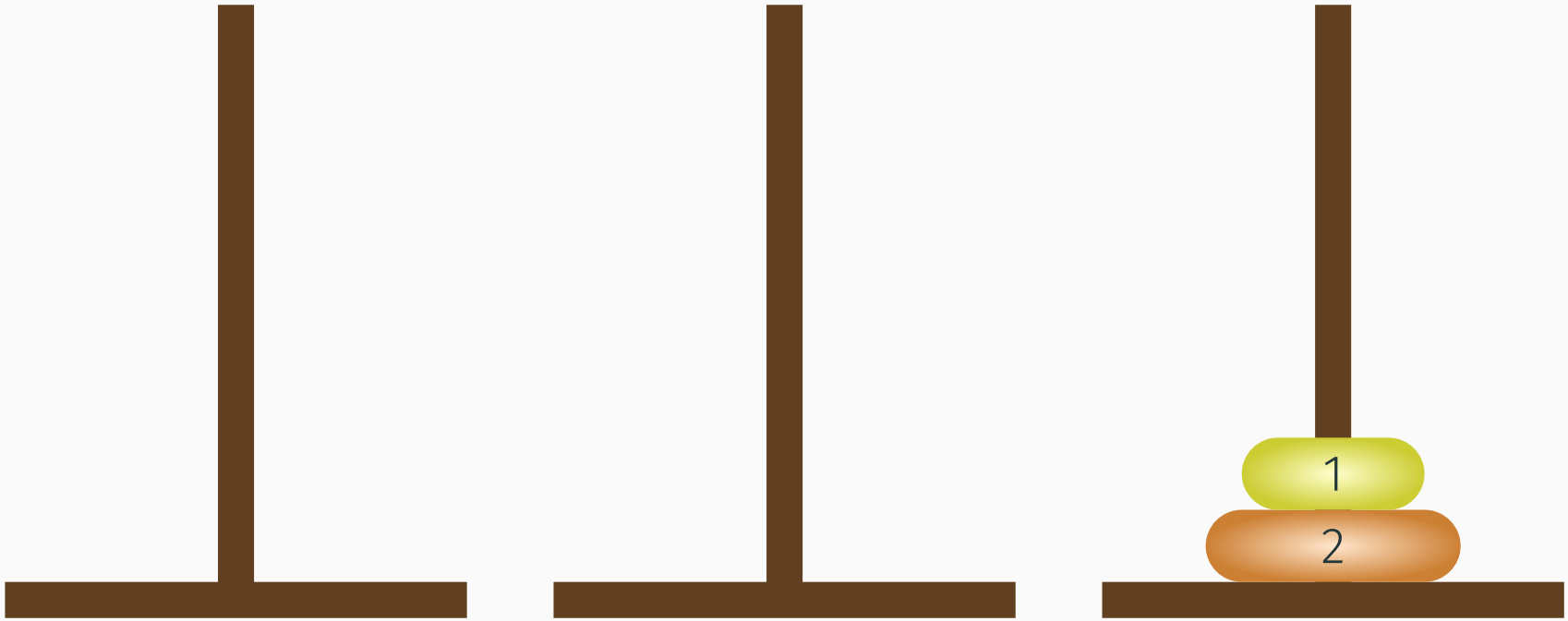
Moved disc from pole 1 to pole 2.

Tower of Hanoi – 2 Discs



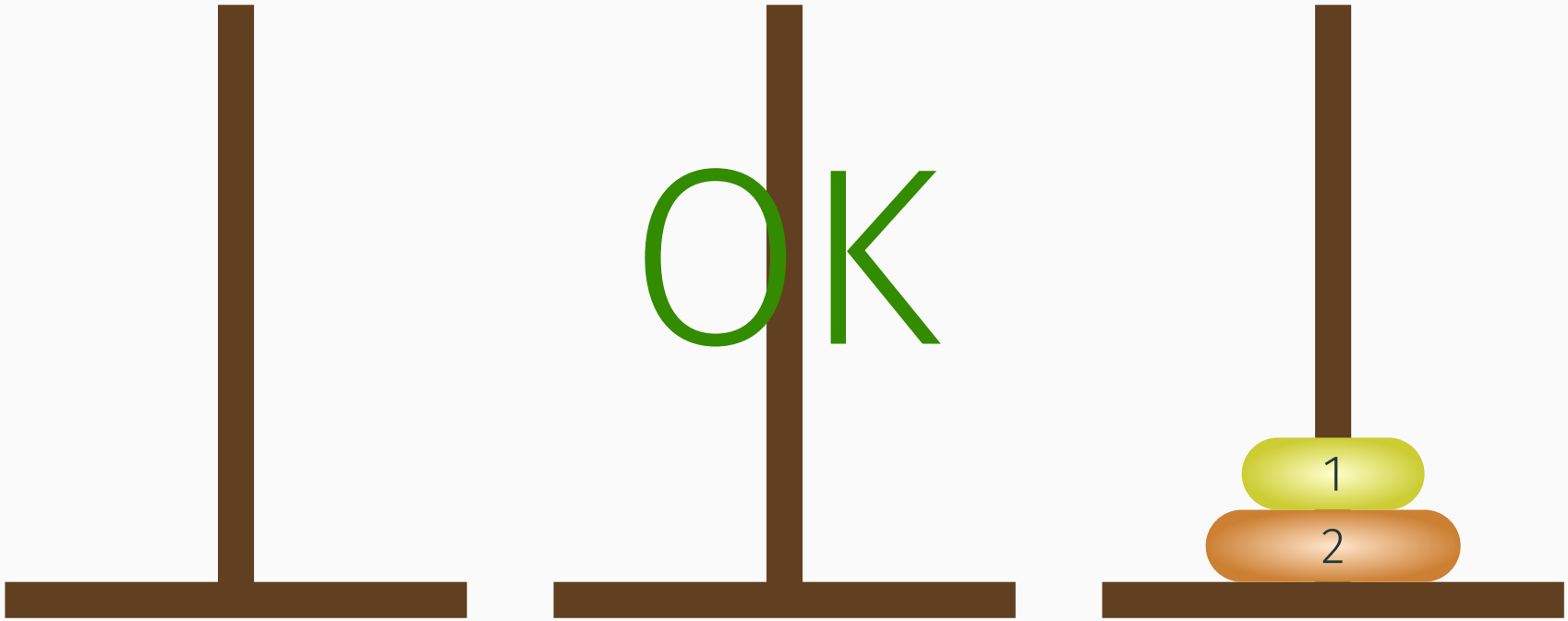
Moved disc from pole 1 to pole 3.

Tower of Hanoi – 2 Discs

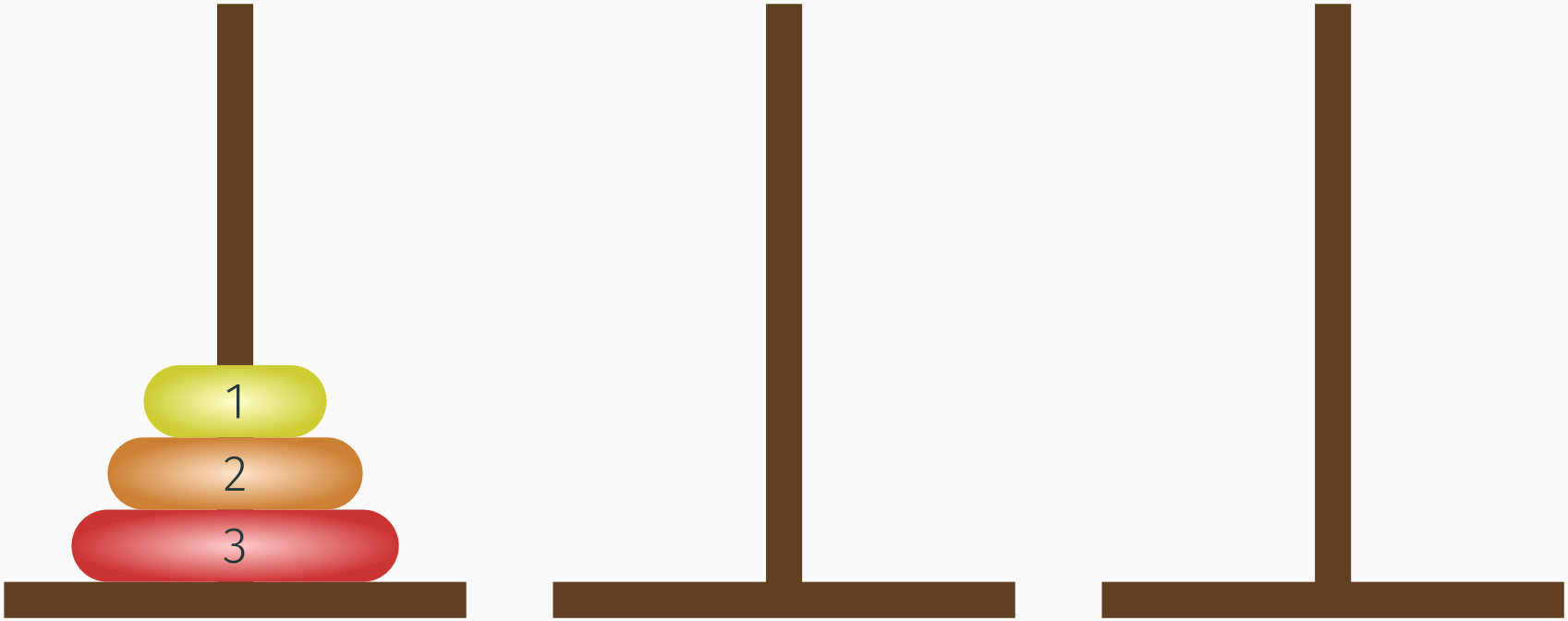


Moved disc from pole 2 to pole 3.

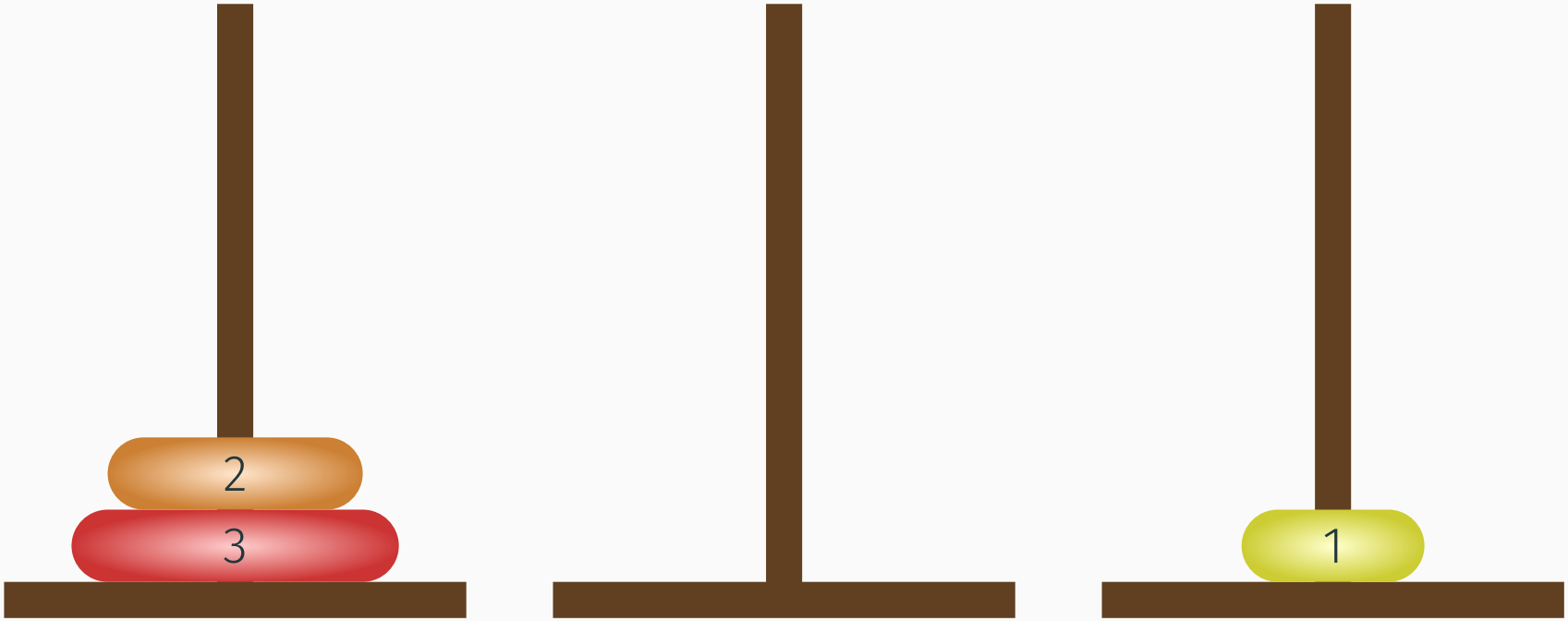
Tower of Hanoi – 2 Discs



Tower of Hanoi – 3 Discs

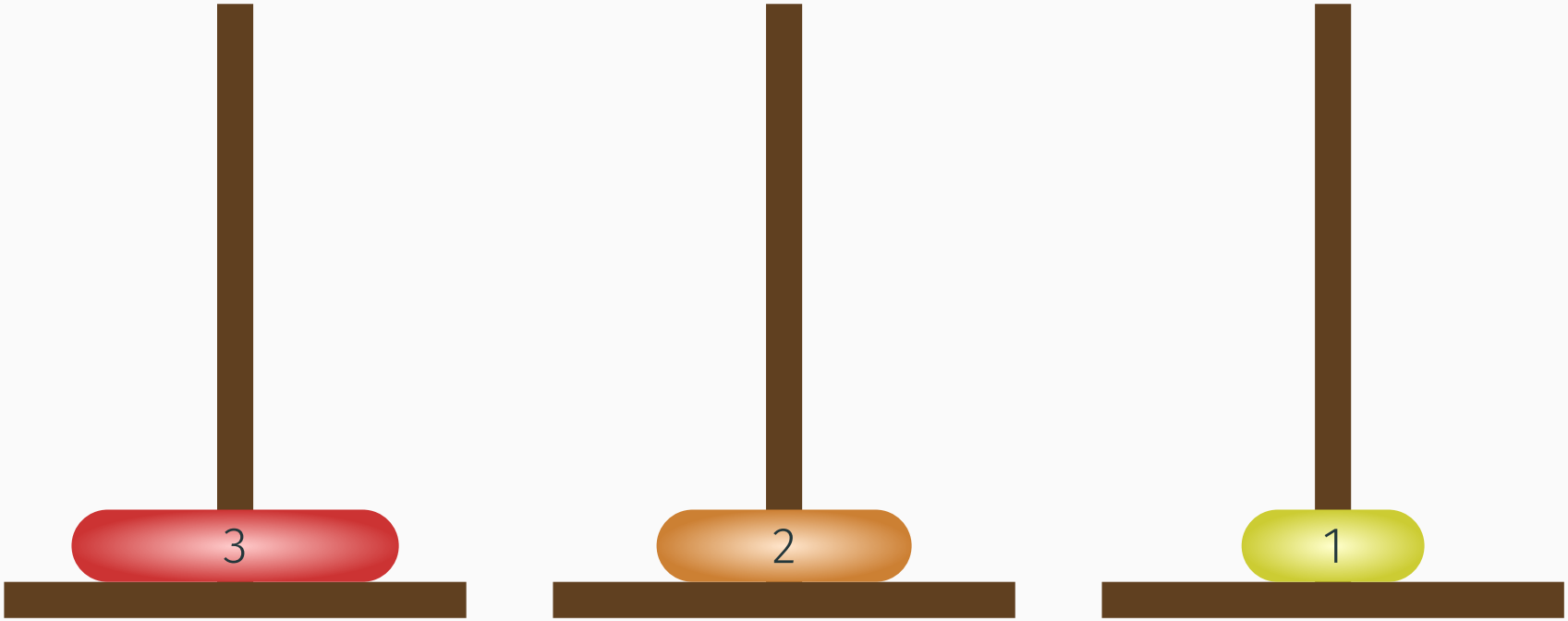


Tower of Hanoi – 3 Discs



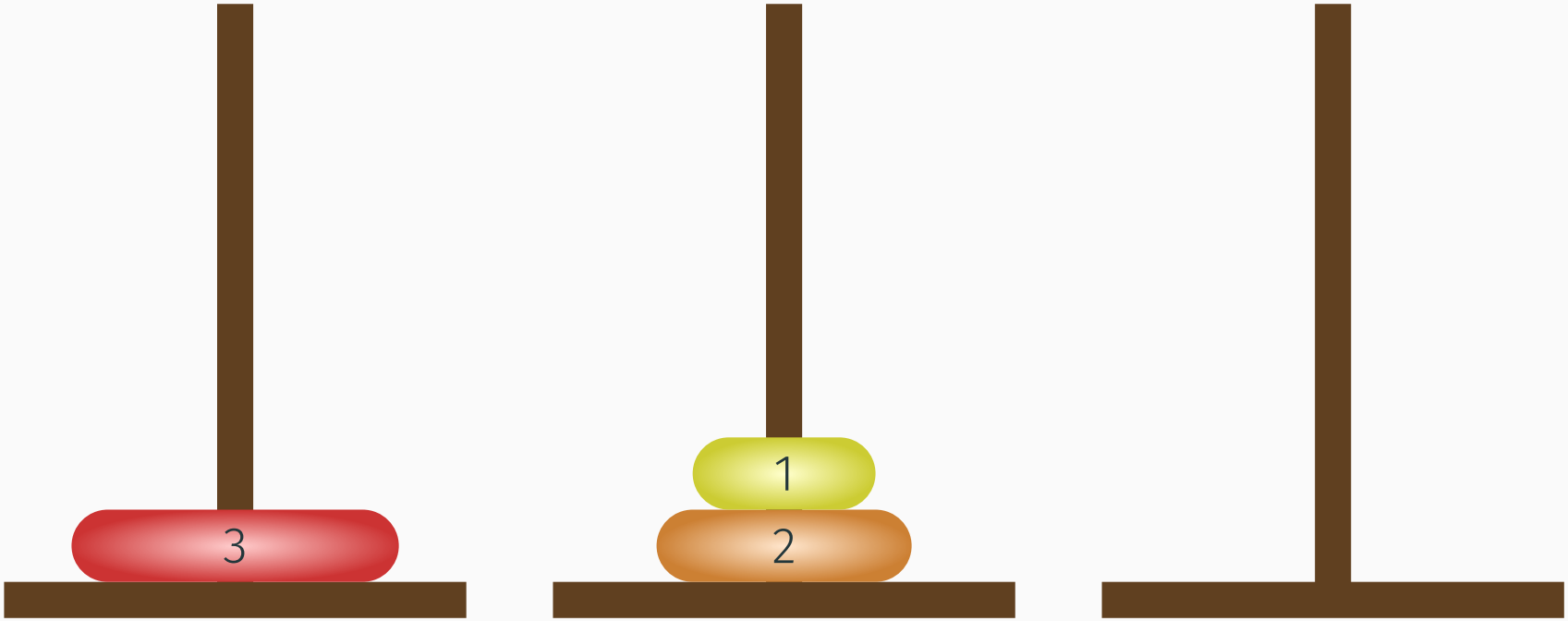
Moved disc from pole 1 to pole 3.

Tower of Hanoi – 3 Discs



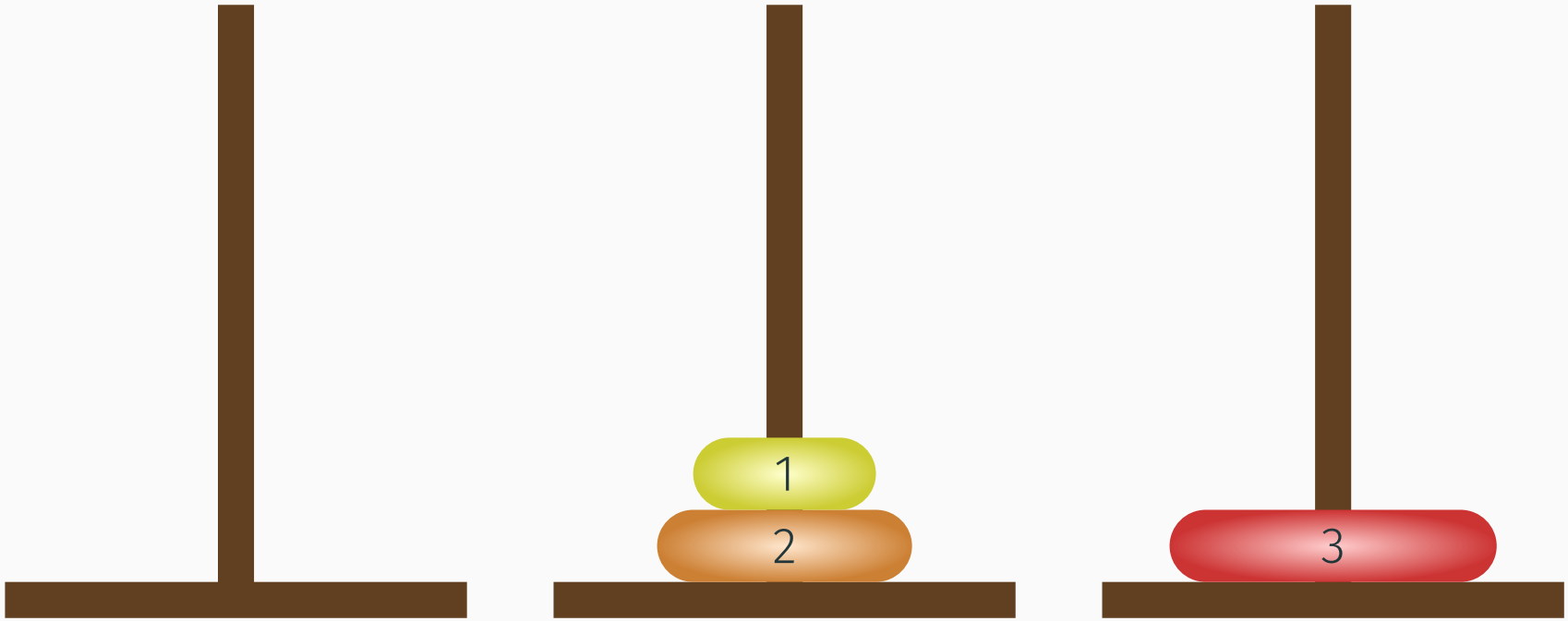
Moved disc from pole 1 to pole 2.

Tower of Hanoi – 3 Discs



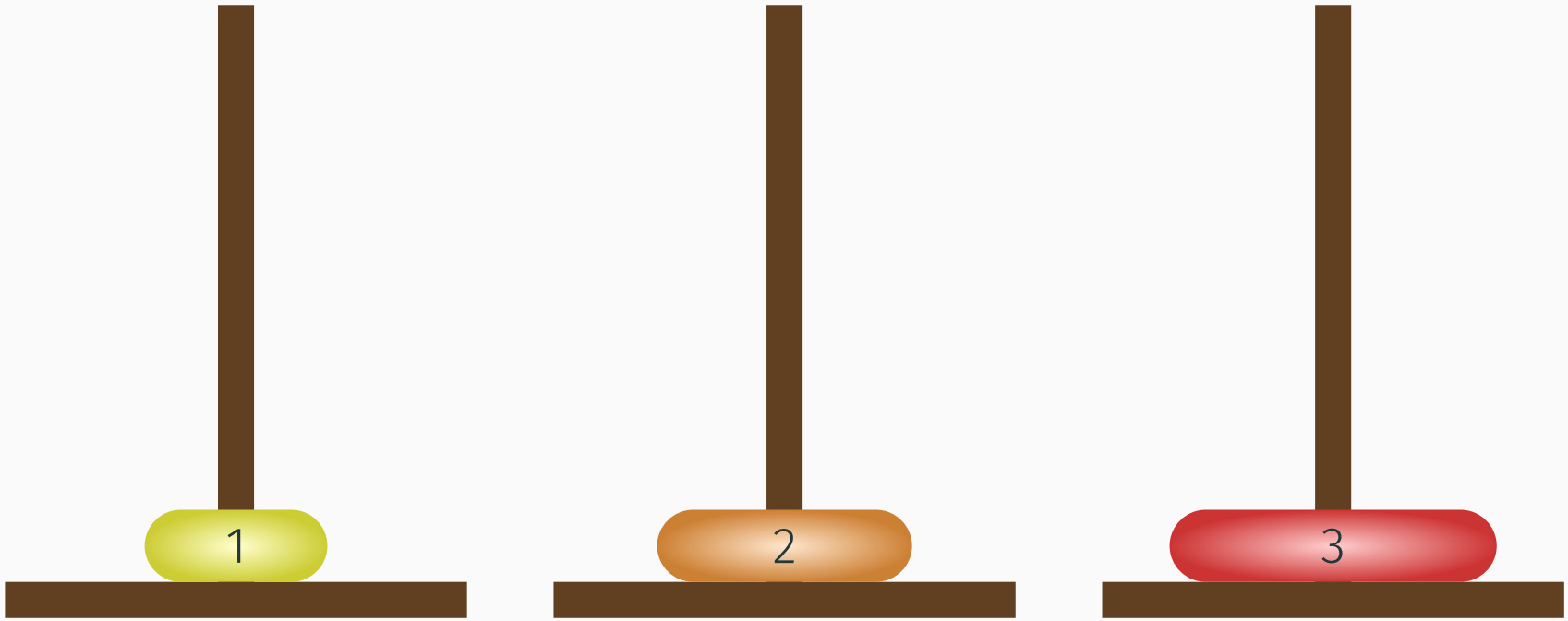
Moved disc from pole 3 to pole 2.

Tower of Hanoi – 3 Discs



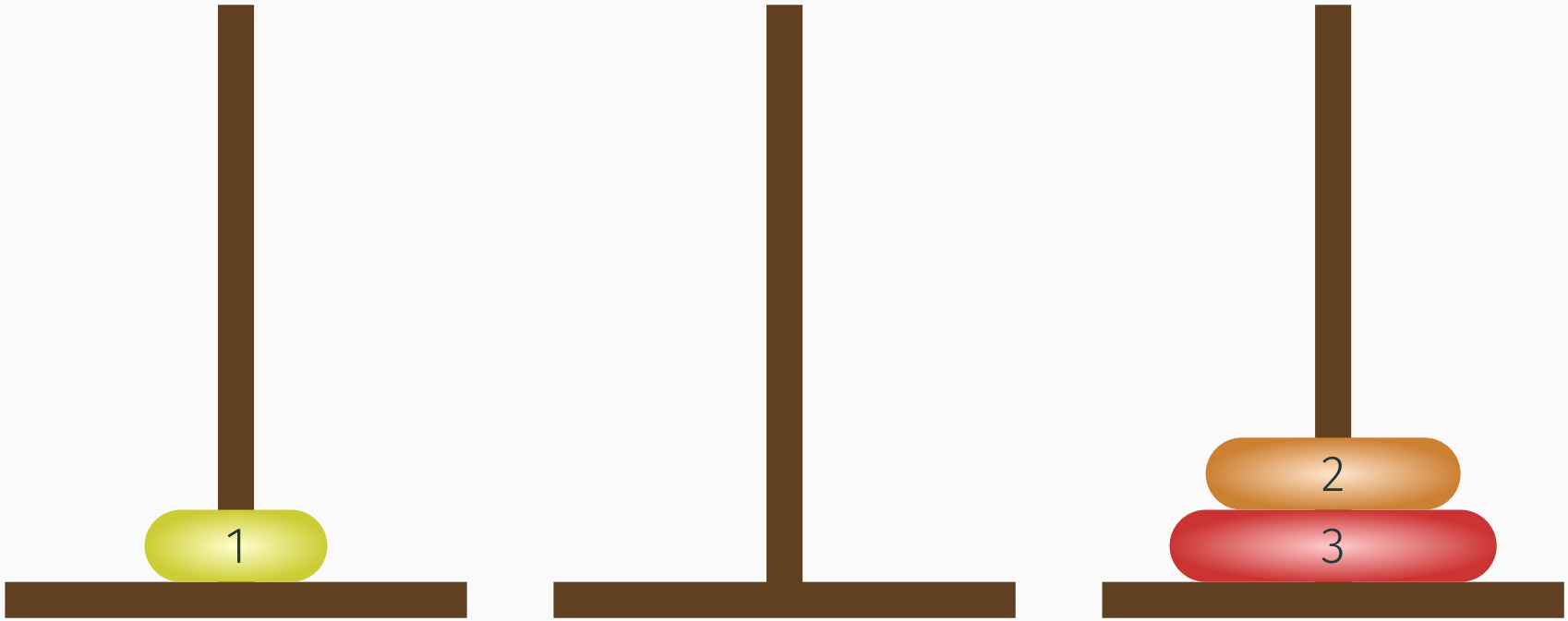
Moved disc from pole 1 to pole 3.

Tower of Hanoi – 3 Discs



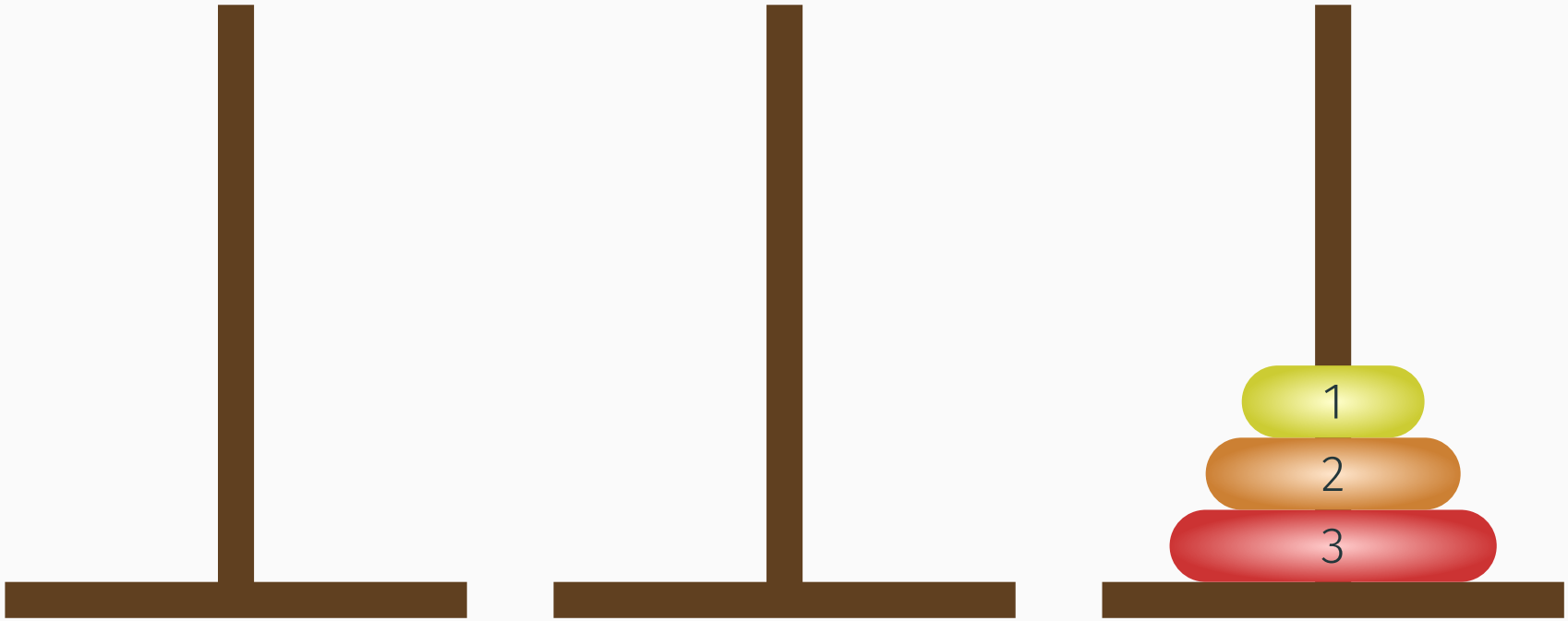
Moved disc from pole 2 to pole 1.

Tower of Hanoi – 3 Discs



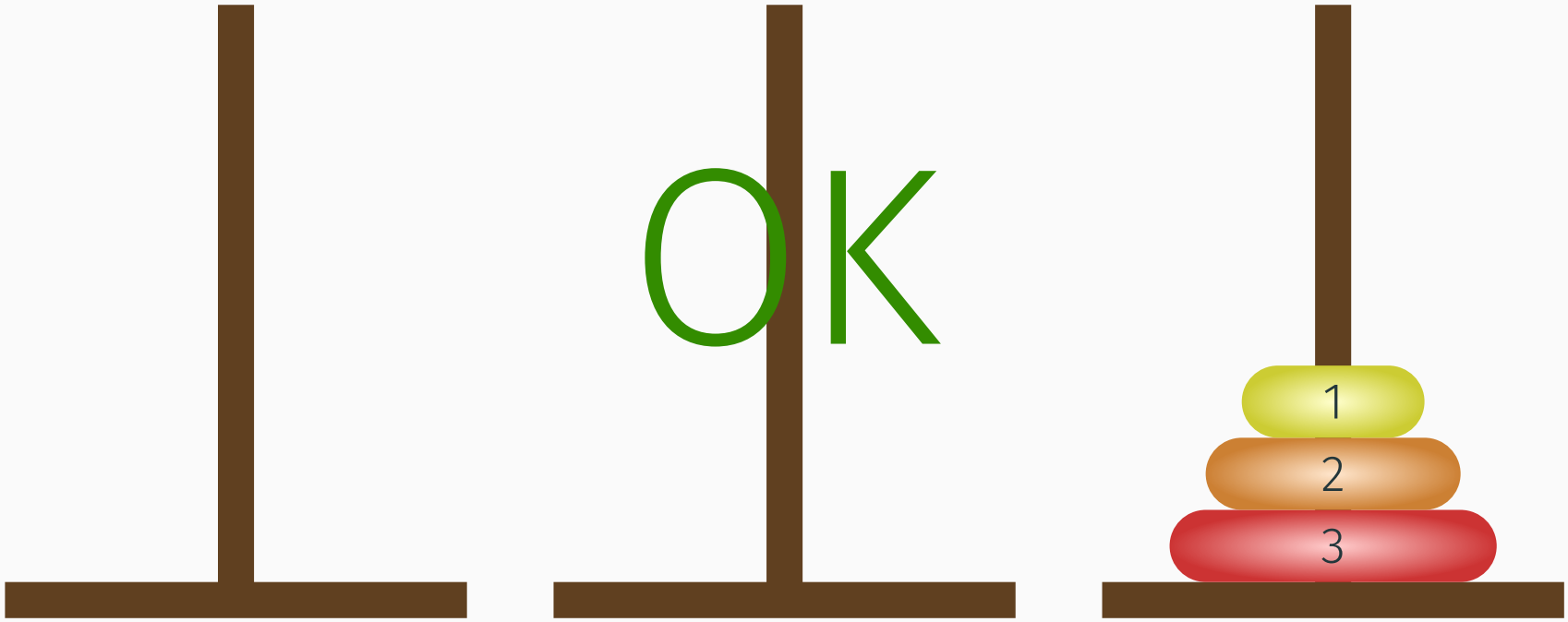
Moved disc from pole 2 to pole 3.

Tower of Hanoi – 3 Discs

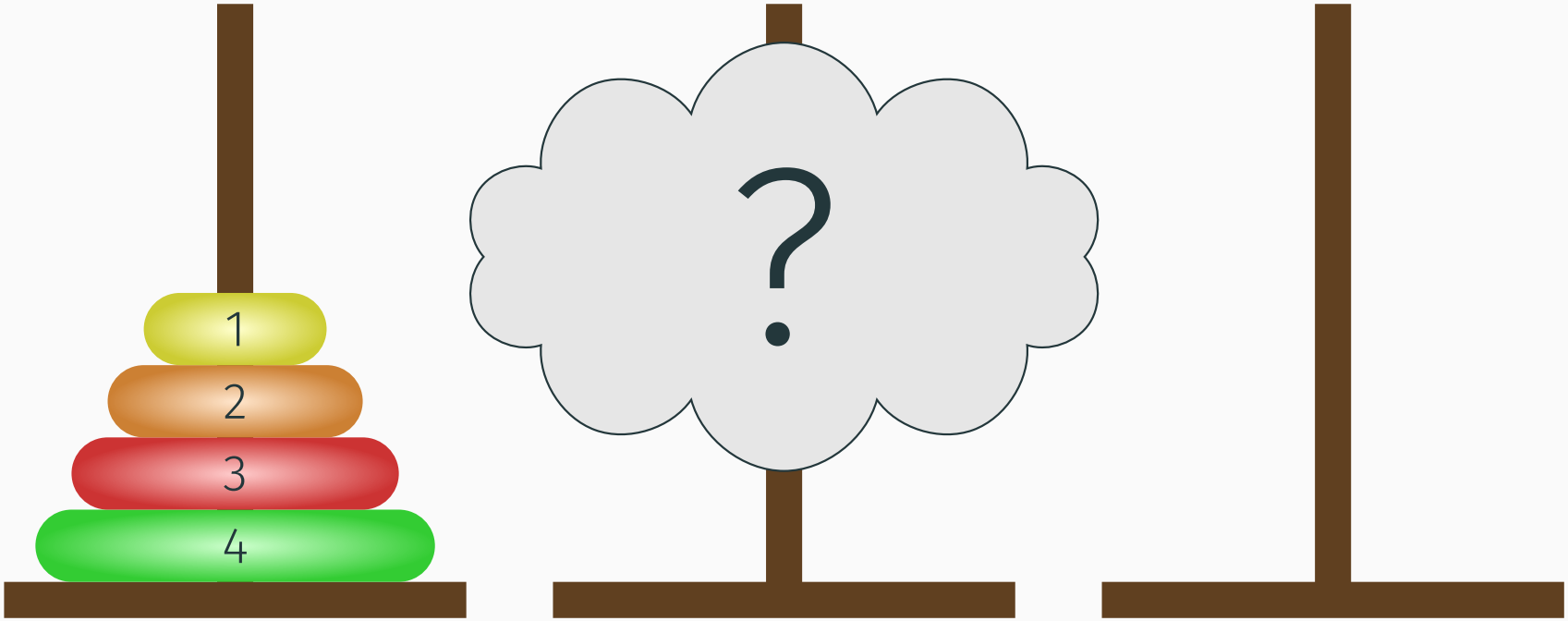


Moved disc from pole 1 to pole 3.

Tower of Hanoi – 3 Discs



Tower of Hanoi – 4 Disc



Example 2: The Tower of Hanoi

```
1 MoveTower(disk, source, dest, spare):
2 if disk == 1:
3     move disk from source to dest
4 else
5     MoveTower(disk - 1, source, spare, dest)
6     move disk from source to dest
7     MoveTower(disk - 1, spare, dest, source)
```

What is the running time?

Example 2: The Tower of Hanoi

```
1 MoveTower(disk, source, dest, spare):
2 if disk == 1:
3     move disk from source to dest
4 else
5     MoveTower(disk - 1, source, spare, dest)
6     move disk from source to dest
7     MoveTower(disk - 1, spare, dest, source)
```

What is the running time?

Assme the running time is $T(n)$, where n is the number of disks. We have the recursive definition accroding to the pseudocode.

$$T(n) = 1 + 2T(n-1)$$

Then

$$T(n) \approx 2T(n-1) \approx 4T(n-2) \approx \cdots \approx 2^n$$

Drawbacks and pitfalls of recursion

The Overhead Costs of Recursion

Many problems are naturally defined recursively, which can lead to simple, elegant code.

However, recursive solutions entail a cost in time and memory: each recursive call requires that the current process state (variables, program counter) be pushed onto the system stack, and popped once the recursion unwinds.

Recursive solutions may still be preferred unless there are very strict time/memory constraints.

The “Curse” in Recursion: Errors to Avoid

```
1  public static int  recursiveFactorial(int n) {
2    return  n * recursiveFactorial(n - 1);
3  }
```

The “Curse” in Recursion: Errors to Avoid

```
1  public static int  recursiveFactorial(int n) {
2    return  n * recursiveFactorial(n - 1);
3  }
```

There must be a base condition: the recursion must ground out!

The “Curse” in Recursion: Errors to Avoid

```
1  public static int  recursiveFactorial(int n) {
2    return  n * recursiveFactorial(n - 1);
3  }
```

There must be a base condition: the recursion must ground out!

```
1  public static int recursiveFactorial(int n) {
2    if (n == 0) return recursiveFactorial(n); // base case
3    else return n * recursiveFactorial(n - 1); // recursive case
4  }
```

The “Curse” in Recursion: Errors to Avoid

```
1  public static int  recursiveFactorial(int n) {
2    return  n * recursiveFactorial(n - 1);
3  }
```

There must be a base condition: the recursion must ground out!

```
1  public static int recursiveFactorial(int n) {
2    if (n == 0) return recursiveFactorial(n); // base case
3    else return n * recursiveFactorial(n - 1); // recursive case
4  }
```

The base condition must not involve more recursion!

The “Curse” in Recursion: Errors to Avoid

```java
public static int  recursiveFactorial(int n) {
  return  n * recursiveFactorial(n - 1);
}
```

There must be a base condition: the recursion must ground out!

```java
public static int recursiveFactorial(int n) {
  if (n == 0) return recursiveFactorial(n); // base case
  else return n * recursiveFactorial(n - 1); // recursive case
}
```

The base condition must not involve more recursion!

```java
public static int recursiveFactorial(int n) {
  if (n == 0) return 1; // base case
  else return (n - 1) * recursiveFactorial(n); // recursive case
}
```

The “Curse” in Recursion: Errors to Avoid

```
1  public static int  recursiveFactorial(int n) {
2    return  n * recursiveFactorial(n - 1);
3  }
```

There must be a base condition: the recursion must ground out!

```
1  public static int recursiveFactorial(int n) {
2    if (n == 0) return recursiveFactorial(n); // base case
3    else return n * recursiveFactorial(n - 1); // recursive case
4  }
```

The base condition must not involve more recursion!

```
1  public static int recursiveFactorial(int n) {
2    if (n == 0) return 1; // base case
3    else return (n - 1) * recursiveFactorial(n); // recursive case
4  }
```

The input must be converging toward the base condition!

Thank you!

Questions?