CS 106B

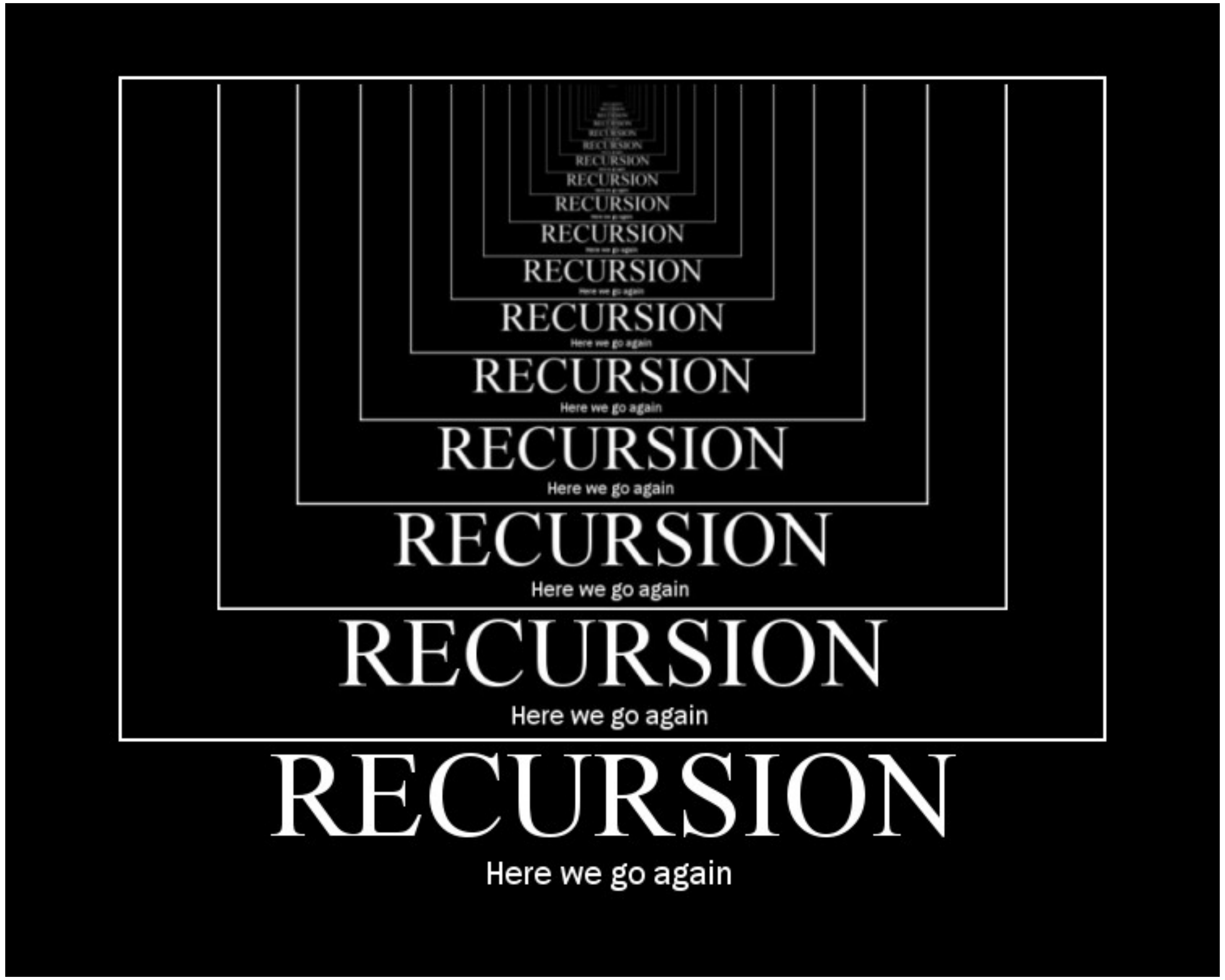
Lecture 7: Introduction to Recursion

Thursday, July 6th, 2017

Programming Abstractions
Summer 2017
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Chapter 5.4-5.6

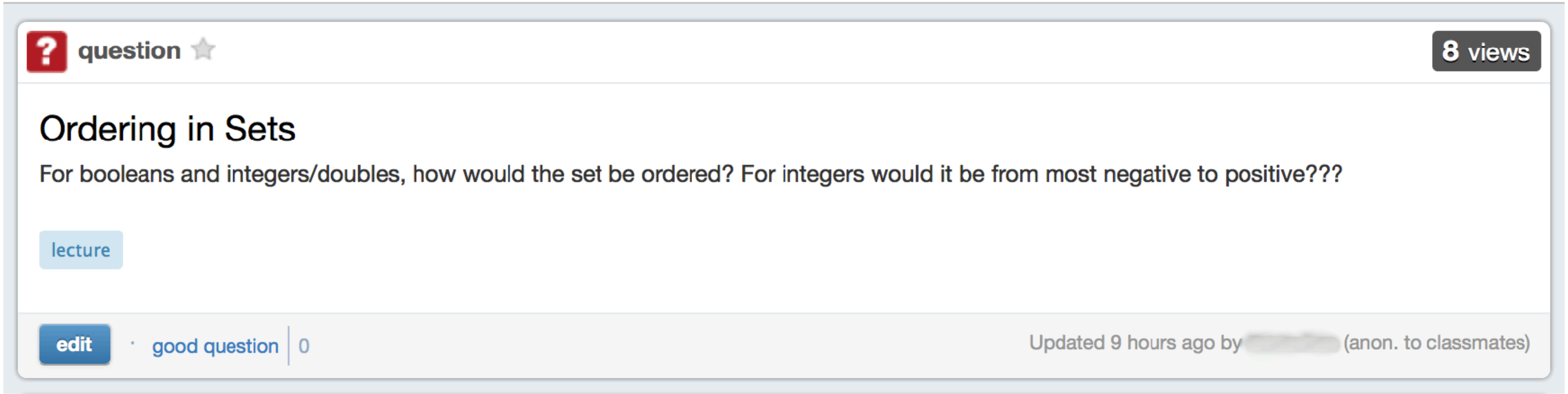


Today's Topics

- Logistics:
- Handout in class: http://web.stanford.edu/class/cs106b//lectures/7-IntroToRecursion/code/handout.pdf
- Writing a simple program all by yourself
- Serafini Due Wednesday, July 12th, noon
  - One submission of two files (wordLadder, Ngrams)
- Recursion!

Today's Topics

- There was a question last quarter on Piazza:


question

8 views

Ordering in Sets

For booleans and integers/doubles, how would the set be ordered? For integers would it be from most negative to positive???

lecture

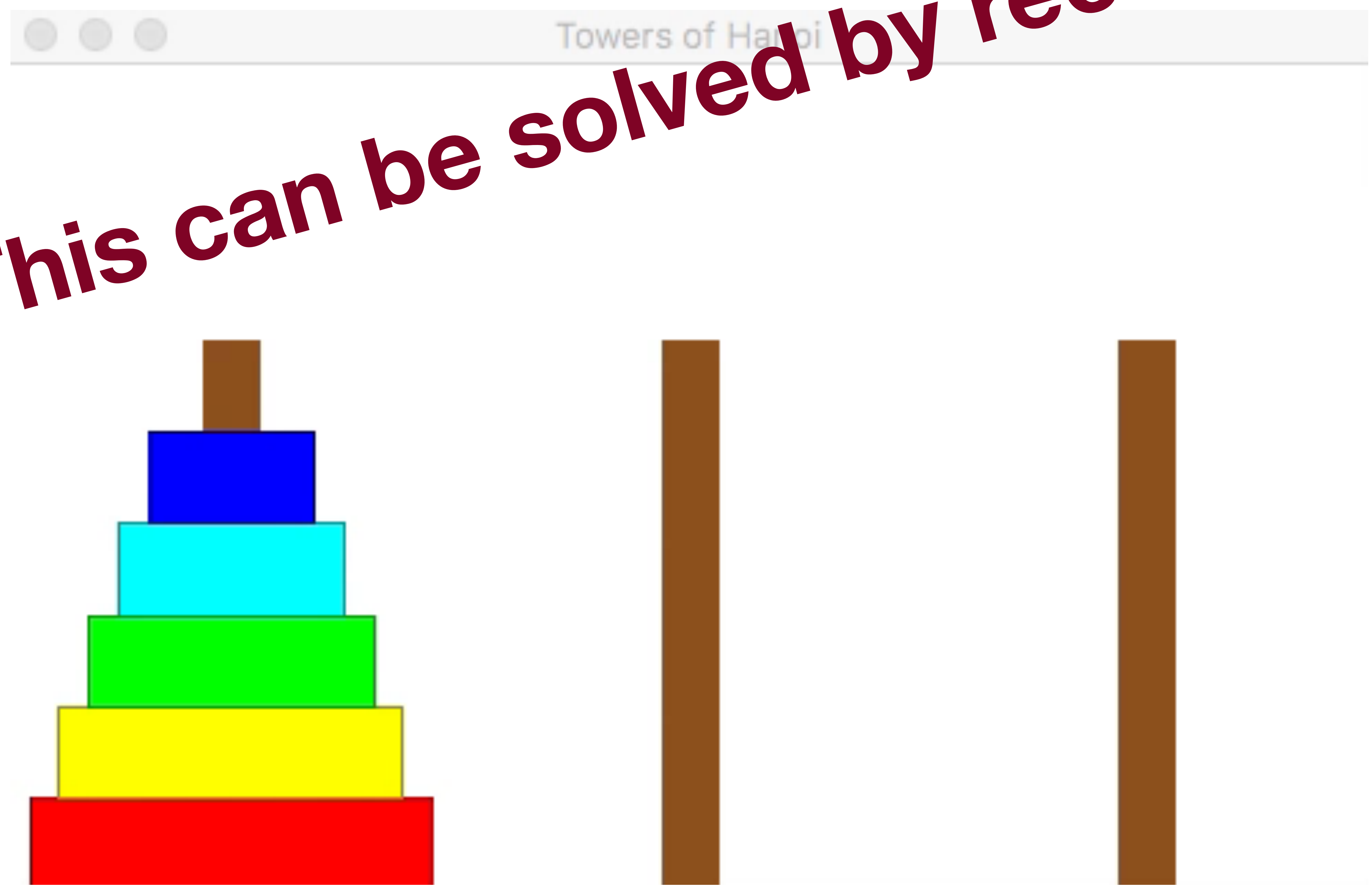
edit · good question | 0

Updated 9 hours ago by [redacted] (anon. to classmates)

- This is a *great* opportunity to write a quick program to test this yourself! Let's see how we might do that!

A Little Demo

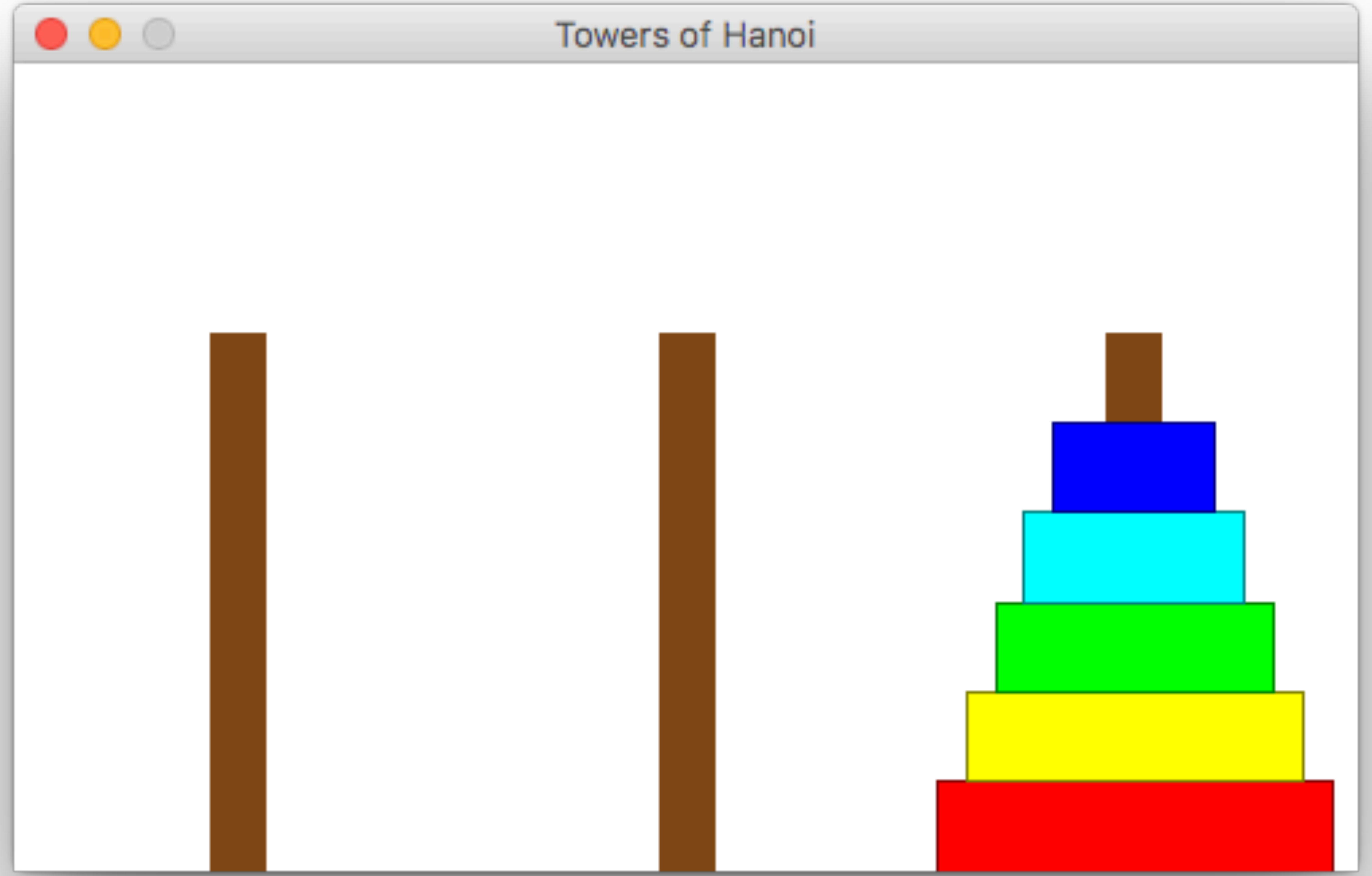
The Towers of Hanoi Puzzle



This can be solved by recursion!

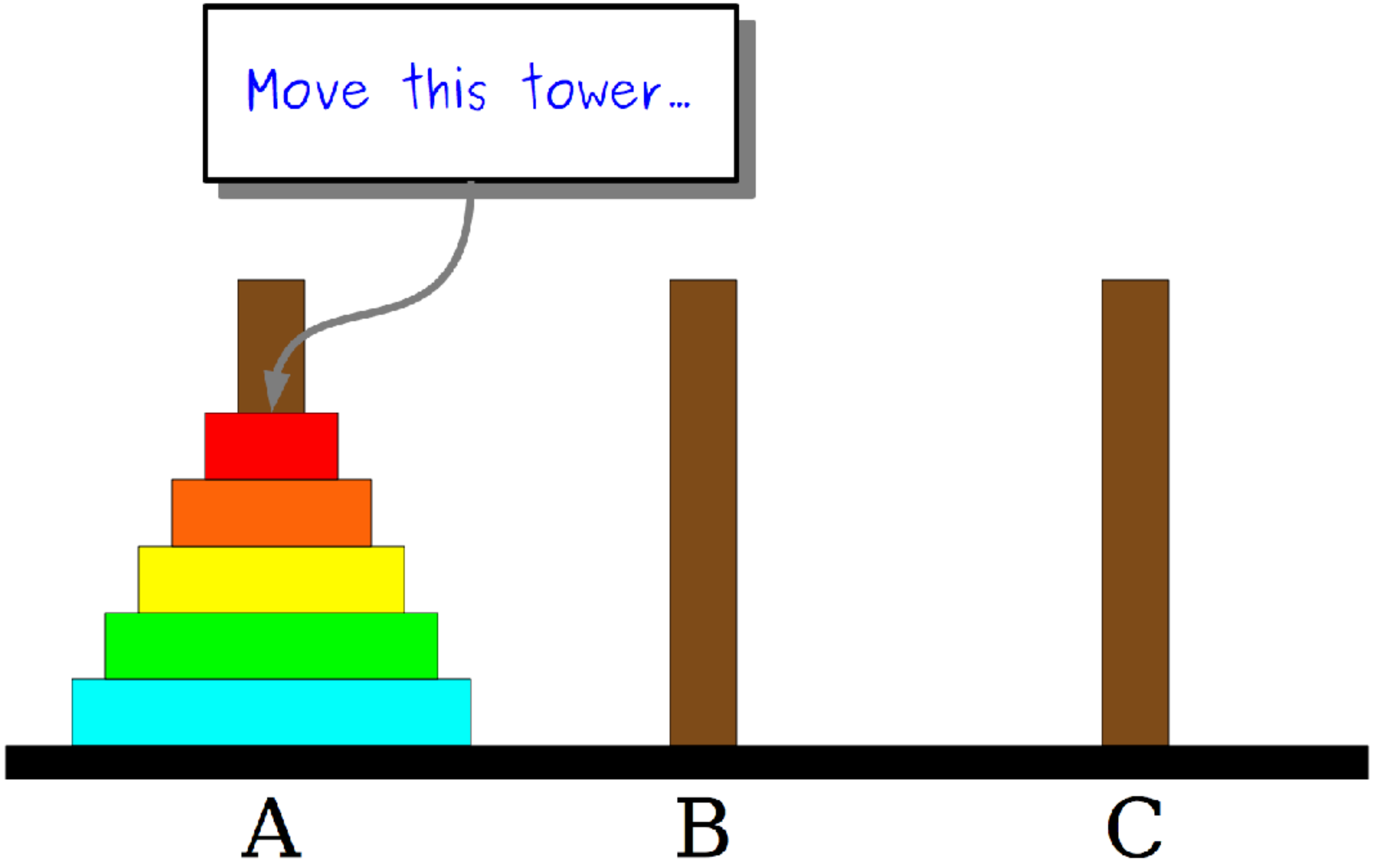
A Little Demo

By the end of today, we will be able to write this program, and you may talk about the algorithm in section



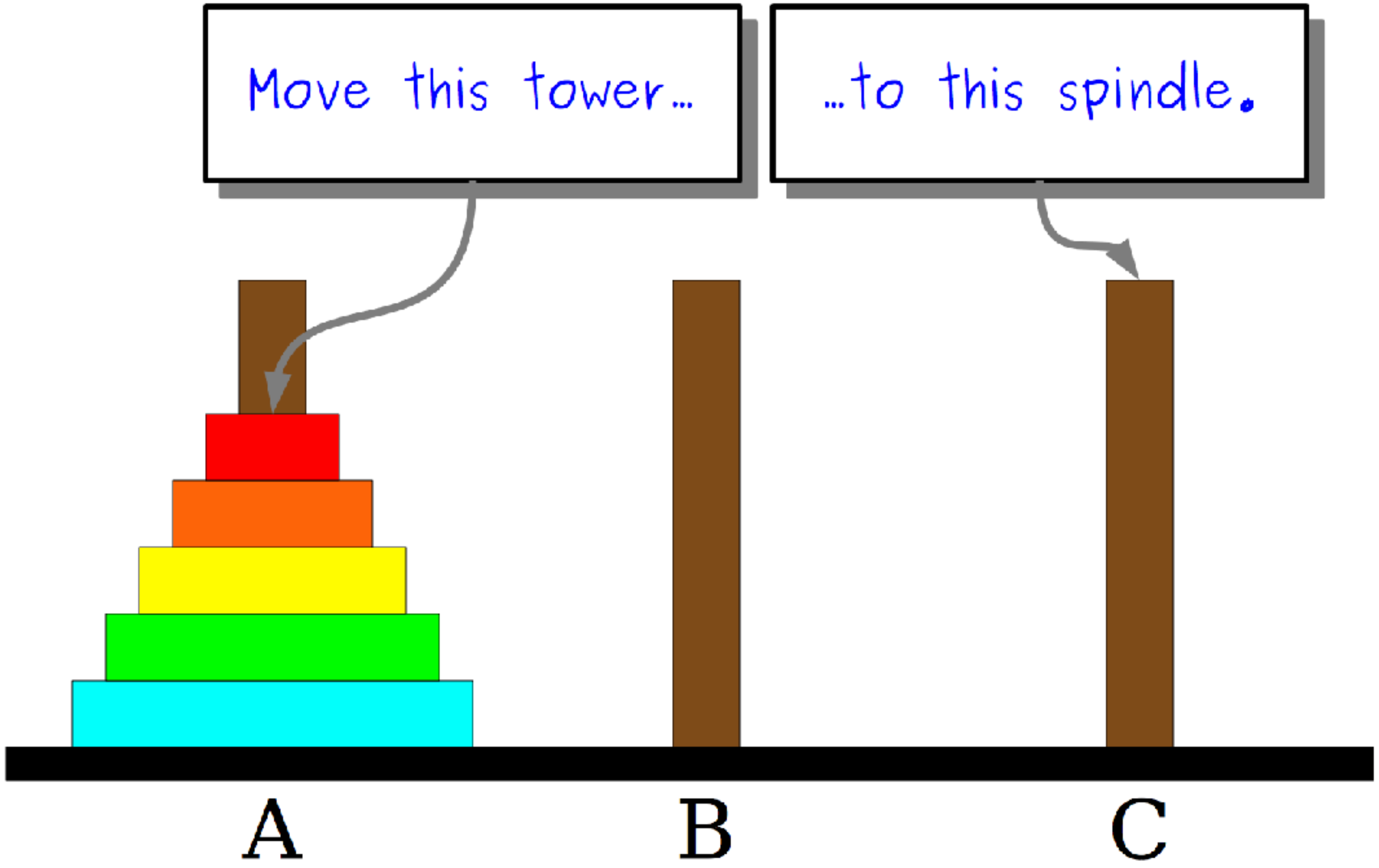
Towers of Hanoi

Here is the way the game is played:



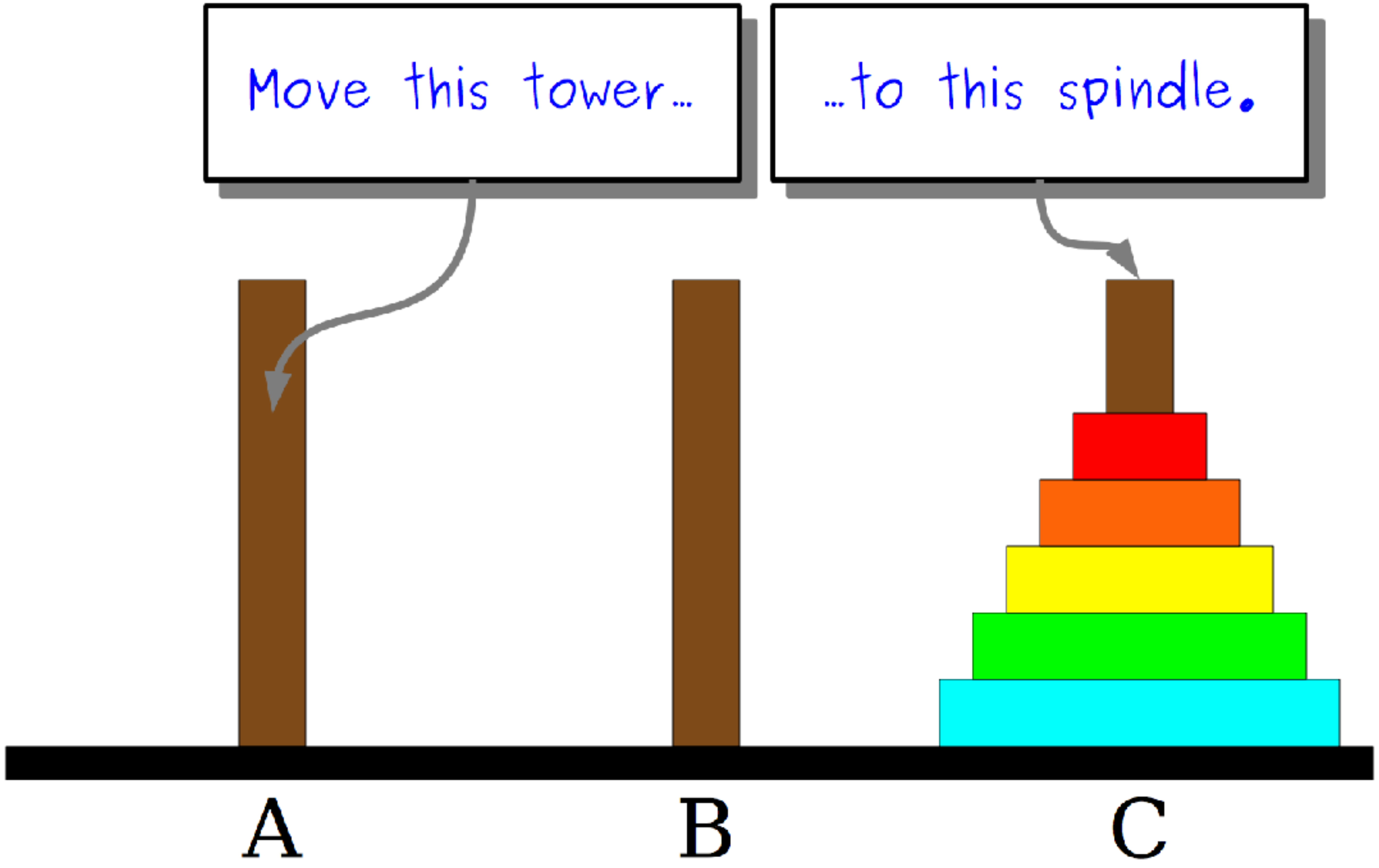
Towers of Hanoi

Here is the way the game is played:



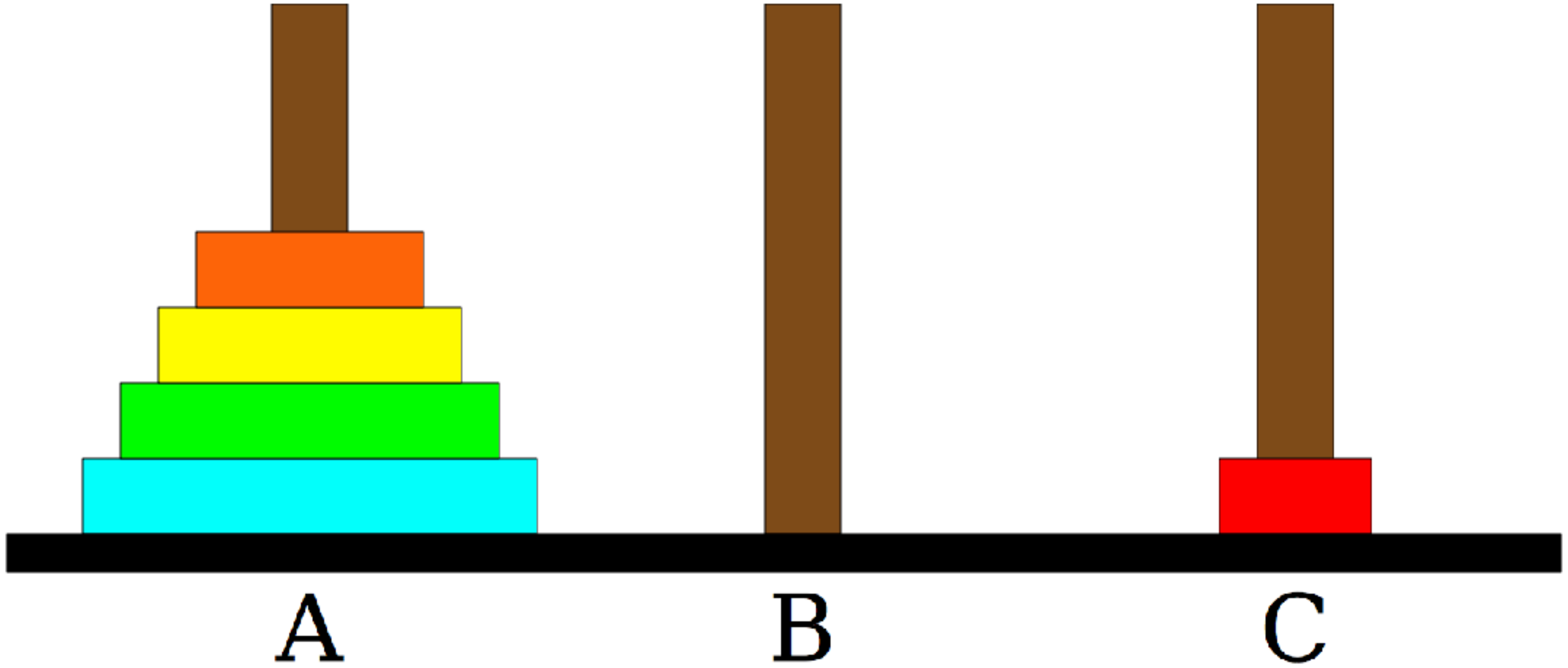
Towers of Hanoi

Here is the way the game is played:



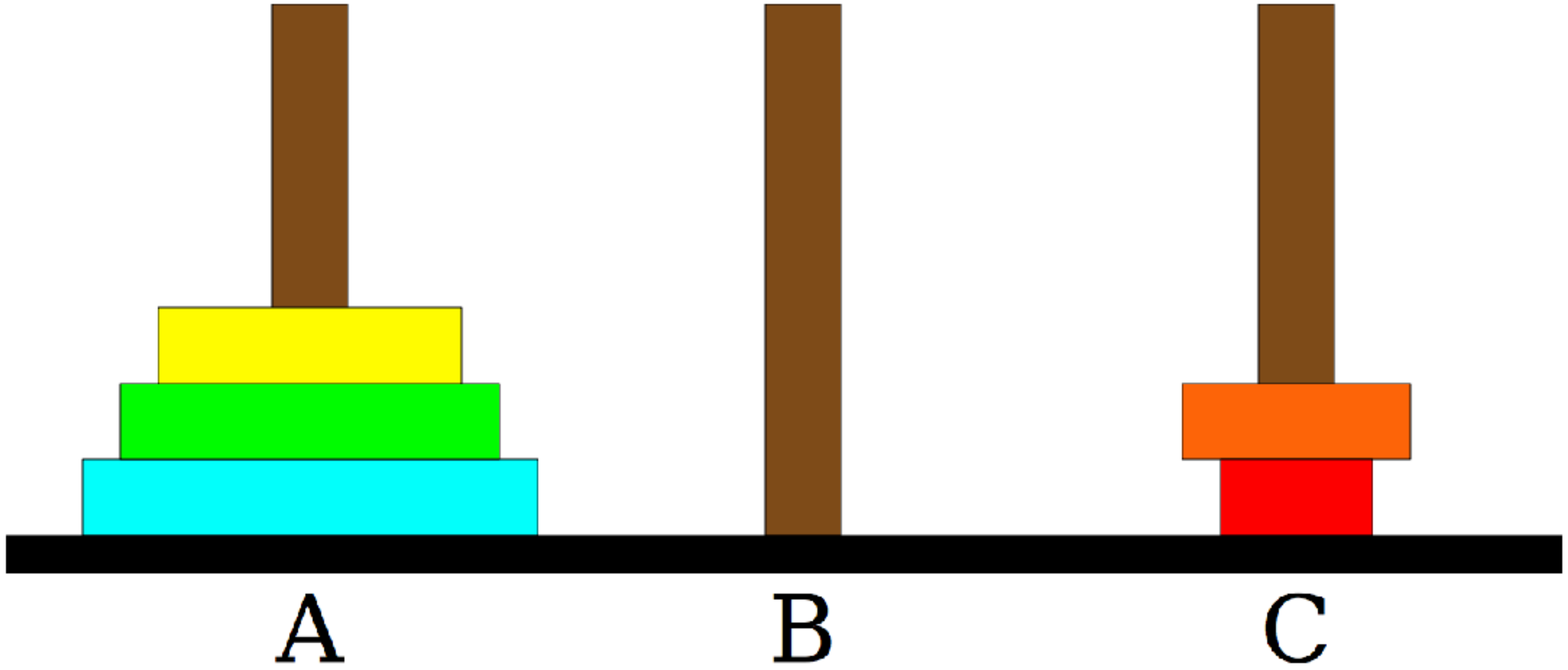
Towers of Hanoi

Here is the way the game is played:



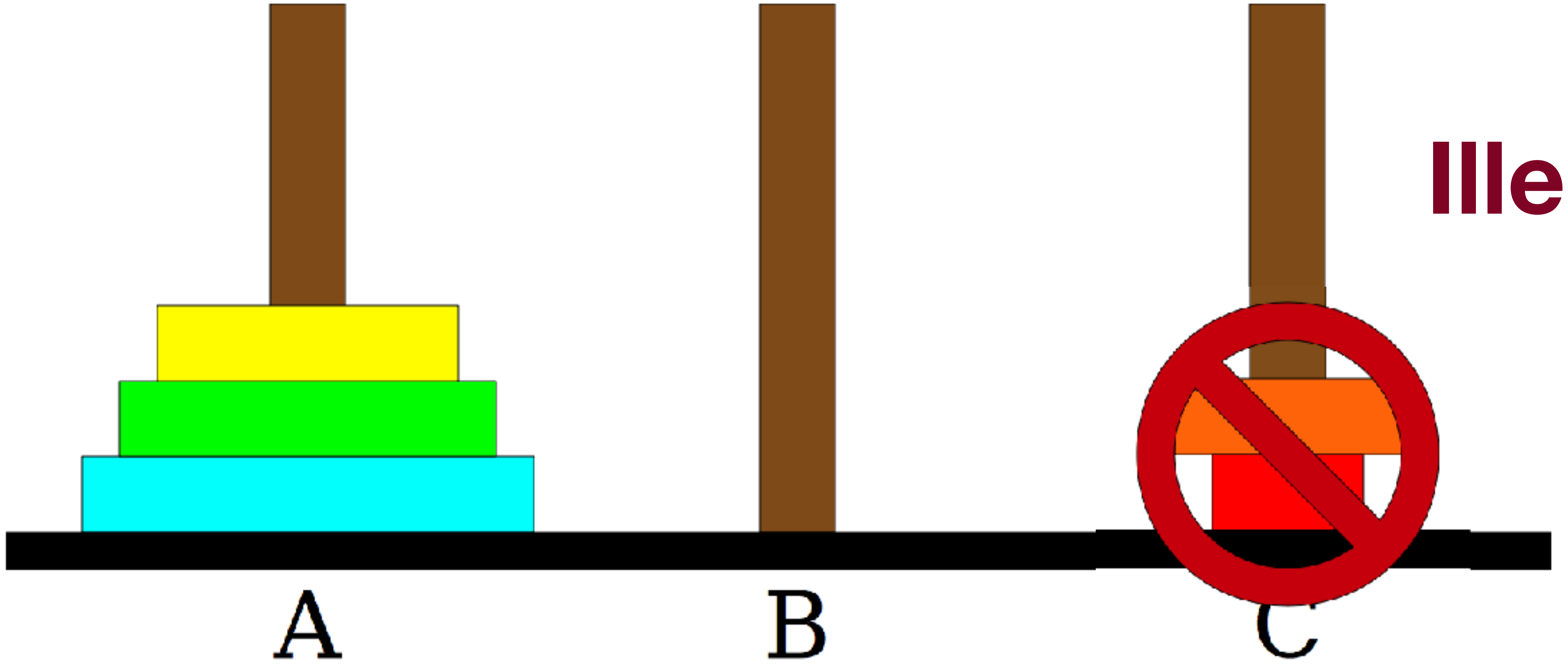
Towers of Hanoi

Here is the way the game is played:



Towers of Hanoi

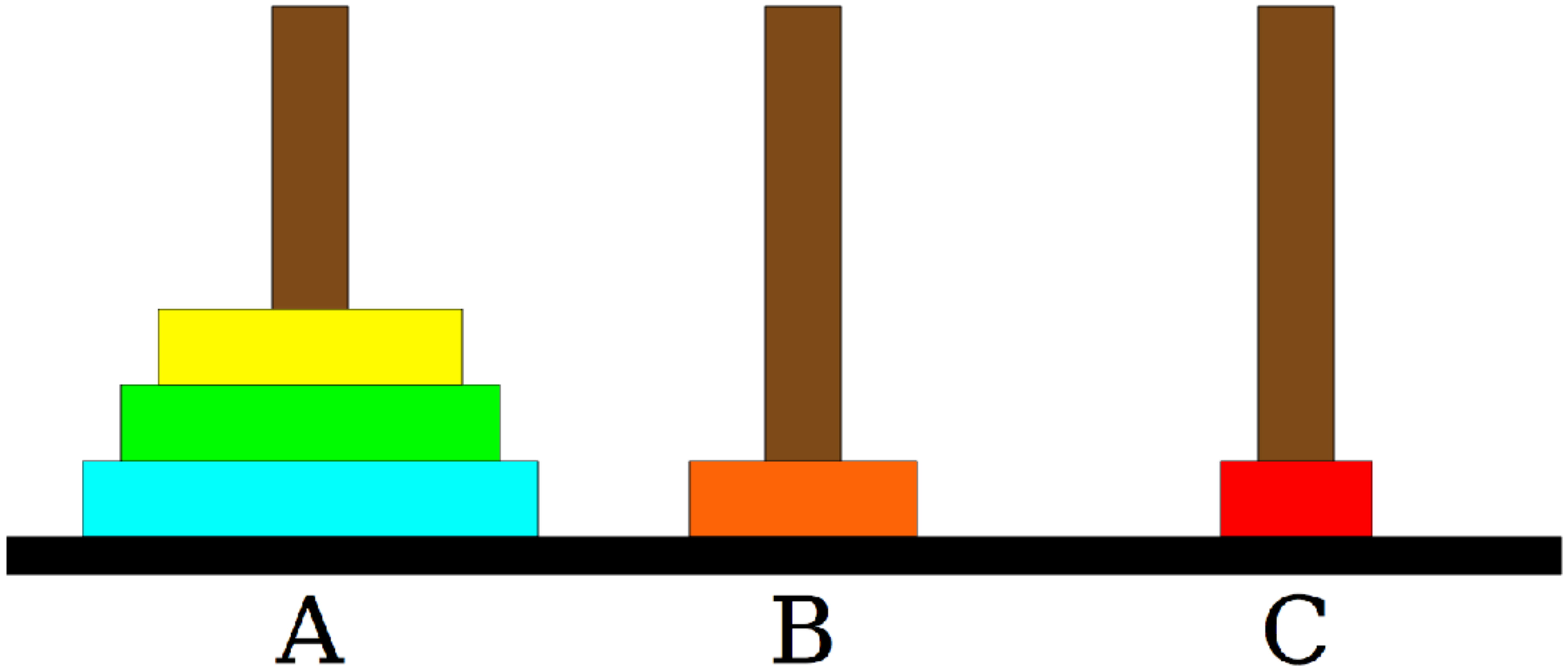
Here is the way the game is played:



Illegal move!

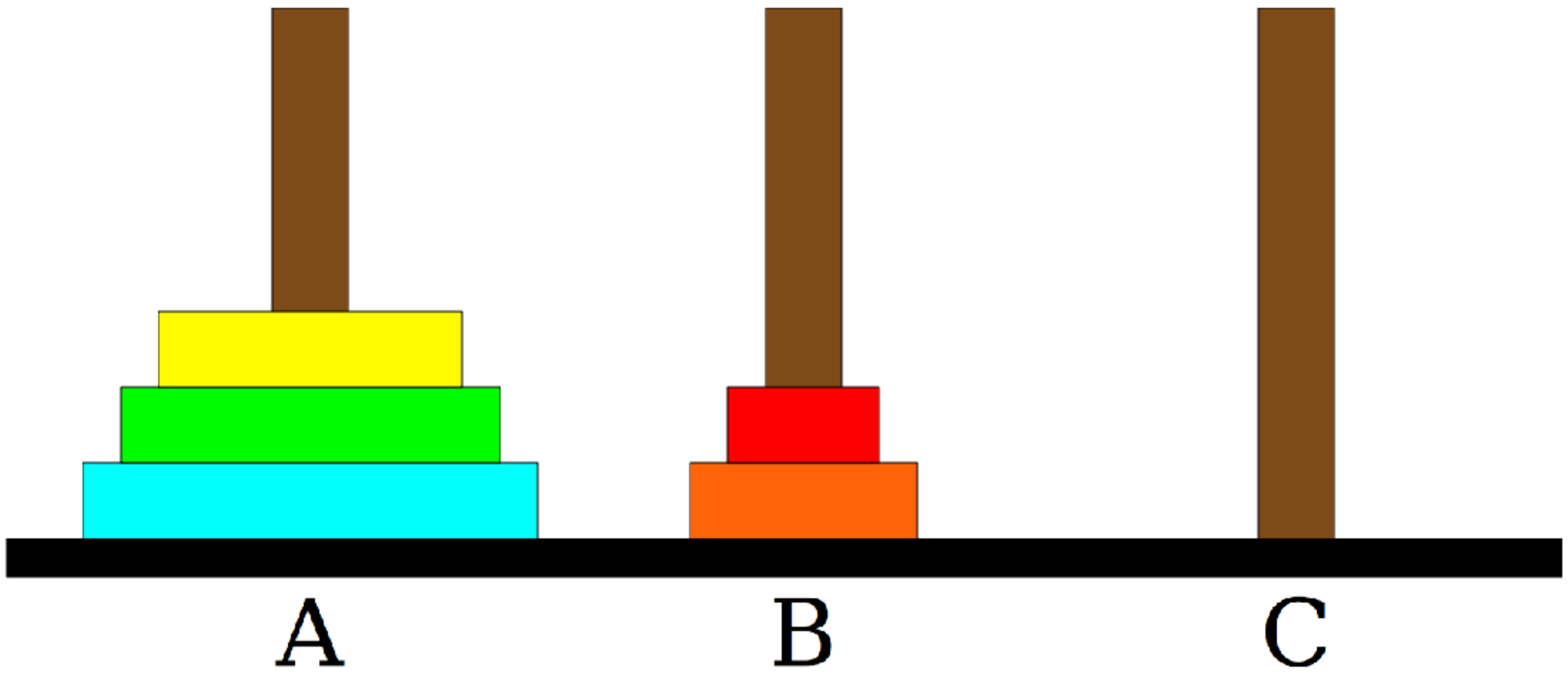
Towers of Hanoi

Here is the way the game is played:



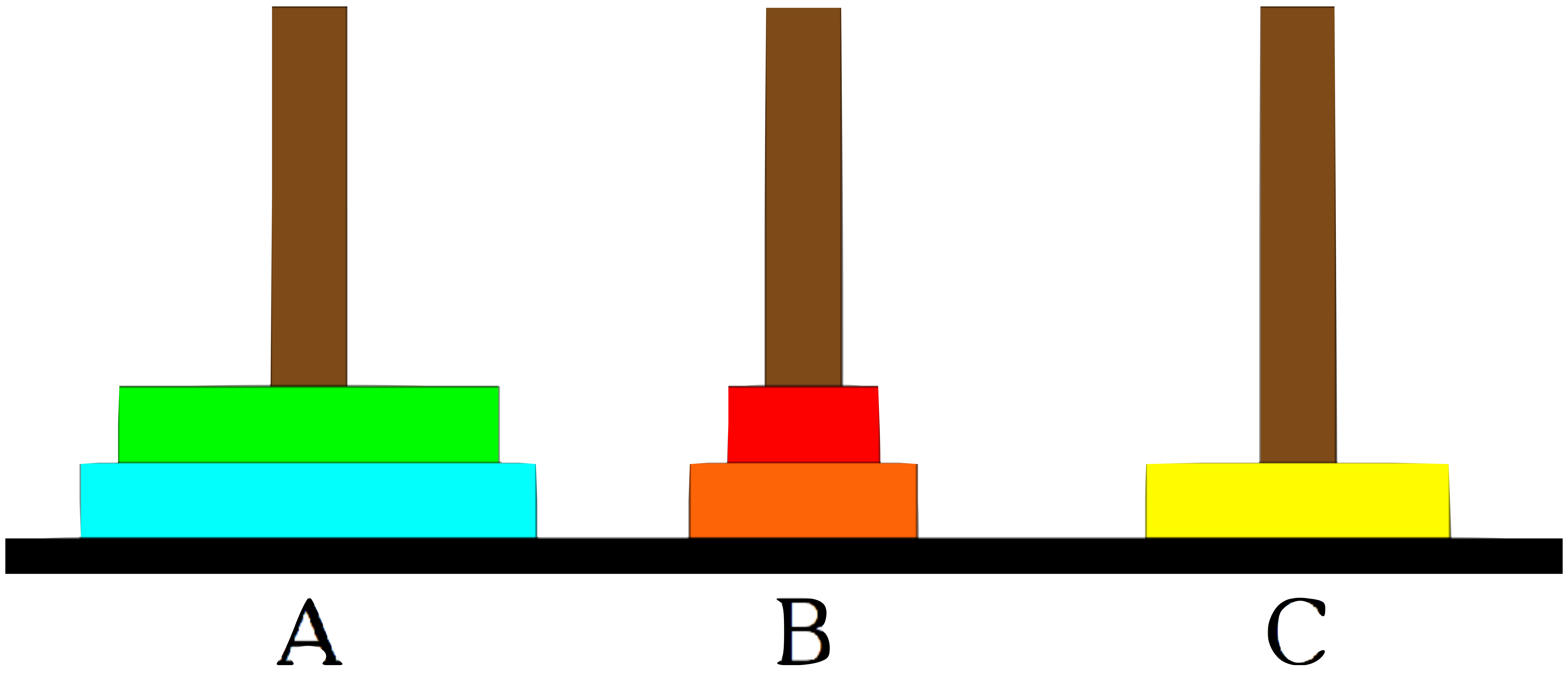
Towers of Hanoi

Here is the way the game is played:



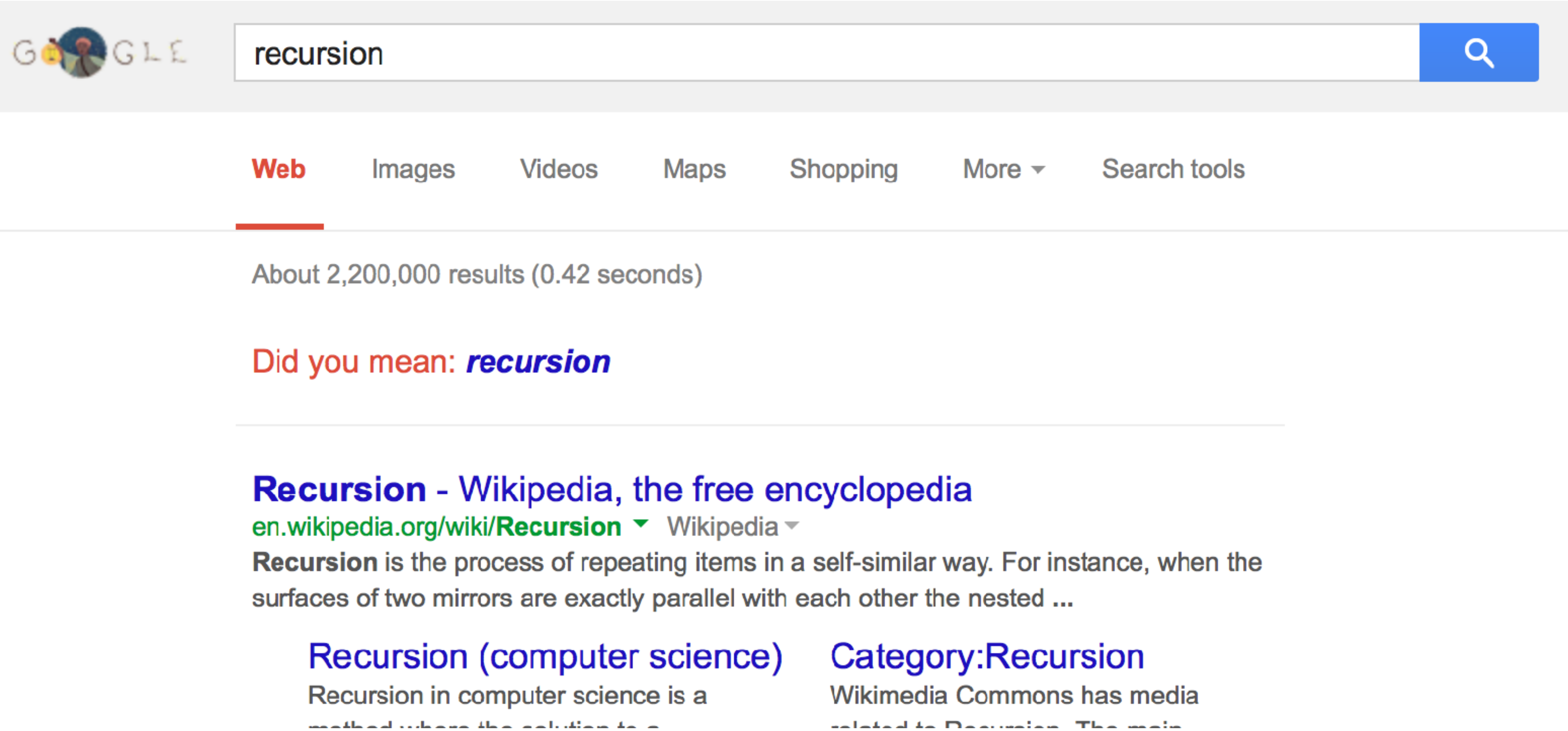
Towers of Hanoi

Here is the way the game is played:



etc.

What is Recursion?



What is Recursion?

Recursion:

A problem solving technique in which problems are solved by reducing them to smaller problems *of the same form.*

Why Recursion?

1. Great style
2. Powerful tool
3. Master of control flow

Pedagogy

Many simple examples

Recursion In Programming

In programming, recursion simply means that a function will call itself:

```
int main() {
    main();
    return 0;
}
```

SEG FAULT!

(this is a terrible example, and will crash!)

main() isn't supposed to call itself, but if we do write this program, what happens?

We'll get back to programming in a minute...

Recursion In Real Life

Recursion

- How to solve a jigsaw puzzle recursively ("solve the puzzle")
  - Is the puzzle finished? If so, stop.
  - Find a correct puzzle piece and place it.
  - Solve the puzzle

ridiculously hard puzzle

Recursion In Real Life

Let's recurse on *you*.

How many students total are directly behind you in your "column" of the classroom?

Rules:

1. You can see only the people directly in front and behind you. So, you can't just look back and count.
2. You *are* allowed to ask questions of the people in front / behind you.

How can we solve this problem *recursively*?

Recursion In Real Life

Answer:

1. The first person looks behind them, and sees if there is a person there. If not, the person responds "0".
2. If there is a person, repeat step 1, and wait for a response.
3. Once a person receives a response, they add 1 for the person behind them, and they respond to the person that asked them.

In C++:

```
int numStudentsBehind(Student curr) {
    if (noOneBehind(curr)) {
        return 0;
    } else {
        Student personBehind = curr.getBehind();
        return numStudentsBehind(personBehind) + 1
    }
}
```

Recursive call!

In C++:

The structure of recursive functions is typically like the following:

```
recursiveFunction() {
    if (test for simple case) {
        Compute the solution without recursion
    } else {
        Break the problem into subproblems of the same form
        Call recursiveFunction() on each subproblem
        Reassamble the results of the subproblems
    }
}
```

In C++:

Every recursive algorithm involves at least **two** cases:

- base case: The simple case; an occurrence that can be answered directly; the case that recursive calls reduce to.

- recursive case: a more complex occurrence of the problem that cannot be directly answered, but can be described in terms of smaller occurrences of the same problem.

In C++:

```
int numStudentsBehind(Student curr) {
    if (noOneBehind(curr)) {
        return 0;
    } else {
        Student personBehind = curr.getBehind();
        return numStudentsBehind(personBehind) + 1
    }
}
```

Base case

In C++:

```
int numStudentsBehind(Student curr) {
    if (noOneBehind(curr)) {
        return 0;
    } else {
        Student personBehind = curr.getBehind();
        return numStudentsBehind(personBehind) + 1
    }
}
```

Base case

Recursive case

In C++:

```
int numStudentsBehind(Student curr) {
    if (noOneBehind(curr)) {
        return 0;
    } else {
        Student personBehind = curr.getBehind();
        return numStudentsBehind(personBehind) + 1
    }
}
```

Recursive call

Three Musts of Recursion 🔑

1. Your code must have a case for all valid inputs

2. You must have a base case that makes no recursive calls

3. When you make a recursive call it should be to a simpler instance and make forward progress towards the base case.

There is a "recursive leap of faith"

More Examples!

The power() function:

Write a recursive function that takes in a number (*x*) and an exponent (*n*) and returns the result of x^n

Powers

$$x^0 = 1$$

$$x^n = x \cdot x^{n-1}$$

Powers

- Let's code it

Powers

- Each previous call waits for the next call to finish (just like any function).

```
cout << power(5, 3) << endl;
```

```
// first call: power (5, 3)
in
}
```

```
// second call: power (5, 2)
in
}
```

```
// third call: power (5, 1)
int
}
```

```
// fourth call: power (5, 0)
int power(int x, int exp) {
    if (exp == 0) {
        return 1;
    } else {
        return x * power(x, exp - 1);
    }
}
```

Powers

- Each previous call waits for the next call to finish (just like any function).

```
cout << power(5, 3) << endl;
```

```
// first call: power (5, 3)
in
}
```

```
// second call: power (5, 2)
in
}
```

```
// third call: power (5, 1)
int
}
```

```
// fourth call: power (5, 0)
int power(int x, int exp) {
    if (exp == 0) {
        return 1;
    } else {
        return x * power(x, exp - 1);
    }
}
```

This call returns 1

Powers

- Each previous call waits for the next call to finish (just like any function).

```
cout << power(5, 3) << endl;
```

```
// first call: power (5, 3)
in
}
```

```
// second call: power (5, 2)
in
}
```

```
// third call: power (5, 1)
int power(int x, int exp) {
    if (exp == 0) {
        return 1;
    } else {
        return x * power(x, exp - 1);
    }
}
```

equals 1 from call

this entire statement returns 5 * 1

Powers

- Each previous call waits for the next call to finish (just like any function).

```
cout << power(5, 3) << endl;
```

```
// first call: power (5, 3)
in
}
```

```
// second call: power (5, 2)
int power(int x, int exp) {
    if (exp == 0) {
        return 1;
    } else {
        return x * power(x, exp - 1);
    }
}
```

equals 5 from call

this entire statement returns 5 * 5

Powers

- Each previous call waits for the next call to finish (just like any function).

```
cout << power(5, 3) << endl;
```

```
// first call: power (5, 3)
int power(int x, int exp) {
    if (exp == 0) {
        return 1;
    } else {
        return x * power(x, exp - 1);
    }
}
```

equals 25 from call

this entire statement returns 5 * 25

the original function call was to this one, so it returns 125, which is 5^3

Faster Method!

```
int power(int x, int exp) {
    if(exp == 0) {
        // base case
        return 1;
    } else {
        if (exp % 2 == 1) {
        // if exp is odd
            return x * power(x, exp - 1);
        } else {
            // else, if exp is even
            int y = power(x, exp / 2);
            return y * y;
        }
    }
}
```

Exponentiation by squaring

Big O???

O(log n) -- yay!

Mystery Recursion: Trace this function

```
int mystery(int n) {
    if (n < 10) {
        return n;
    } else {
        int a = n/10;
        int b = n % 10;
        return mystery(a + b);
    }
}
```

What is the result of mystery(648)?

A. 8
B. 9
C. 54
D. 72
E. 648

Mystery Recursion: Trace this function

```
int mystery(int n) { // n = 648
    if (n < 10) {
        return n;
    } else {
        int a = n/10; // a = 64
        int b = n % 10; // b = 8
        return mystery(a + b); // mystery(72);
    }
}
```

Mystery Recursion: Trace this function

```
int mystery(int n) { // n = 648
}
```

```
int mystery(int n) { // n = 72
    if (n < 10) {
        return n;
    } else {
        int a = n/10; // a = 7
        int b = n % 10; // b = 2
        return mystery(a + b); // mystery(9);
    }
}
```

Mystery Recursion: Trace this function

```
int mystery(int n) { // n = 648
```

```
int mystery(int n) { // n = 72
```

```
int mystery(int n) { // n = 9
    if (n < 10) {
        return n; // return 9;
    } else {
        int a = n/10;
        int b = n % 10;
        return mystery(a + b);
    }
}
```

```
}
```

```
}
```

Mystery Recursion: Trace this function

```
int mystery(int n) { // n = 648
}
```

```
int mystery(int n) { // n = 72
    if (n < 10) {
        return n;
    } else {
        int a = n/10; // a = 7
        int b = n % 10; // b = 2
        return mystery(a + b); // mystery(9);
    }
}
```

returns 9

Mystery Recursion: Trace this function

```
int mystery(int n) { // n = 648
    if (n < 10) {
        return n;
    } else {
        int a = n/10; // a = 64
        int b = n % 10; // b = 8
        return mystery(a + b); //
    }
}
```

returns 9

What is the result of mystery(648)?

A. 8
B. 9
C. 54
D. 72
E. 648

More Examples! isPalendrome(string s)

Write a recursive function isPalindrome accepts a string and returns true if it reads the same forwards as backwards.

```
isPalindrome("madam") → true
isPalindrome("racecar") → true
isPalindrome("step on no pets") → true
isPalindrome("Java") → false
isPalindrome("byebye") →false
```

Three Musts of Recursion 🔑

1. Your code must have a case for all valid inputs

2. You must have a base case that makes no recursive calls

3. When you make a recursive call it should be to a simpler instance and make forward progress towards the base case.

isPalendrome

```
// Returns true if the given string reads the same
// forwards as backwards.
// Trivially true for empty or 1-letter strings.
bool isPalindrome(const string& s) {
    if (s.length() < 2) { // base case
        return true;
    } else { // recursive case
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        string middle = s.substr(1, s.length() - 2);
        return isPalindrome(middle);
    }
}
```

Flashback to 106A: Hailstone

```
// Couts the sequence of numbers from n to one
// produced by the Hailstone (aka Collatz) procedure
void hailstone(int n) {
    cout << n << endl;
    if(n == 1) {
        return;
    } else {
        if(n % 2 == 0) {
            // n is even so we repeat with n/2
            hailstone(n / 2);
        } else {
            // n is odd so we repeat with 3 * n + 1
            hailstone(3 * n + 1);
        }
    }
}
```

Flashback to 106A: Hailstone

```
// Couts the sequence of numbers from n to one
// produced by the Hailstone (aka Collatz) procedure
void hailstone(int n) {
    cout << n << endl;
    if(n == 1) {
        return;
```

3. When you make a recursive call it should be to a simpler instance and make forward progress towards the base case.

```
            // n is odd so we repeat with 3 * n + 1
            hailstone(3 * n + 1);
        }
    }
}
```

Is this simpler???

Flashback to 106A: Hailstone

```
hailstone(int n)
```

Hailstone has been checked for values up to 5×10^{18}

but no one has proved that it always reaches 1!

There is a cash prize for proving it!

The prize is $1400.

Flashback to 106A: Hailstone

Print the sequences of numbers that you take to get from N until 1, using the Hailstone (Collatz) production rules:

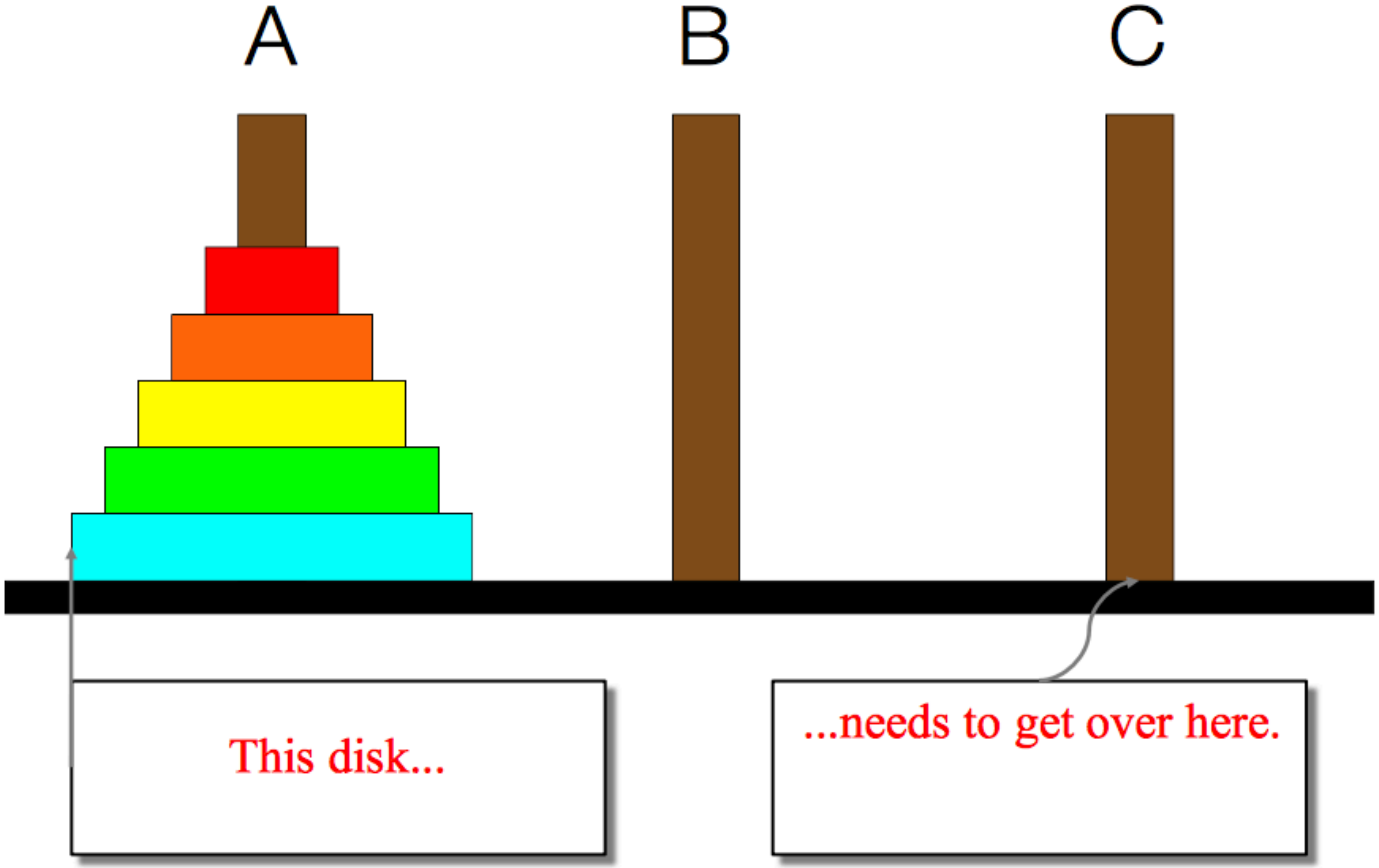
If n == 1, you are done.

If n is even your next number is n / 2.

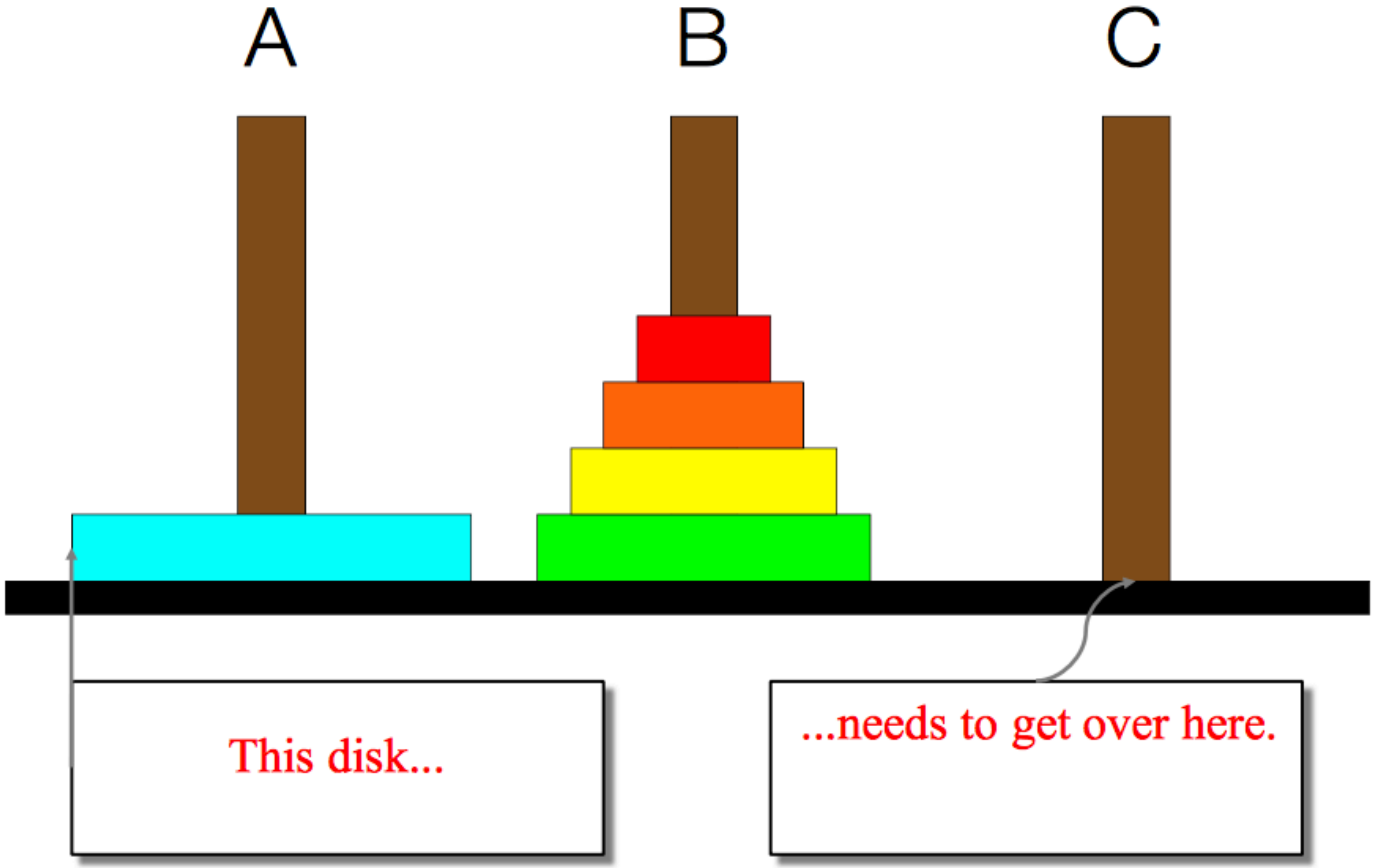
If n is odd your next number is 3*n + 1.

Back to Towers of Hanoi

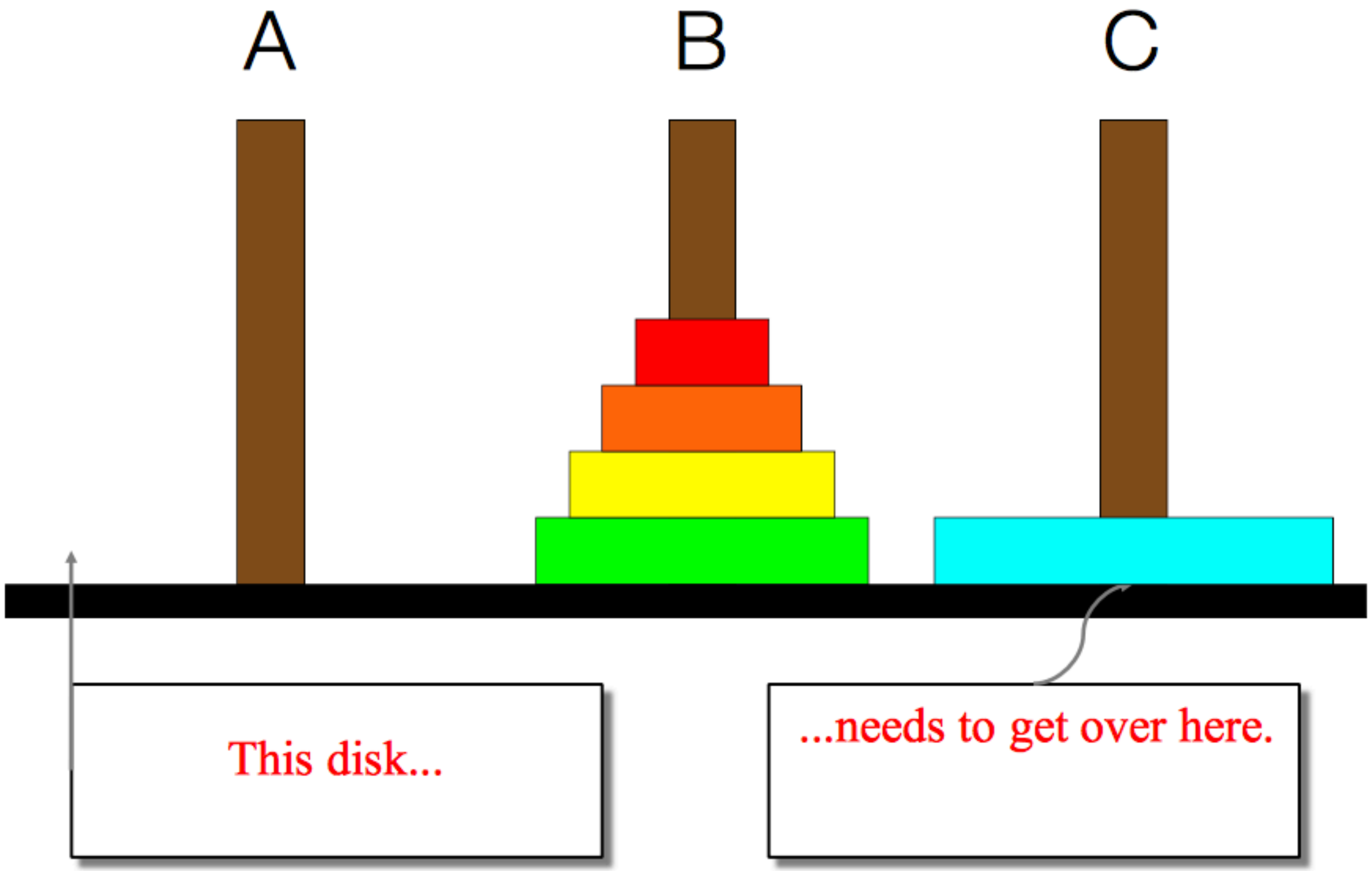
This is a hard problem to solve iteratively, but can be done recursively (though the recursive insight is not trivial to figure out)



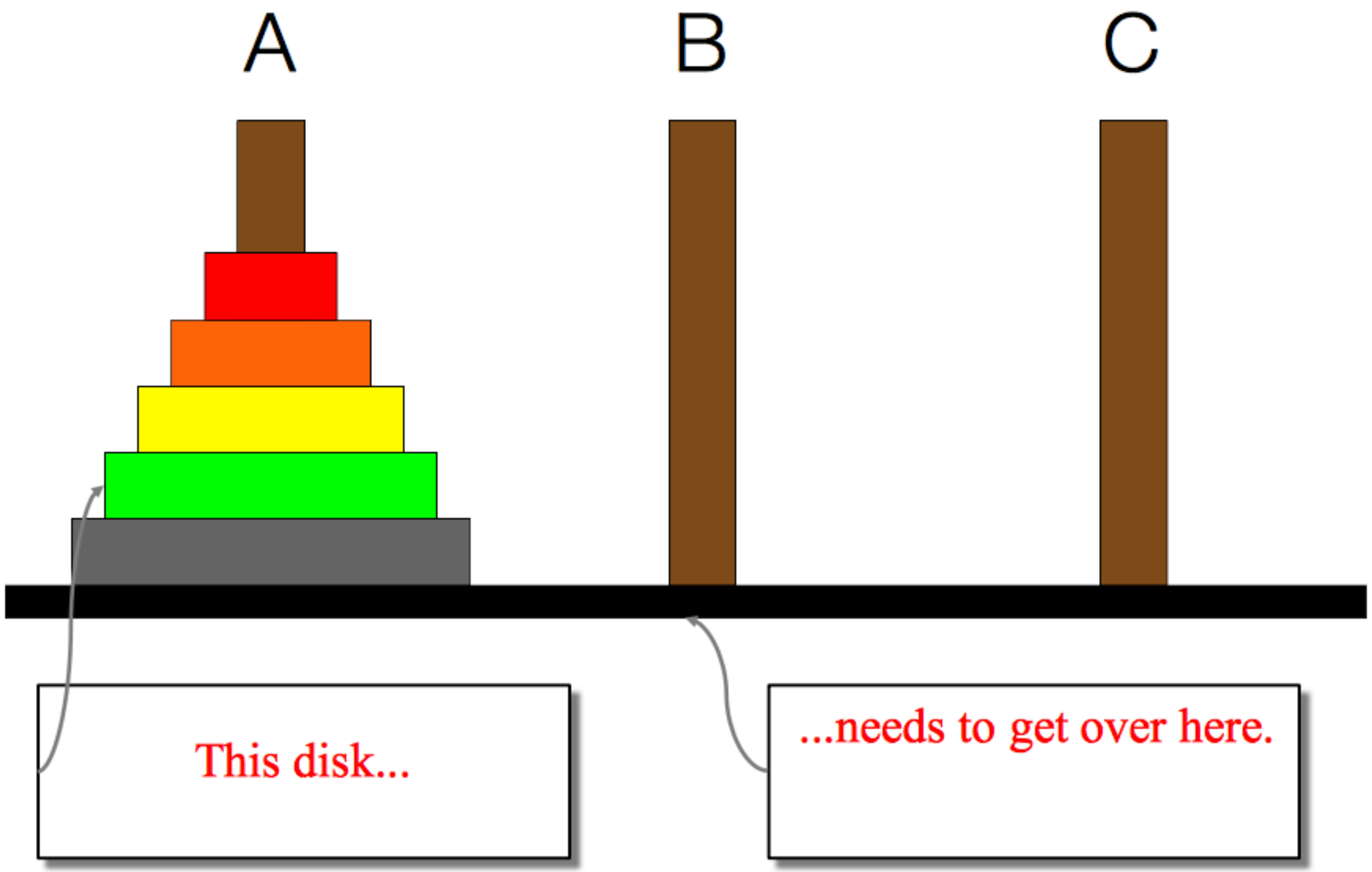
Back to Towers of Hanoi



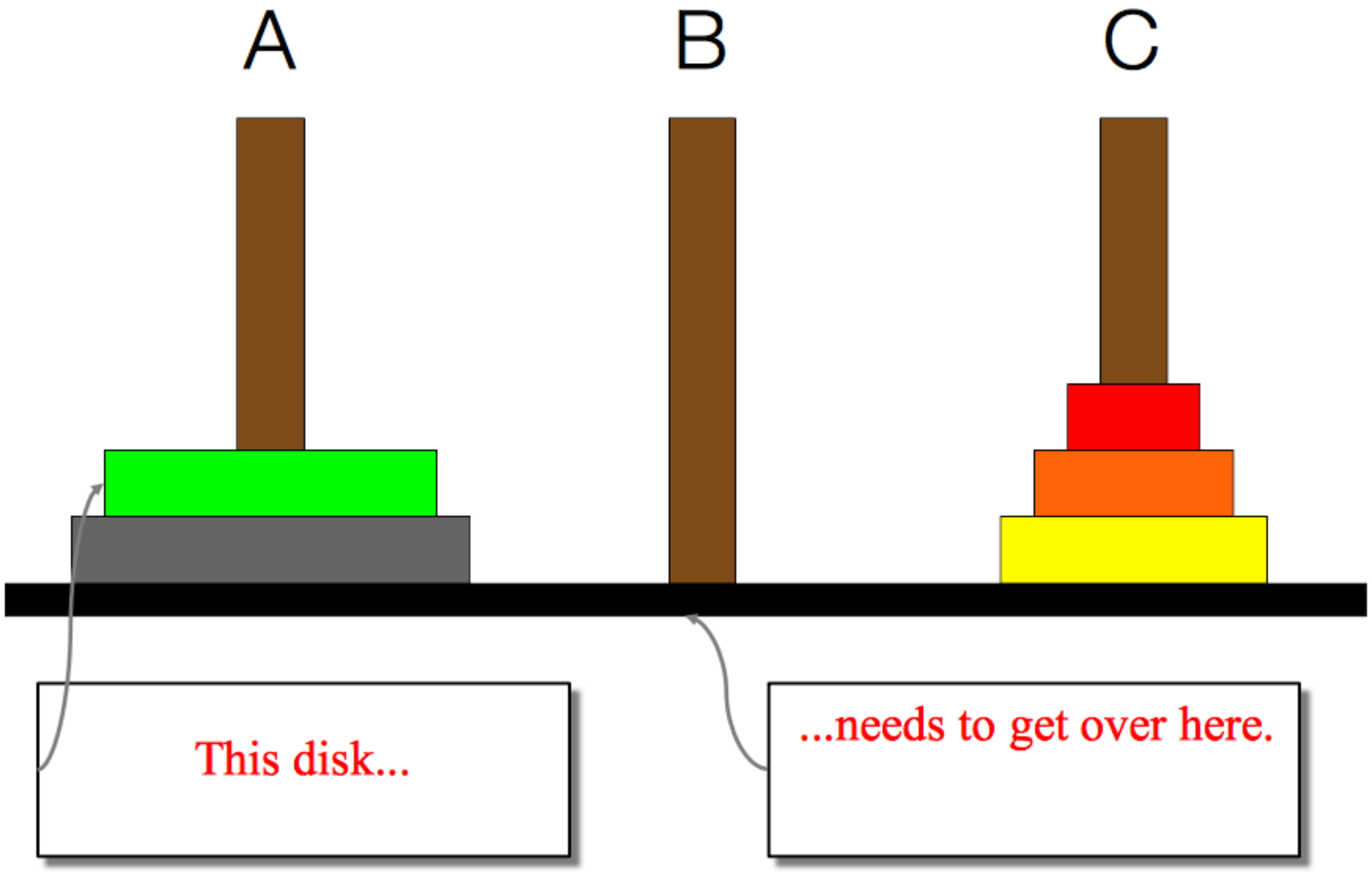
Back to Towers of Hanoi



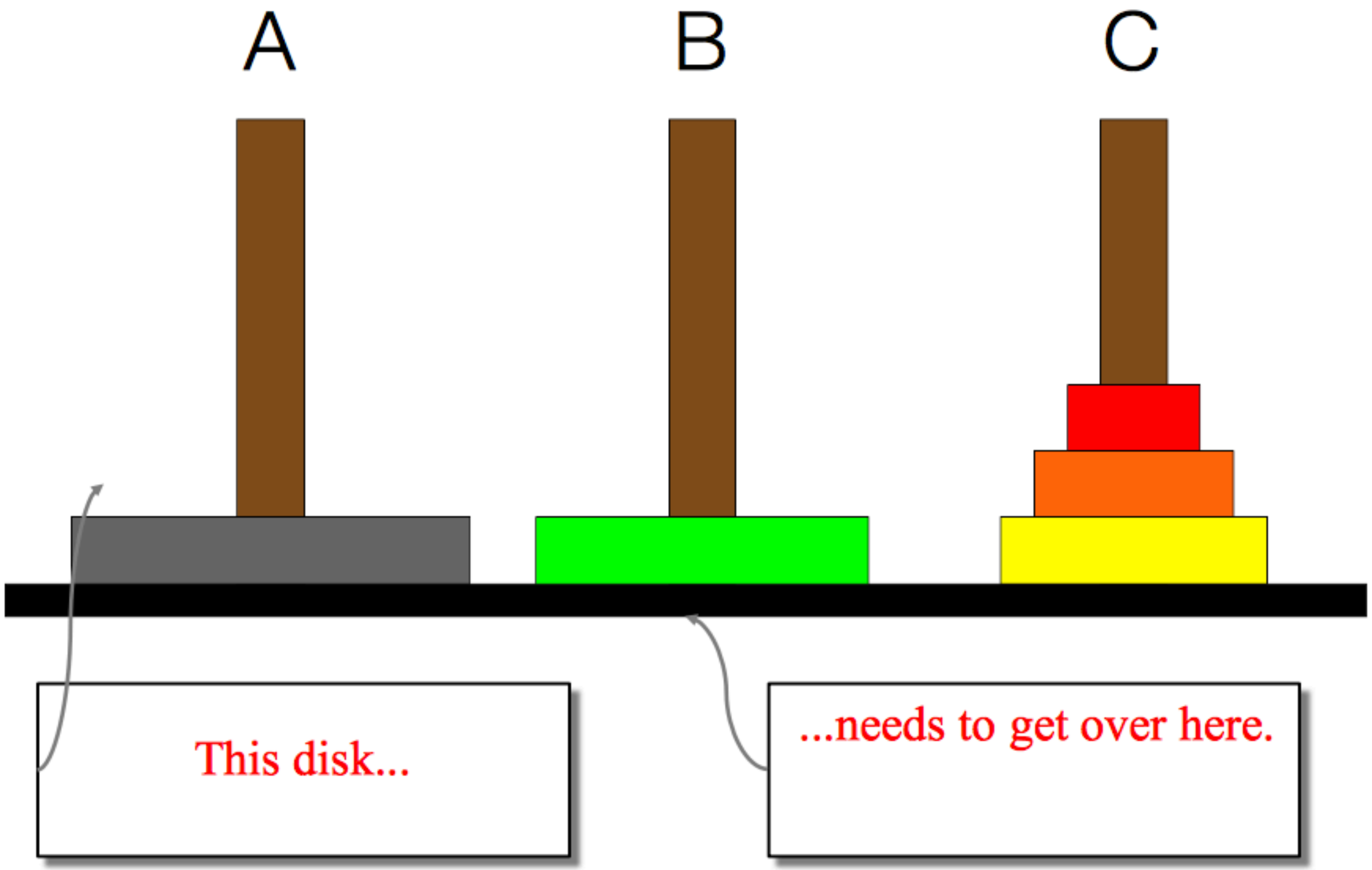
Back to Towers of Hanoi



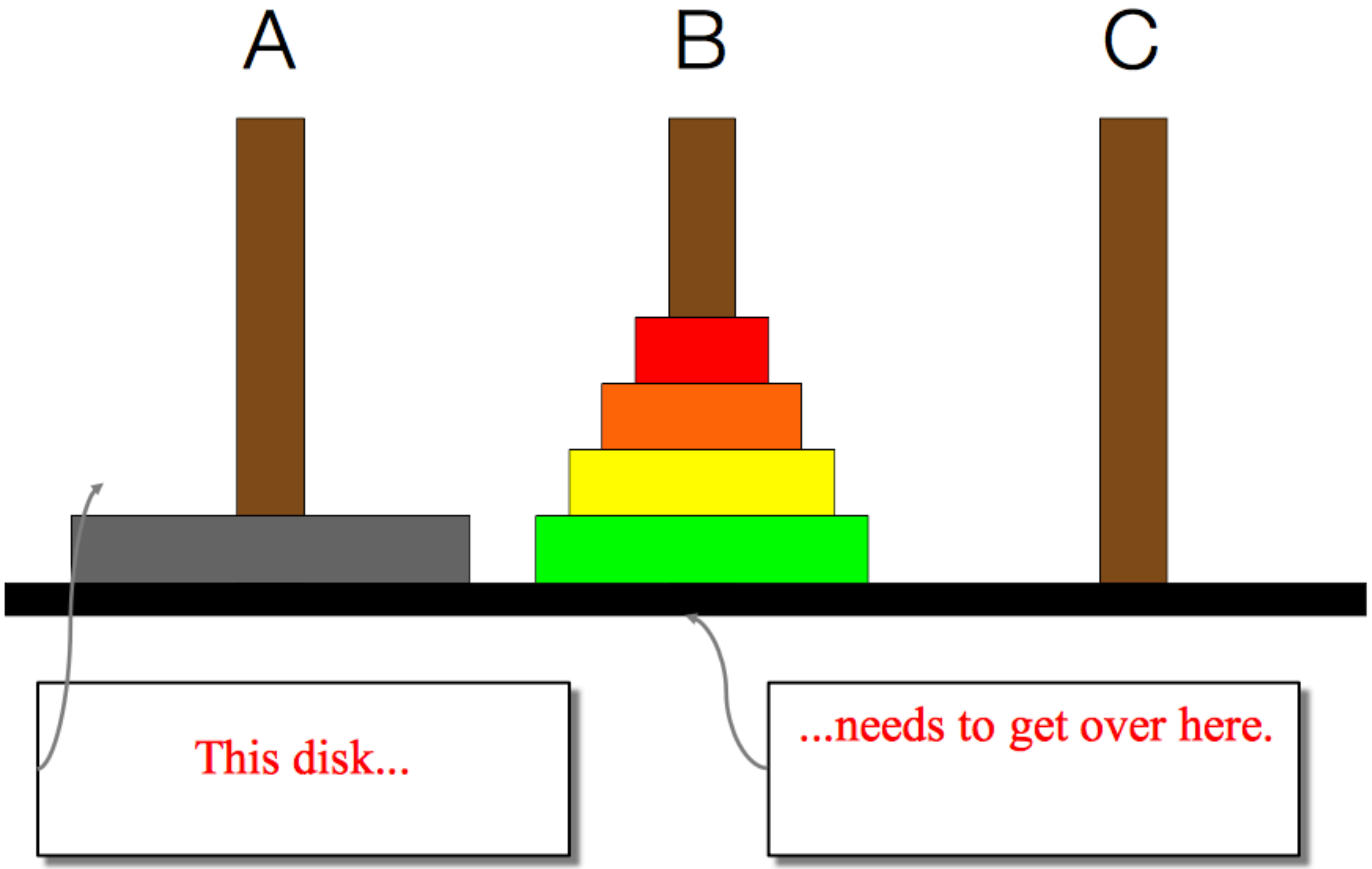
Back to Towers of Hanoi



Back to Towers of Hanoi



Back to Towers of Hanoi



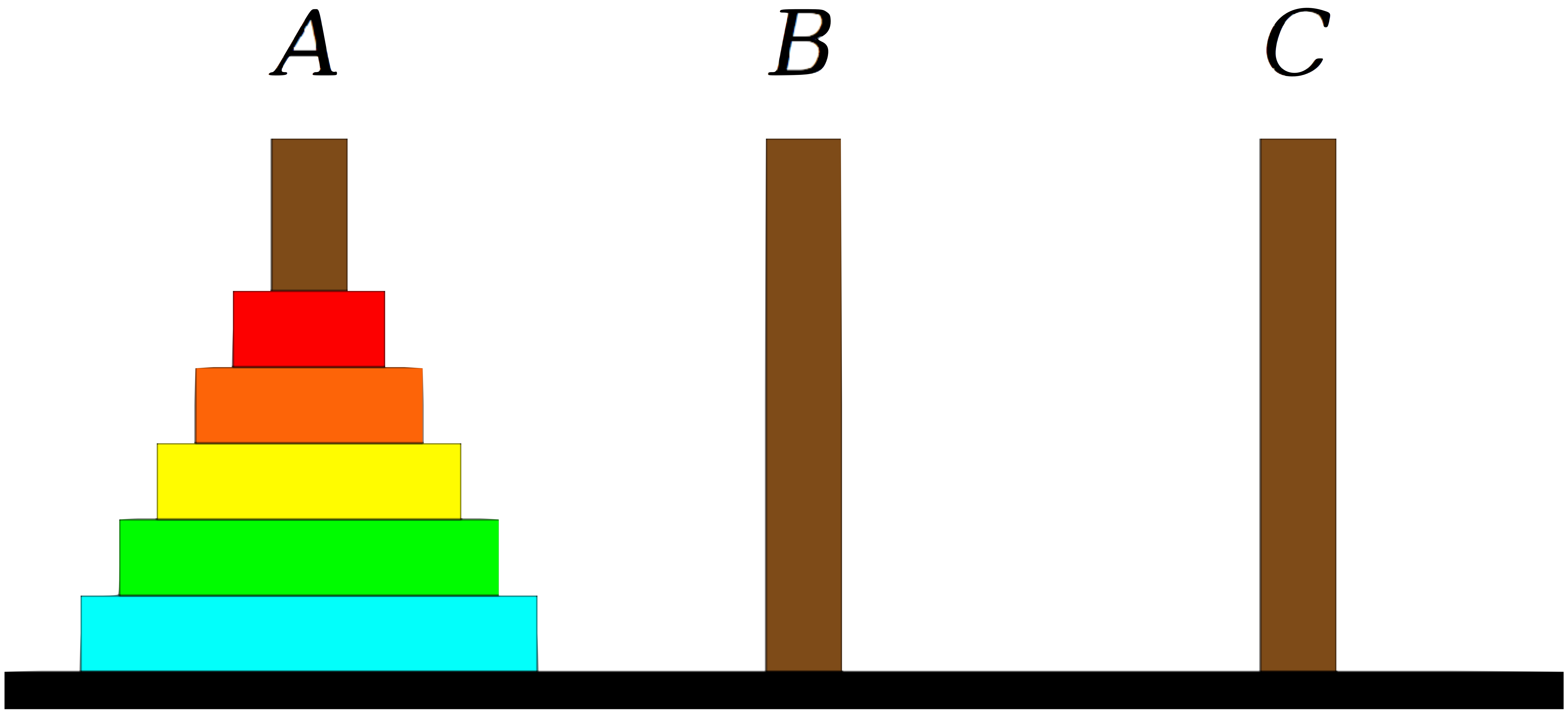
Back to Towers of Hanoi

- We need to find a very simple case that we can solve directly in order for the recursion to work.
- If the tower has size one, we can just move that single disk from the source to the destination.
- If the tower has more than one, we have to use the auxiliary spindle.

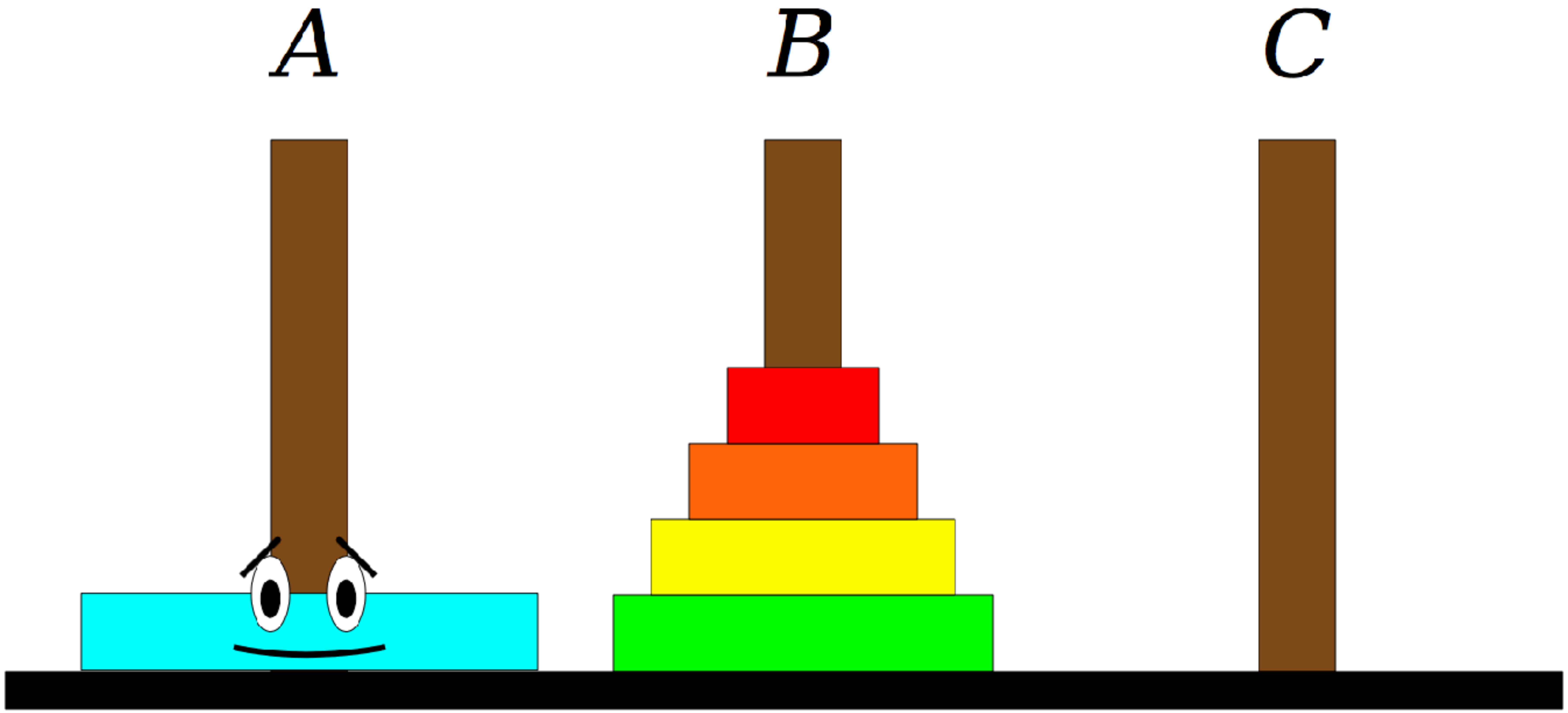
Back to Towers of Hanoi

- We can break the entire process down into very simple steps -- not necessarily easy to think of steps, but simple ones!

Back to Towers of Hanoi

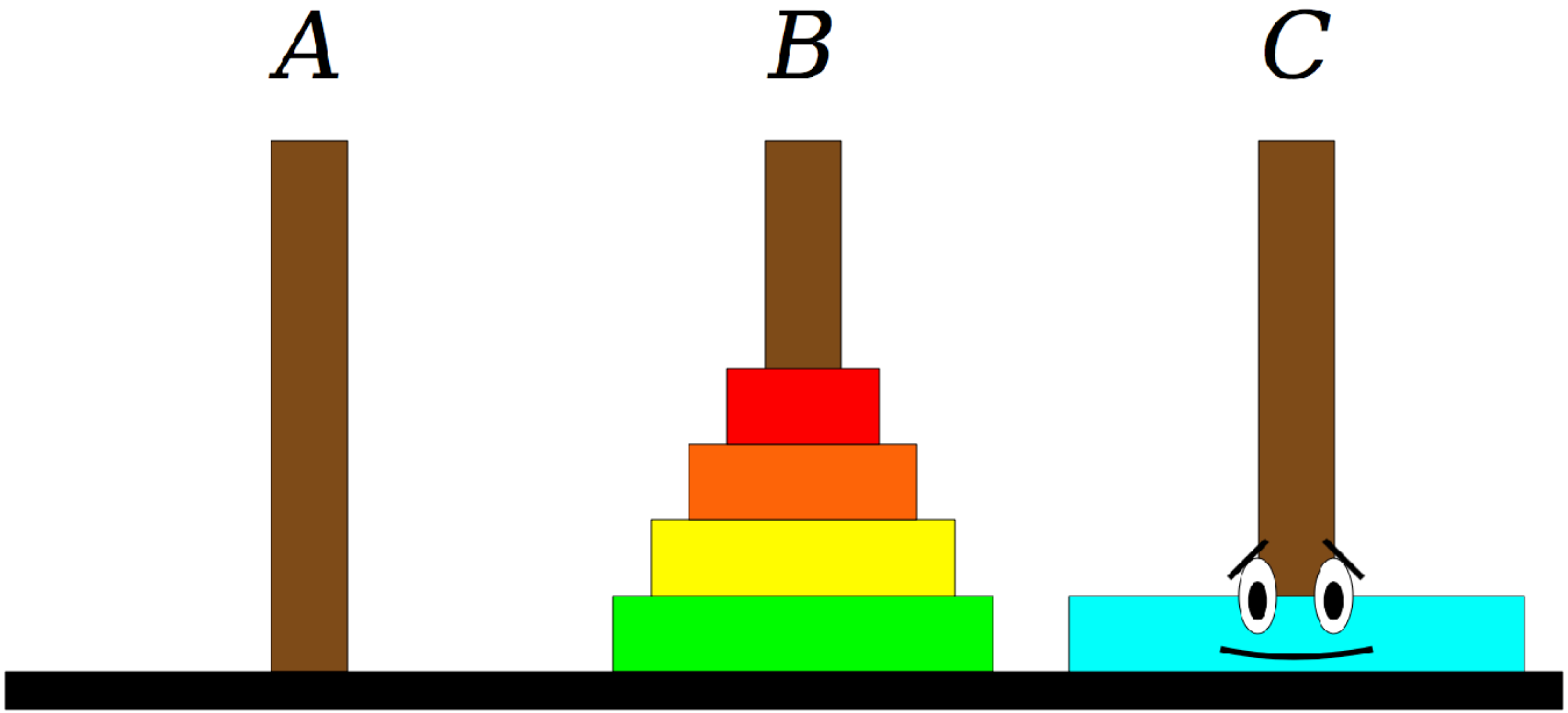


Back to Towers of Hanoi



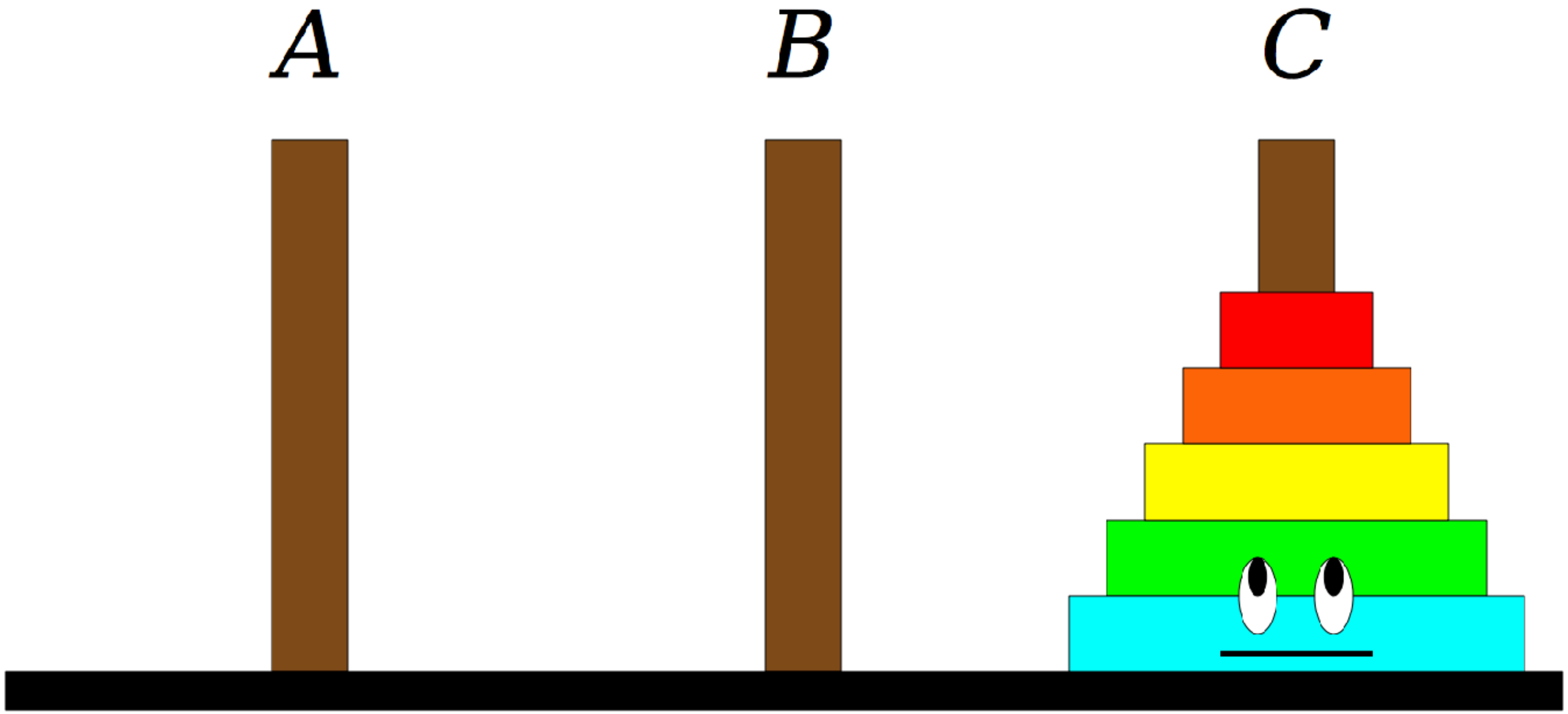
Step One: Move the four smaller disks from Spindle A to Spindle B.

Back to Towers of Hanoi



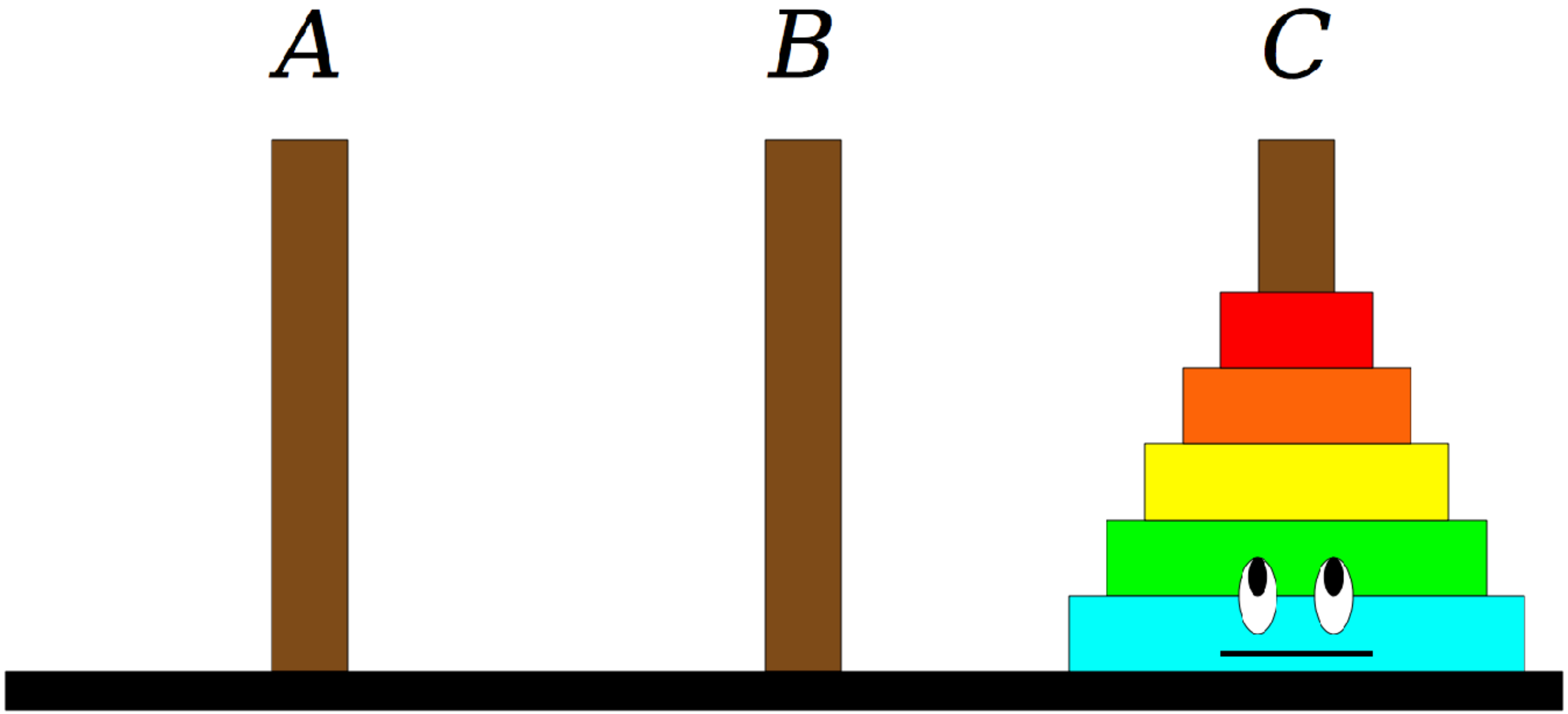
Step One: Move the four smaller disks from Spindle *A* to Spindle *B*.
Step Two: Move the blue disk from Spindle *A* to Spindle *C*.

Back to Towers of Hanoi



Step One: Move the four smaller disks from Spindle A to Spindle B.
Step Two: Move the blue disk from Spindle A to Spindle C.
Step Three: Move the four smaller disks from Spindle B to Spindle C.

Back to Towers of Hanoi



Step One: Move the four smaller disks from Spindle A to Spindle B.
Step Two: Move the blue disk from Spindle A to Spindle C.
Step Three: Move the four smaller disks from Spindle B to Spindle C.

Repeat these steps at each stage!

Back to Towers of Hanoi

Recap

- **Recursion**
  - Break a problem into smaller subproblems of the same form, and call the same function again on that smaller form.
  - Super powerful programming tool
  - Not always the perfect choice, but often a good one
  - Some beautiful problems are solved recursively

- **Three Musts for Recursion:**
  1. Your code must have a case for all valid inputs
  2. You must have a base case that makes no recursive calls
  3. When you make a recursive call it should be to a simpler instance and make forward progress towards the base case.

References and Advanced Reading

- **References:**
  - http://www.cs.utah.edu/~germain/PPS/Topics/recursion.html
  - Why is iteration generally better than recursion? http://stackoverflow.com/a/3093/561677

- **Advanced Reading:**
  - Tail recursion: http://stackoverflow.com/questions/33923/what-is-tail-recursion
  - Interesting story on the history of recursion in programming languages: http://goo.gl/P6Einb

Extra Slides

Converting Decimal to Binary

Recursion is about solving a small piece of a large problem.

– What is 69743 in binary?
  - Do we know anything about its representation in binary?

– Case analysis:
  - What is/are easy numbers to print in binary?
  - Can we express a larger number in terms of a smaller number(s)?

Converting Decimal to Binary

Suppose we are examining some arbitrary integer N.

– if N's binary representation is 10010101011

– (N / 2)'s binary representation is 1001010101

– (N % 2)'s binary representation is 1

– What can we infer from this relationship?

Converting Decimal to Binary

```
// Prints the given integer's binary representation.
// Precondition: n >= 0
void printBinary(int n) {
    if (n < 2) {
        // base case; same as base 10
        cout << n;
    } else {
        // recursive case; break number apart
        printBinary(n / 2);
        printBinary(n % 2);
    }
}
```