

# Thinking Recursively

## Part II

# Announcements

- Assignment 2 due Wednesday
- Michael's OH changed this week to 3-5PM, Monday and Tuesday in Gates 160

Review from Yesterday..

# Recursive Problem-Solving

**if** (*problem is sufficiently simple*) {

*Directly solve the problem.*

*Return the solution.*

} **else** {

*Split the problem up into one or more smaller problems with the same structure as the original.*

*Solve each of those smaller problems.*

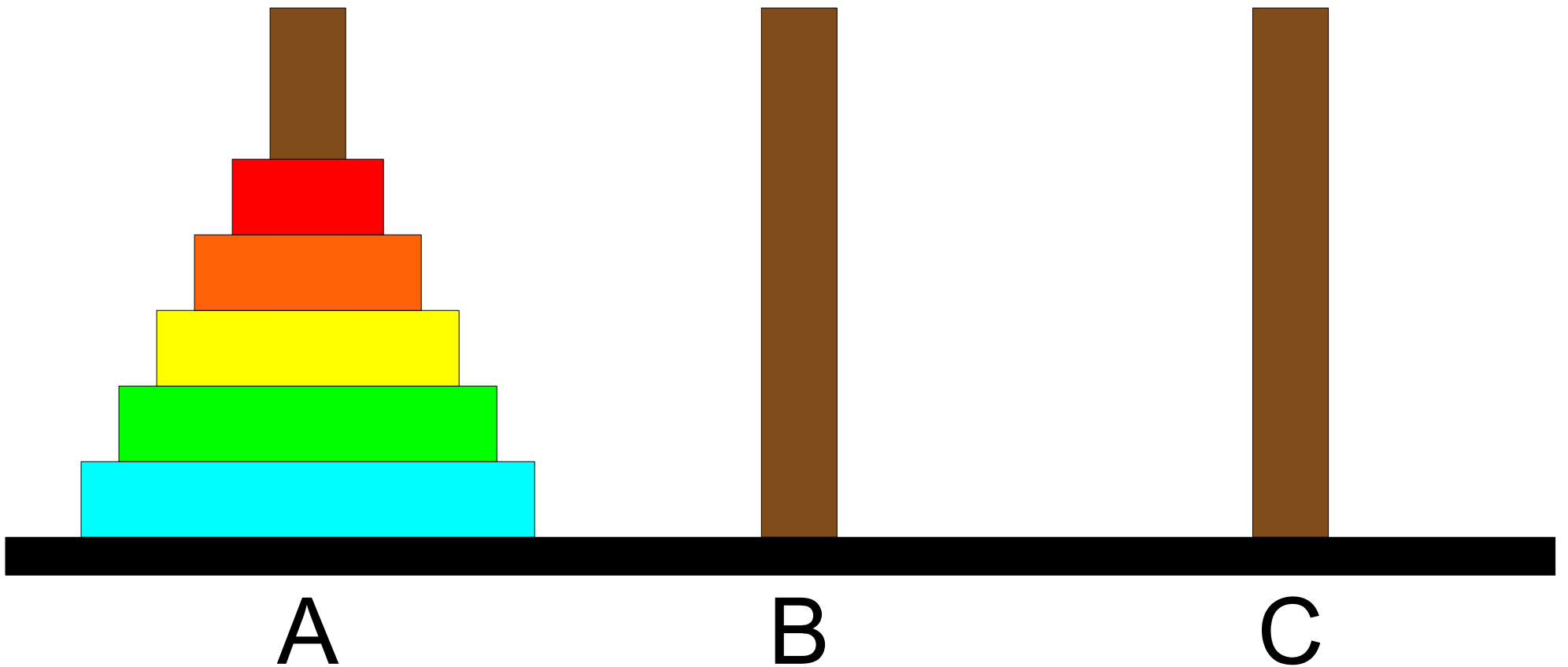
*Combine the results to get the overall solution.*

*Return the overall solution.*

}

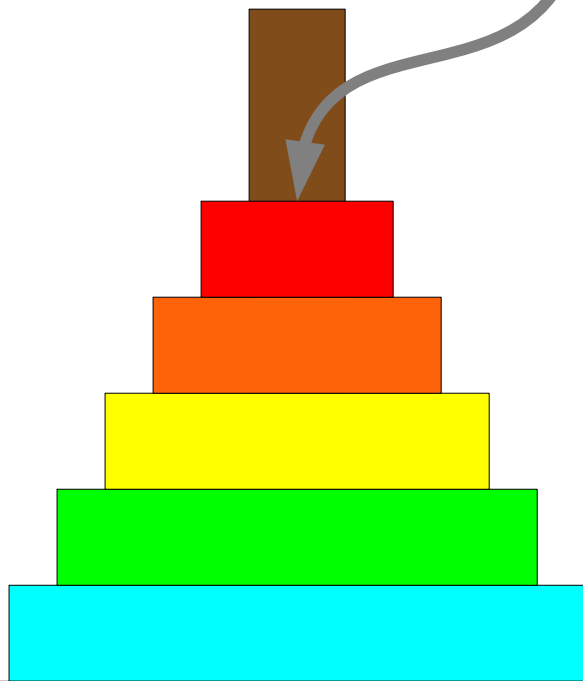
# Solving the Towers of Hanoi

# Towers of Hanoi



# Towers of Hanoi

Move this tower...



A

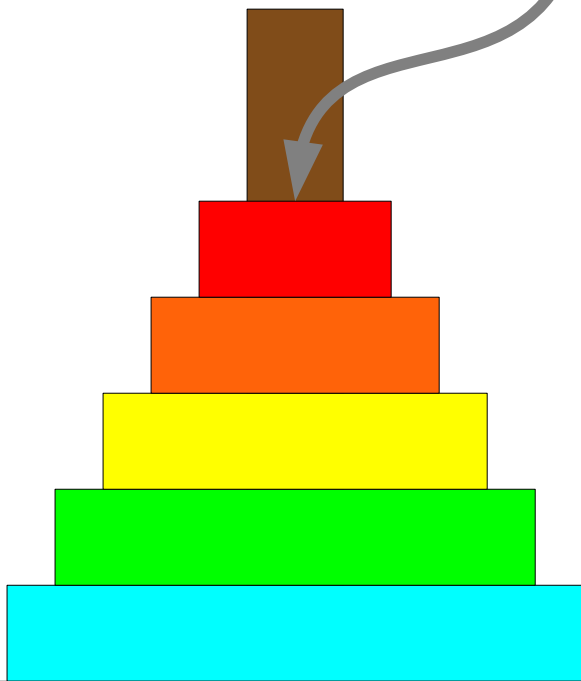
B

C

# Towers of Hanoi

Move this tower...

...to this spindle.



A



B



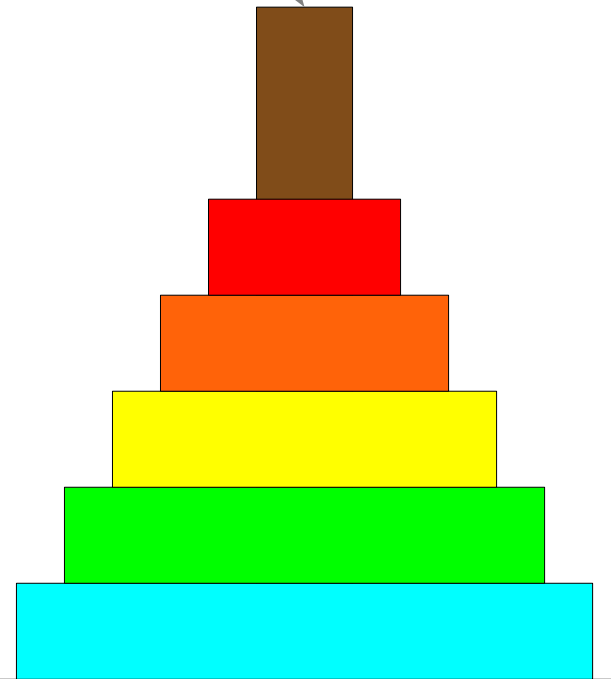
C



# Towers of Hanoi

Move this tower...

...to this spindle.

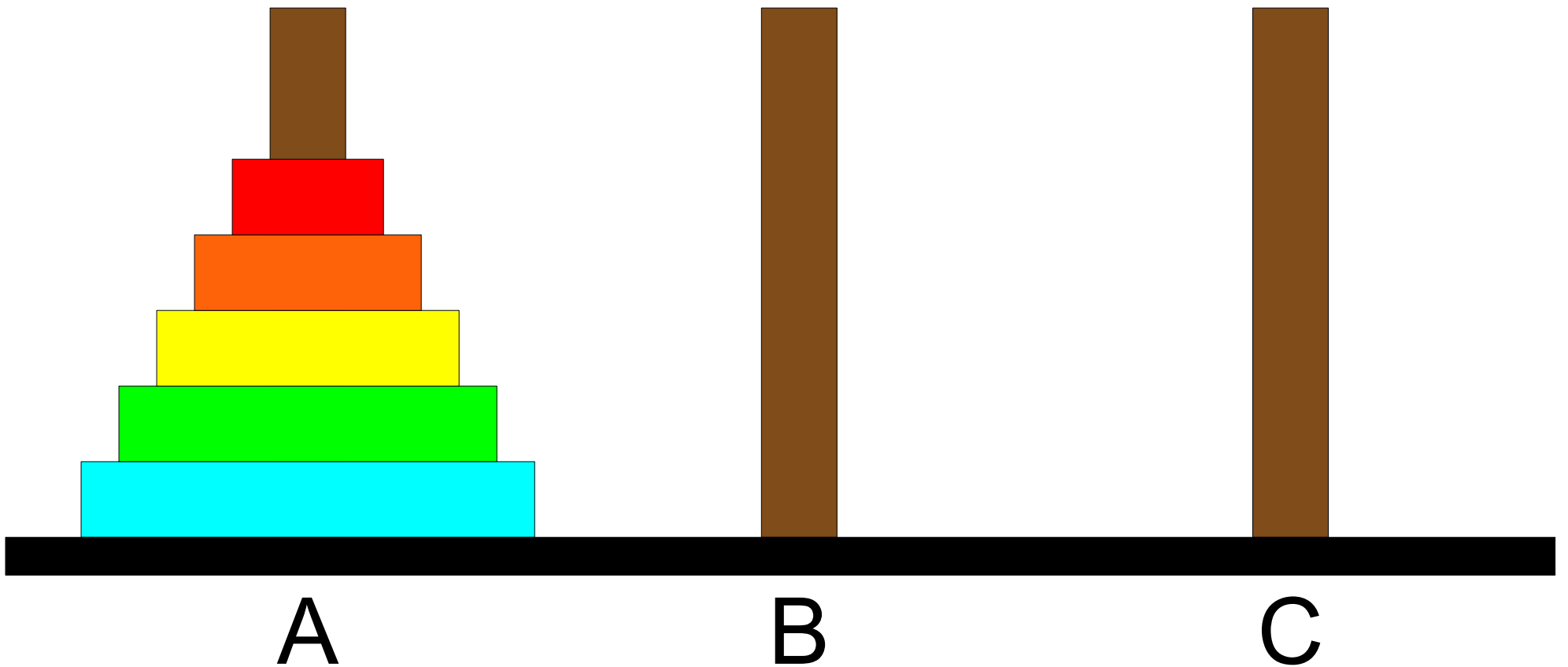


A

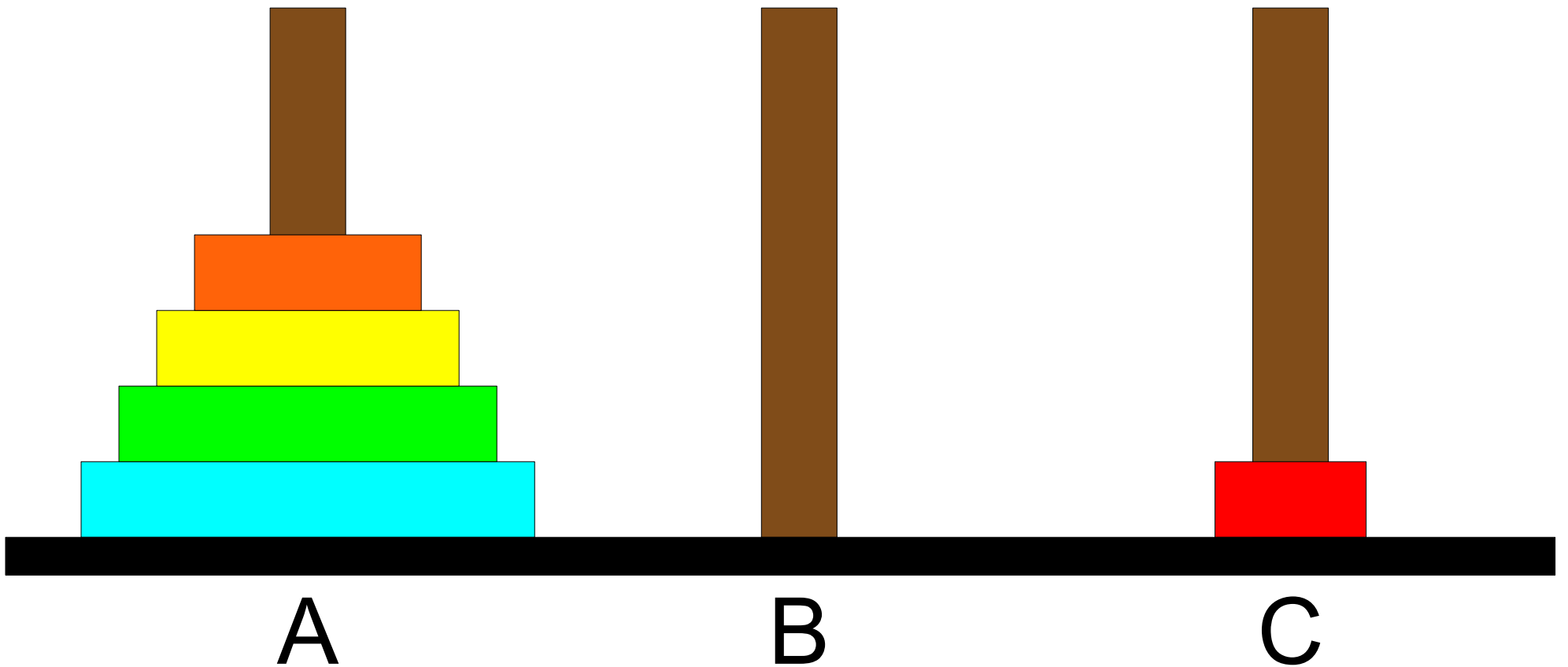
B

C

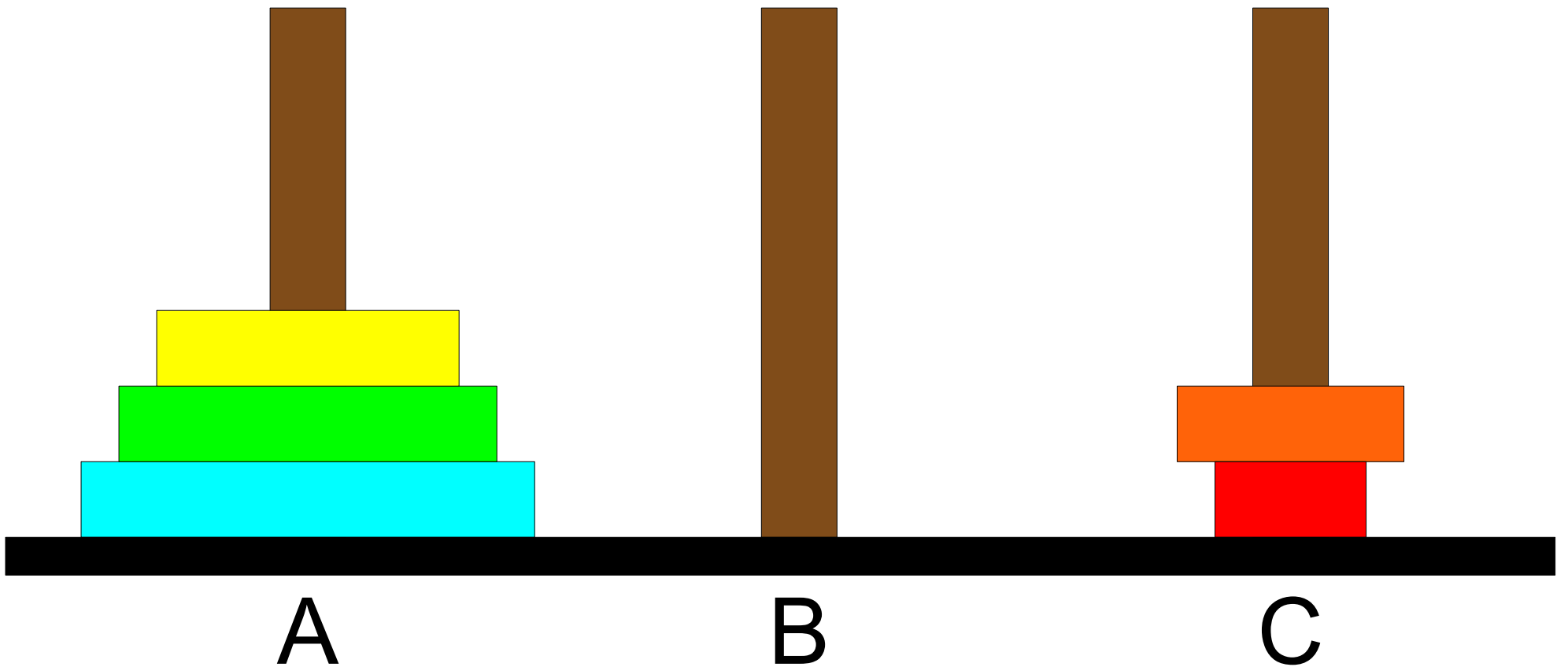
# Towers of Hanoi



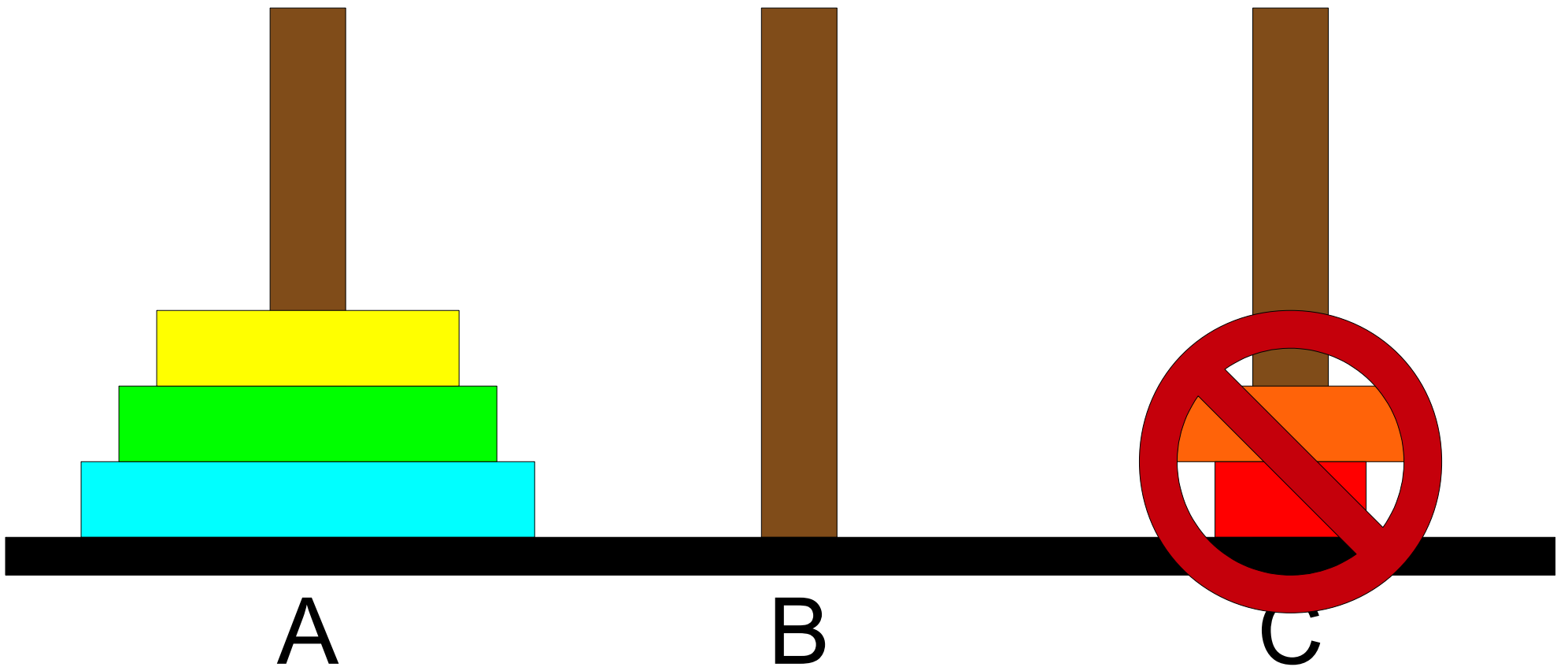
# Towers of Hanoi



# Towers of Hanoi



# Towers of Hanoi

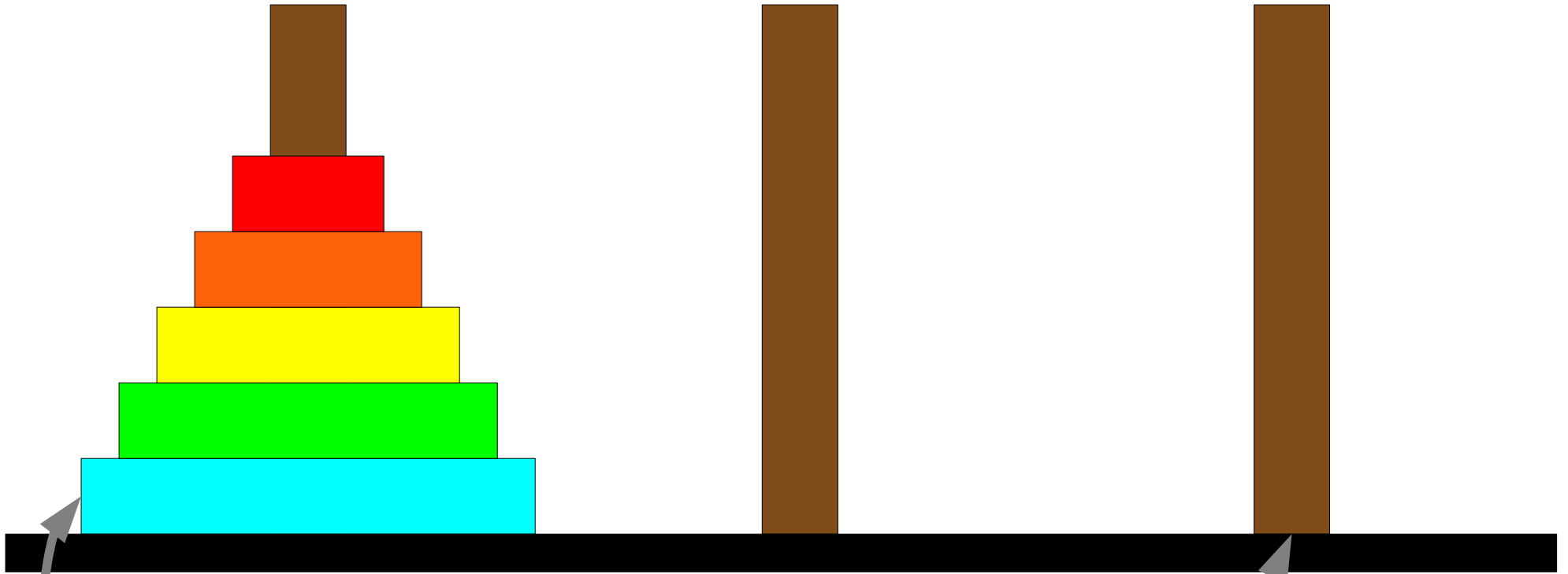


# Solving the Towers of Hanoi

A

B

C



This disk...

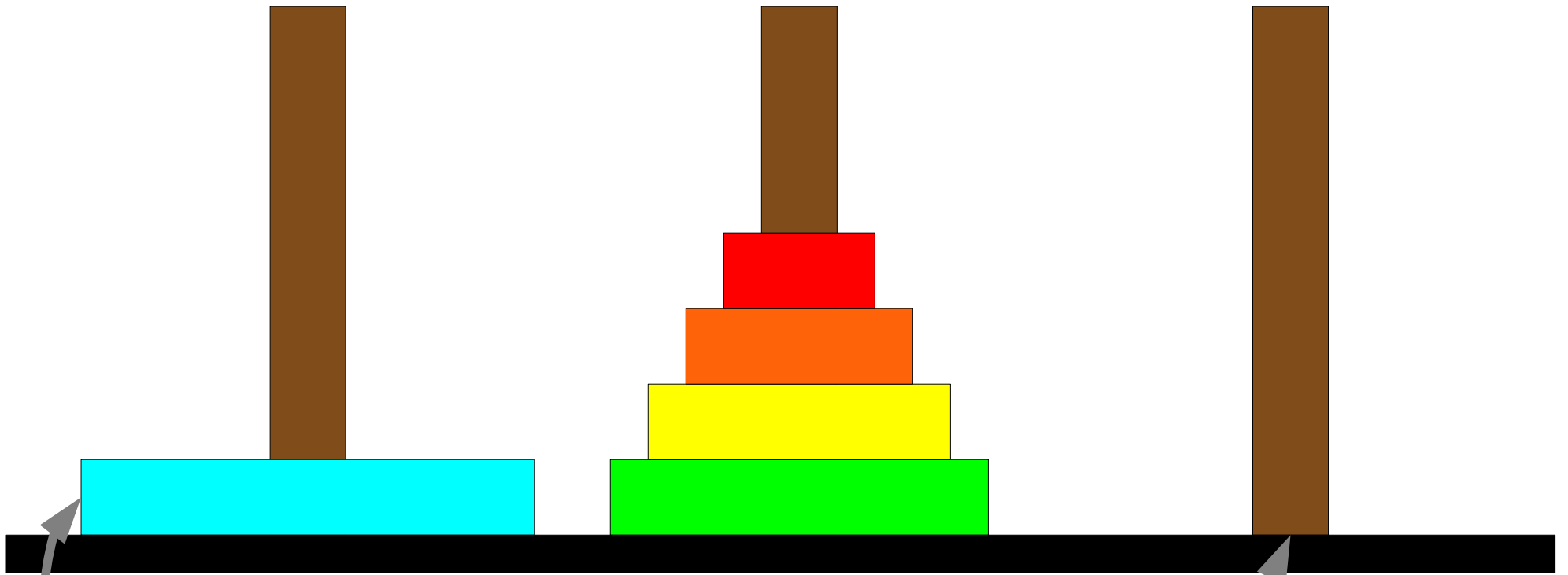
...needs to get over here.

# Solving the Towers of Hanoi

A

B

C



This disk...

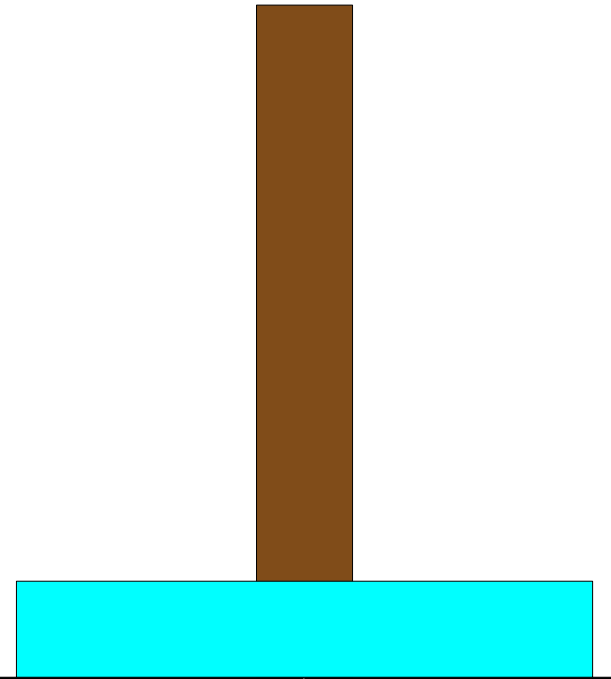
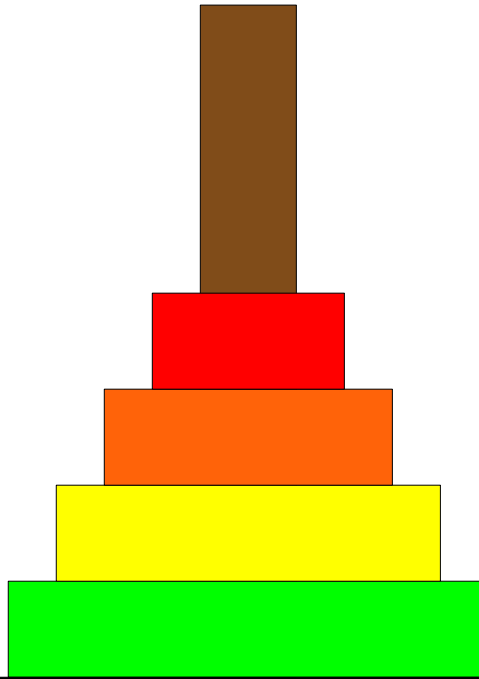
...needs to get over here.

# Solving the Towers of Hanoi

A

B

C



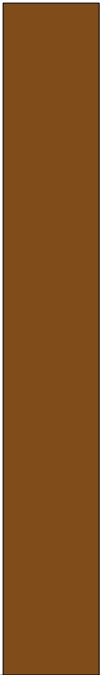
This disk...

...needs to get over here.

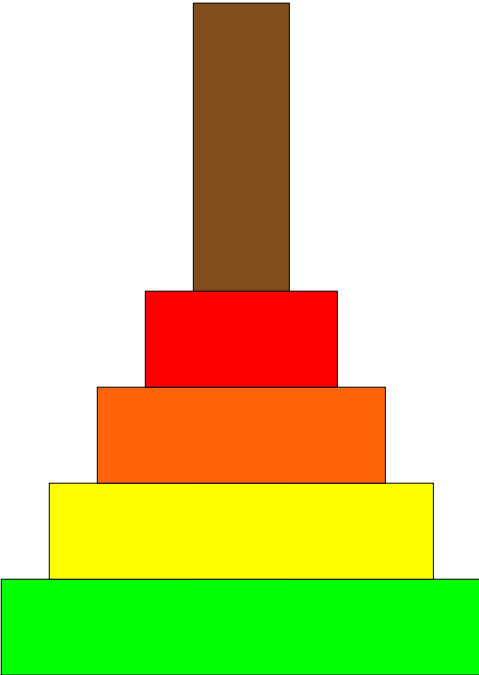


# Solving the Towers of Hanoi

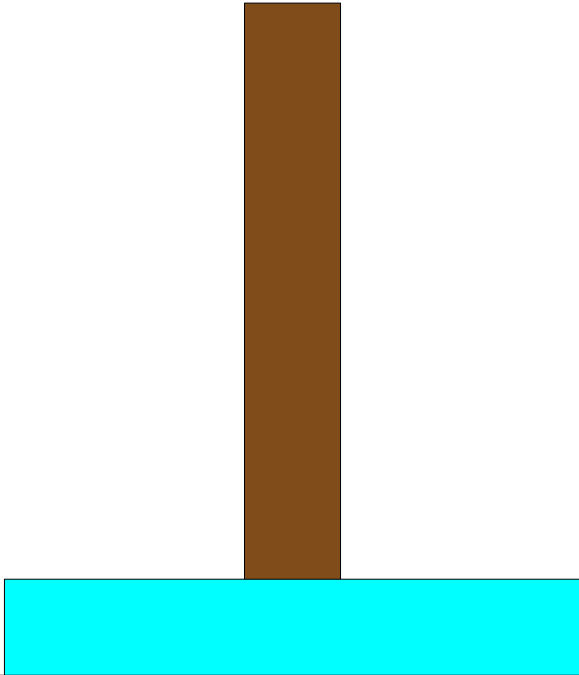
A



B



C



# Solving the Towers of Hanoi

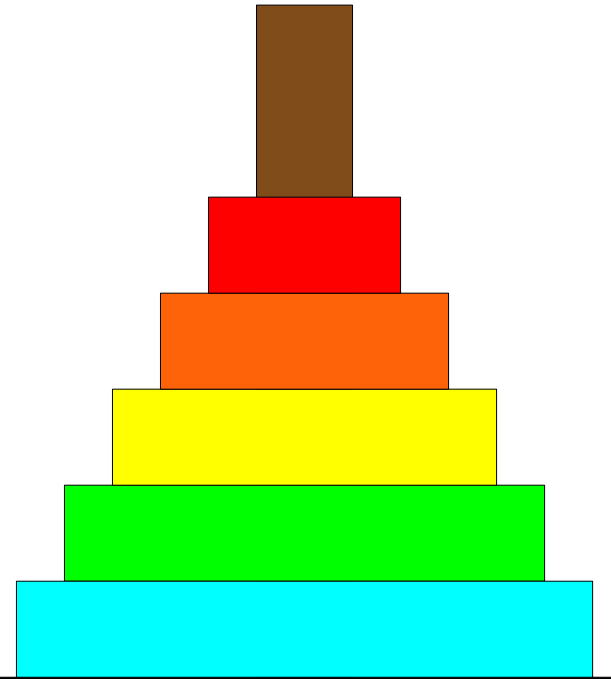
A



B



C

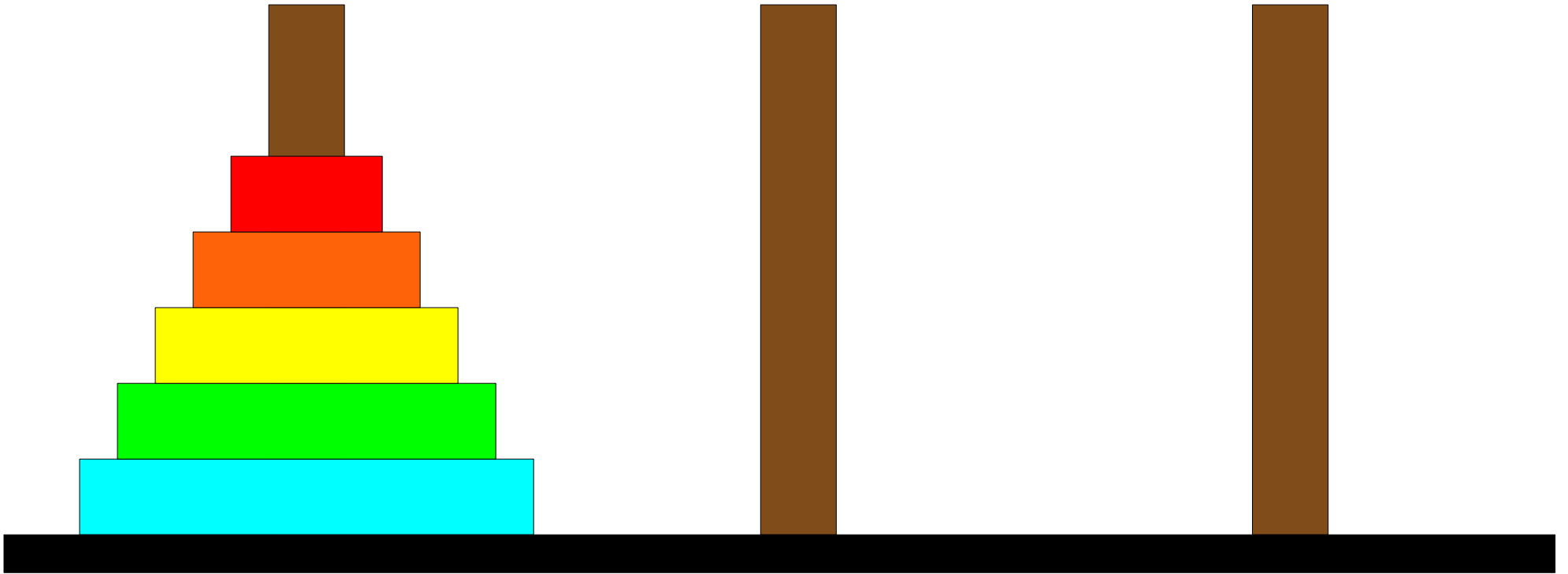


# Recursive “Leap of Faith”

A

B

C

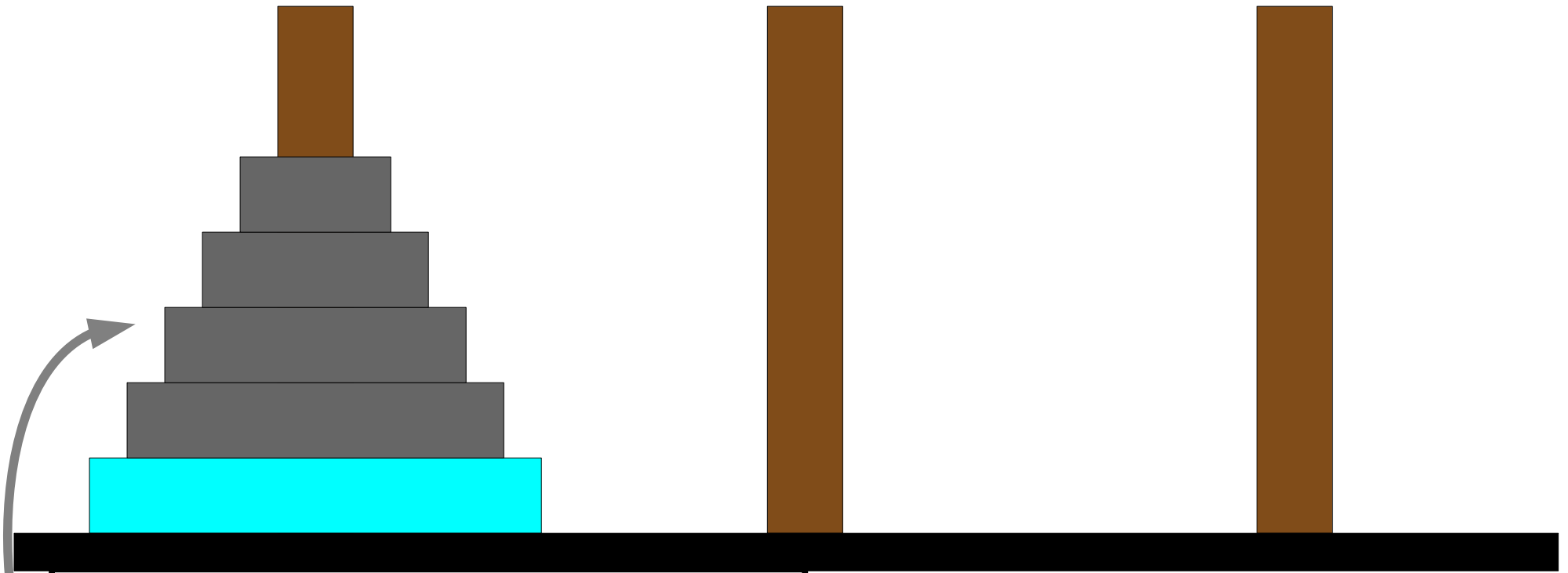


# Recursive “Leap of Faith”

A

B

C



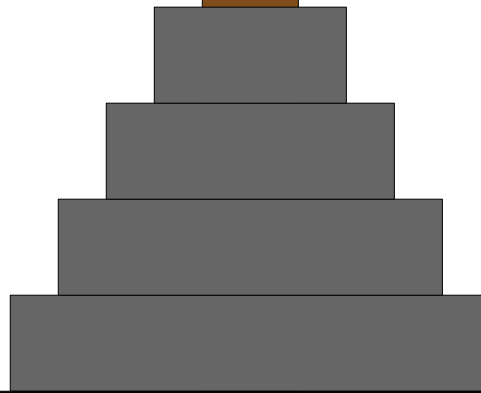
Take a “leap of faith” that the recursive call will move this tower correctly.

# Recursive “Leap of Faith”

A

B

C



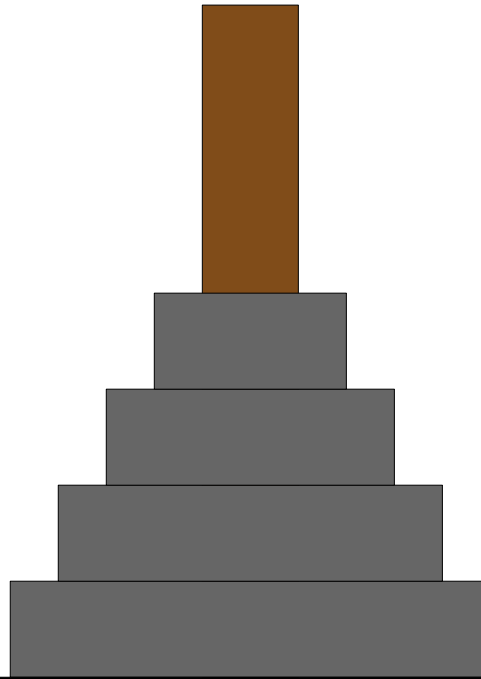
Take a “leap of faith” that the recursive call will move this tower correctly.

# Recursive “Leap of Faith”

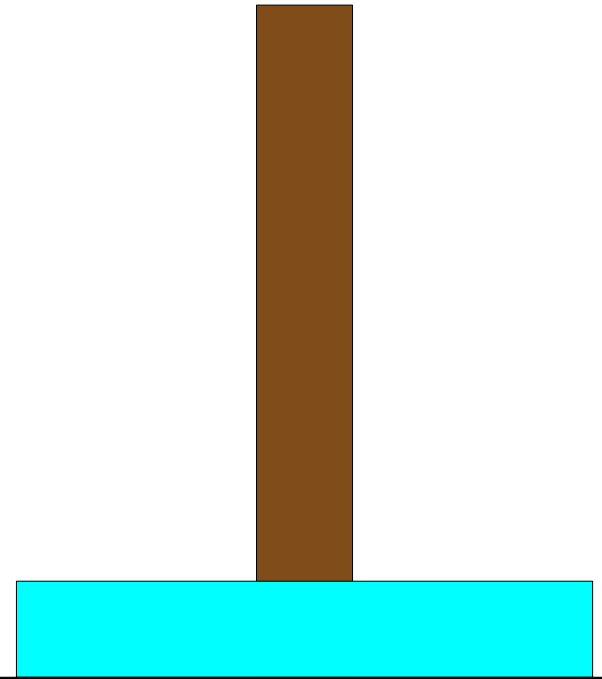
A



B



C



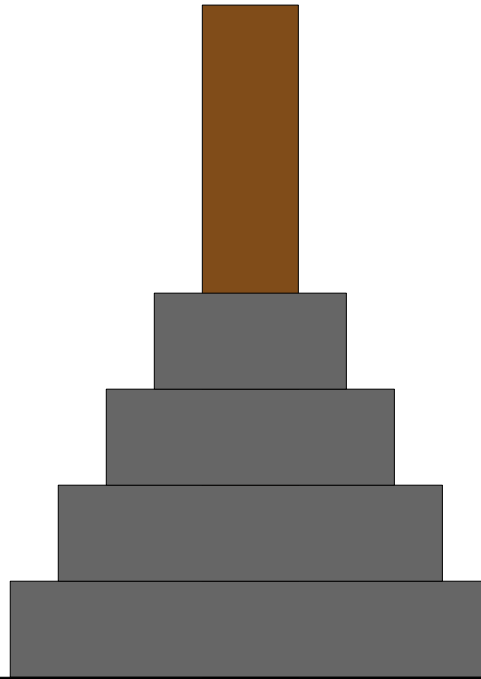
Take a “leap of faith” that the recursive call will move this tower correctly.

# Recursive “Leap of Faith”

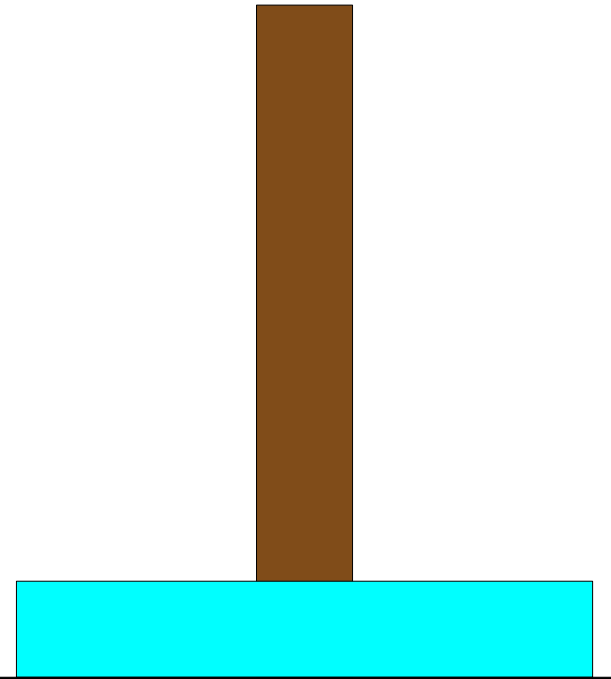
A



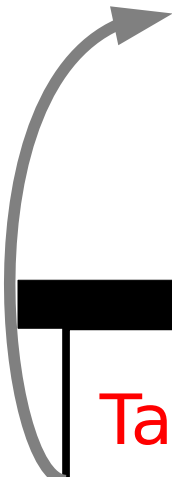
B



C



Take a “leap of faith” that the recursive call will move this tower correctly.



# Recursive “Leap of Faith”

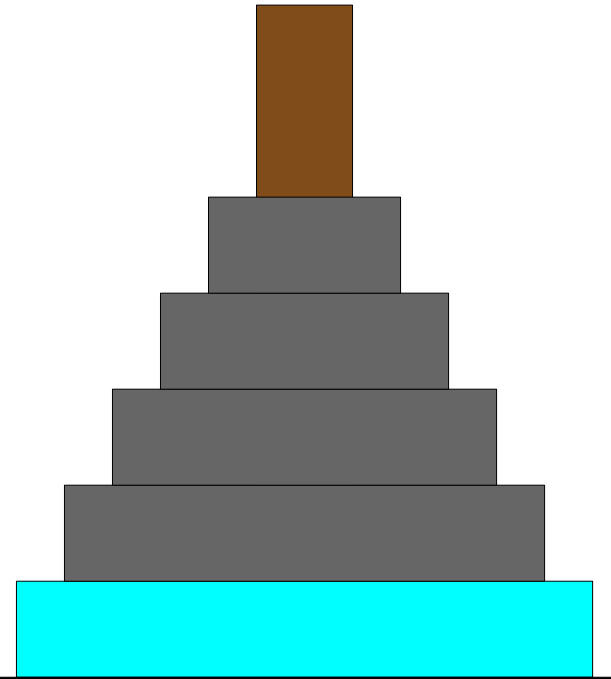
A



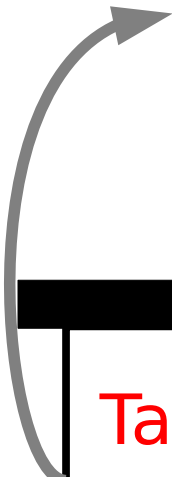
B



C



Take a “leap of faith” that the recursive call will move this tower correctly.

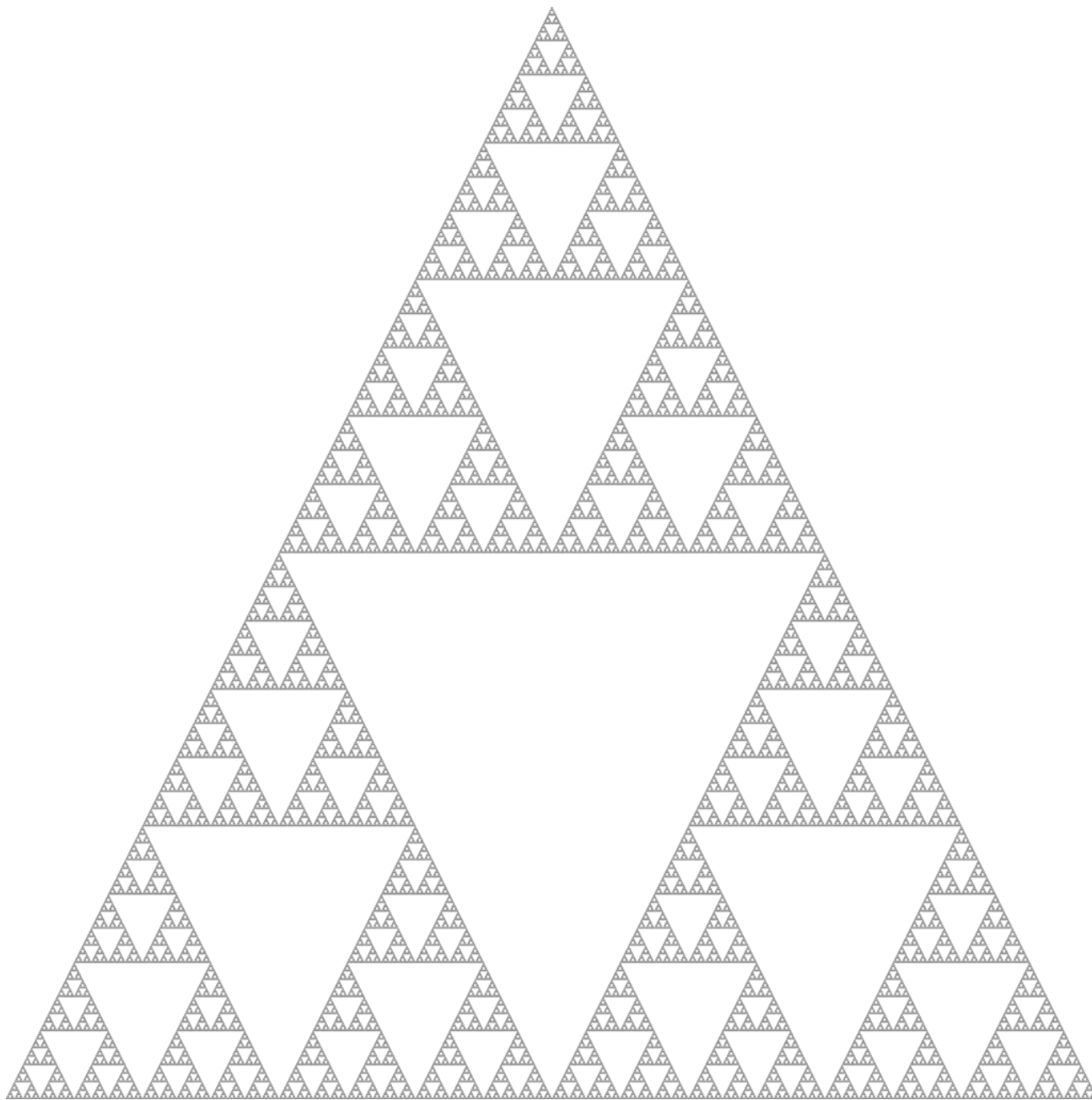




# Writing Recursive Functions

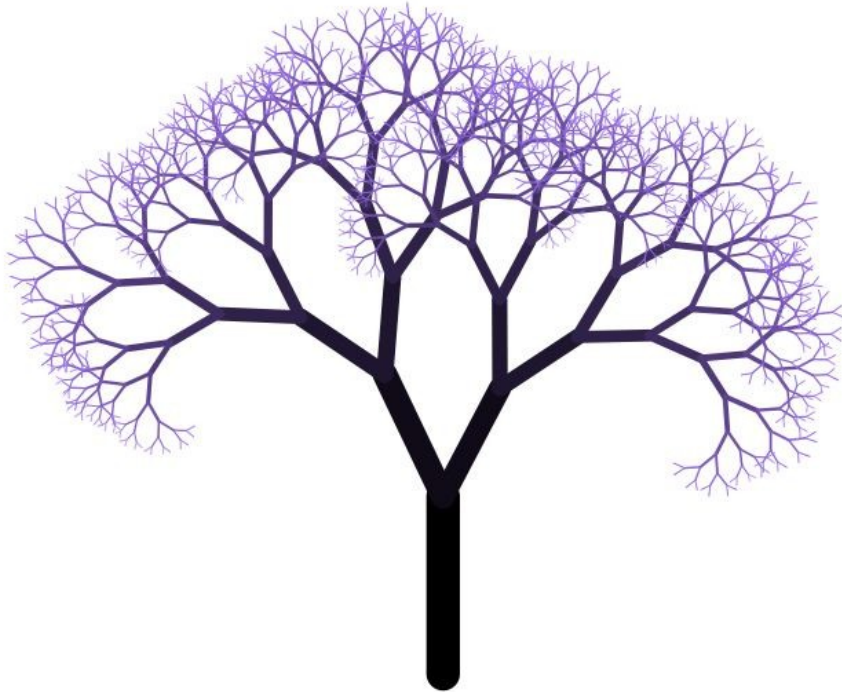
- Another way of putting this: when writing recursive functions it can be very helpful to **assume** that the recursive call will do the right thing.
- This is helpful because it's very hard to think through an **exponential** number of recursive calls.

New Stuff...



A **fractal image** is an image that is defined in terms of smaller versions of itself.

# Fractal Trees



- We can generate a fractal tree as follows:
  - Grow in some direction for a period of time.
  - Then, split and grow smaller trees outward at some angle.

Trees  
(Pseudocode)

Trees.cpp  
(Computer)

# More Trees

- What if you change the amount of branching?
- What if you make the lines thicker?
- What if you allow the tree to keep growing after it branches?
- What if you color the branches and leaves differently?
- What if you try to space the branches apart more realistically?
- Stanford **Dryad** program uses a combination of recursion, machine learning, and human feedback to design aesthetically pleasing trees.
  - Check it out at <http://dryad.stanford.edu/>

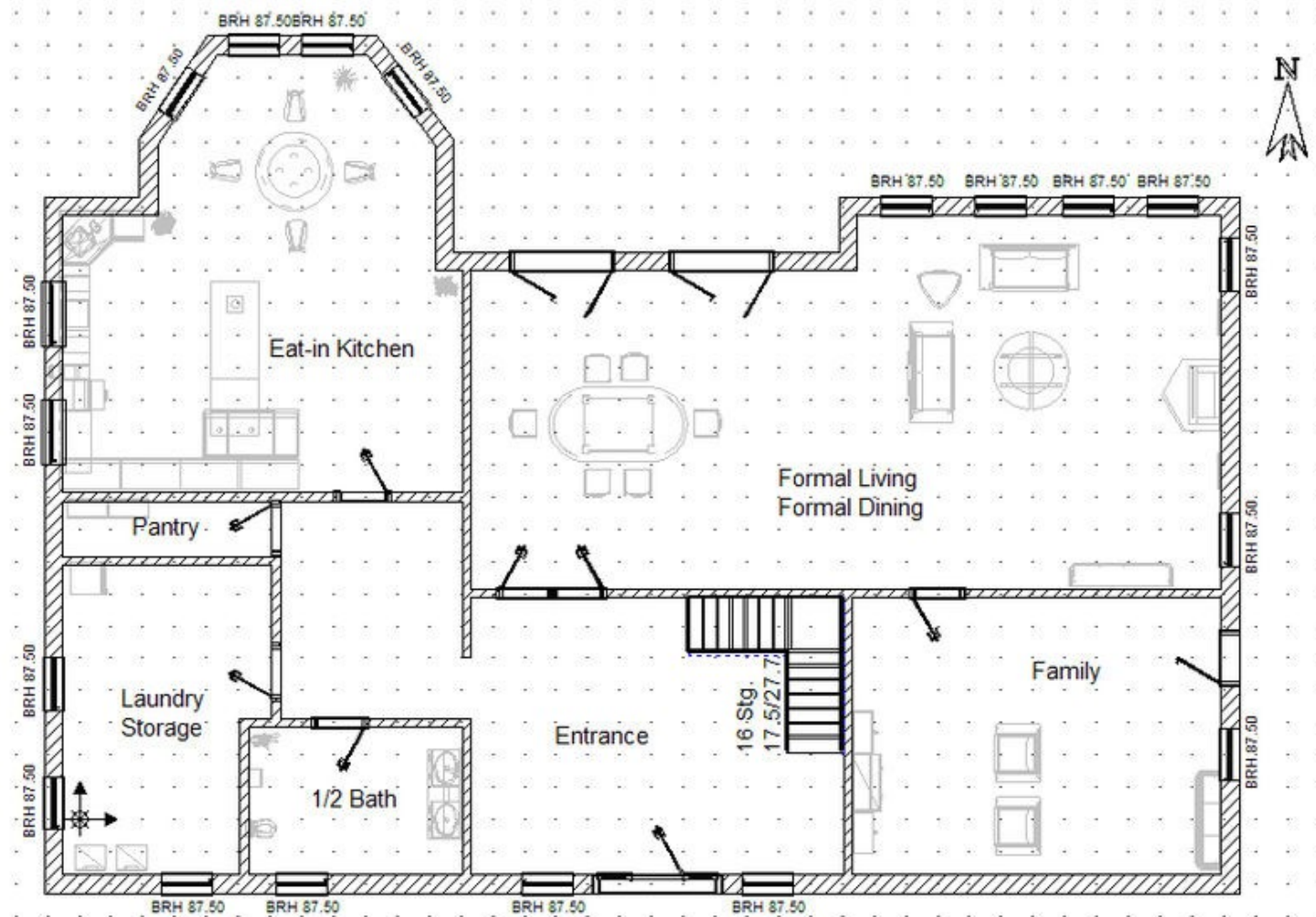
An Amazing Website

**<http://recursivedrawing.com/>**

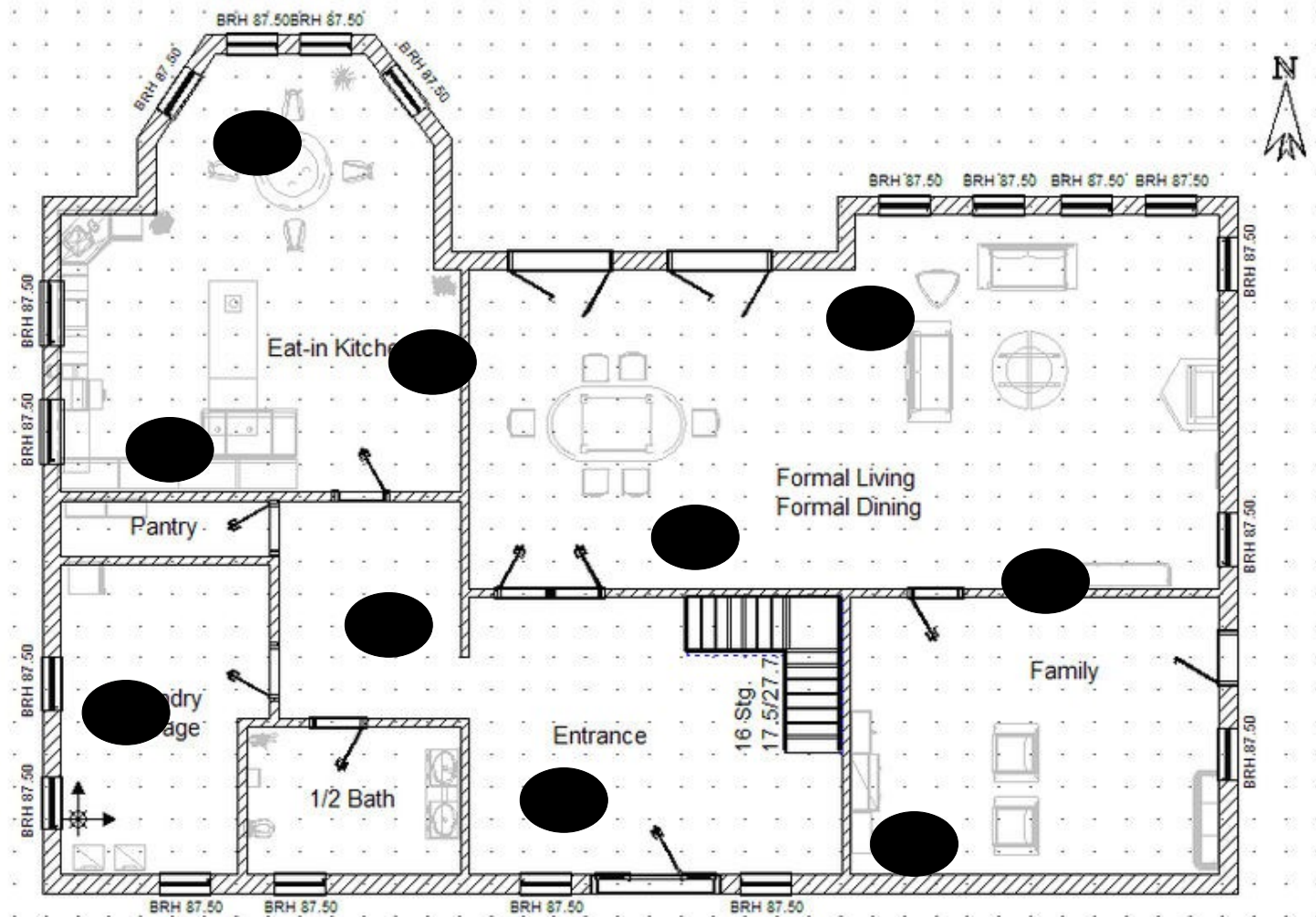


# Exhaustive Recursion

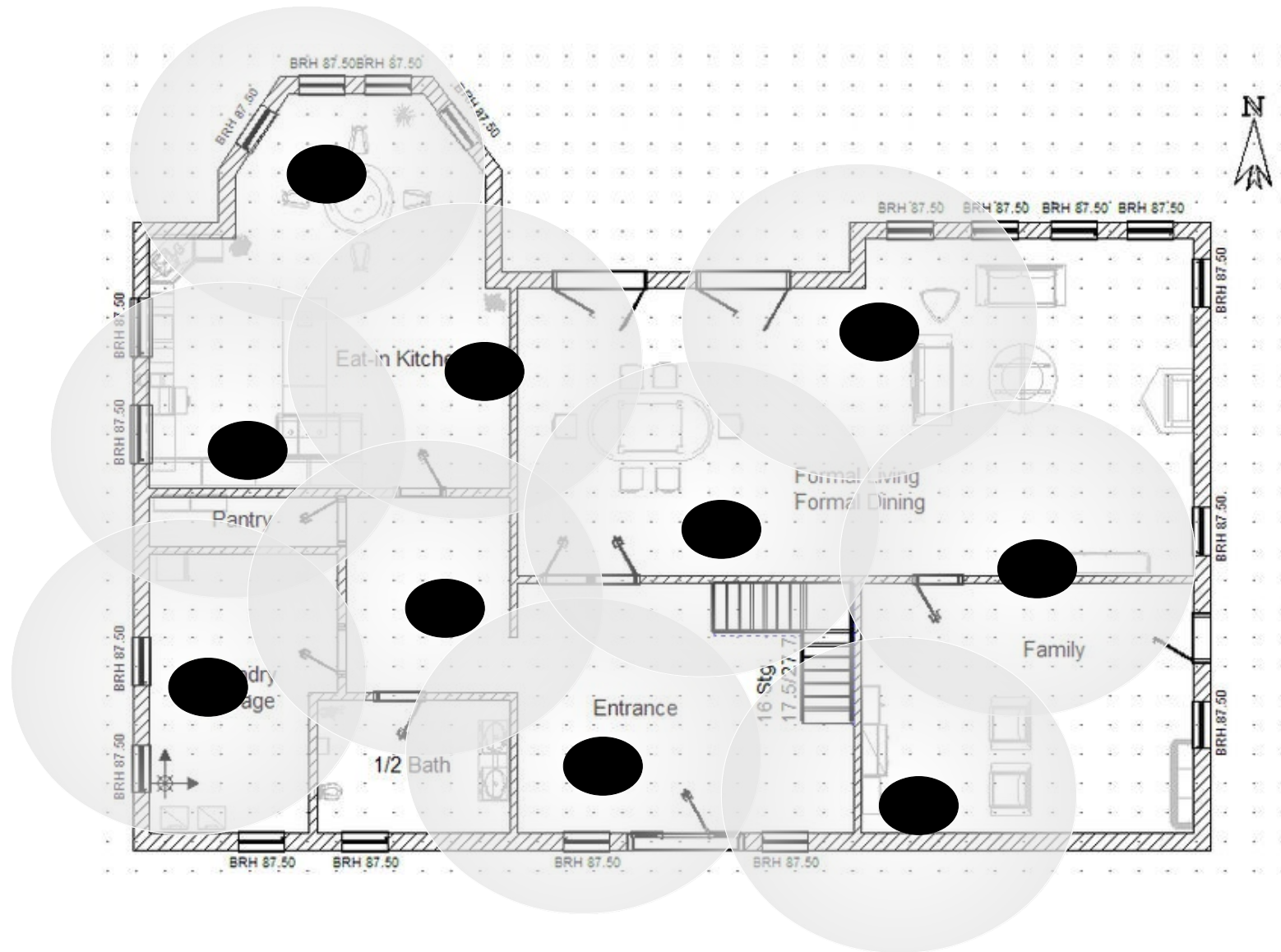
# Sensor Placement



# Sensor Placement



# Sensor Placement



# Sensor Placement



# Sensor Placement

- **No known** efficient algorithms for solving this problem *perfectly*
  - Fast algorithms exist for *approximately* solving this
  - Still an active area of research
- If you want the correct answer, the best you can do is **consider all possible choices of sensors** and return the best one.
- How can we generate all possible choices?

# Subsets

- Given set  $S$ , a **subset** of  $S$  is a set formed by choosing some number of elements from  $S$ .
  - Note: A set can only hold a single “copy” of an element.
    - e.g The set  $\{0, 1, 1\}$  is identical to the set  $\{0, 1\}$
- Examples:
  - $\{0, 1, 4\}$  is a subset of  $\{0, 1, 2, 3, 4, 5\}$
  - $\{\text{dikdik}, \text{ibex}\}$  is a subset of  $\{\text{dikdik}, \text{ibex}\}$
  - $\{\ } \subseteq \{a, b, c\}$
  - $\{\ } \subseteq \{\ }$

# Generating Subsets

- Many important problems in computer science can be solved by listing all the subsets of a set  $S$  and finding the “best” one out of every option.
  - Like optimal sensor placement!



# Generating Subsets

$$\{0, 1, 2\}$$

$$\{\}$$

$$\{0\}$$

$$\{2\}$$

$$\{0, 2\}$$

$$\{1\}$$

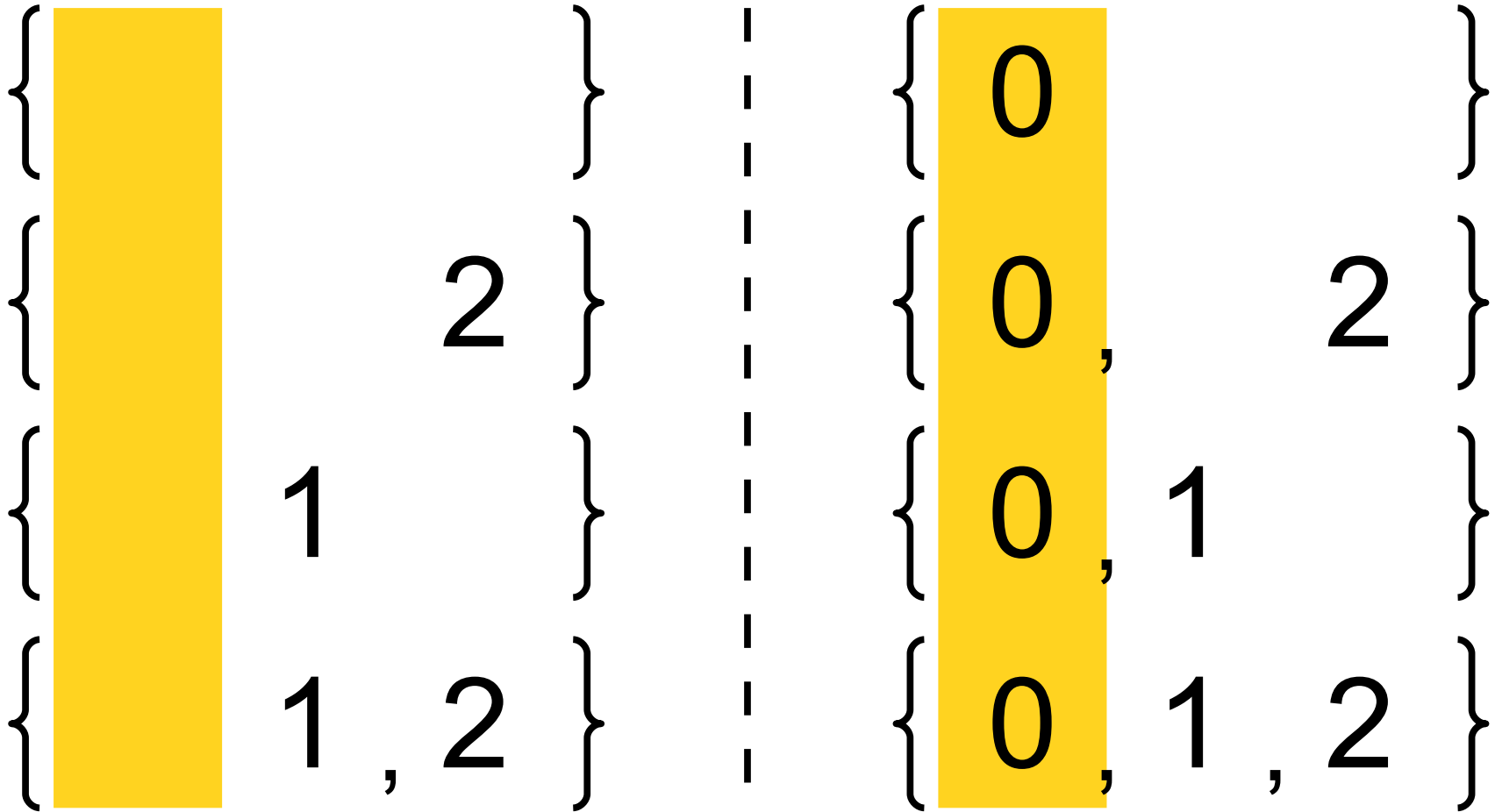
$$\{0, 1\}$$

$$\{1, 2\}$$

$$\{0, 1, 2\}$$

# Generating Subsets

$$\{0, 1, 2\}$$



# Generating Subsets

$$\{0, 1, 2\}$$

$$\{\}$$

$$\{0\}$$

$$\{2\}$$

$$\{0, 2\}$$

$$\{1\}$$

$$\{0, 1\}$$

$$\{1, 2\}$$

$$\{0, 1, 2\}$$

# Generating Subsets

$\{0, 1, 2\}$

$\{\}$

$\{2\}$

$\{1\}$

$\{1, 2\}$

$\{0\}$

$\{0, 2\}$

$\{0, 1\}$

$\{0, 1, 2\}$

# Generating Subsets

$$\{0, 1, 2\}$$

$$\{\}$$

$$\{0\}$$

$$\{2\}$$

$$\{0, 2\}$$

$$\{1\}$$

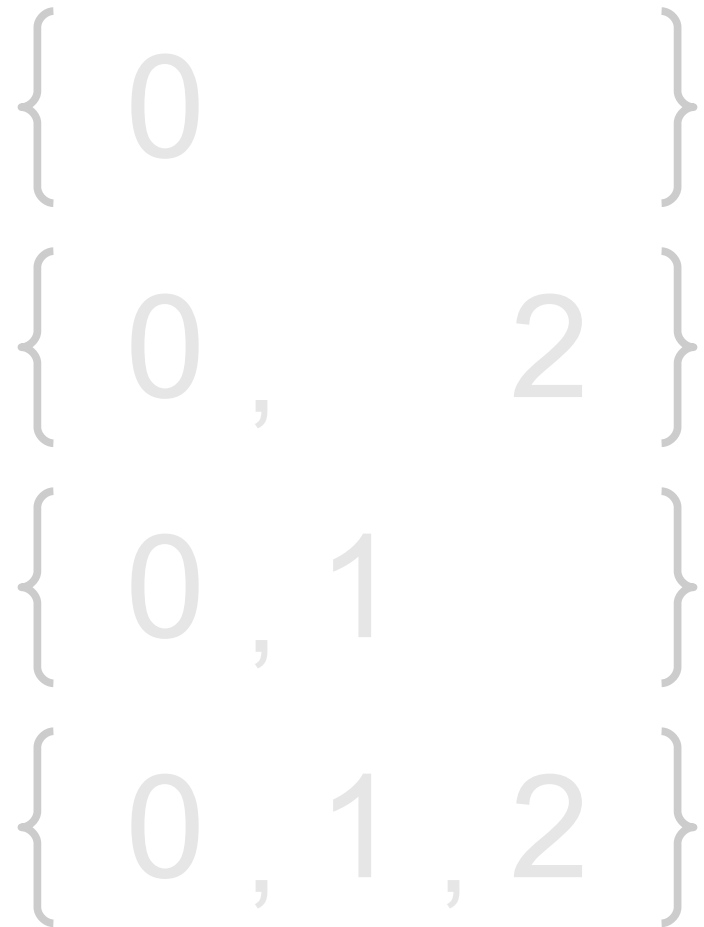
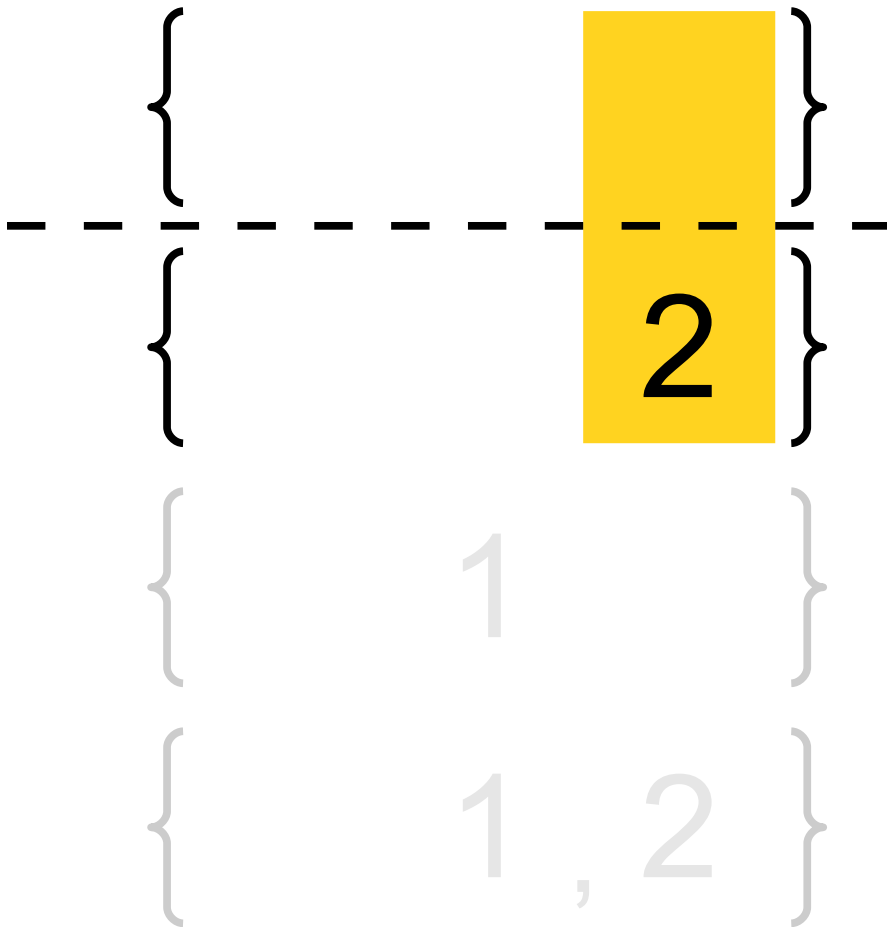
$$\{0, 1\}$$

$$\{1, 2\}$$

$$\{0, 1, 2\}$$

# Generating Subsets

$\{0, 1, 2\}$



subsetsOf()  
(Pseudocode)

# Generating Subsets

- **Base Case:**
  - The only subset of the empty set is the empty set.
- **Recursive Step:**
  - Fix some element  $x$  of the set.
  - Generate all subsets of the set formed by removing  $x$  from the main set.
  - These subsets are subsets of the original set.
  - All of the sets formed by adding  $x$  into those subsets are subsets of the original set.



```
subsets.cpp::subsetsOf()  
  (Computer)
```

# Tracing the Recursion

Input Set

Subsets Generated

# Tracing the Recursion

Input Set

Subsets Generated

{ A, H, I }

# Tracing the Recursion

Input Set

{ A, H, I }

Subsets Generated

{ H, I }

# Tracing the Recursion

Input Set

Subsets Generated

{ A, H, I }

{ H, I }

{ I }

# Tracing the Recursion

Input Set

Subsets Generated

{ A, H, I }

{ H, I }

{ I }

{ }

# Tracing the Recursion

Input Set

Subsets Generated

{ A, H, I }

{ H, I }

{ I }

{ }

{ }

# Tracing the Recursion

Input Set

Subsets Generated

{ A, H, I }

{ H, I }

{ I }

{ }

{ I }, { }

{ }



# Tracing the Recursion

Input Set

Subsets Generated

{ A, H, I }

{ H, I }

{H, I}, {H}, {I}, { }

{ I }

{I}, { }

{ }

{ }

# Tracing the Recursion

Input Set

Subsets Generated

{ A, H, I }

{A, H, I}, {A, H}, {A, I}, {A}  
{H, I}, {H}, {I}, { }

{ H, I }

{H, I}, {H}, {I}, { }

{ I }

{I}, { }

{ }

{ }

# Analyzing Our Function

- How many subsets are there of a set with  $n$  elements?
- We can make a subset by choosing, for each element, whether to include it in the subset or exclude it from the subset.
- We make  $n$  choices with 2 options for each choice, so there are  $2^n$  possible subsets.
- The returned collection of sets will use at least  $2^n$  bytes of memory.

# A Quick Calculation

- On my computer, an `int` is four bytes ( $4 = 2^2$ ).
- My computer has about 4GB of memory (about  $2^{32}$  bytes).
- If we need  $2^n$  space to hold the return value, what is the largest  $n$  we can pick without blowing up my computer (again)?
- **Answer:**  $n = 30$ .

# Reducing Memory Usage

- In many cases, we need to perform some operation on each subset, but don't need to actually store those subsets.
- **Idea:** Generate each subset, process it, and then discard it.
- **Question:** How do we do this?

# Recursively Exploring Options

- Our recursive function needs to keep track of
  - What choices we've made so far, and
  - What choices we still need to make.
- **Base Case:**
  - If there are no choices left, output the set we formed from the choices we made.
- **Recursive Step:**
  - Find the next choice to make.
  - For each possible choice, recursively explore all options formed from making that choice.

visitSubsets  
(Pseudocode)

```
subsets.cpp::visitSubsets  
(Computer)
```



# Visualizing Subset Generation

- A very useful way of visualizing what happens in a recursive function is with a **decision tree**.
- Idea: Visualize the parameters and “choices” made in the recursion.
- Can provide valuable insights into how solutions are generated via recursive functions.

Subset Decision Tree  
(Board)

# A Decision Tree

$\{\}$

$\{I\}$

$\{H\}$

$\{H, I\}$

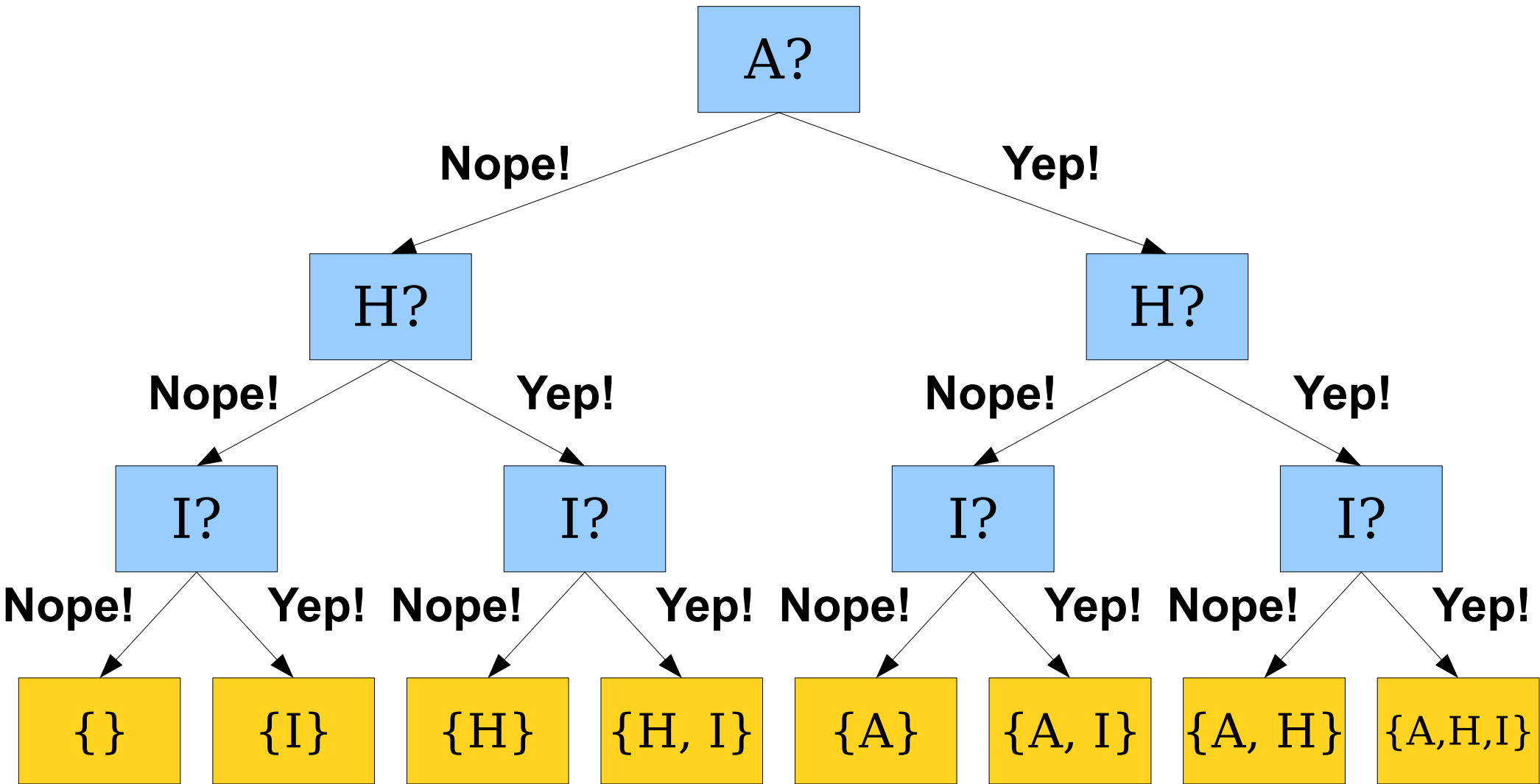
$\{A\}$

$\{A, I\}$

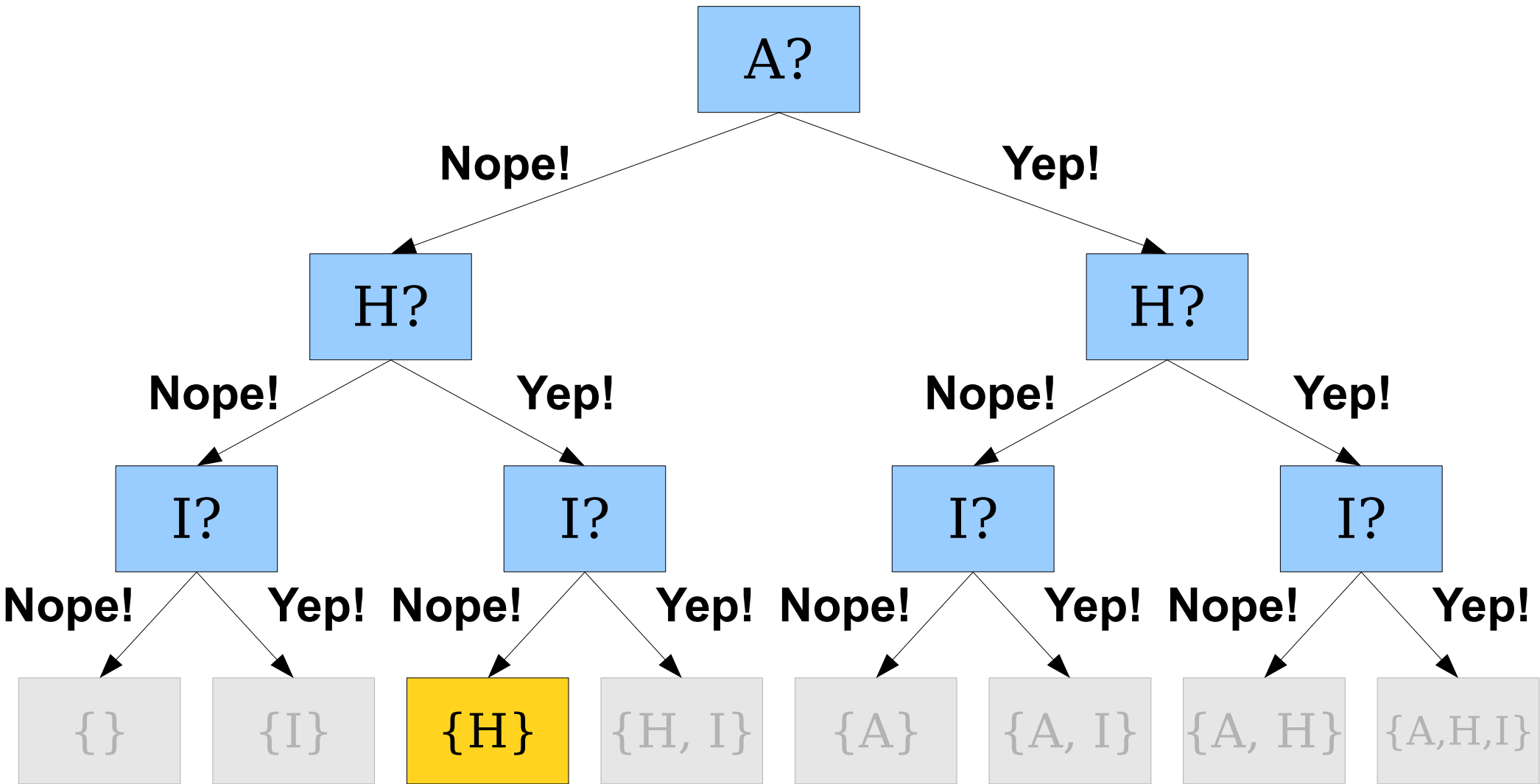
$\{A, H\}$

$\{A, H, I\}$

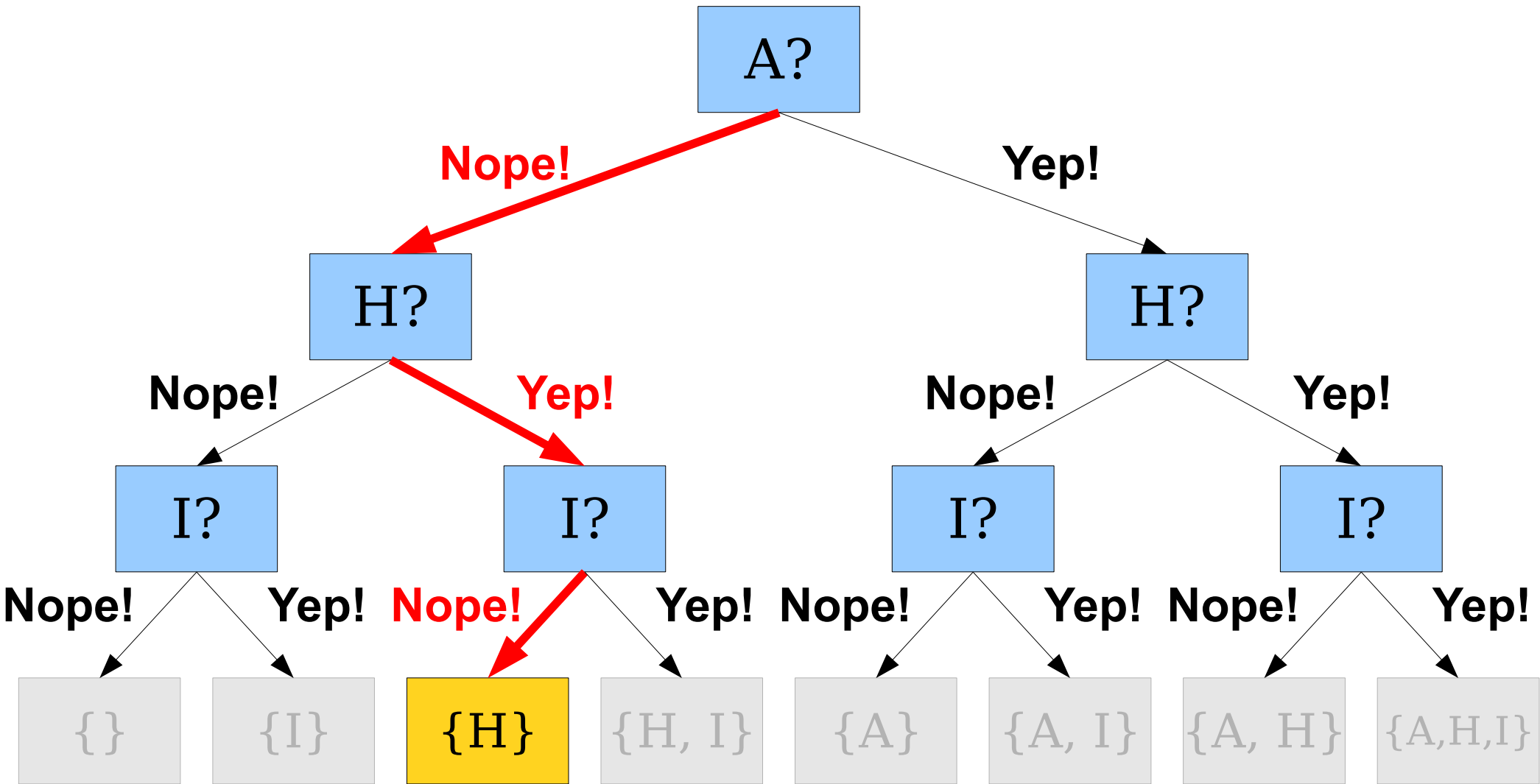
# A Decision Tree



# A Decision Tree



# A Decision Tree



# Next Time

- **Exhaustive Recursion II**
  - What other structures can we generate?
  - How do we do so efficiently?
- **Recursive Backtracking**
  - How do you find a needle in a haystack?