

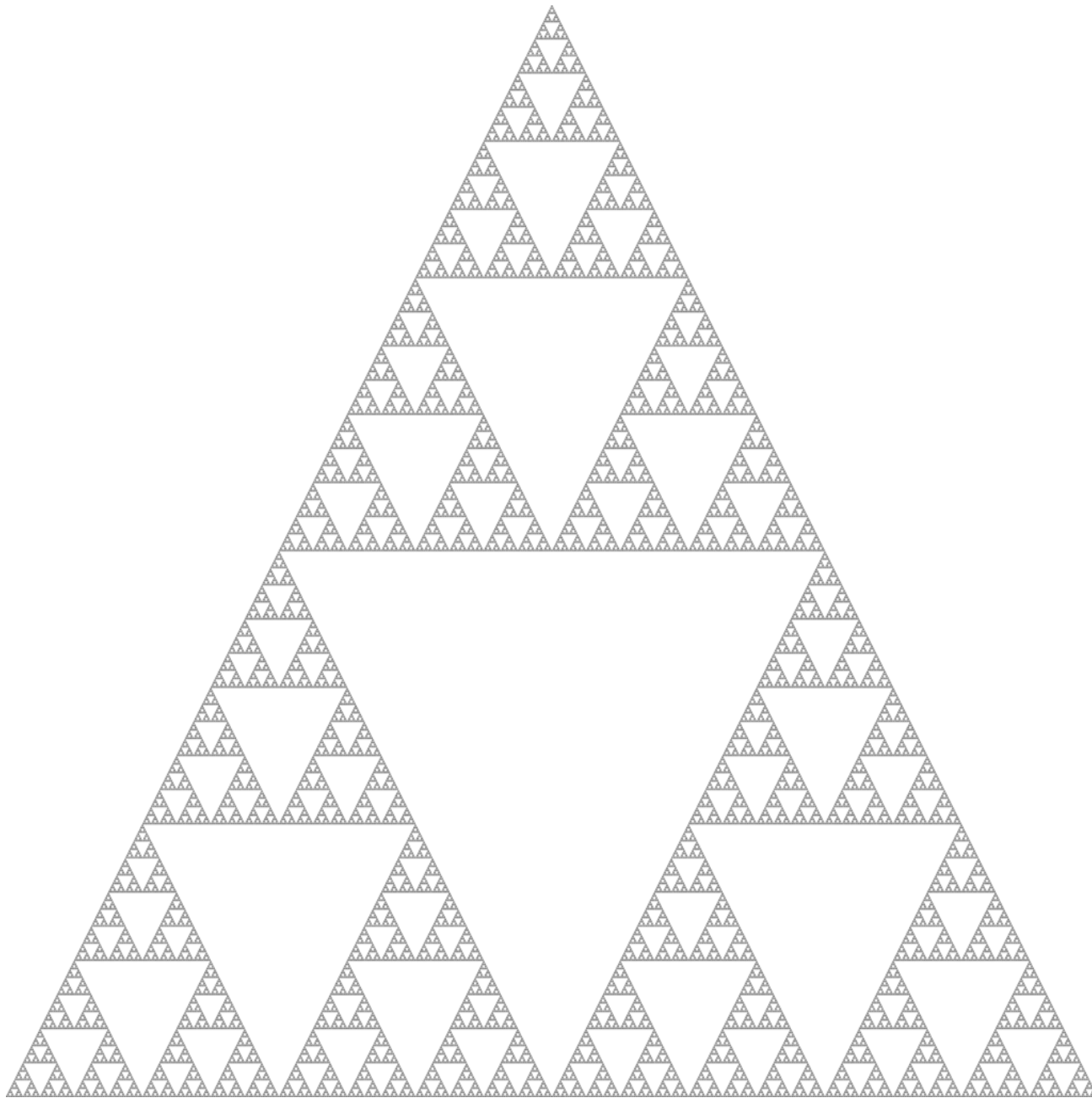
# Thinking Recursively

## Part II

Friday Four Square!  
Today at 4:15PM, Outside Gates

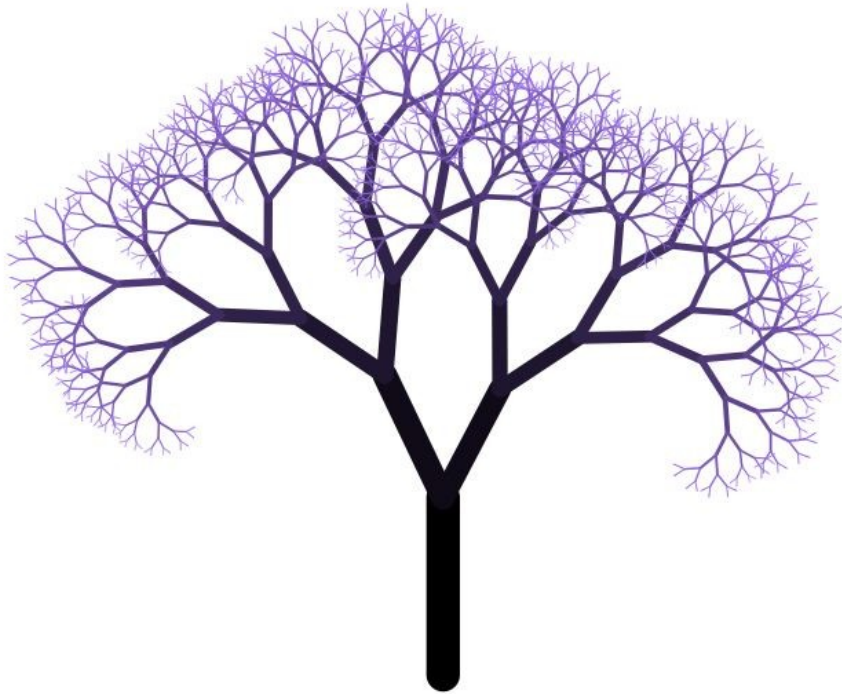
# Recursive Problem-Solving

```
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.  
}
```

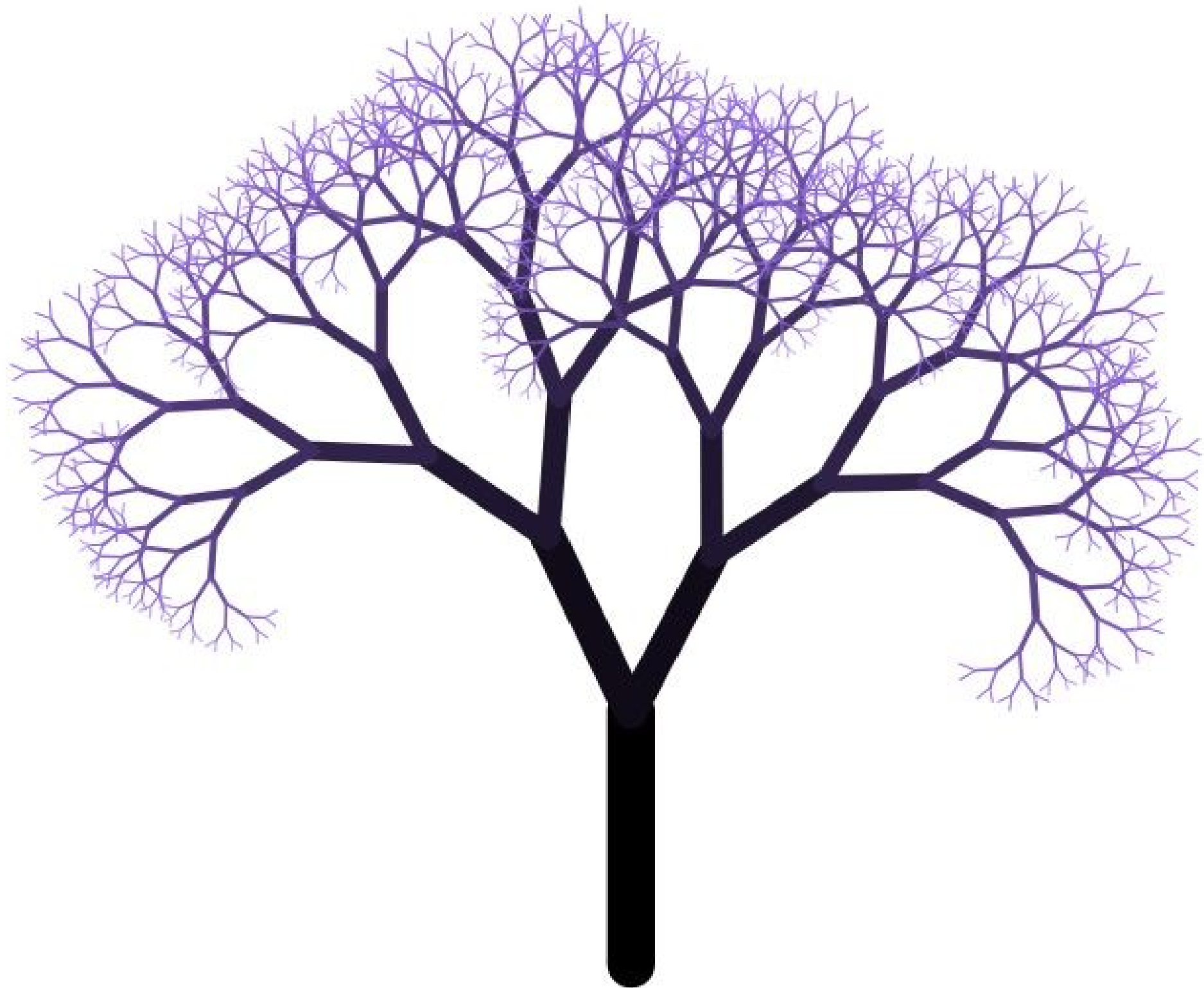


A **fractal image** is an image that is defined in terms of smaller versions of itself.

# 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.



# Methods in the Graphics Library

**GWindow gw**(*width*, *height*)

Creates a graphics window with the specified dimensions.

**gw.drawLine**(*x0*, *y0*, *x1*, *y1*)

Draws a line connecting the points (*x0*, *y0*) and (*x1*, *y1*).

**gw.drawPolarLine**(*x0*, *y0*, *r*, *theta*)

Draws a line *r* pixels long in direction *theta* from (*x0*, *y0*). To make chaining line segments easier, this function returns the ending coordinates as a **GPoint**.

**gw.getWidth**()

Returns the width of the graphics window.

**gw.getHeight**()

Returns the height of the graphics window.

Many more functions exist in the **gwindow.h** and **gobjects.h** interfaces. The full documentation is available on the web site.

# 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

# Generating All Possibilities

- Commonly, you will need to generate all objects matching some criteria.
  - Word Ladders: Generate all words that differ by exactly one letter.
- Often, structures can be generated iteratively.
- In many cases, however, it is best to think about generating all options recursively.

# Subsets

- Given set  $S$ , a **subset** of  $S$  is a set formed by choosing some number of elements from  $S$ .
- Examples:
  - $\{0, 1, 2\}$  is a subset of  $\{0, 1, 2, 3, 4, 5\}$
  - $\{\text{dikdik}, \text{ibex}\}$  is a subset of  $\{\text{dikdik}, \text{ibex}\}$
  - $\{A, G, C, T\}$  is a subset of  $\{A, B, C, D, \dots, Z\}$
  - $\{\} \subseteq \{a, b, c\}$
  - $\{\} \subseteq \{\}$

# 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.
- Example:
  - You have a collection of sensors on an autonomous vehicle, each of which has data coming in.
  - Which **subset** of the sensors do you choose to listen to, given that each takes a different amount of time to read?

# Generating Subsets

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

$$\{\}$$

$$\{2\}$$

$$\{1\}$$

$$\{1, 2\}$$

$$\{0\}$$

$$\{0, 2\}$$

$$\{0, 1\}$$

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

# Generating Subsets

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

{		}
{		}
{	2	}
{	1	}
{	1, 2	}

{	0	}
{	0, 2	}
{	0, 1	}
{	0, 1, 2	}

# Generating Subsets

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

$$\{\}$$

$$\{2\}$$

$$\{1\}$$

$$\{1, 2\}$$

$$\{0\}$$

$$\{0, 2\}$$

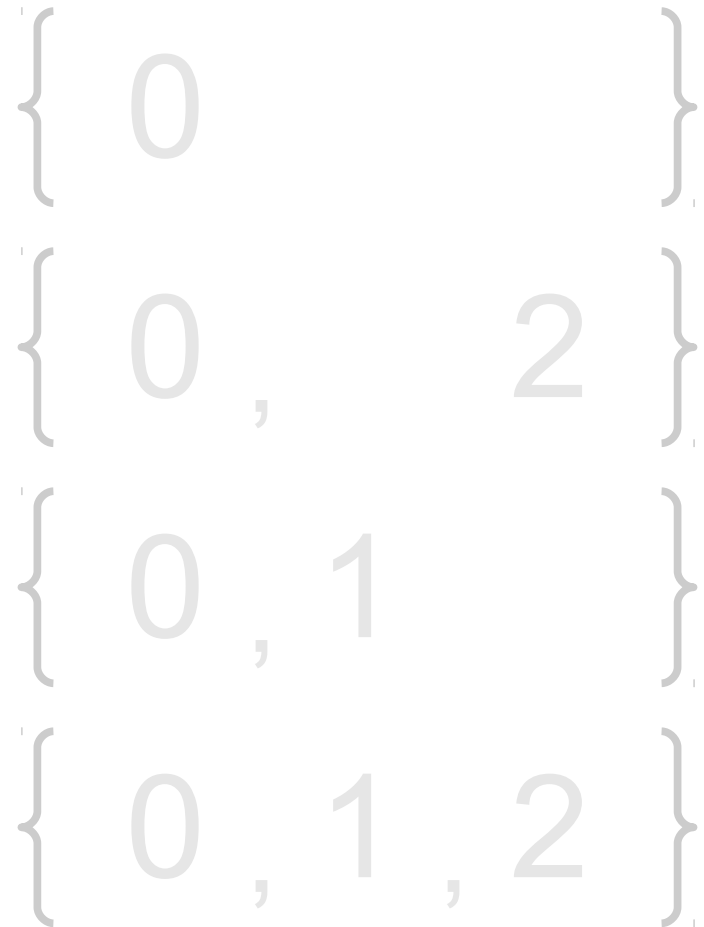
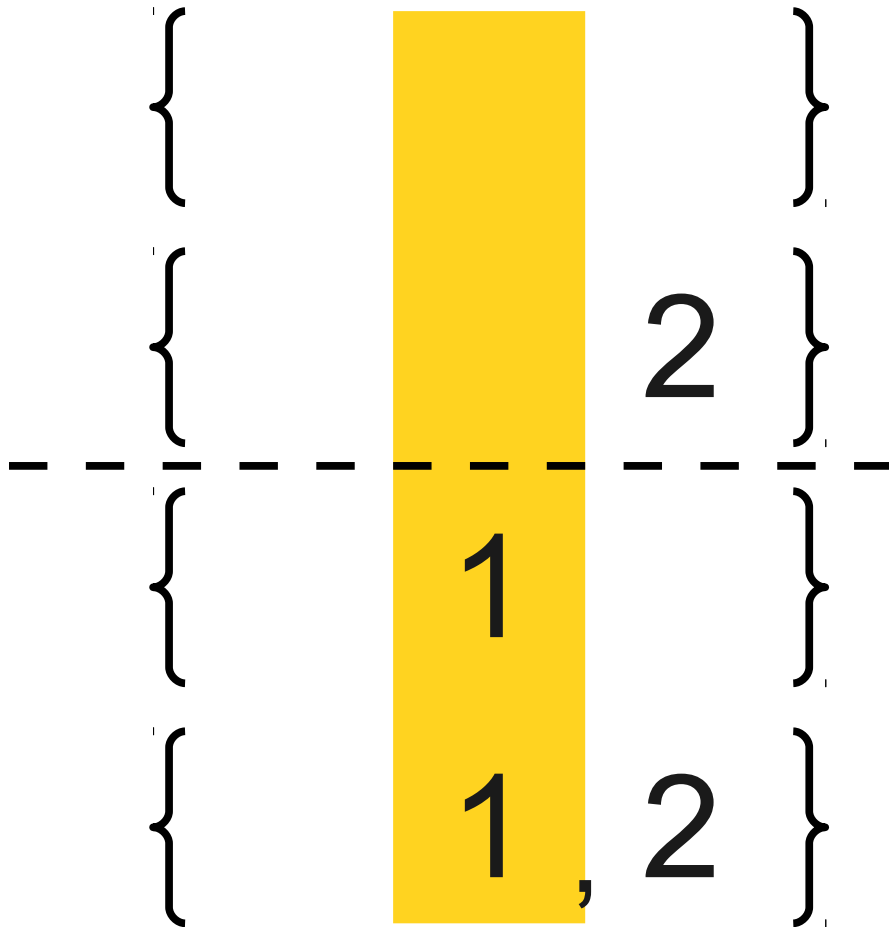
$$\{0, 1\}$$

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



# Generating Subsets

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



# Generating Subsets

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

$$\{\}$$

$$\{2\}$$

$$\{1\}$$

$$\{1, 2\}$$

$$\{0\}$$

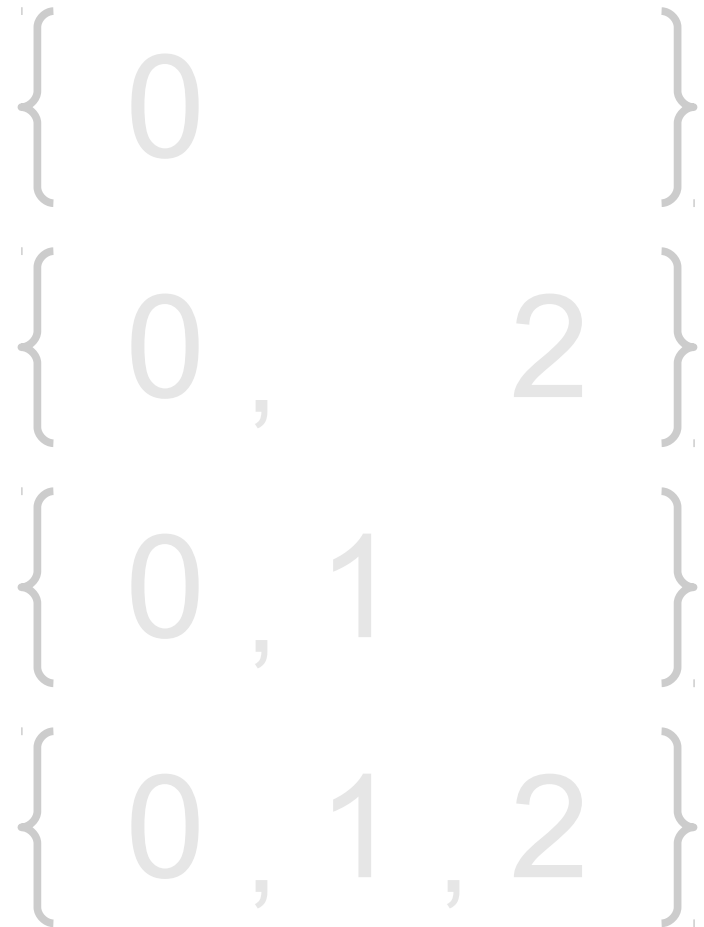
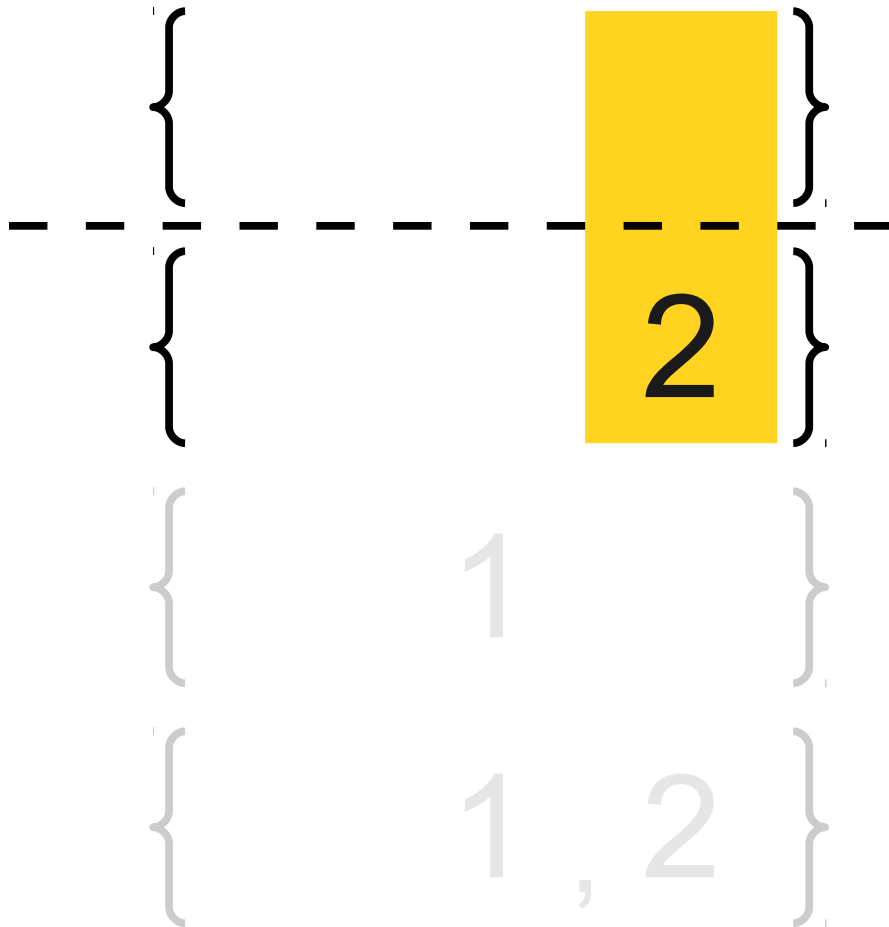
$$\{0, 2\}$$

$$\{0, 1\}$$

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

# Generating Subsets

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



# 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.

# Tracing the Recursion

# Tracing the Recursion

{ A, H, I }

# Tracing the Recursion

{ A, H, I }

{ H, I }

# Tracing the Recursion

{ A, H, I }

{ H, I }

{ I }



# Tracing the Recursion

{ A, H, I }

{ H, I }

{ I }

{ }

# Tracing the Recursion

{ A, H, I }

{ H, I }

{ I }

{ }

{ }

# Tracing the Recursion

{ A, H, I }

{ H, I }

{ I }

{ }

{I}, { }

{ }

# Tracing the Recursion

$\{ A, H, I \}$

$\{ H, I \}$

$\{ I \}$

$\{ \}$

$\{ H, I \}, \{ H \}, \{ I \}, \{ \}$

$\{ I \}, \{ \}$

$\{ \}$

# Tracing the Recursion

{ 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 about  $2^n$  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?



# A Decision Tree

$\{\}$

$\{I\}$

$\{H\}$

$\{H, I\}$

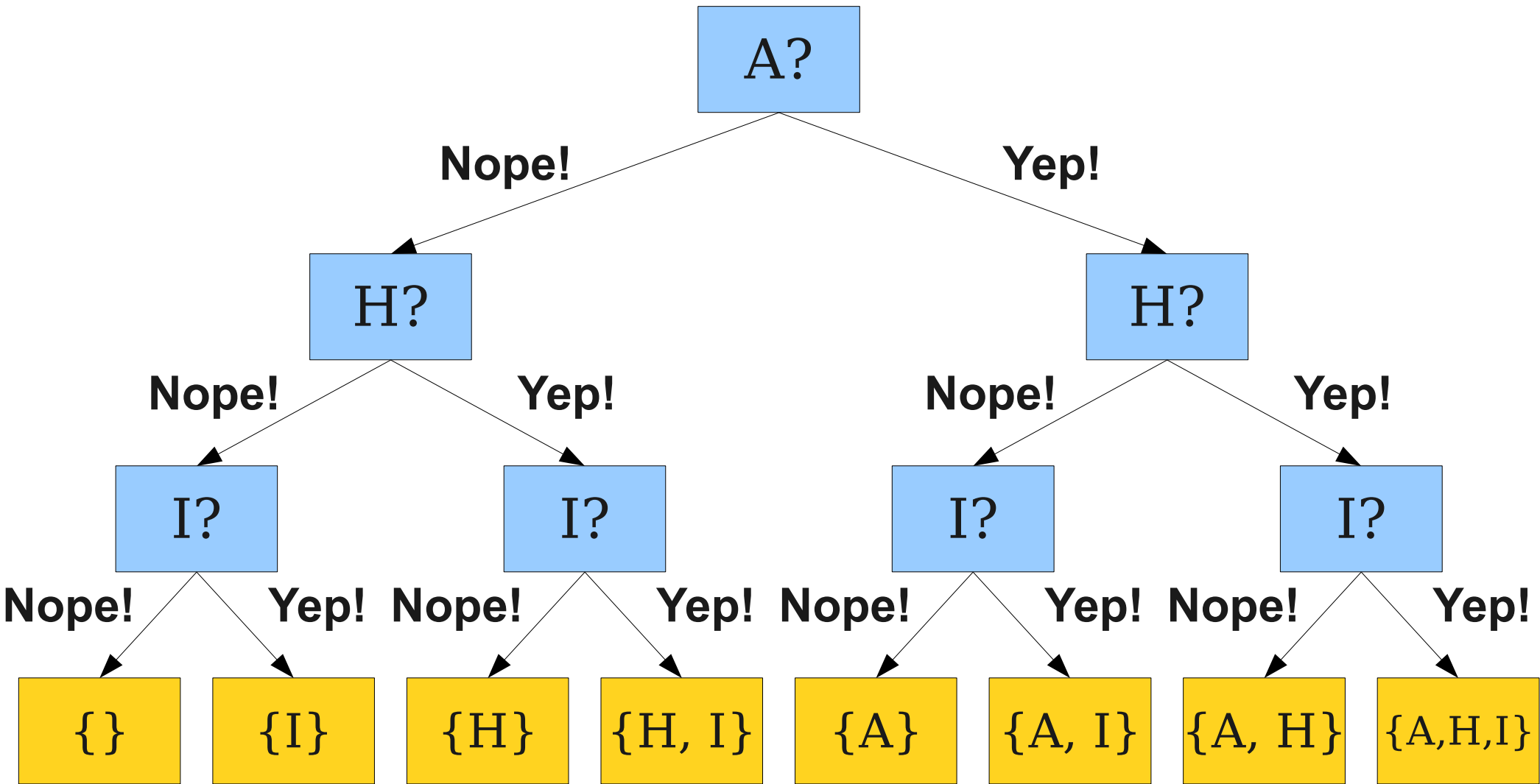
$\{A\}$

$\{A, I\}$

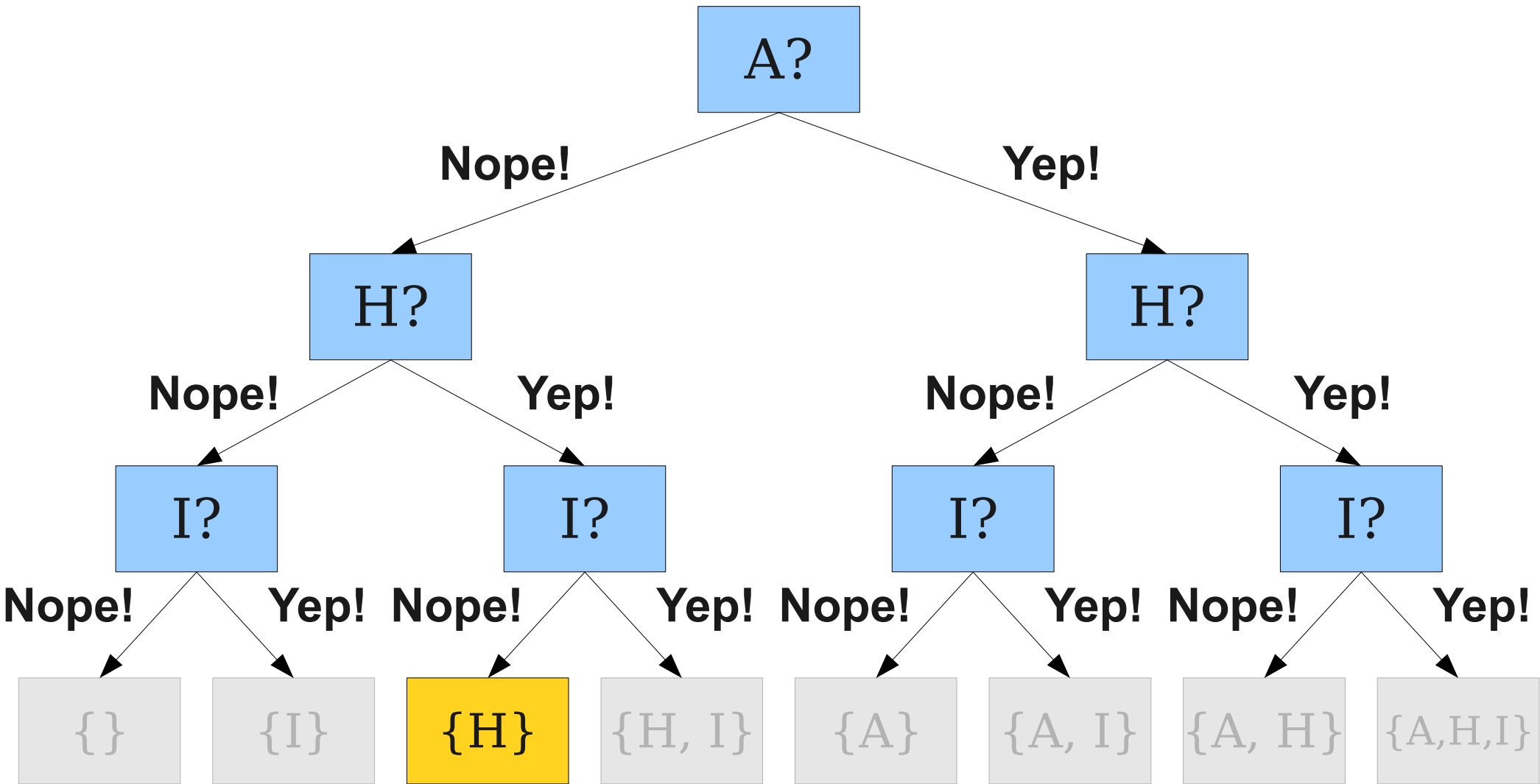
$\{A, H\}$

$\{A, H, I\}$

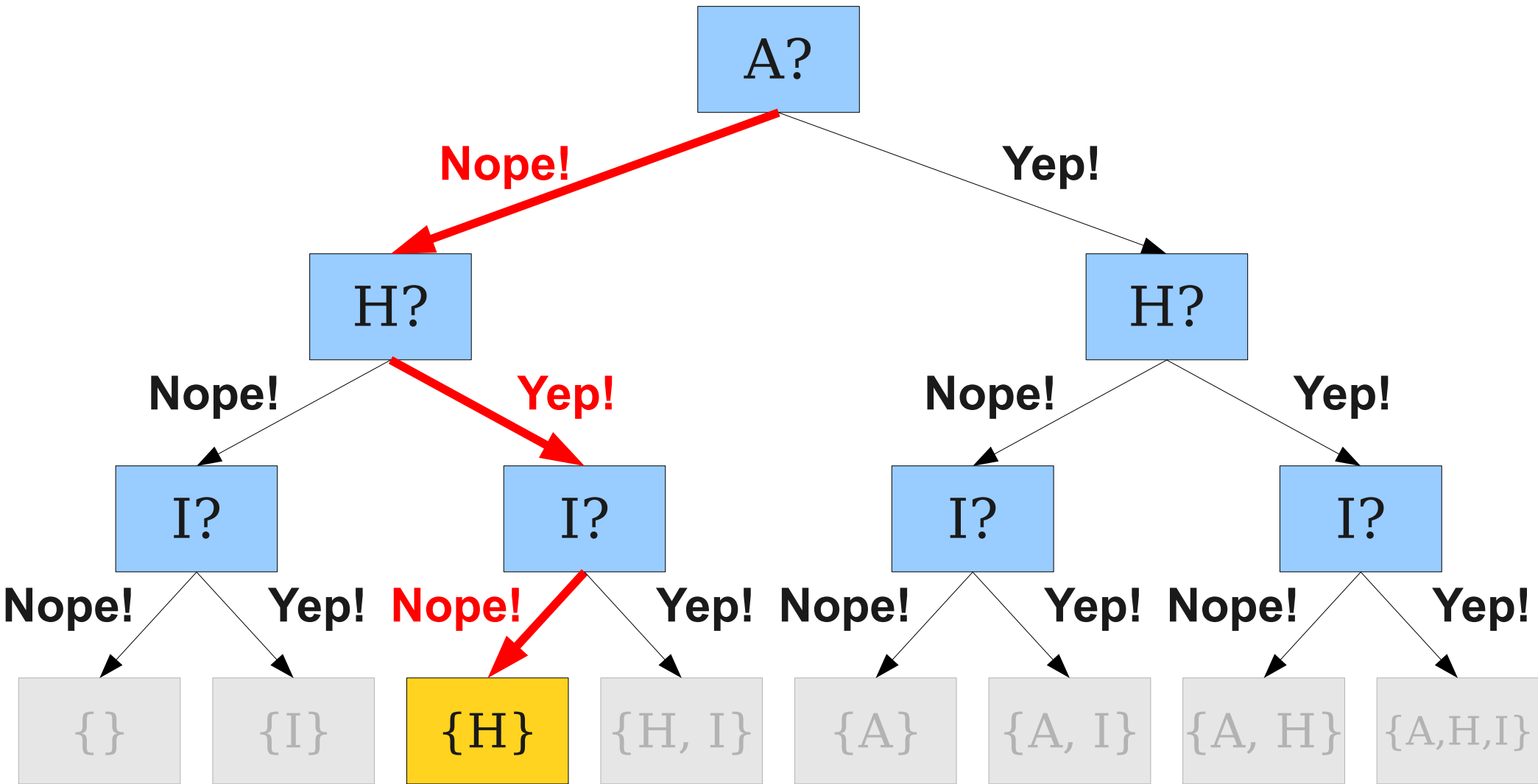
# A Decision Tree



# A Decision Tree



# A Decision Tree



# 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.

# 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?