Thinking Recursively

Part II

Outline for Today

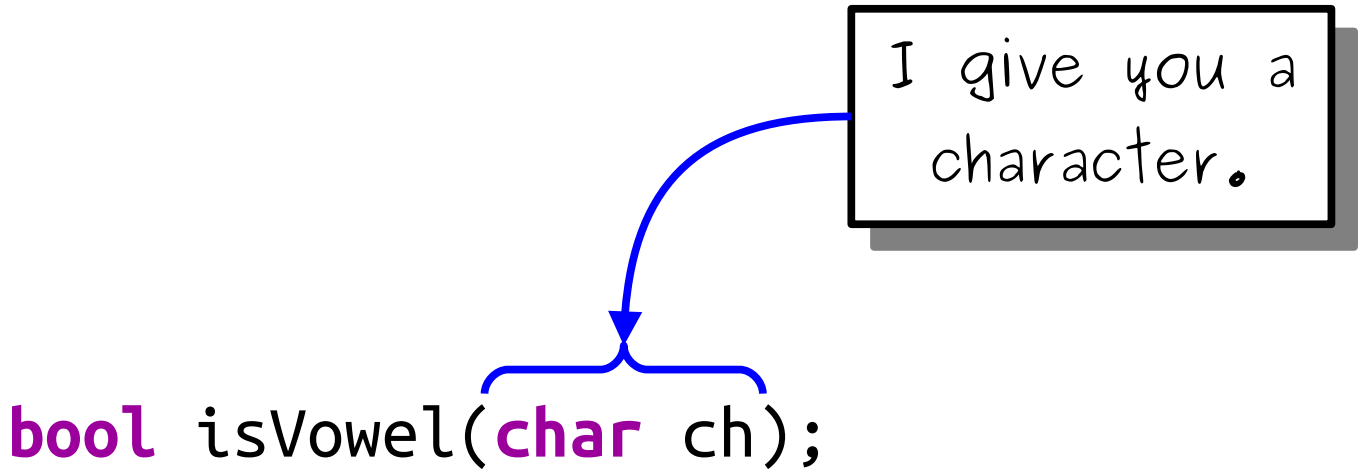
- ***The Recursive Leap of Faith***
  - On trusting the contract.
- ***Enumerating Subsets***
  - A classic combinatorial problem.
- ***Decision Trees***
  - Generating all solutions to a problem.
- ***Wrapper Functions***
  - Hiding parameters and keeping things clean.

The Recursive Leap of Faith

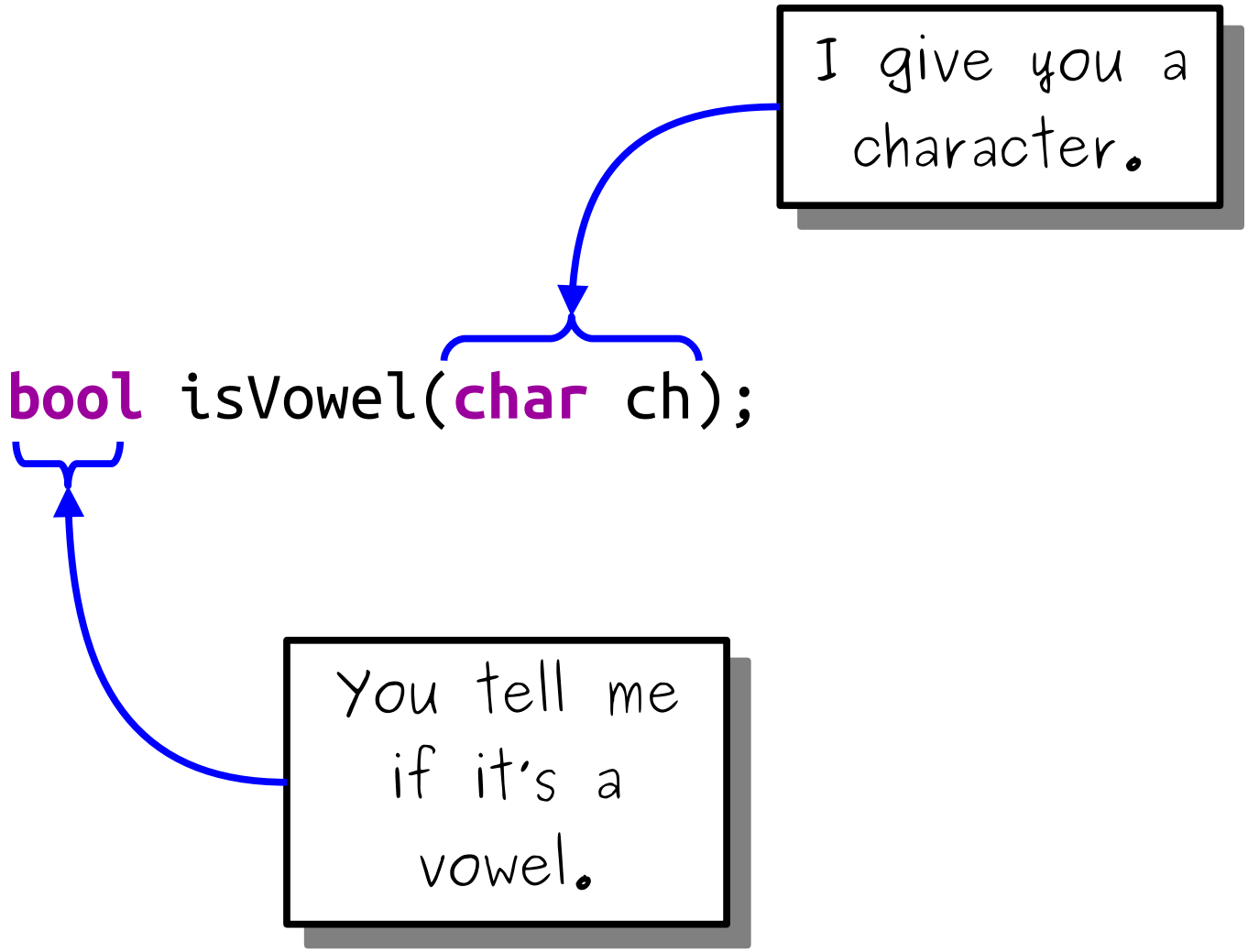
The Contract

```
bool isVowel(char ch);
```

The Contract



The Contract



The Contract

```
bool isVowel(char ch) {
    ch = toLowerCase(ch);
    return ch == 'a' ||
           ch == 'e' ||
           ch == 'i' ||
           ch == 'o' ||
           ch == 'u' ||
           ch == 'y';
}
```

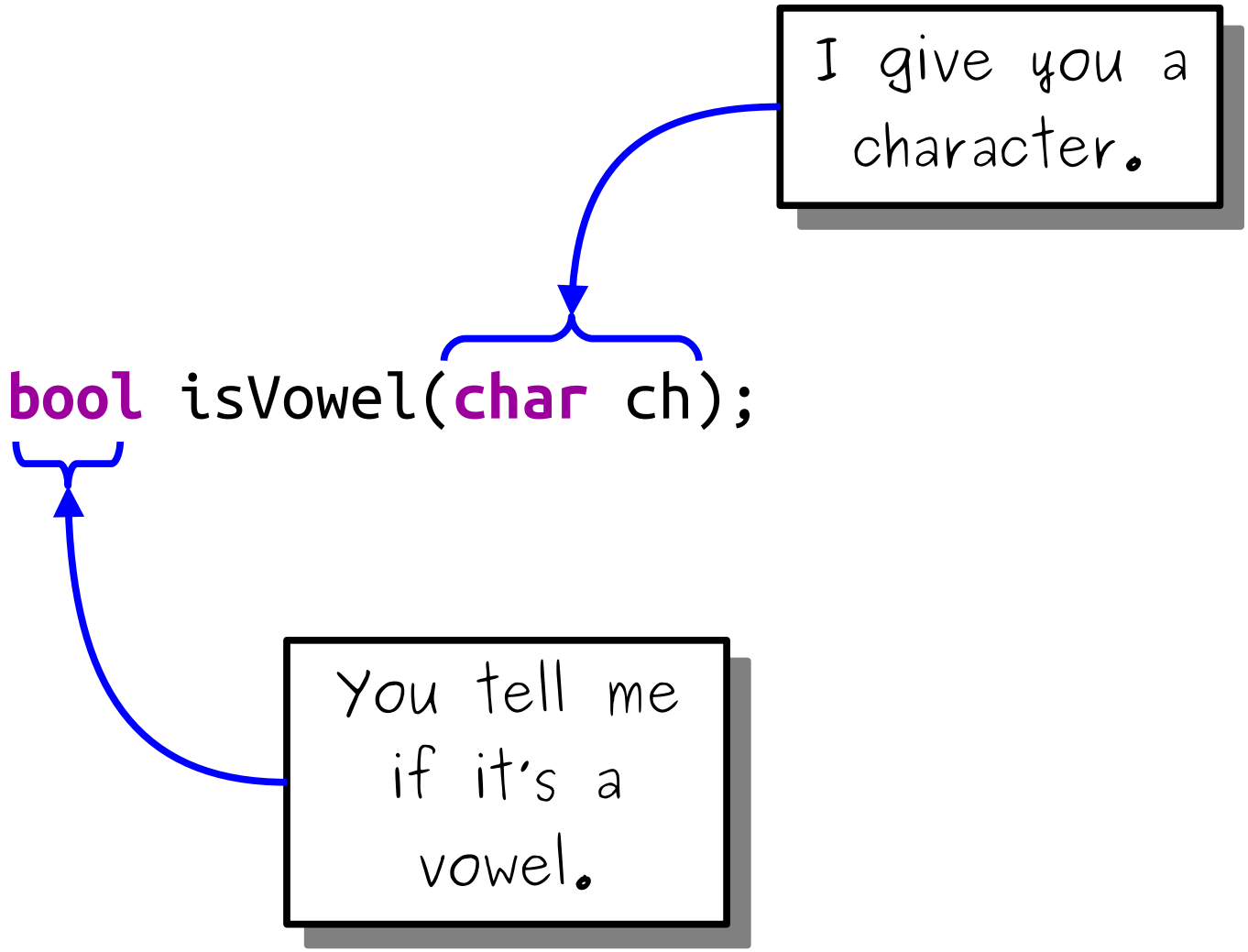
The Contract

```
bool isVowel(char ch) {
    switch(ch) {
        case 'A': case 'a':
        case 'E': case 'e':
        case 'I': case 'i':
        case 'O': case 'o':
        case 'U': case 'u':
        case 'Y': case 'y':
            return true;
        default:
            return false;
    }
}
```

The Contract

```
bool isVowel(char ch) {
    ch = tolower(ch);
    return string("aeiouy").find(ch) != string::npos;
}
```

The Contract

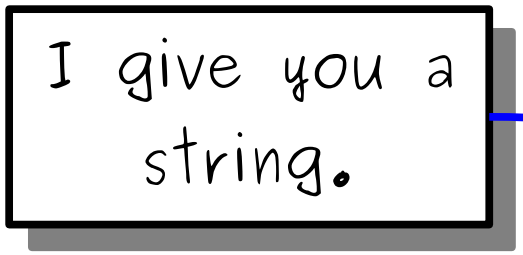


The Contract

The Contract

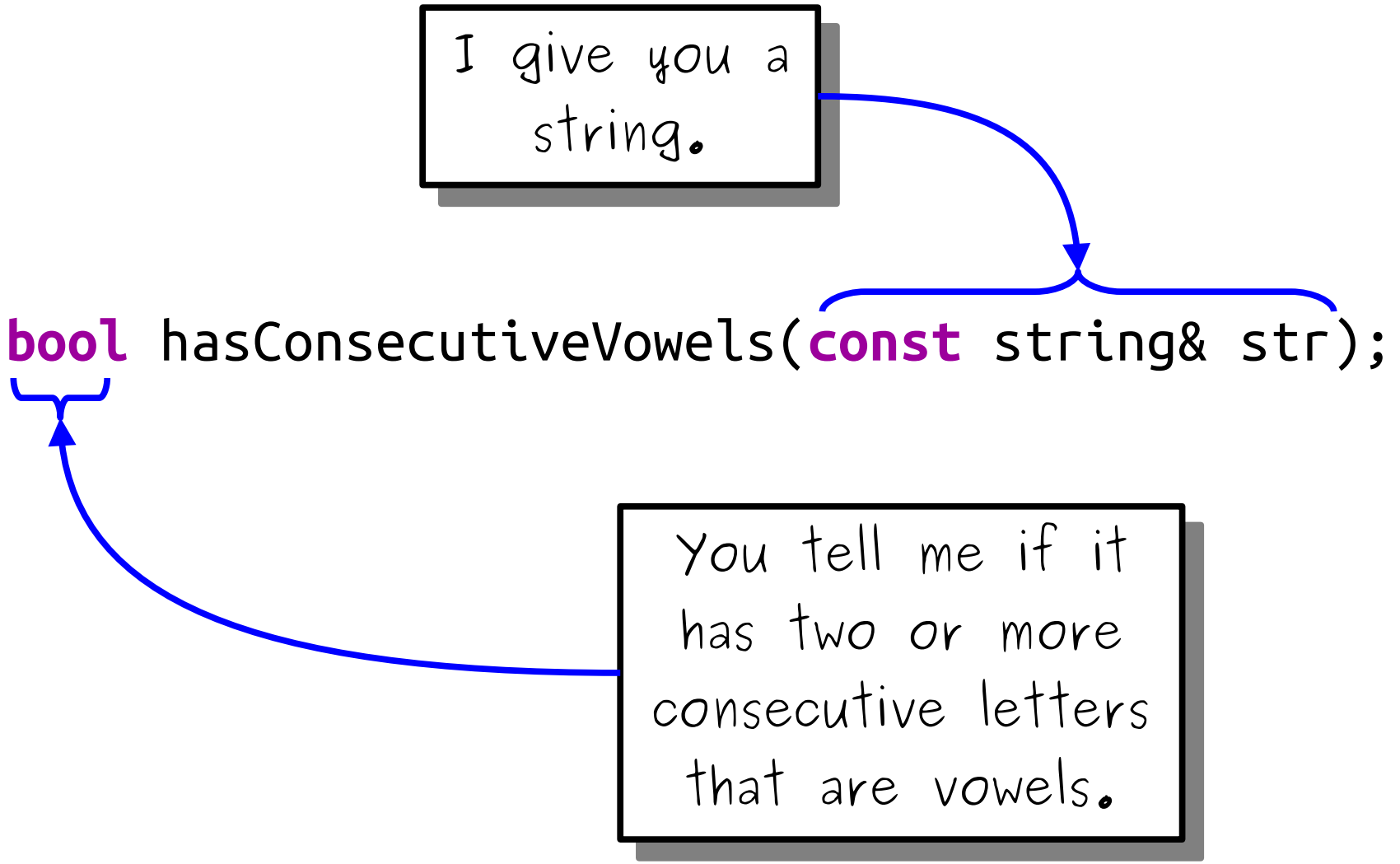
```
bool hasConsecutiveVowels(const string& str);
```

The Contract




```
bool hasConsecutiveVowels(const string& str);
```

The Contract



Trusting the Contract

```
bool isVowel(char ch);

bool hasConsecutiveVowels(const string& str) {




}
```

Trusting the Contract

```
bool isVowel(char ch);

bool hasConsecutiveVowels(const string& str) {
  for (int i = 1; i < str.length(); i++) {



  }

}
```

Trusting the Contract

```
bool isVowel(char ch);

bool hasConsecutiveVowels(const string& str) {
  for (int i = 1; i < str.length(); i++) {
    if (str[i - 1] is a vowel && str[i] is a vowel) {
       return true;
    }
  }

}
```

Trusting the Contract

```
bool isVowel(char ch);

bool hasConsecutiveVowels(const string& str) {
  for (int i = 1; i < str.length(); i++) {
    if (str[i - 1] is a vowel && str[i] is a vowel) {
       return true;
    }
  }
  return false;
}
```

Trusting the Contract

```
bool isVowel(char ch);

bool hasConsecutiveVowels(const string& str) {
  for (int i = 1; i < str.length(); i++) {
    if (isVowel(str[i - 1]) && isVowel(str[i])) {
       return true;
    }
  }
  return false;
}
```

Trusting the Contract

```
bool isVowel(char ch);

bool hasConsecutiveVowels(const string& str) {
  for (int i = 1; i < str.length(); i++) {
    if (isVowel(str[i - 1]) && isVowel(str[i])) {
       return true;
    }
  }
  return
}
```

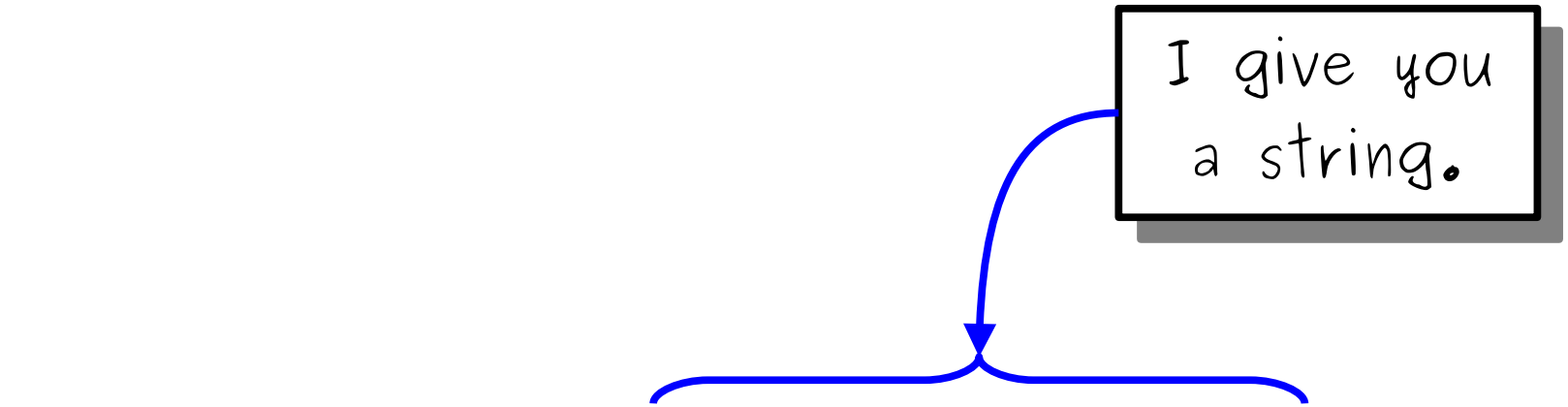
It doesn't matter how `isVowel` is implemented. We just trust that it works.

The Contract

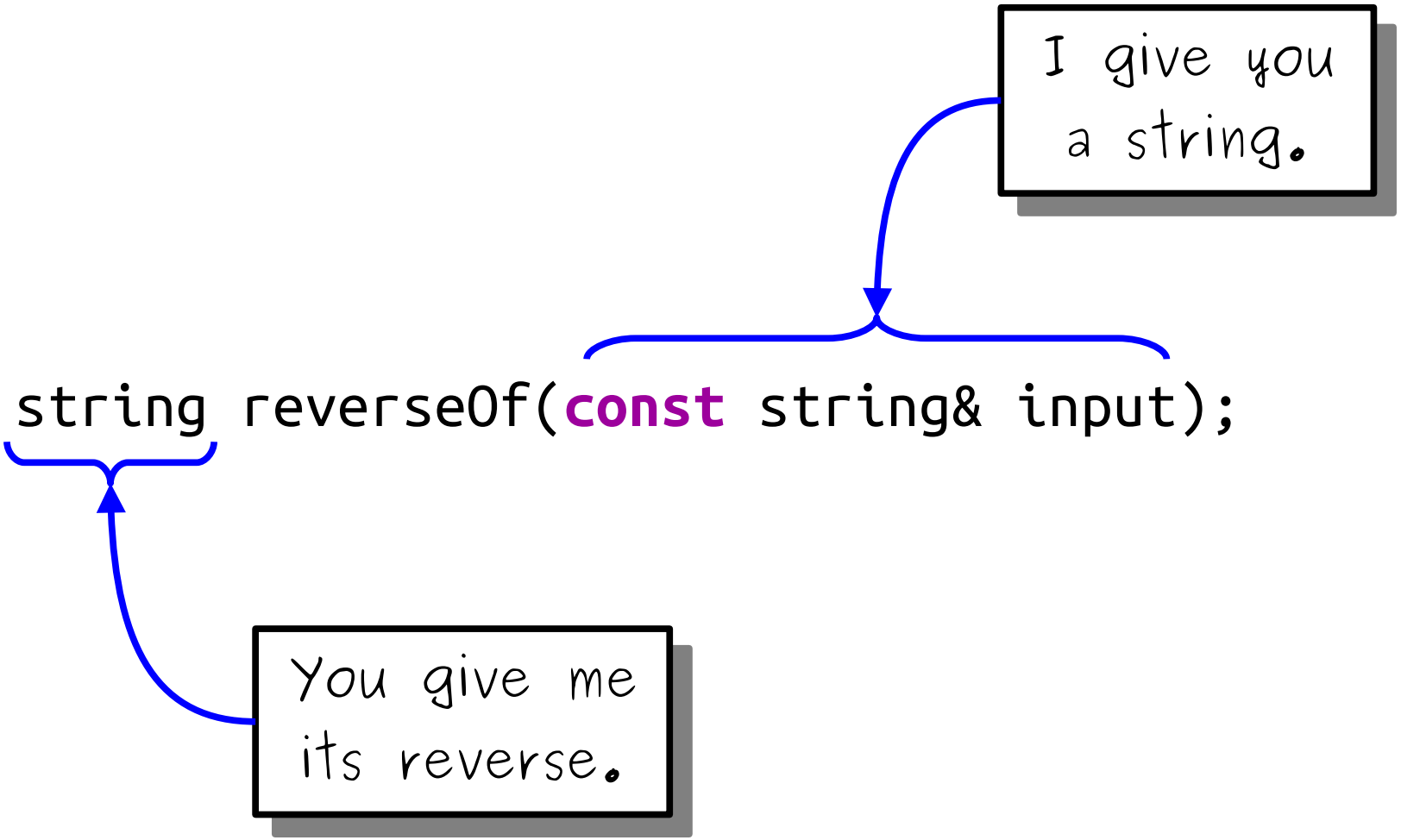
The Contract

```
string reverseOf(const string& input);
```

The Contract



The Contract



Trusting the Contract

```
string reverseOf(const string& input);

string reverseOf(const string& input) {



}
```

Trusting the Contract

```
string reverseOf(const string& input);

string reverseOf(const string& input) {
    if (input == "") {

    } else {

    }
}
```

Trusting the Contract

```
string reverseOf(const string& input);

string reverseOf(const string& input) {
    if (input == "") {
        return "";
    } else {

    }
}
```

Trusting the Contract

```
string reverseOf(const string& input);

string reverseOf(const string& input) {
    if (input == "") {
        return "";
    } else {
        return the reverse of input.substr(1) + input[0];
    }
}
```

Trusting the Contract

```
string reverseOf(const string& input);

string reverseOf(const string& input) {
    if (input == "") {
        return "";
    } else {
        return reverseOf(input.substr(1)) + input[0];
    }
}
```

Trusting the Contract

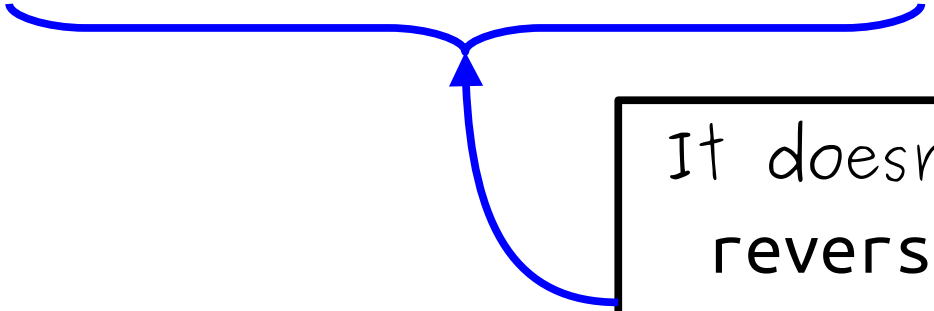
```
string reverseOf(const string& input);

string reverseOf(const string& input) {
    if (input == "") {
        return "";
    } else {
        return reverseOf(input.substr(1)) + input[0];
    }
}
```

Trusting the Contract

```
string reverseOf(const string& input);

string reverseOf(const string& input) {
    if (input == "") {
        return "";
    } else {
        return reverseOf(input.substr(1)) + input[0];
    }
}
```



It doesn't matter how `reverseOf` reverses the string. It just matters that it does.

The Contract

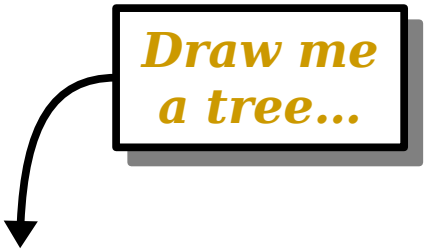
The Contract

```
void drawTree(GWindow& window,
              double x, double y,
              double height,
              double angle,
              int order);
```

The Contract

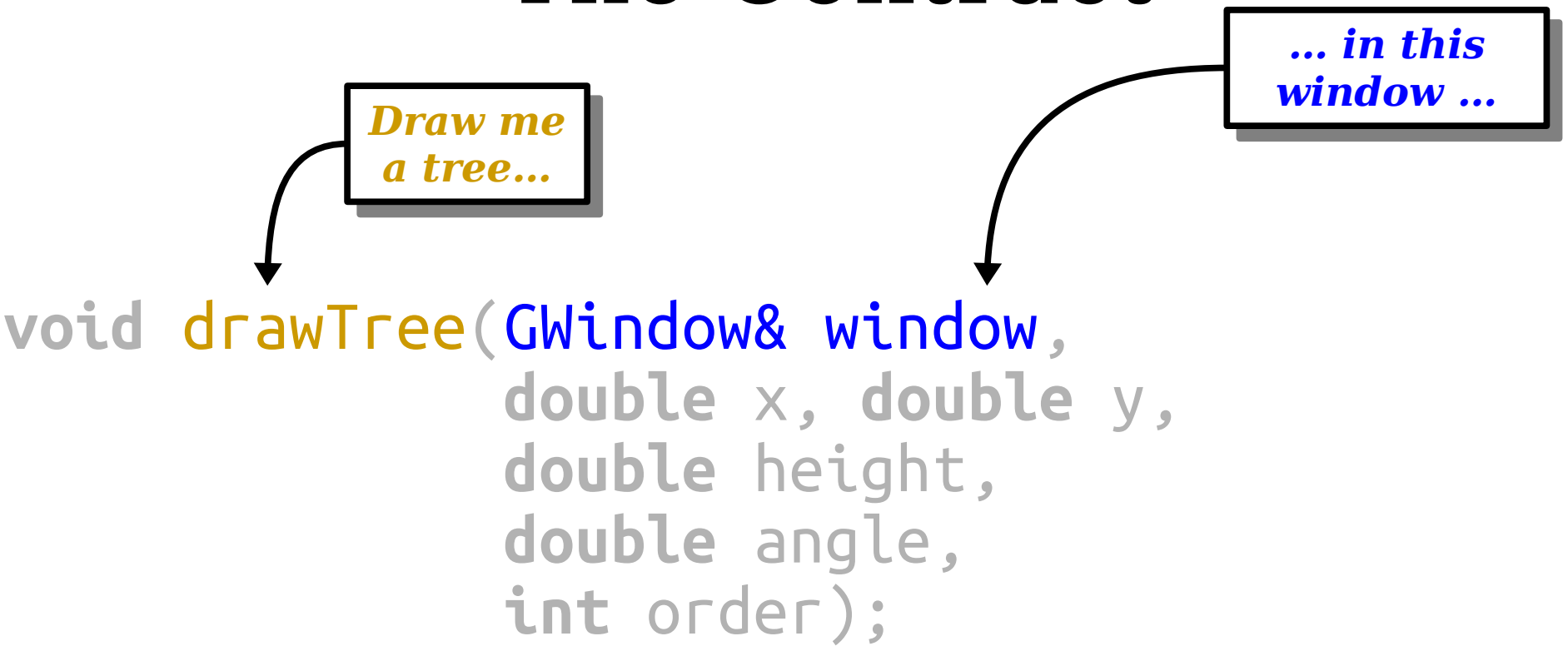
```
void drawTree(GWindow& window,
              double x, double y,
              double height,
              double angle,
              int order);
```

The Contract

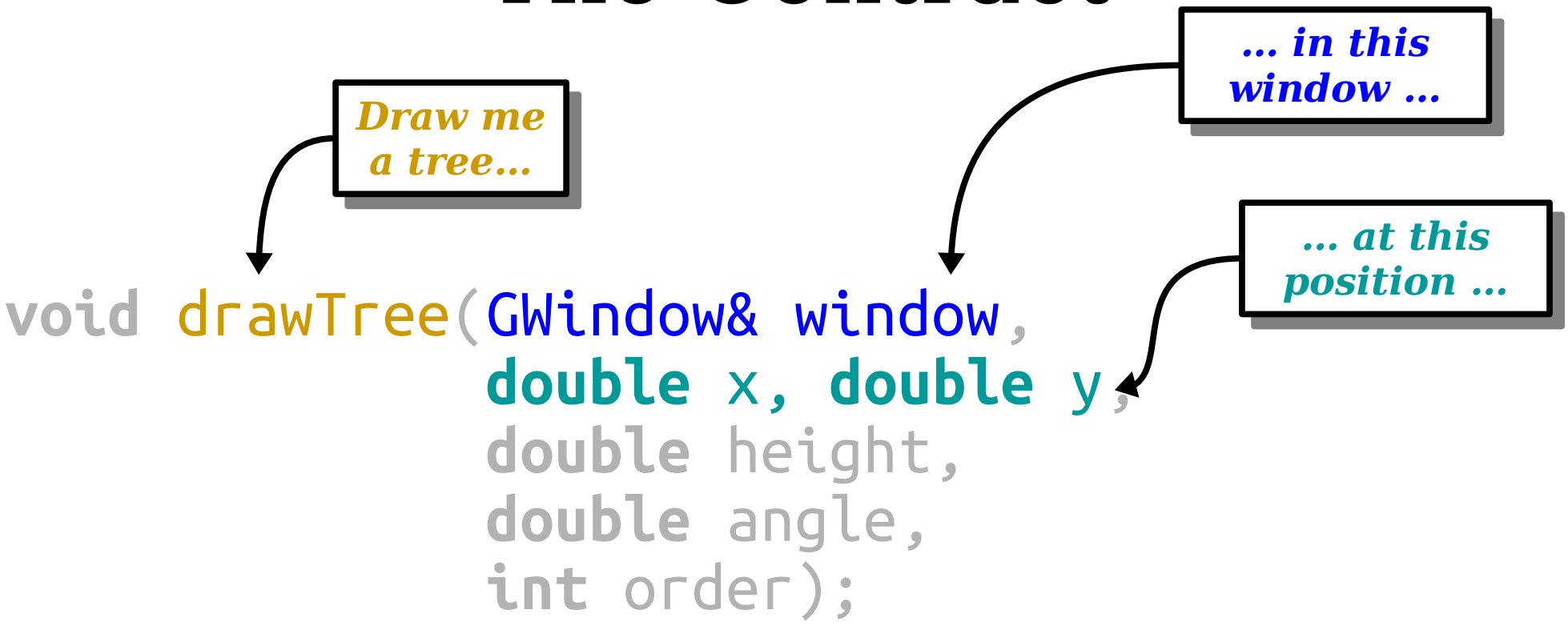



```
void drawTree(GWindow& window,
              double x, double y,
              double height,
              double angle,
              int order);
```

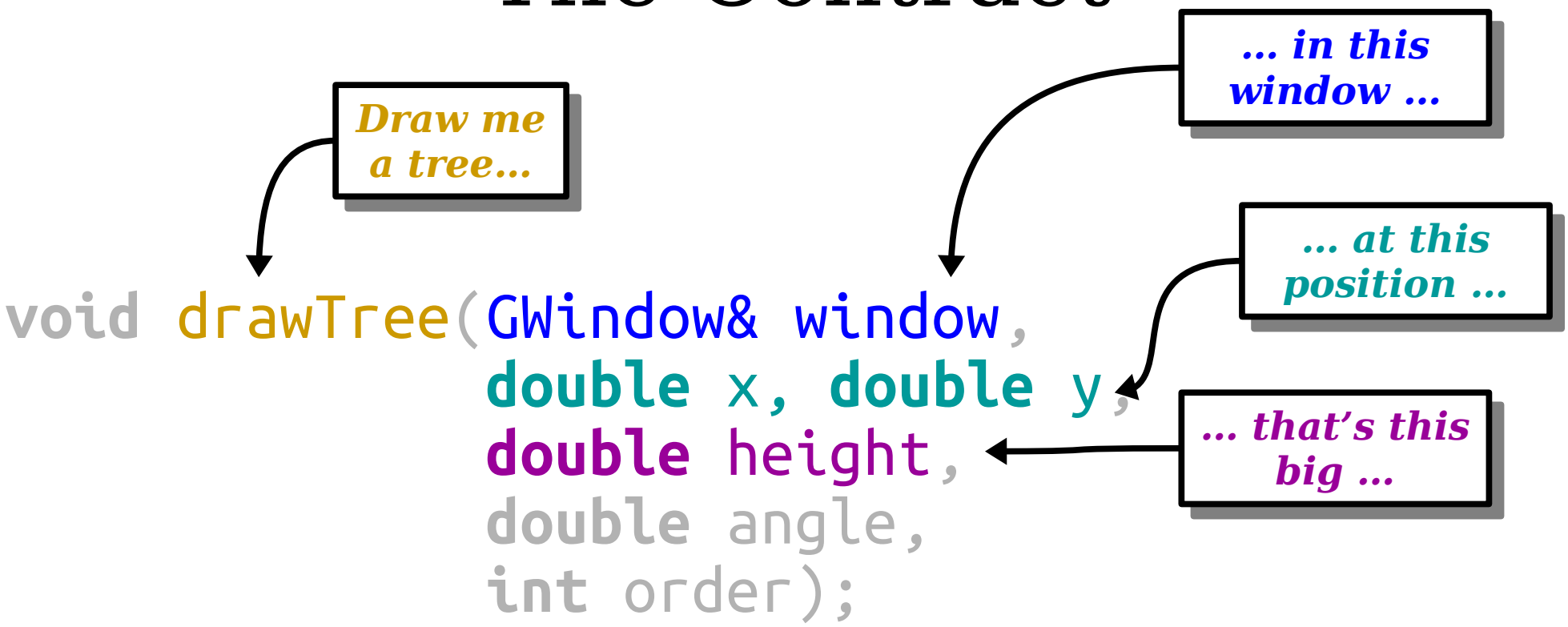
The Contract



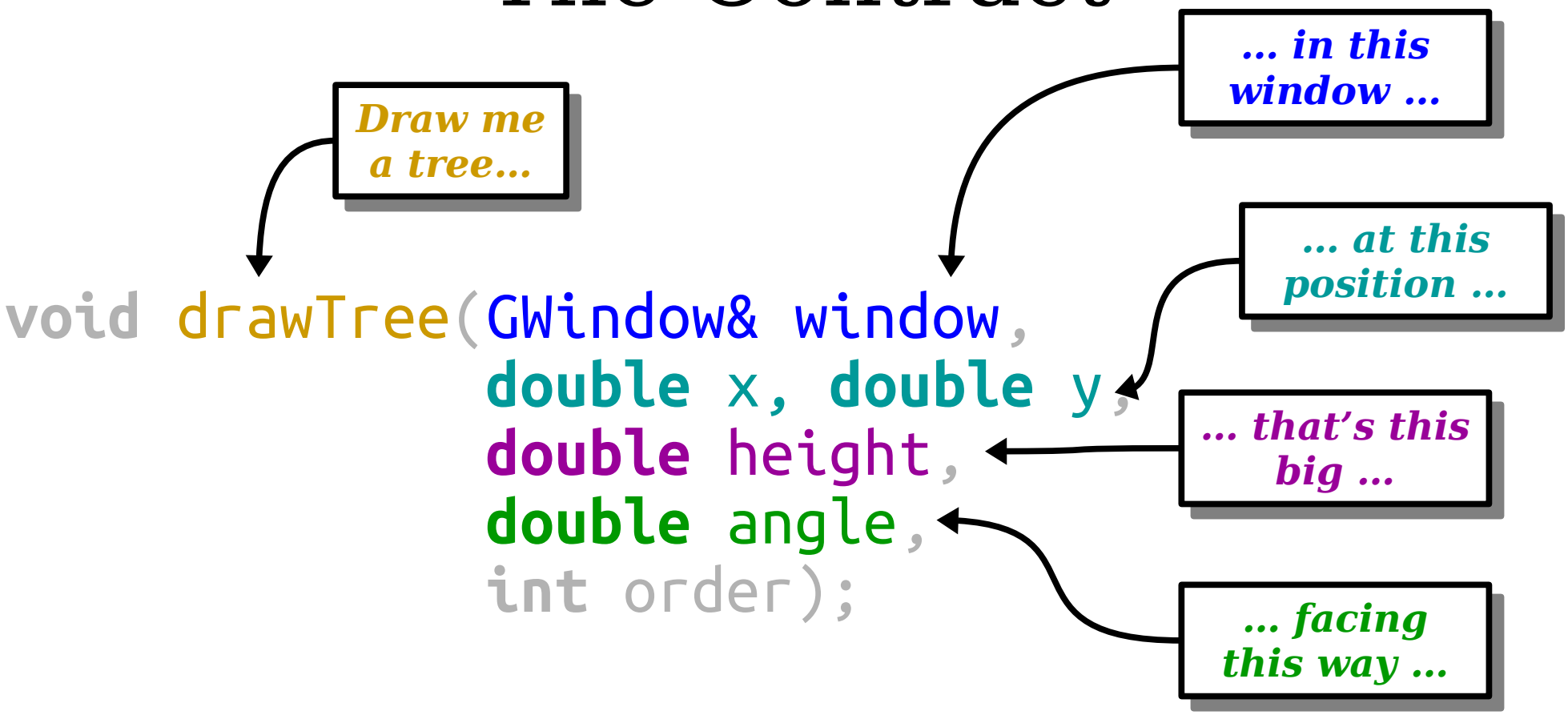
The Contract



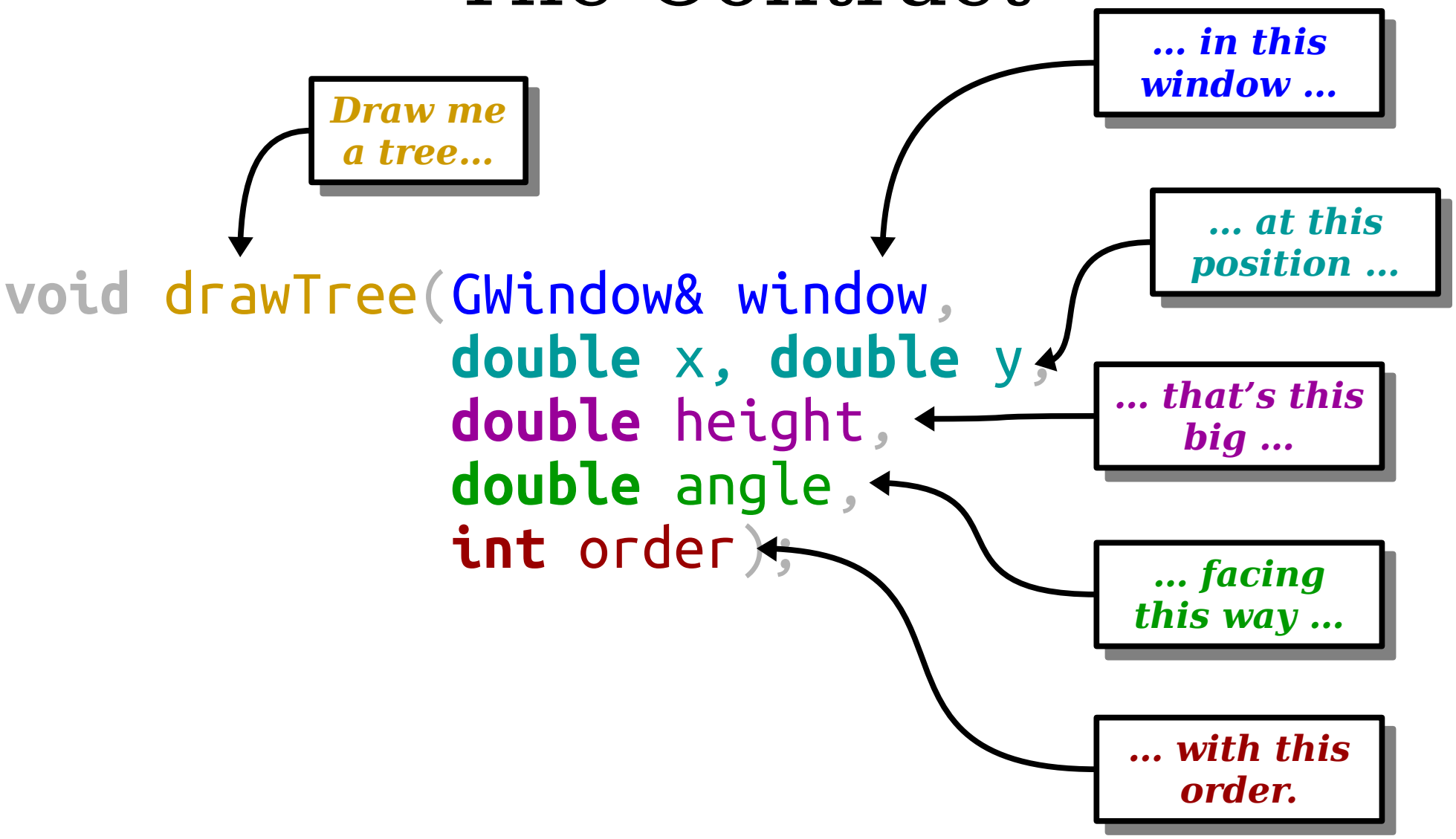
The Contract



The Contract



The Contract



Trusting the Contract

```
void drawTree(GWindow& window,
              double x, double y,
              double height, double angle,
              int order);

void drawTree(GWindow& window,
              double x, double y,
              double height, double angle,
              int order) {









}
```

Trusting the Contract

```
void drawTree(GWindow& window,
              double x, double y,
              double height, double angle,
              int order);

void drawTree(GWindow& window,
              double x, double y,
              double height, double angle,
              int order) {
    if (order == 0) return;




}
```

Trusting the Contract

```
void drawTree(GWindow& window,
              double x, double y,
              double height, double angle,
              int order);

void drawTree(GWindow& window,
              double x, double y,
              double height, double angle,
              int order) {
    if (order == 0) return;

    GPoint endpoint = drawPolarLine(/* … */);



}
```

Trusting the Contract

```
void drawTree(GWindow& window,
              double x, double y,
              double height, double angle,
              int order);

void drawTree(GWindow& window,
              double x, double y,
              double height, double angle,
              int order) {
    if (order == 0) return;

    GPoint endpoint = drawPolarLine(/* … */);

    draw a tree angling to the left
    draw a tree angling to the right
}
```

Trusting the Contract

```
void drawTree(GWindow& window,
              double x, double y,
              double height, double angle,
              int order);

void drawTree(GWindow& window,
              double x, double y,
              double height, double angle,
              int order) {
    if (order == 0) return;

    GPoint endpoint = drawPolarLine(/* … */);

    drawTree(/* … */);
    drawTree(/* … */);
}
```

Trusting the Contract

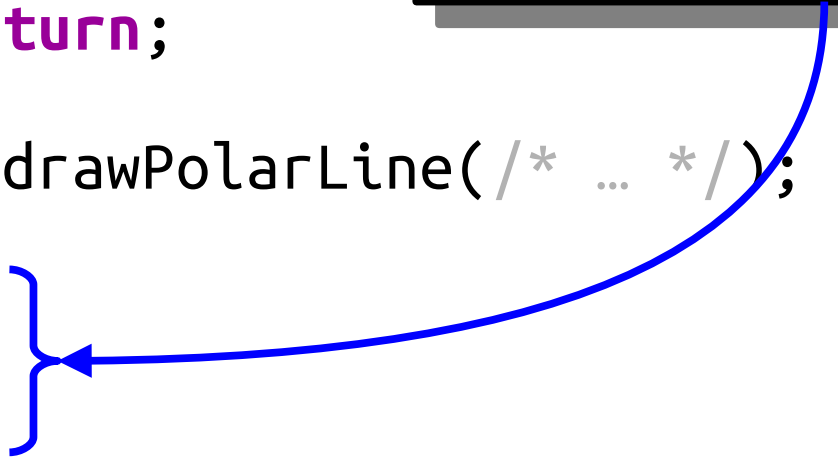
```
void drawTree(GWindow& window,
              double x, double y,
              double height, double angle,
              int order);

void drawTree(GWindow& window,
              double x, double y,
              double height, doubl
              int order) {
    if (order == 0) return;

    GPoint endpoint = drawPolarLine(/* … */);

    drawTree(/* … */);
    drawTree(/* … */);
}
```

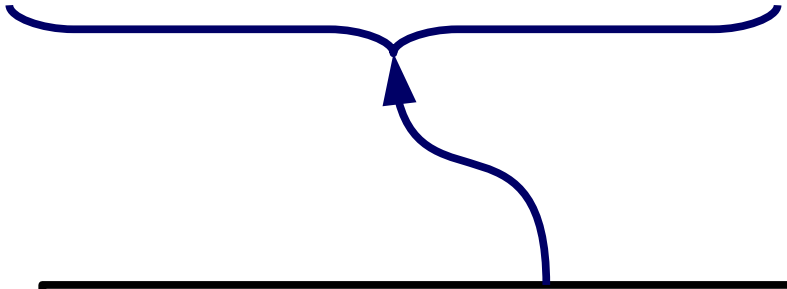
It doesn't matter how **drawTree** draws a tree. It just matters that it does.



The Recursive Leap of Faith

- When writing a recursive function, it helps to take a ***recursive leap of faith***.
- Before writing the function, answer these questions:
  - What does the function take in?
  - What does it return?
- Then, as you're writing the function, trust that your recursive calls to the function just "work" without asking how.
- This can take some adjustment to get used to, but is a necessary skill for writing more complex recursive functions.

Recursive Enumeration



e·nu·mer·a·tion

noun

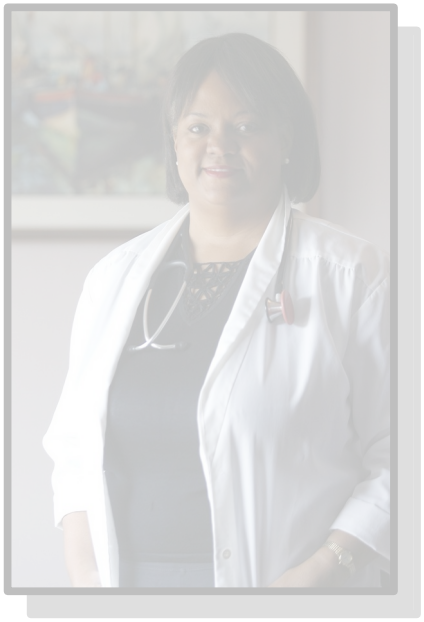
The act of mentioning a number of things one by one.

(Source: Google)

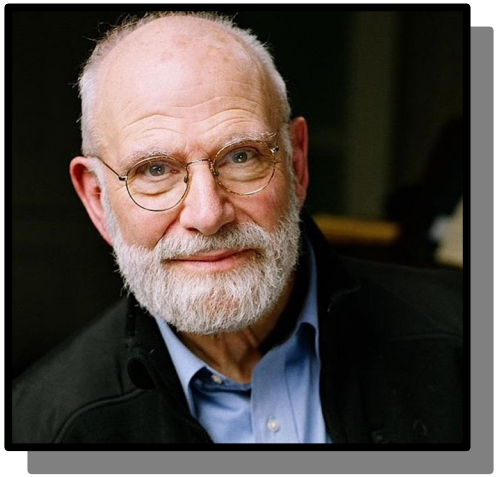
You need to send an emergency team of doctors to an area.

You know which doctors you have available to send.

List all the possible teams you can make from your list of all the doctors.



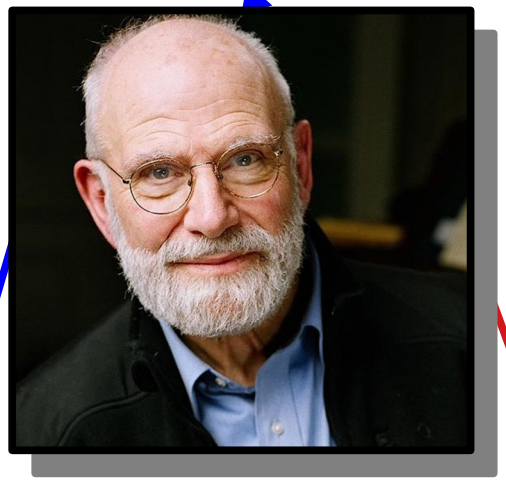

NYU
Medical
Center

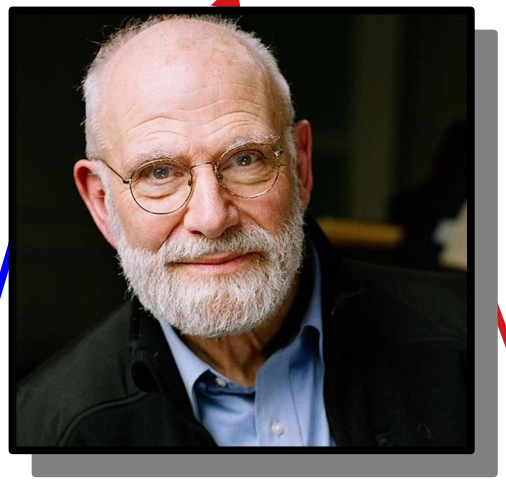



NYU
Medical
Center

✓

×





✓

×

✓

×

…

…

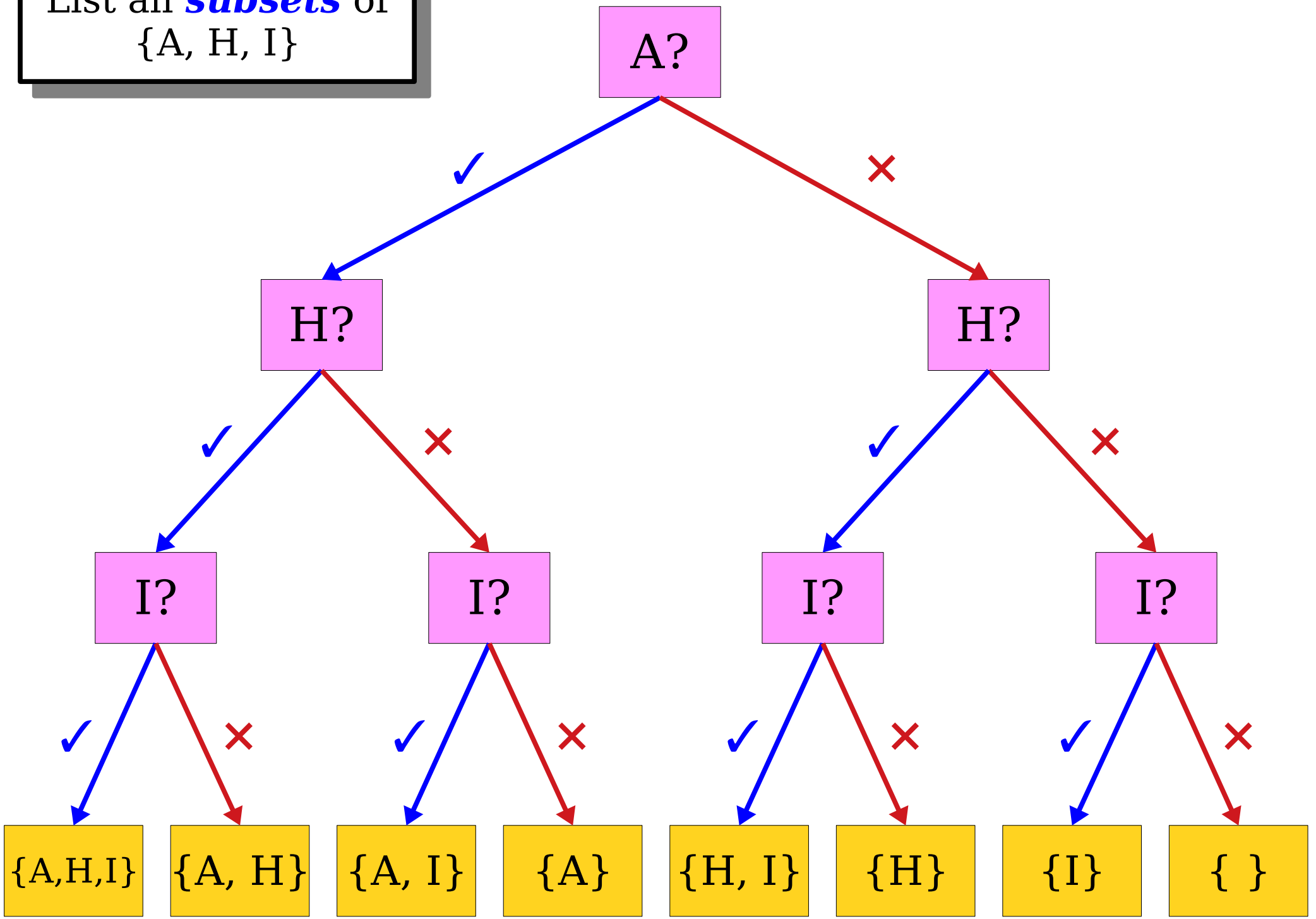
…

…

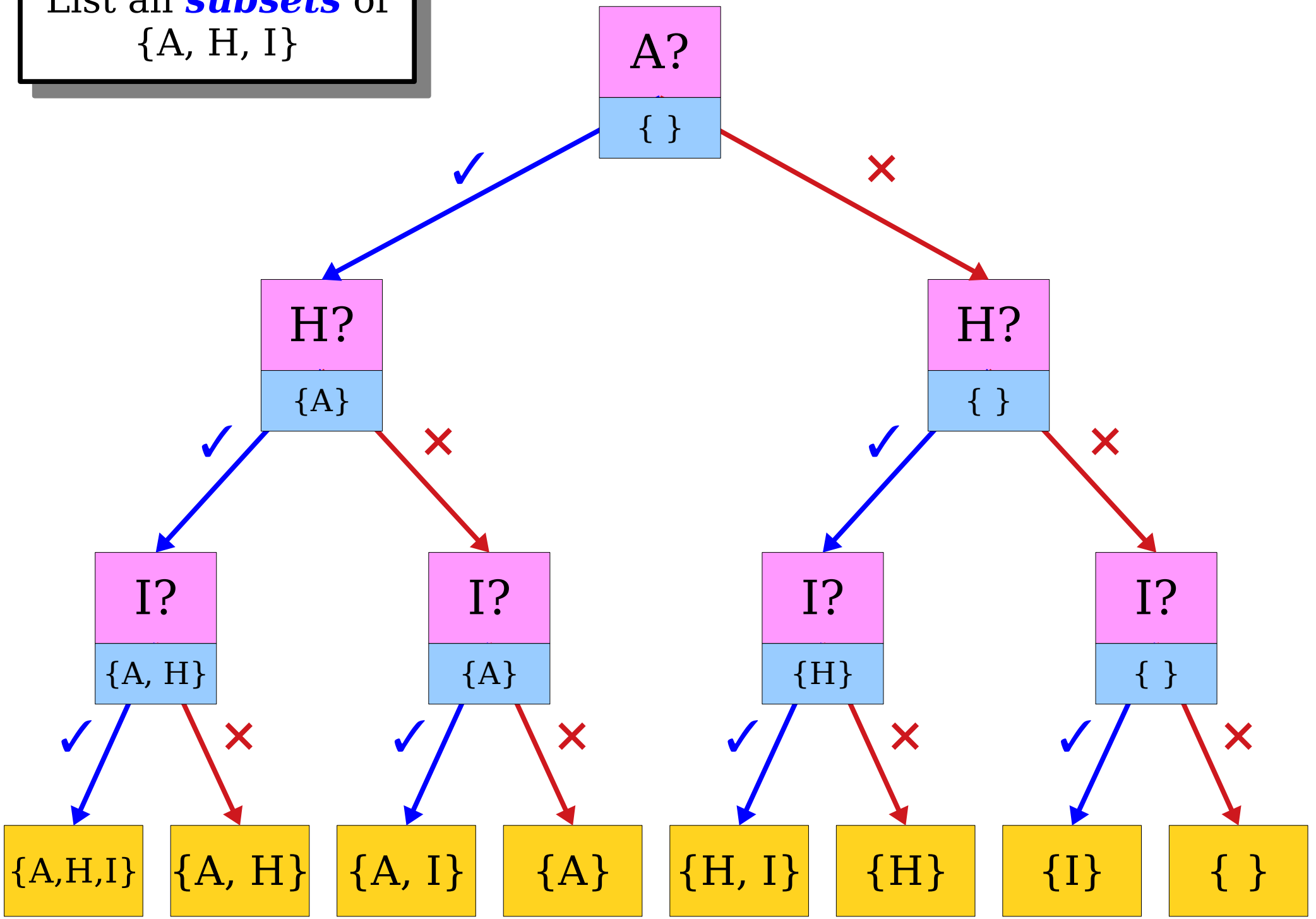
This structure is called a ***decision tree***.

List all ***subsets*** of {A, H, I}

List all ***subsets*** of {A, H, I}



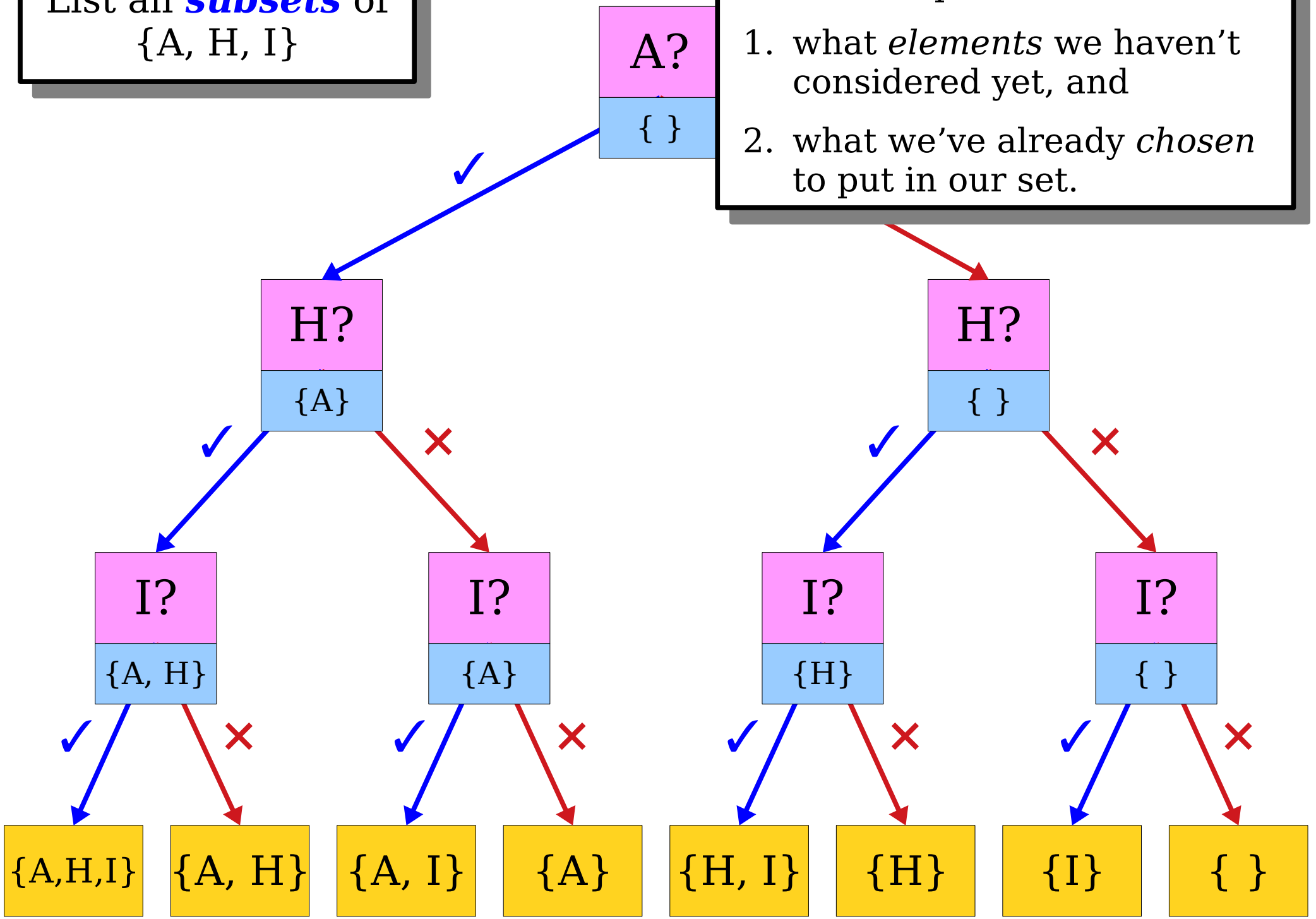
List all ***subsets*** of {A, H, I}



List all ***subsets*** of {A, H, I}

At each step, we need to know

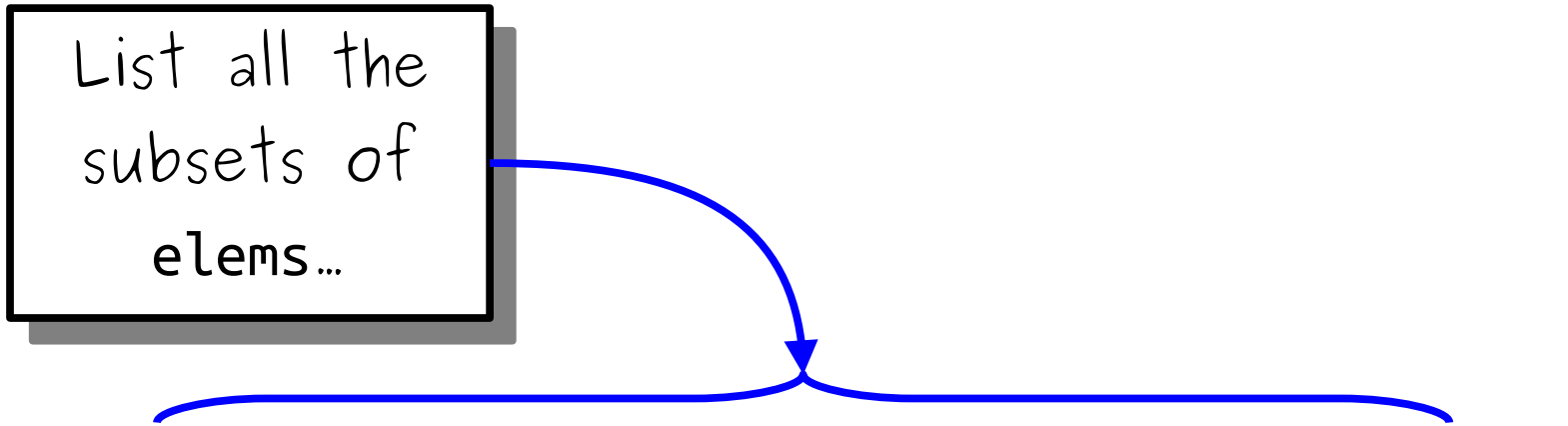
1. what *elements* we haven't considered yet, and
2. what we've already *chosen* to put in our set.



The Contract

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& chosen);
```

The Contract




```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& chosen);
```

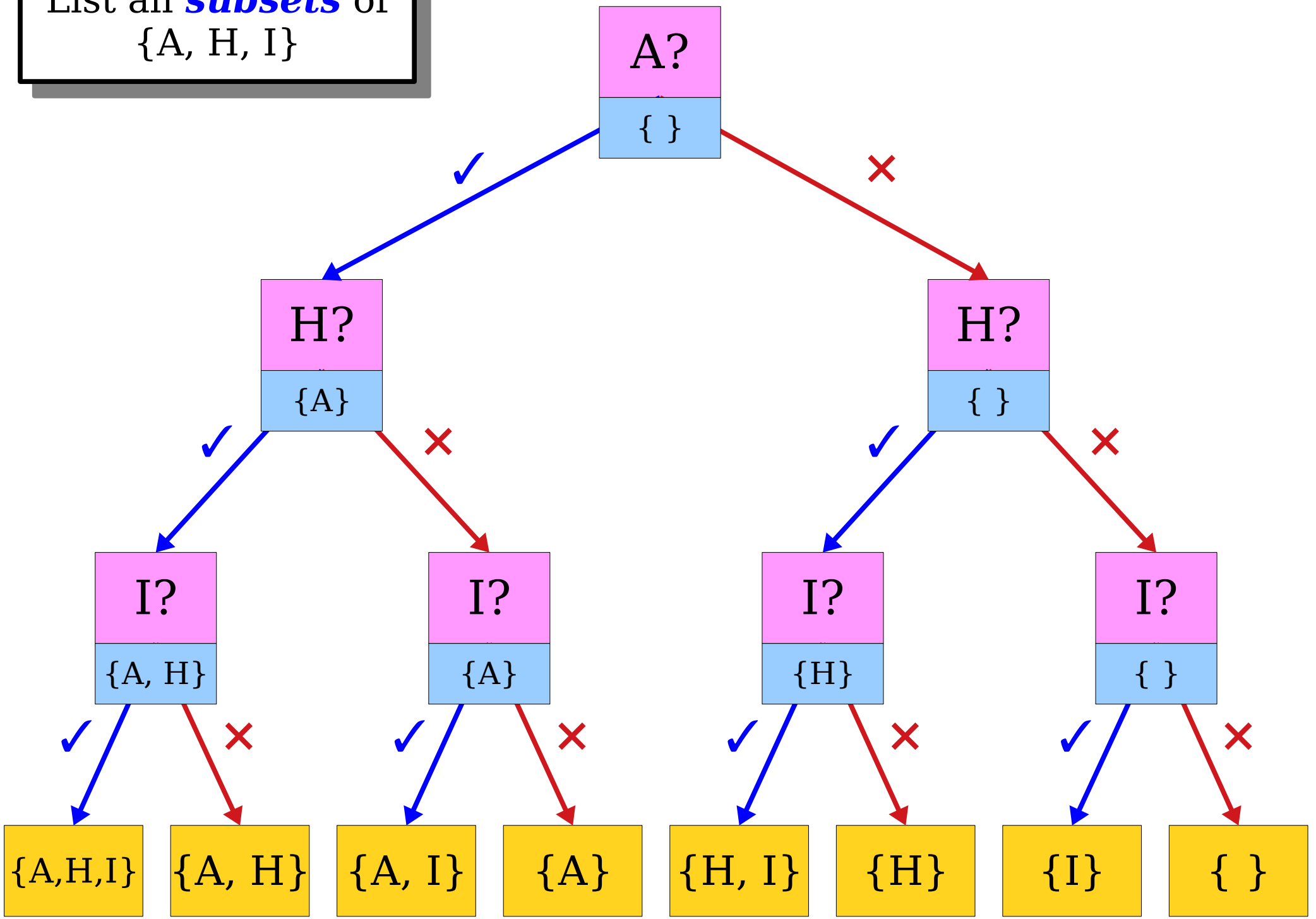
The Contract

List all the subsets of **elems**…

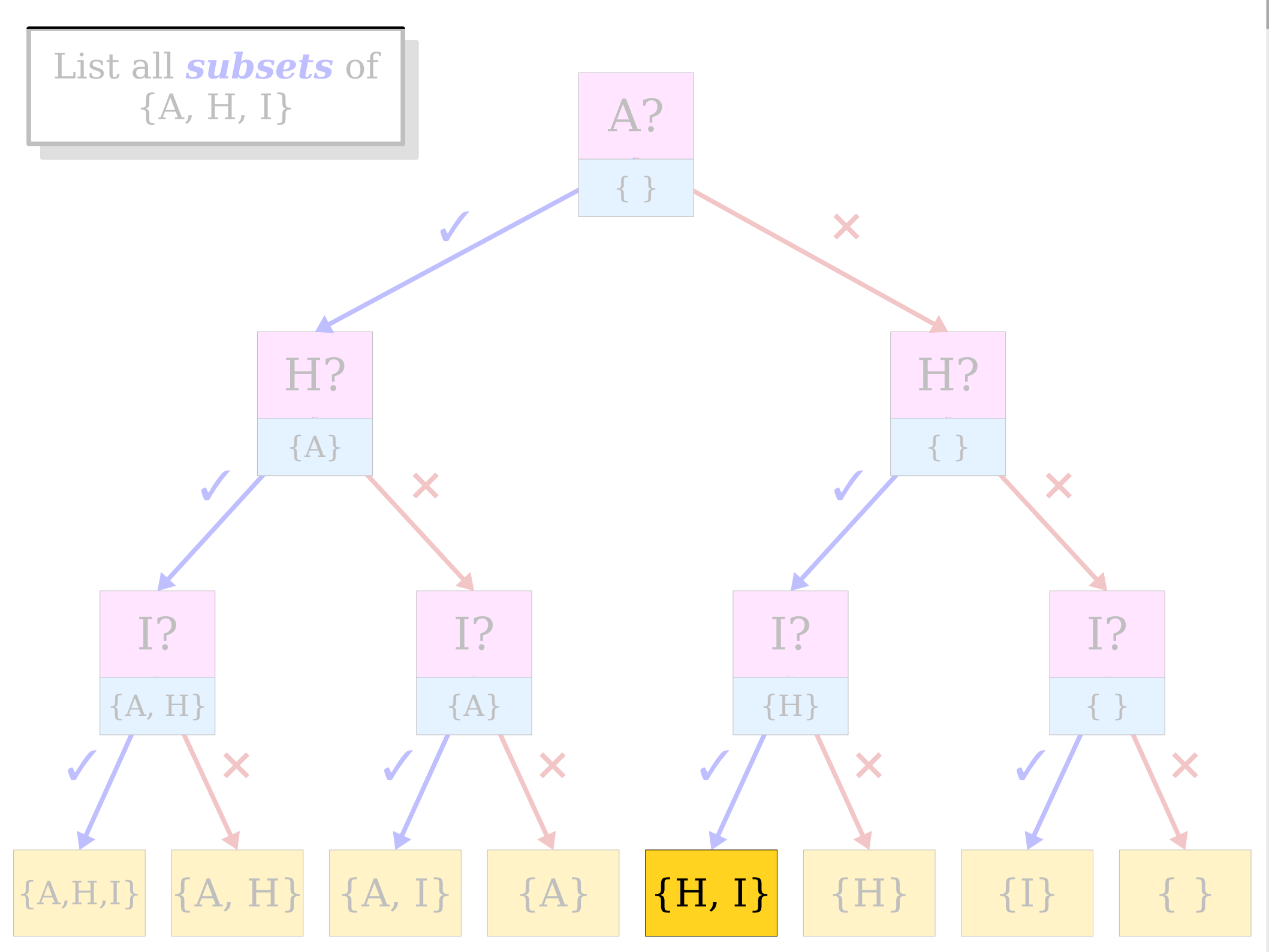
```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& chosen);
```

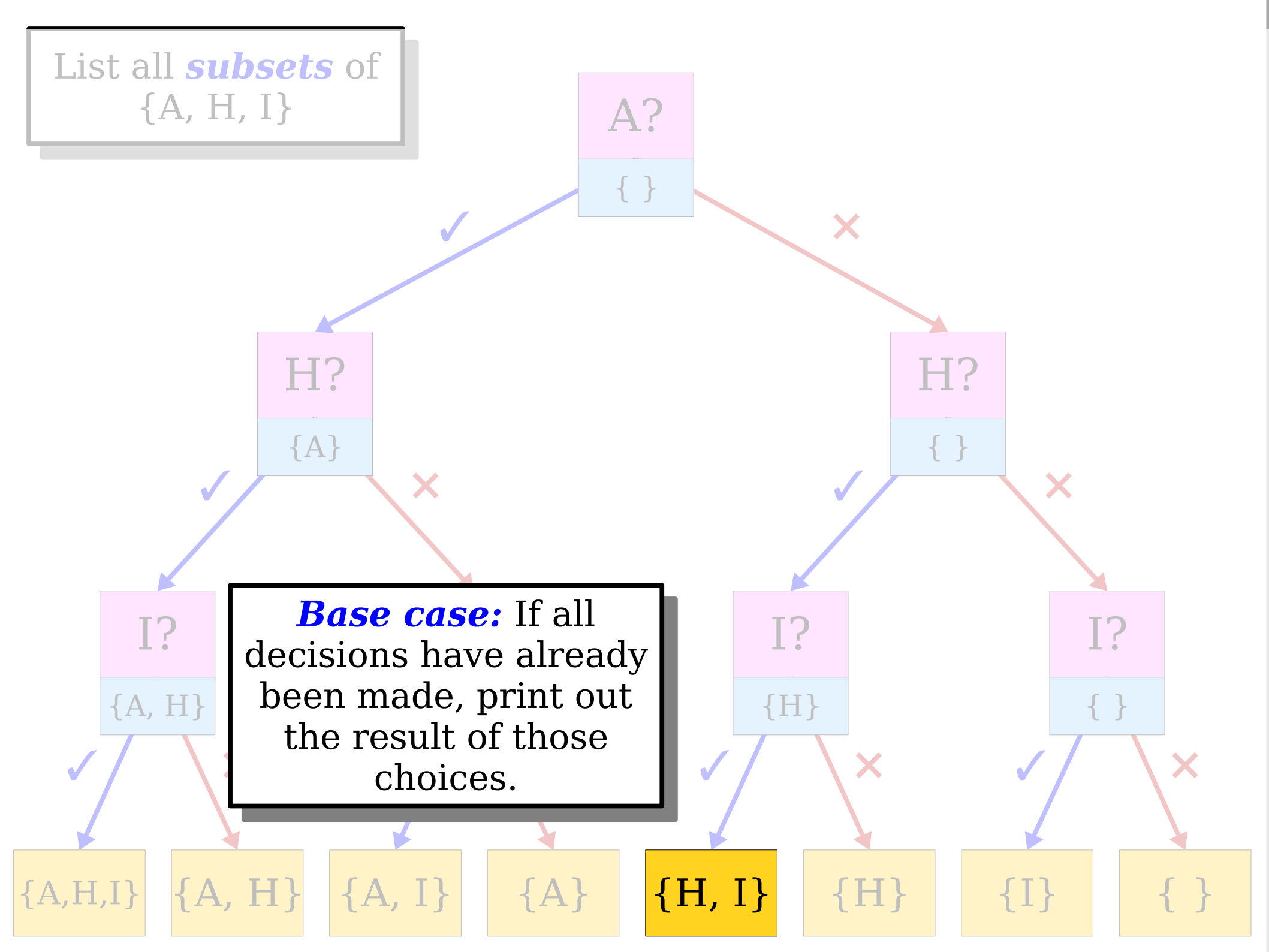
… given that we've already committed to choosing the integers in **chosen**.

List all ***subsets*** of {A, H, I}

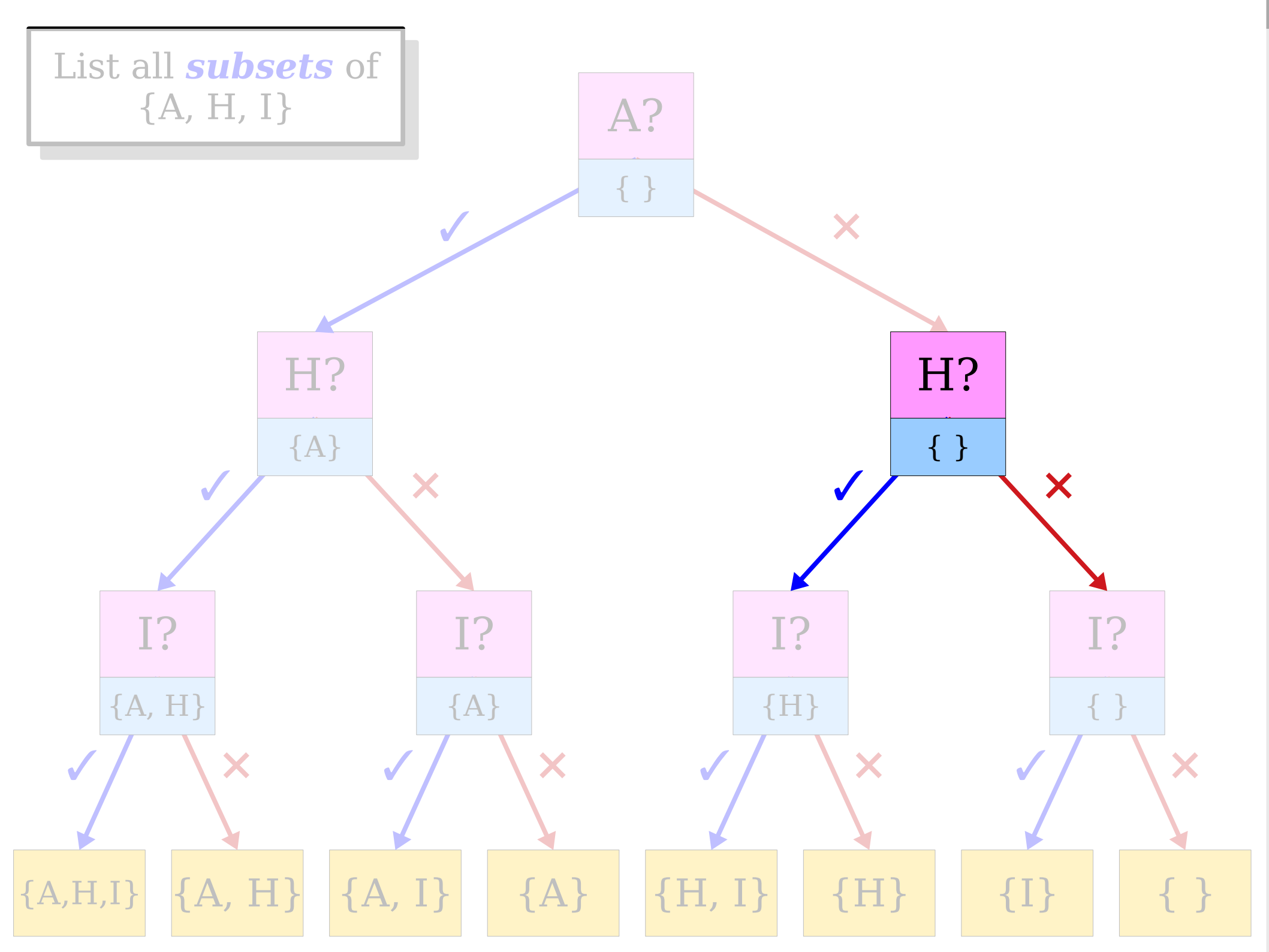


List all ***subsets*** of {A, H, I}

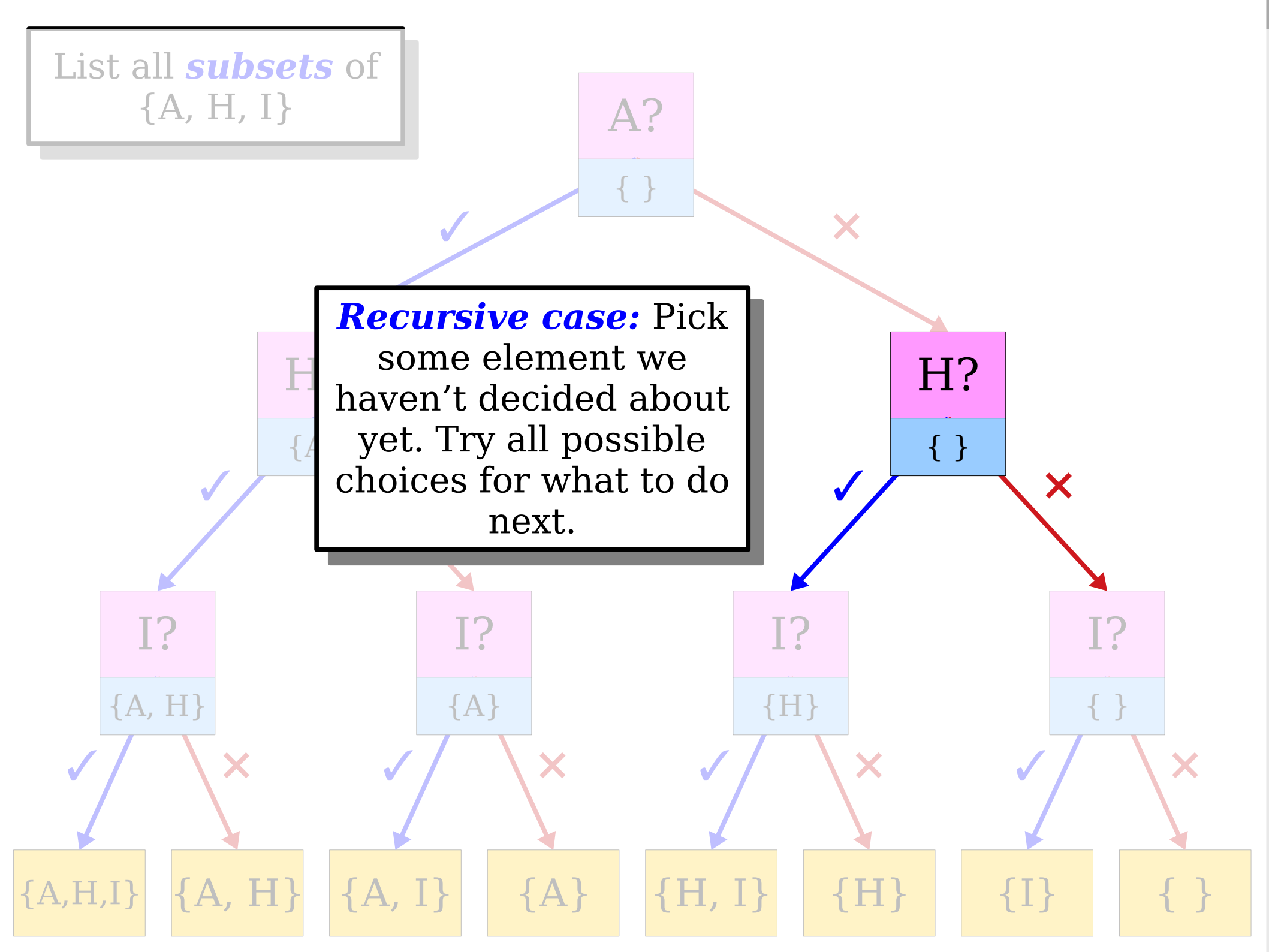



List all subsets of {A, H, I}
A?
{ }
✓
×
H?
{A}
H?
{ }
✓
×
✓
×
I?
{A, H}
I?
{H}
I?
{ }
Base case: If all decisions have already been made, print out the result of those choices.
✓
×
✓
×
✓
×
{A,H,I}
{A, H}
{A, I}
{A}
{H, I}
{H}
{I}
{ }

List all ***subsets*** of {A, H, I}

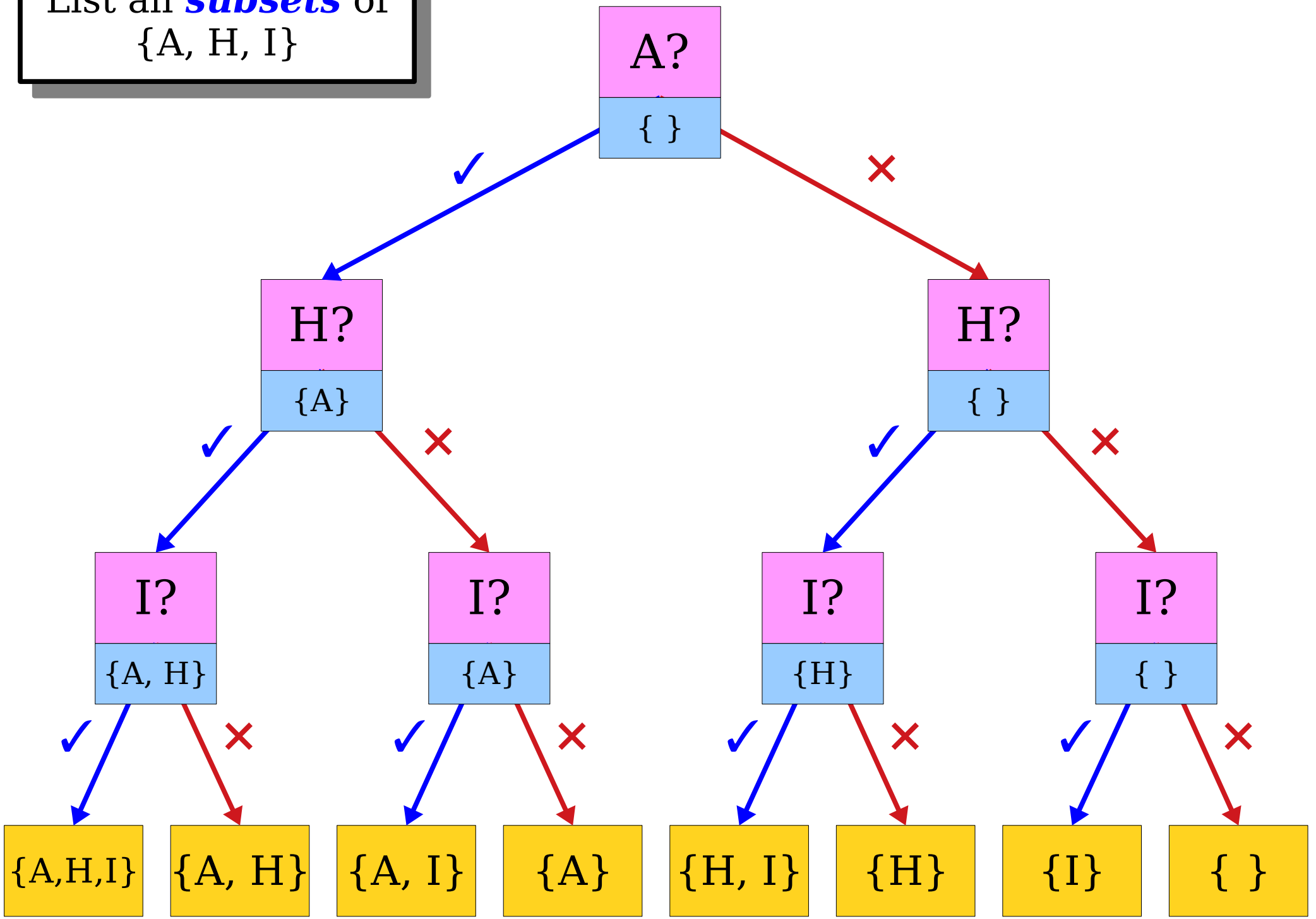


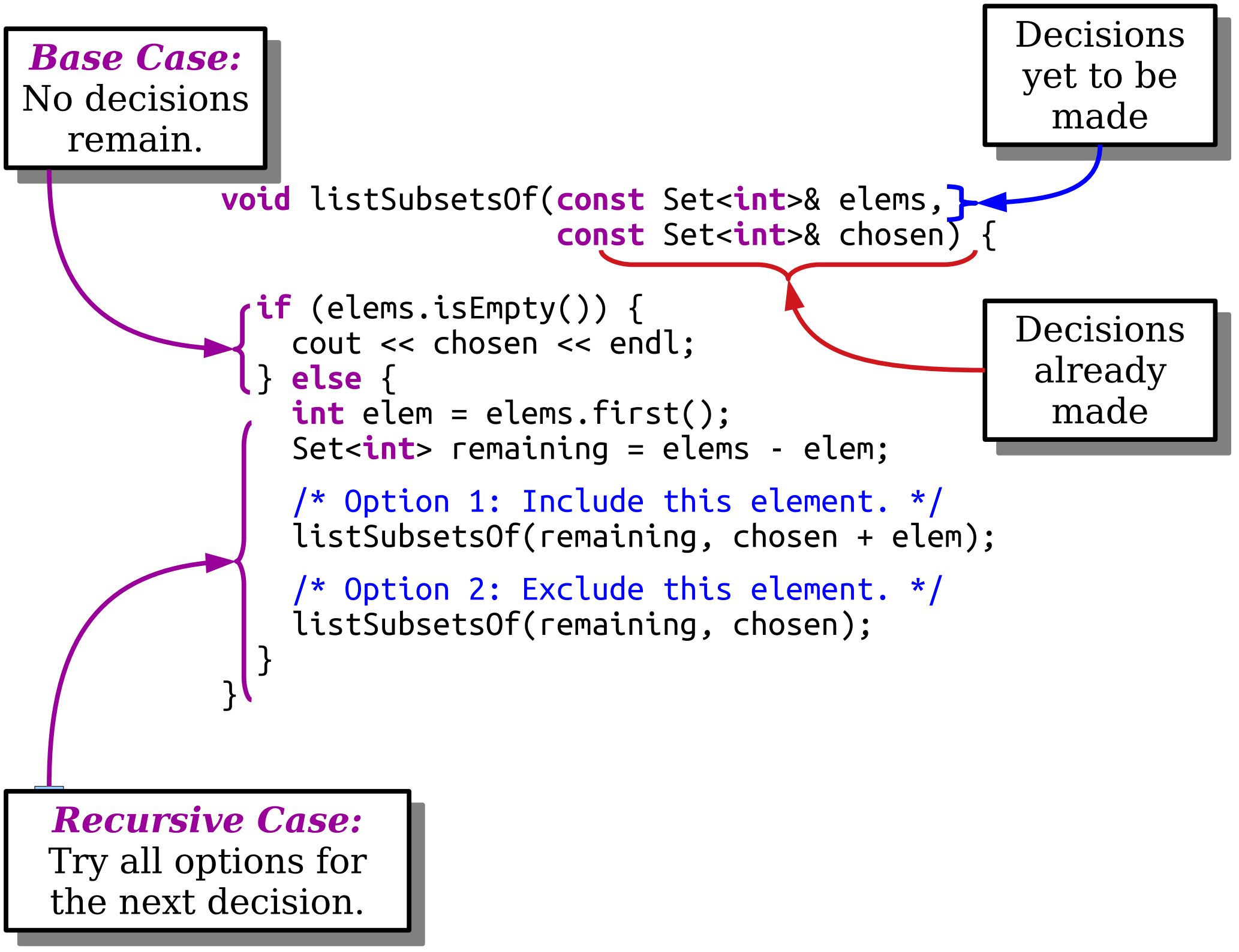
List all ***subsets*** of {A, H, I}



Recursive case: Pick some element we haven't decided about yet. Try all possible choices for what to do next.

List all ***subsets*** of {A, H, I}



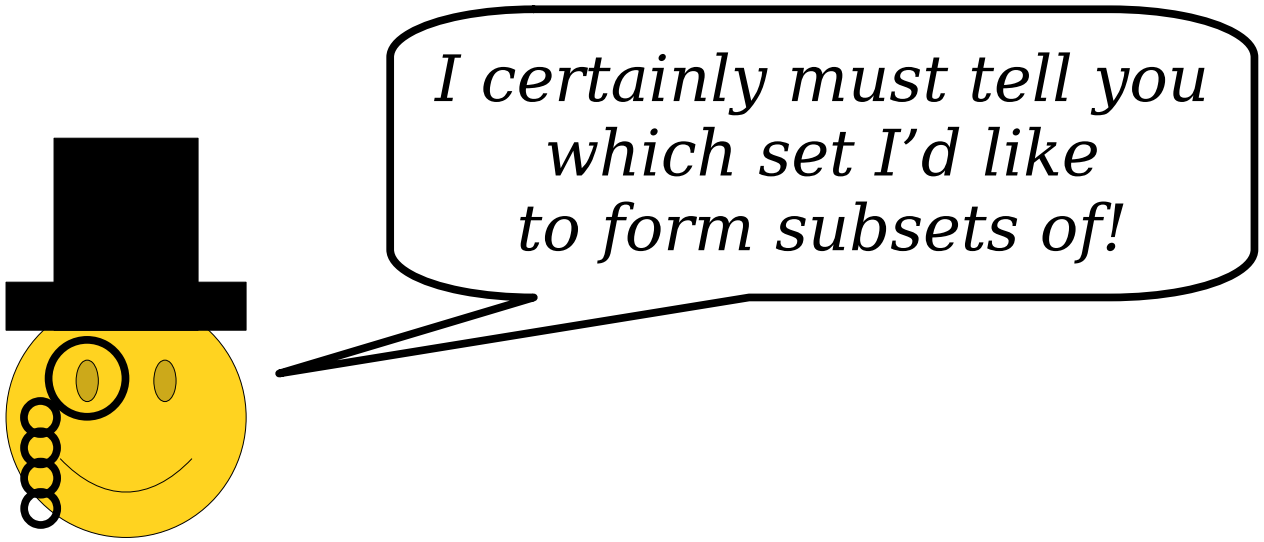

Base Case:
No decisions remain.
Decisions yet to be made
void listSubsetsOf(const Set<int>& elems,
const Set<int>& chosen) {
if (elems.isEmpty()) {
cout << chosen << endl;
} else {
int elem = elems.first();
Set<int> remaining = elems - elem;
/* Option 1: Include this element. */
listSubsetsOf(remaining, chosen + elem);
/* Option 2: Exclude this element. */
listSubsetsOf(remaining, chosen);
}
}
Decisions already made
Recursive Case:
Try all options for the next decision.

A Question of Parameters

```
listSubsetsOf({1, 2, 3}, {});
```

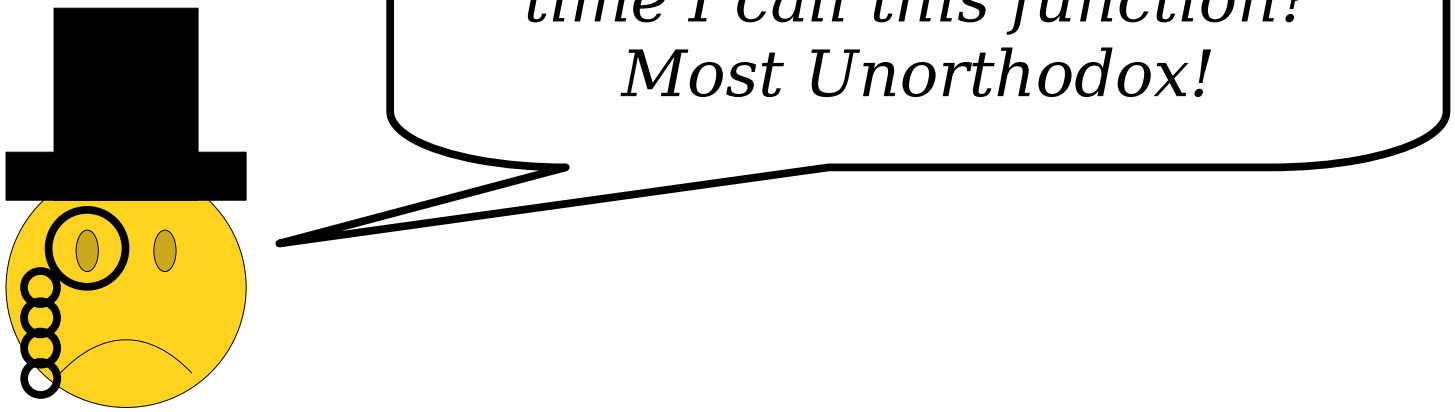
```
listSubsetsOf({1, 2, 3}, {});
```

```
listSubsetsOf({1, 2, 3}, {});
```



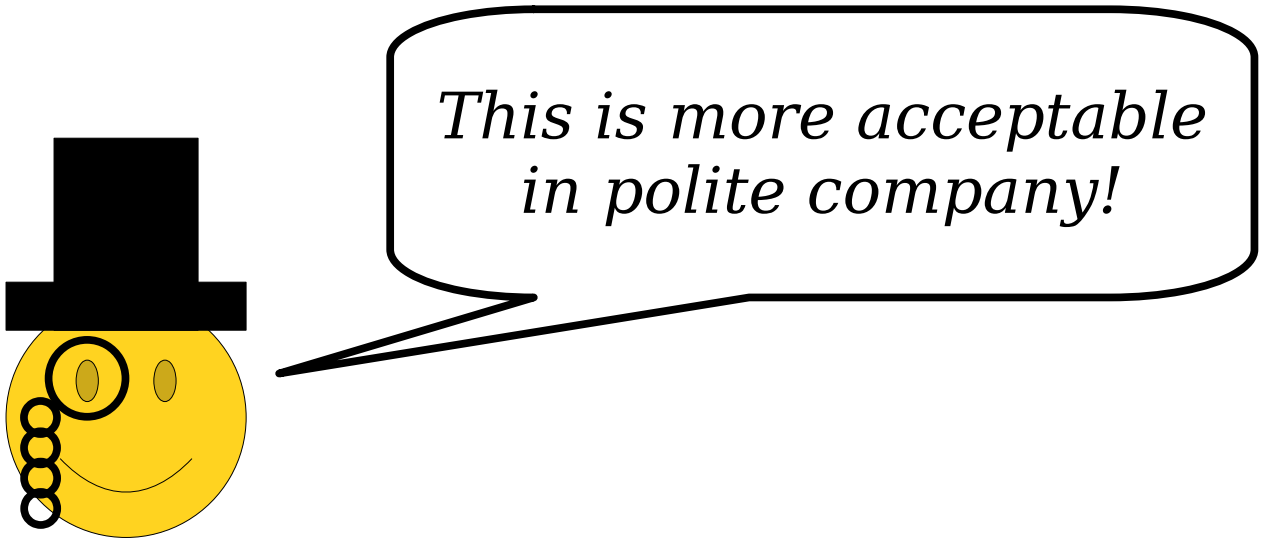
```
listSubsetsOf({1, 2, 3}, {});
```

```
listSubsetsOf({1, 2, 3}, {});
```



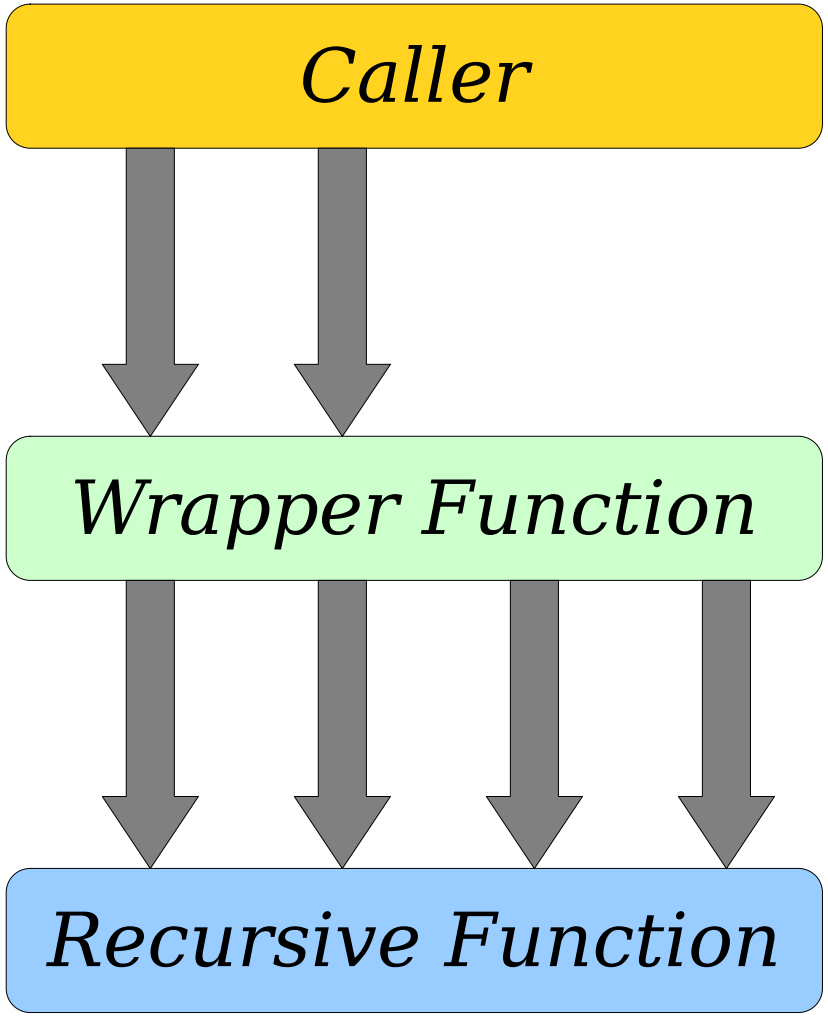
```
listSubsetsOf({1, 2, 3});
```

```
listSubsetsOf({1, 2, 3});
```



Wrapper Functions

- Some recursive functions need extra arguments as part of an implementation detail.
  - In our case, the set of things chosen so far is not something we want to expose.
- A ***wrapper function*** is a function that does some initial prep work, then fires off a recursive call with the right arguments.



Summary For Today

- Making the ***recursive leap of faith*** and trusting that your recursive calls will perform as expected helps simplify writing recursive code.
- A ***decision tree*** models all the ways you can make choices to arrive at a set of results.
- A ***wrapper function*** makes the interface of recursive calls cleaner and harder to misuse.

Your Action Items

- ***Read Chapter 8.***
  - There's a lot of great information there about recursive problem-solving, and it's a great resource.
- ***Finish Assignment 2***
  - Hopefully you've finished Rising Tides by now. Aim to complete You Got Hufflepuff! by our next lecture.

Next Time

- ***Iteration + Recursion***
  - Combining two techniques together.
- ***Enumerating Permutations***
  - What order should we do these tasks in?