

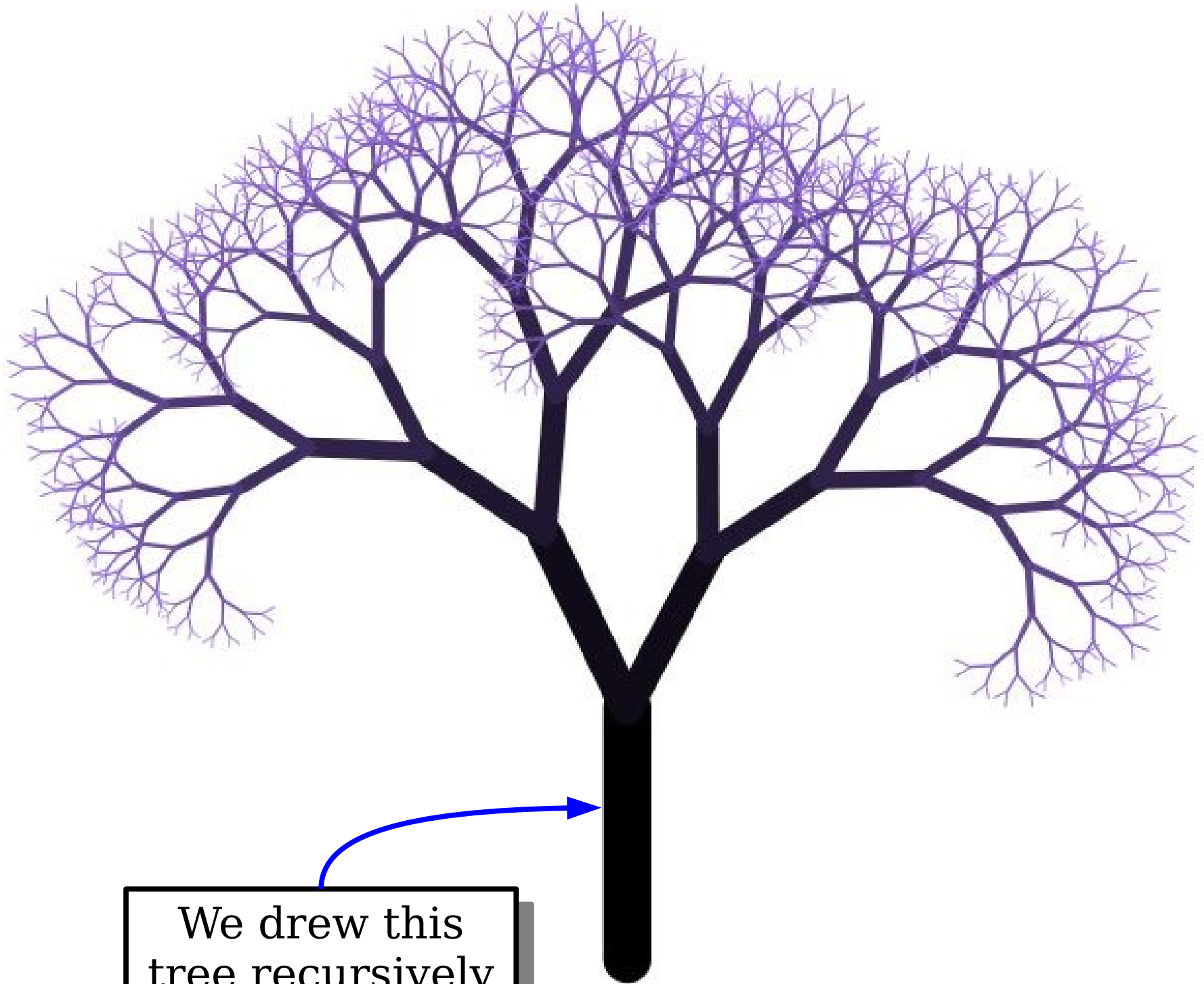
# Thinking Recursively

## Part II

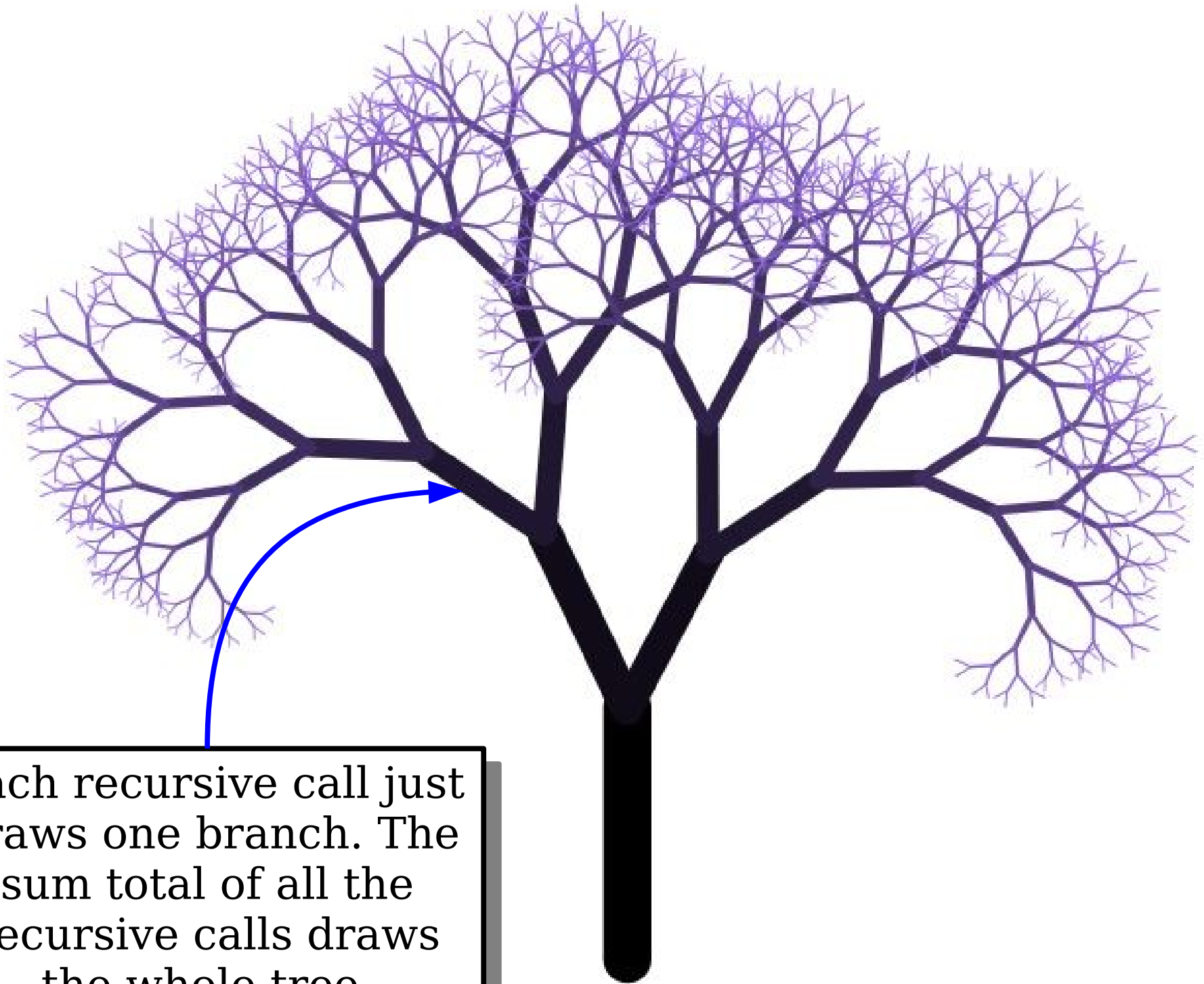
# Outline for Today

- ***Recap from Last Time***
  - Where are we, again?
- ***Enumerating Permutations***
  - What order should we do things?
- ***Enumerating Combinations***
  - Finding the right group of the right size.

Recap from Last Time



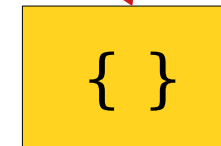
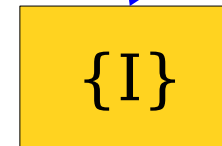
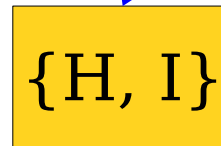
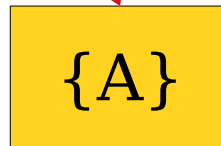
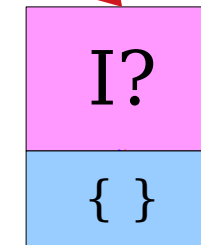
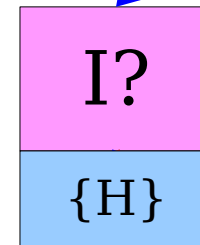
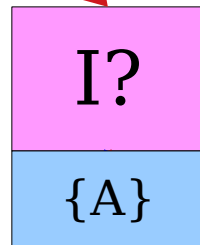
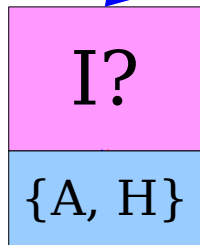
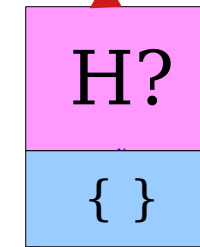
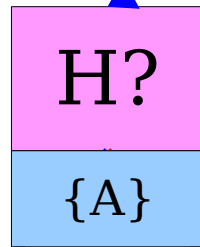
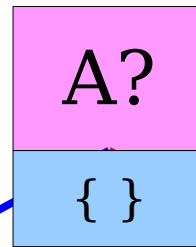
We drew this tree recursively



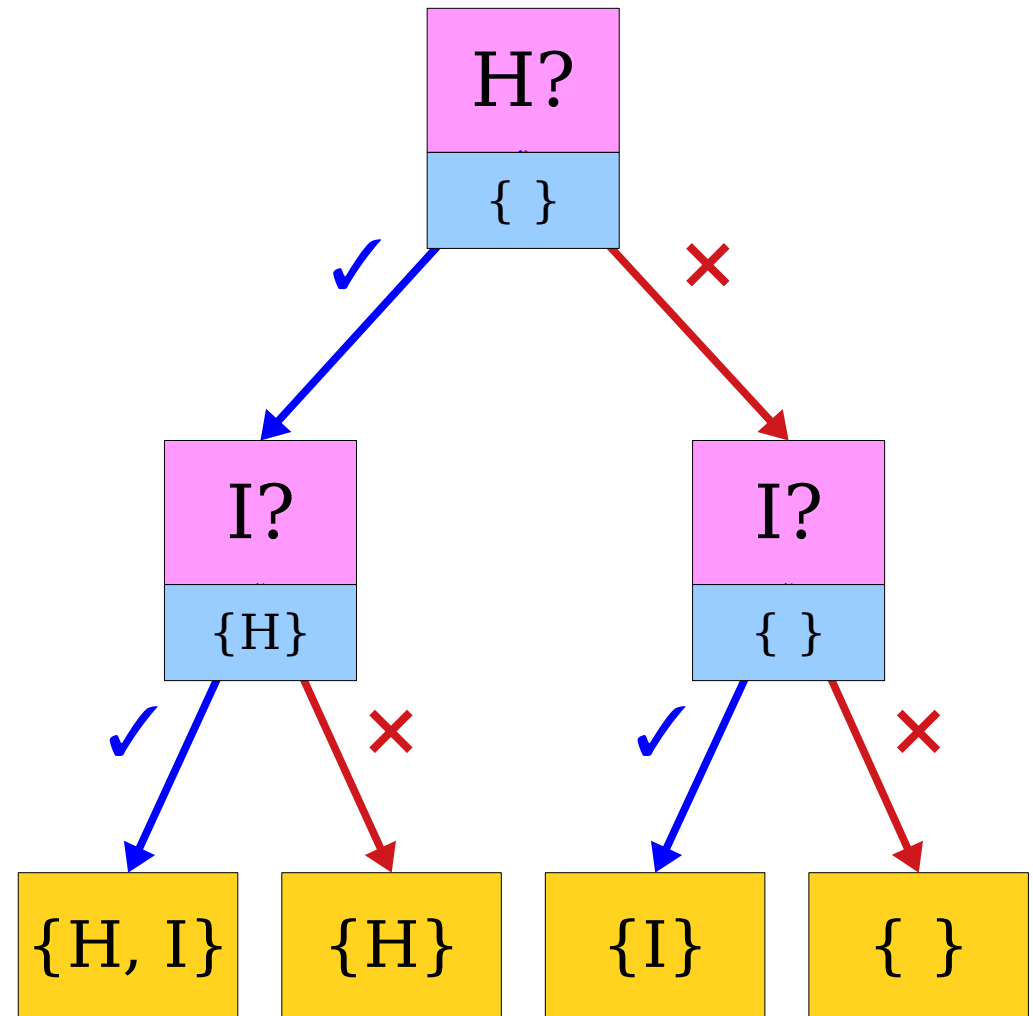
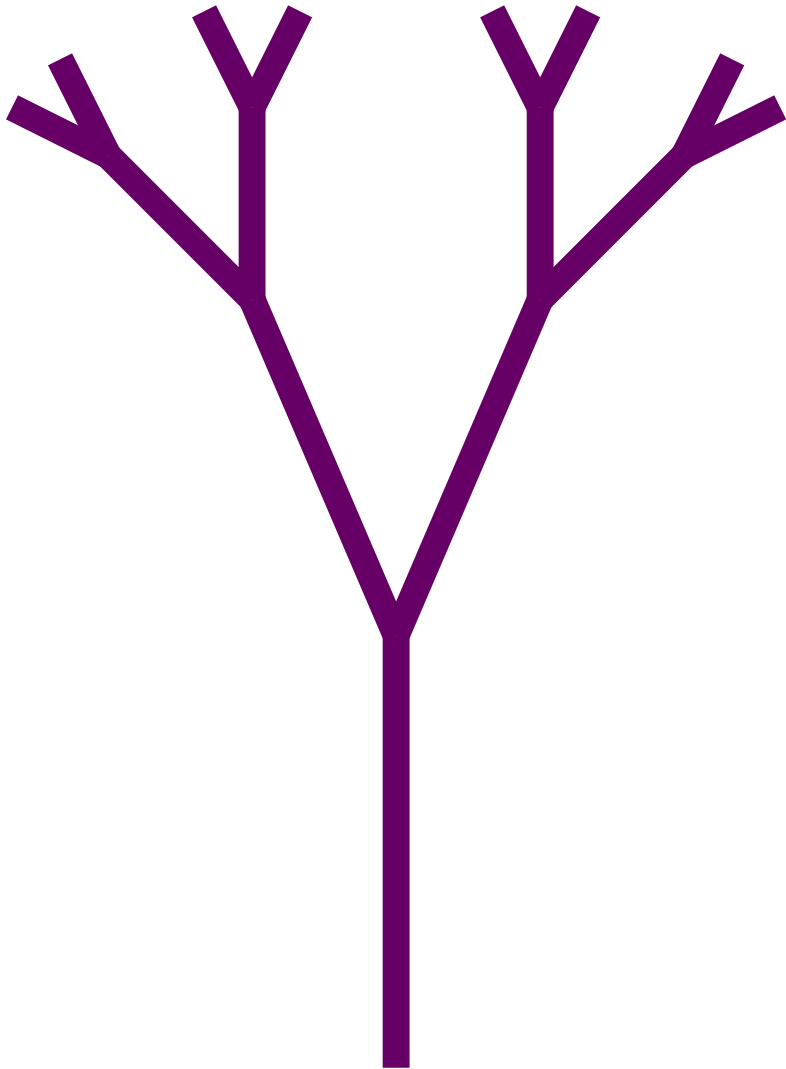
Each recursive call just draws one branch. The sum total of all the recursive calls draws the whole tree.

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

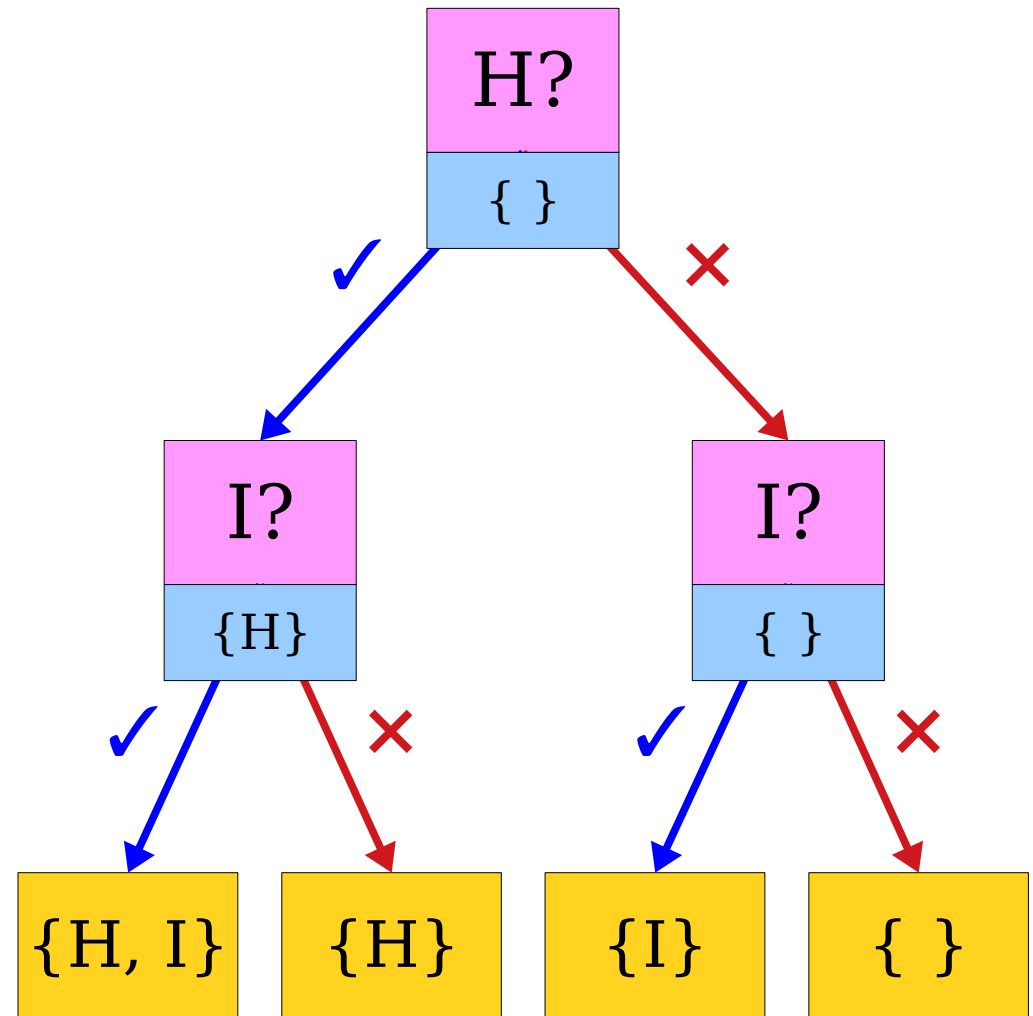
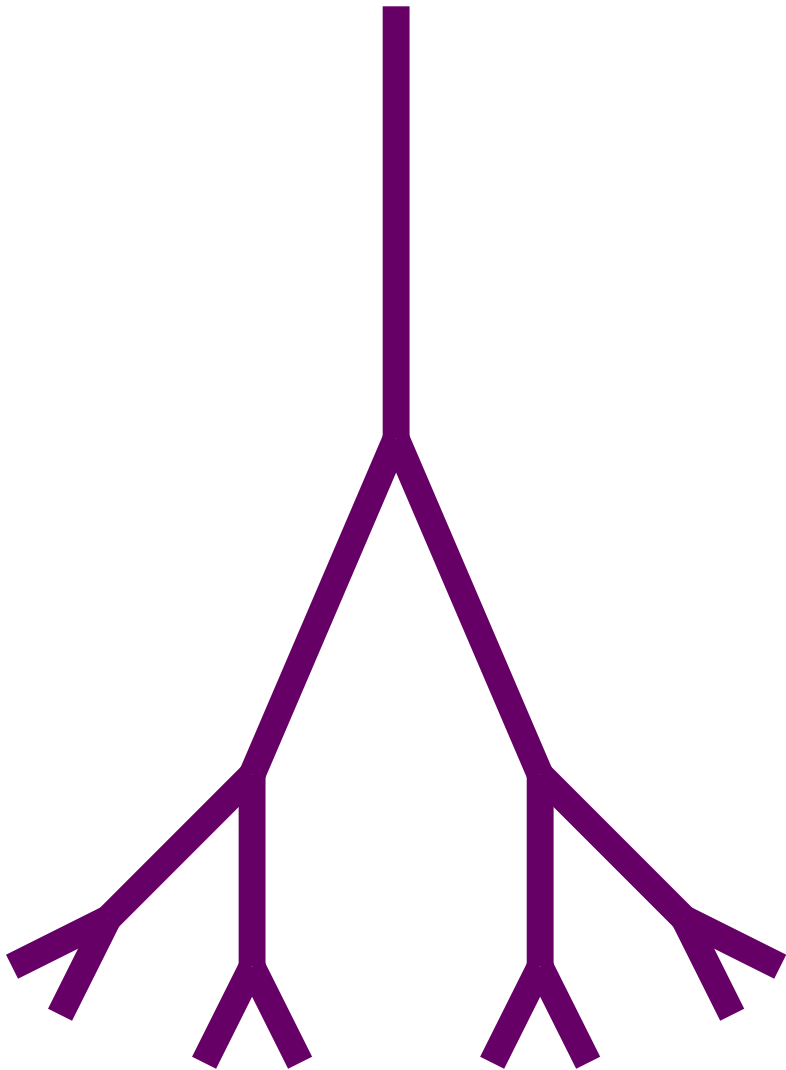
This is called a  
*decision tree*.



# Two Trees

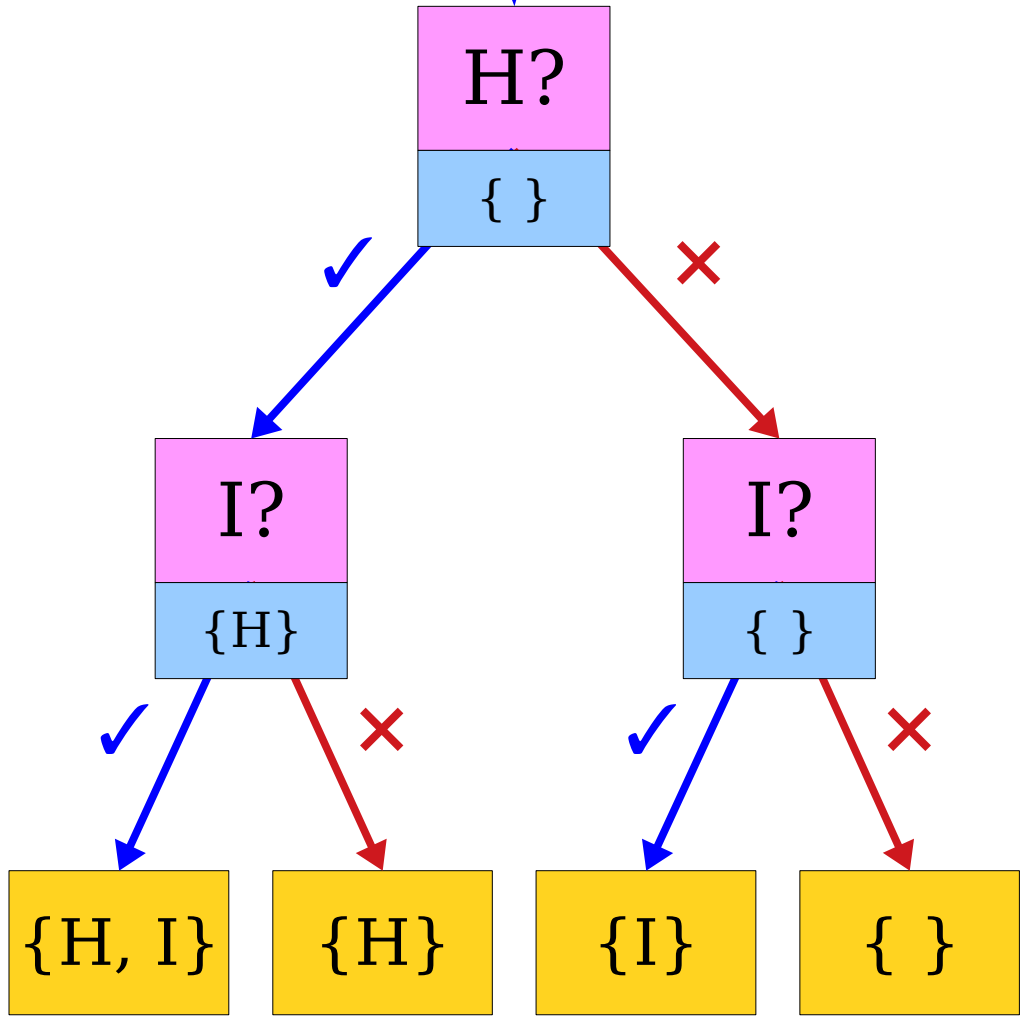
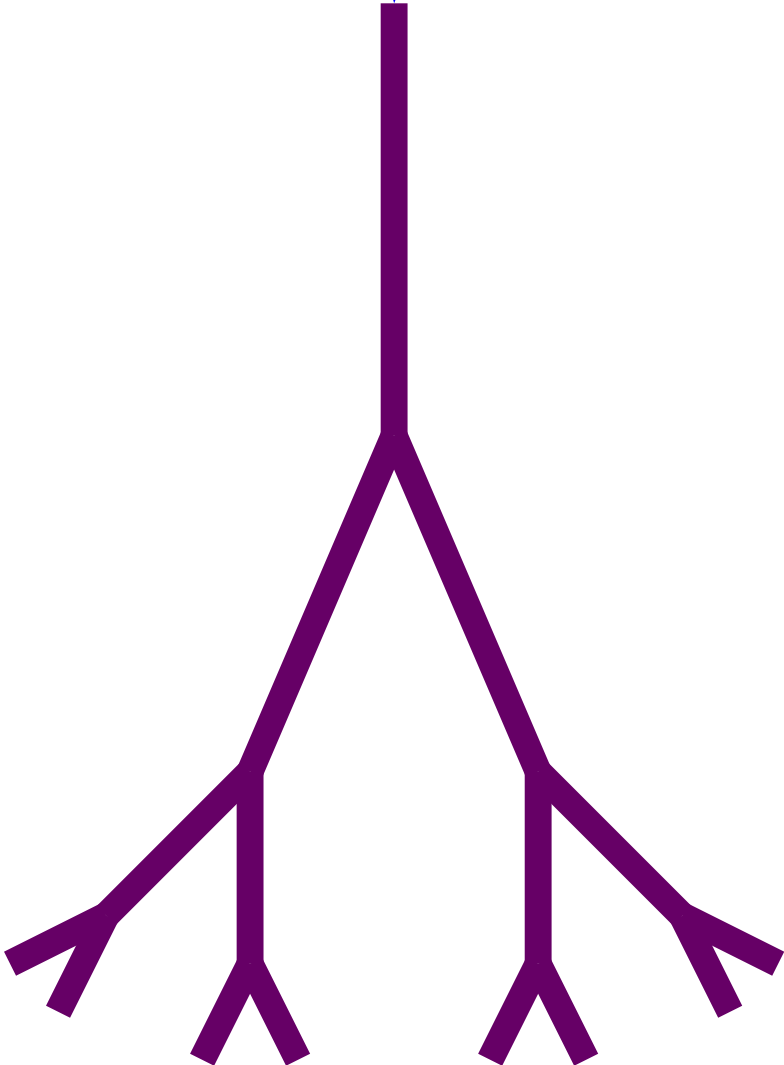


# Two Trees

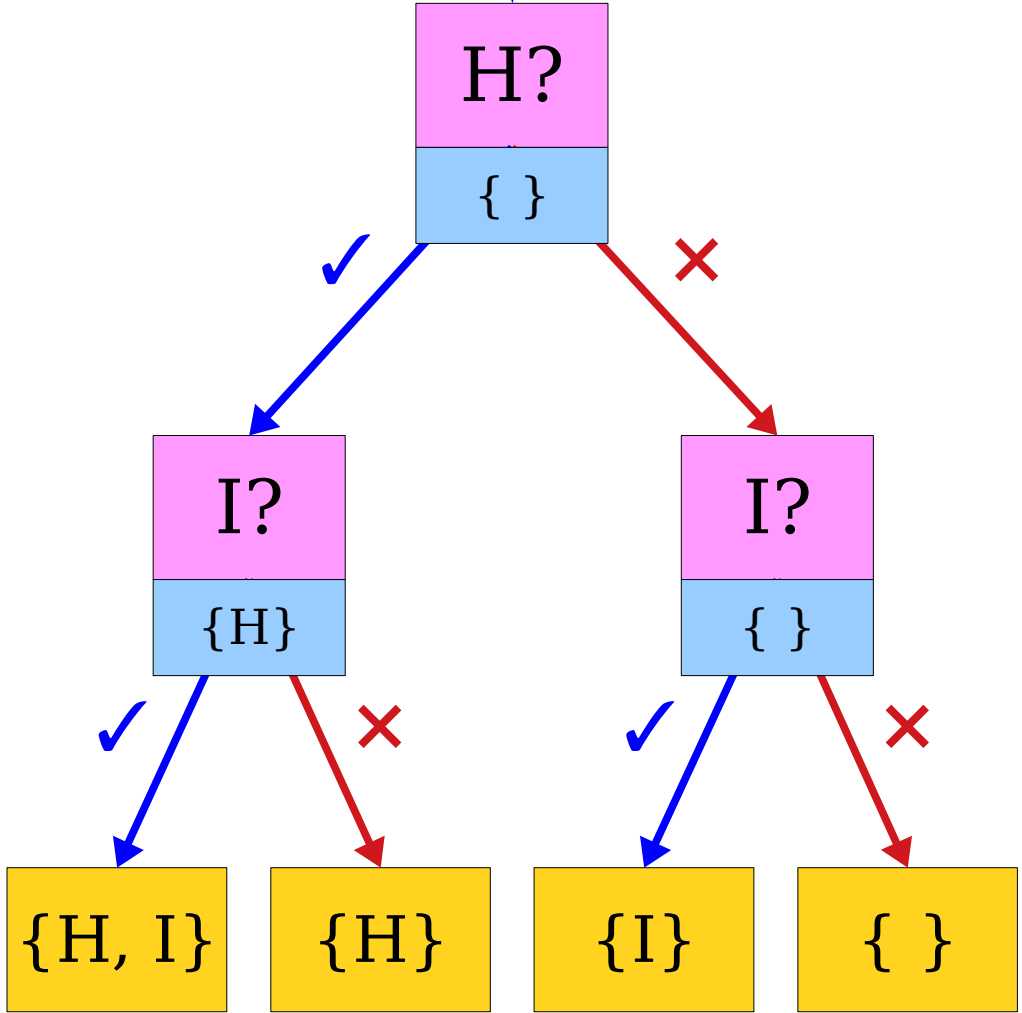
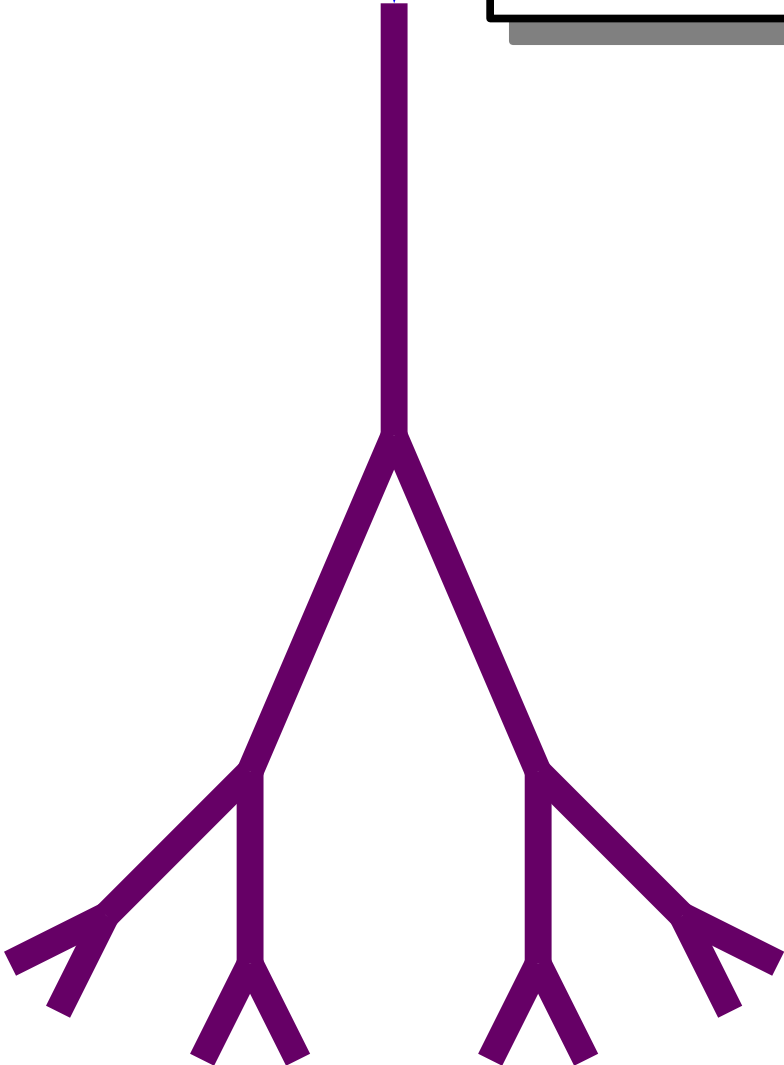




We'll process these trees recursively.



Each recursive call just processes one part of the tree. The sum of all recursive calls processes the whole tree.



**Base Case:**  
No decisions remain.

Decisions yet to be made

```
void listSubsetsRec(const Set<int>& remaining,  
                  const Set<int>& used) {
```

```
    if (remaining.isEmpty()) {  
        cout << used << endl;  
    } else {  
        int elem = remaining.first();
```

```
        /* Option 1: Include this element. */  
        listSubsetsRec(remaining - elem, used + elem);
```

```
        /* Option 2: Exclude this element. */  
        listSubsetsRec(remaining - elem, used);
```

```
    }  
}
```

**Recursive Case:**  
Try all options for the next decision.

Decisions already made

New Stuff!

# Enumerating Permutations



You own a classy print shop.

You have a list of jobs to print.

Each job requires some amount of time and has a hard deadline.

Which jobs should you pick to maximize your profit?

# Permutations

- A ***permutation*** of a sequence is a sequence with the same elements, though possibly in a different order.

# Permutations

- A ***permutation*** of a sequence is a sequence with the same elements, though possibly in a different order.





# Permutations

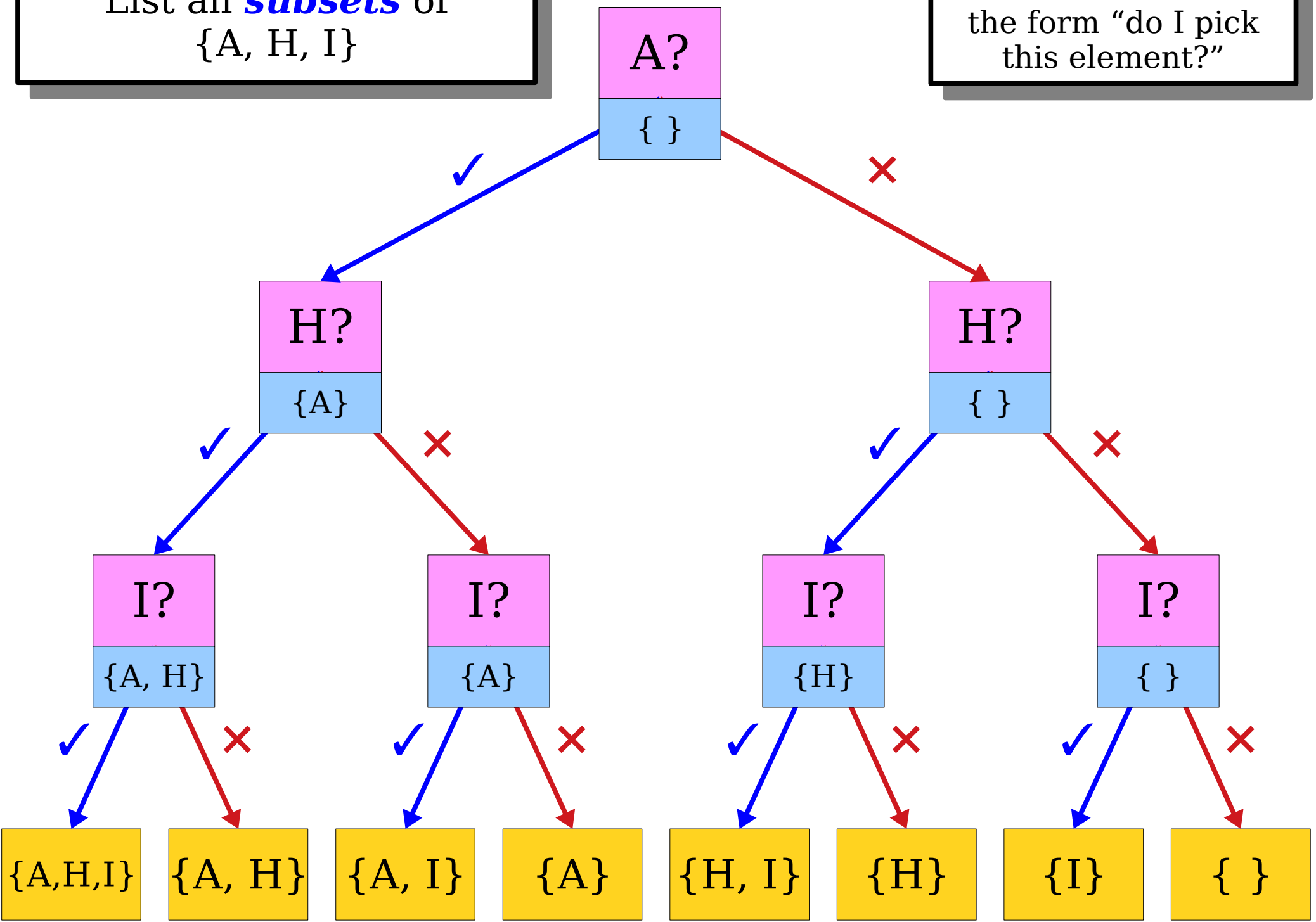
- A ***permutation*** of a sequence is a sequence with the same elements, though possibly in a different order.
- For example:
  - E Pluribus Unum
  - E Unum Pluribus
  - Pluribus E Unum
  - Pluribus Unum E
  - Unum E Pluribus
  - Unum Pluribus E



# Enumerating Permutations

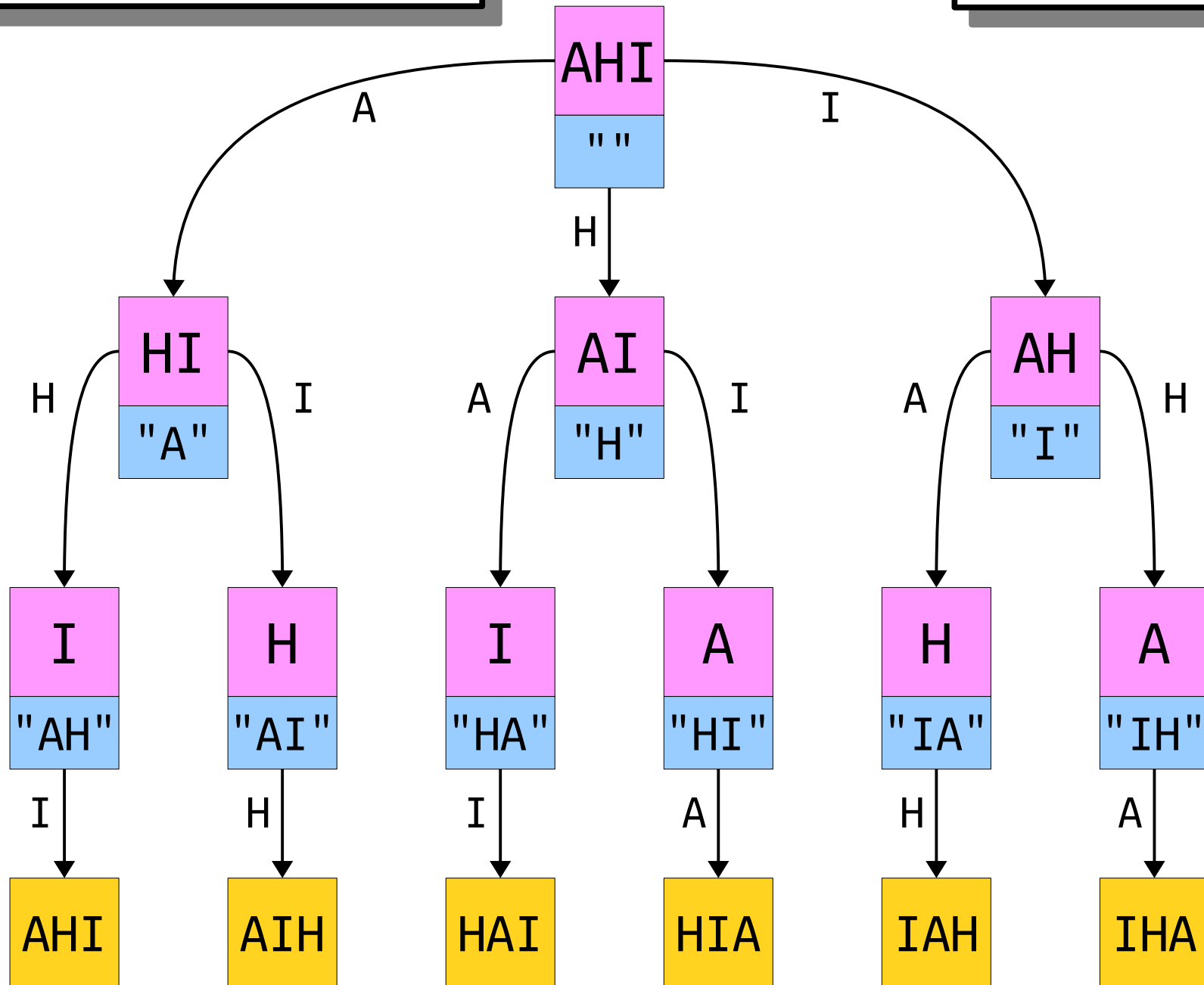
List all *subsets* of  
 $\{A, H, I\}$

Each decision is of the form  
"do I pick  
this element?"



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

Each decision is of  
the form "what do I  
pick next?"



**Base Case:**  
No decisions remain.

Decisions yet to be made

```
void listPermutationsRec(const string& remaining,  
                        const string& used) {
```

```
    if (remaining == "") {  
        cout << used << endl;  
    } else {
```

```
        /* Decide what comes next. */
```

```
        for (int i = 0; i < remaining.size(); i++) {  
            listPermutationsRec(remaining.substr(0, i) +  
                                remaining.substr(i + 1),  
                                used + remaining[i]);  
        }
```

Decisions already made

**Recursive Case:**  
Try all options for the next decision.

**Base Case:**  
No decisions remain.

Decisions yet to be made

```
void listSubsetsRec(const Set<int>& remaining,  
                  const Set<int>& used) {
```

```
    if (remaining.isEmpty()) {  
        cout << used << endl;  
    } else {  
        int elem = remaining.first();
```

```
        /* Option 1: Include this element. */  
        listSubsetsRec(remaining - elem, used + elem);
```

```
        /* Option 2: Exclude this element. */  
        listSubsetsRec(remaining - elem, used);
```

```
    }  
}
```

**Recursive Case:**  
Try all options for the next decision.

Decisions already made

**Base Case:** No decisions remain.

```
void exploreRec(decisions remaining,  
               decisions already made) {
```

```
  if (no decisions remain) {  
    process decisions made;  
  } else {  
    for (each possible next choice) {  
      exploreRec(all remaining decisions,  
                decisions made + that choice);  
    }  
  }  
}
```

Decisions yet to be made

Decisions already made

**Recursive Case:**  
Try all options for the next decision.

```
void exploreAllTheThings(initial state) {  
  exploreRec(initial state, no decisions made);  
}
```

# Enumerating Combinations





You need to pick 11 people to serve as starters on your soccer (football) team. You have a good way of evaluating, roughly speaking, how any given team of 11 players will get along.

How do you decide which 11 players to pick?

# Generating Combinations

- Suppose that we want to find every way to choose exactly *one* element from a set.
- We could do something like this:

```
for (int x: mySet) {  
    cout << x << endl;  
}
```

# Generating Combinations

- Suppose that we want to find every way to choose exactly *two* elements from a set.
- We could do something like this:

```
for (int x: mySet) {  
    for (int y: mySet) {  
        if (x != y) {  
            cout << x << ", " << y << endl;  
        }  
    }  
}
```

# Generating Combinations

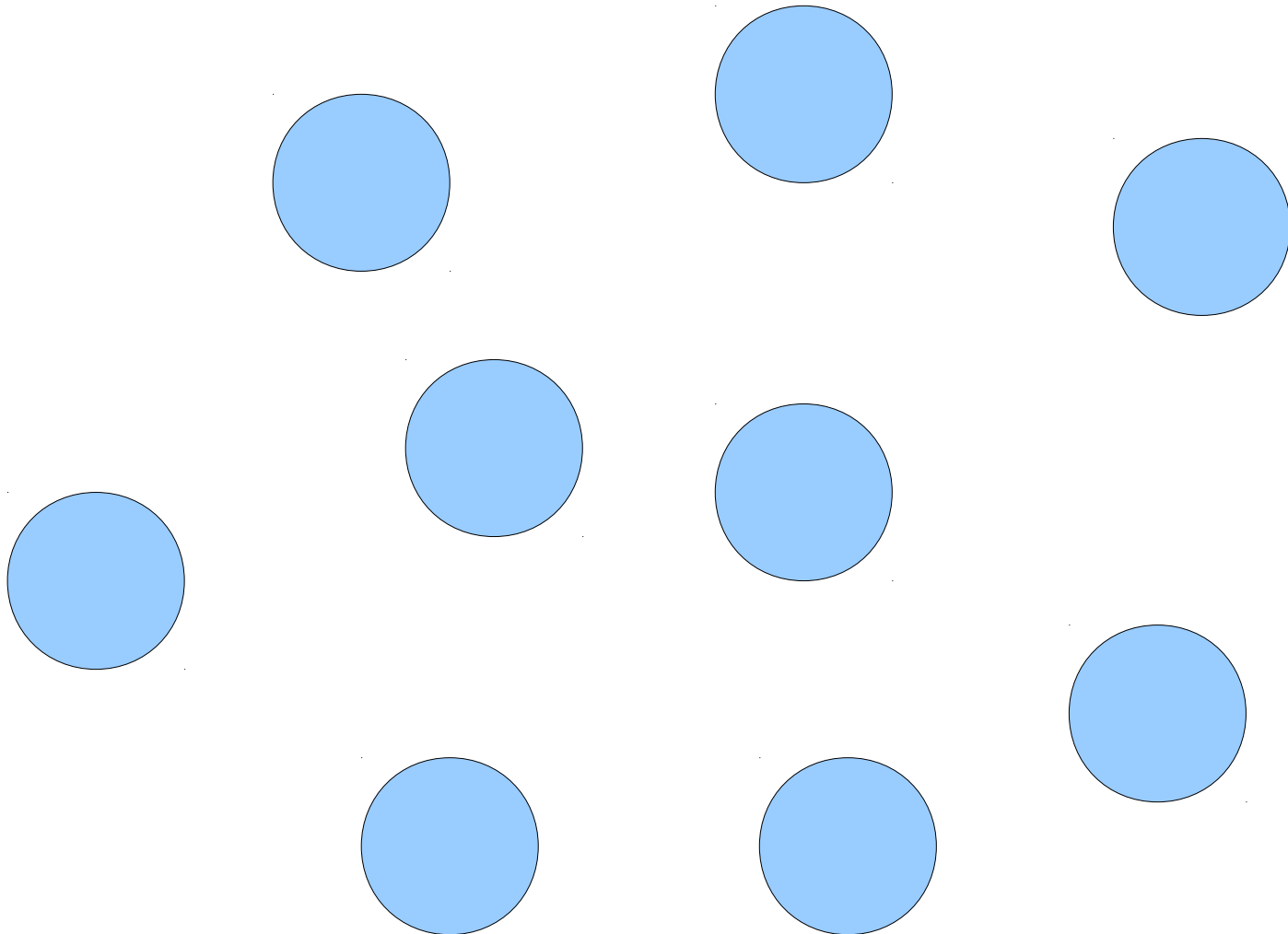
- Suppose that we want to find every way to choose exactly *three* elements from a set.
- We could do something like this:

```
for (int x: mySet) {
    for (int y: mySet) {
        for (int z: mySet) {
            if (x != y && x != z && y != z) {
                cout << x << ", " << y << ", " << z << endl;
            }
        }
    }
}
```

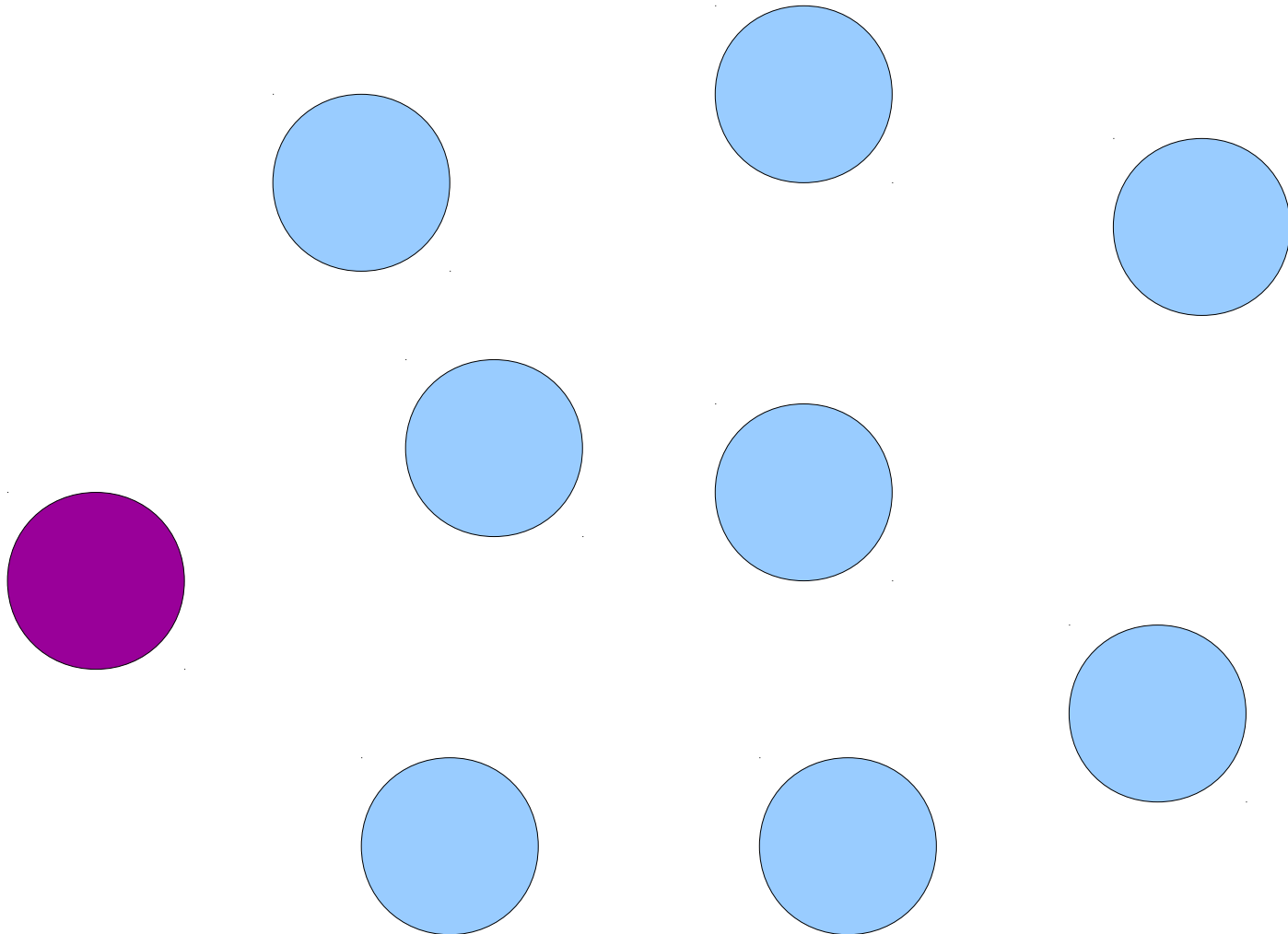
# Generating Combinations

- If we know how many elements we want in advance, we can always just nest a whole bunch of loops.
- But what if we don't know in advance?
- Or we *do* know in advance, but it's a large number and we don't want to type until our fingers bleed?

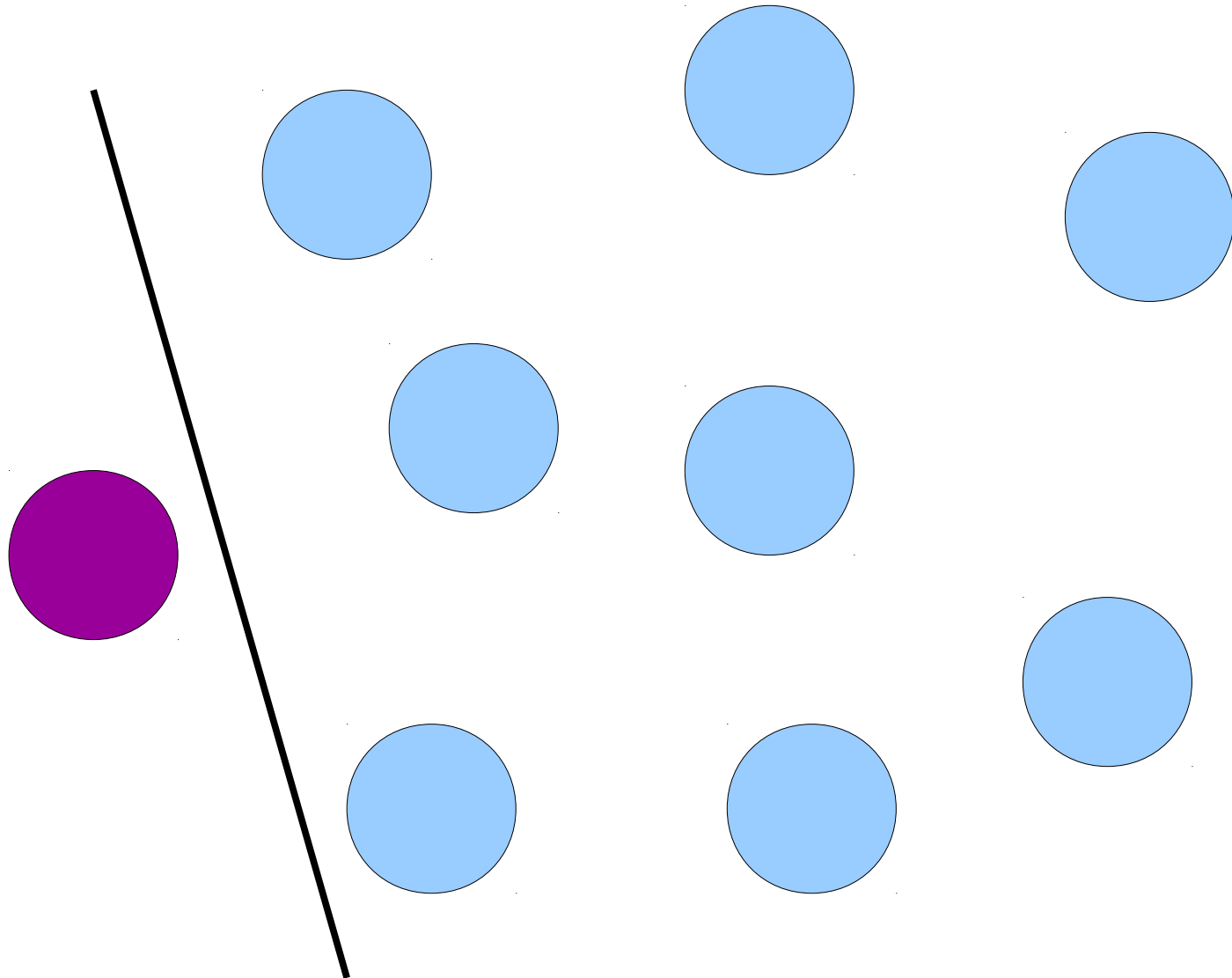
# Generating Combinations



# Generating Combinations

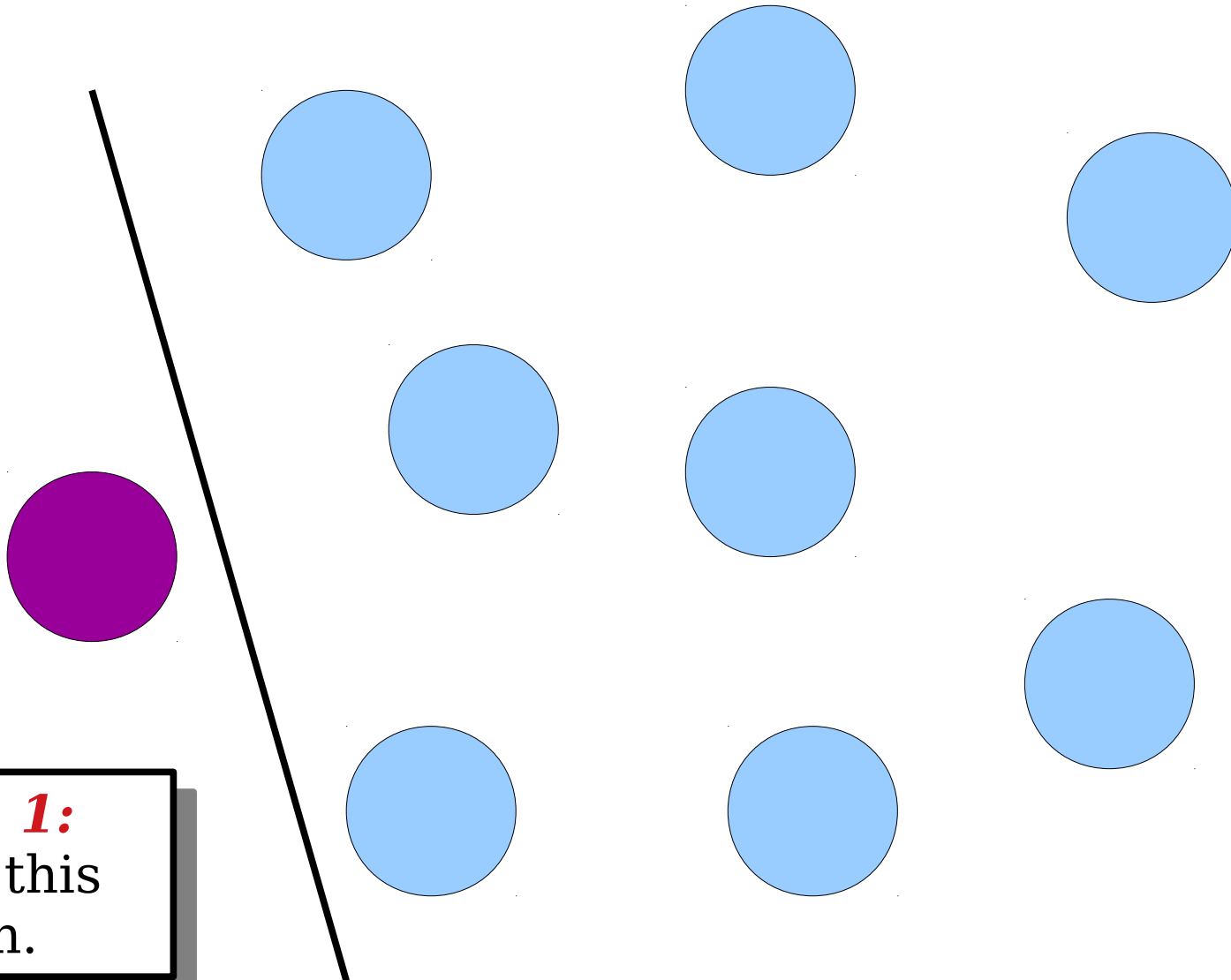


# Generating Combinations



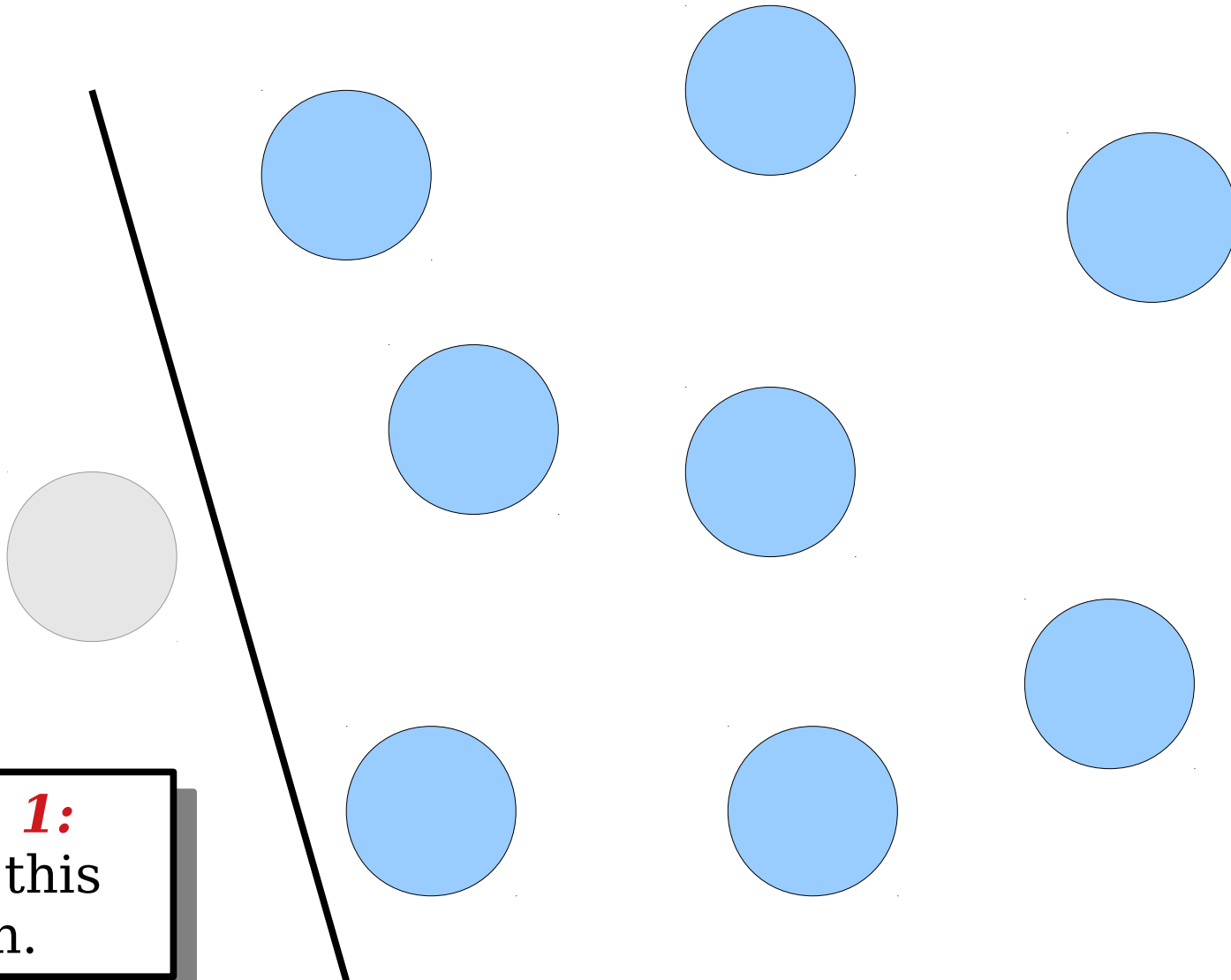


# Generating Combinations



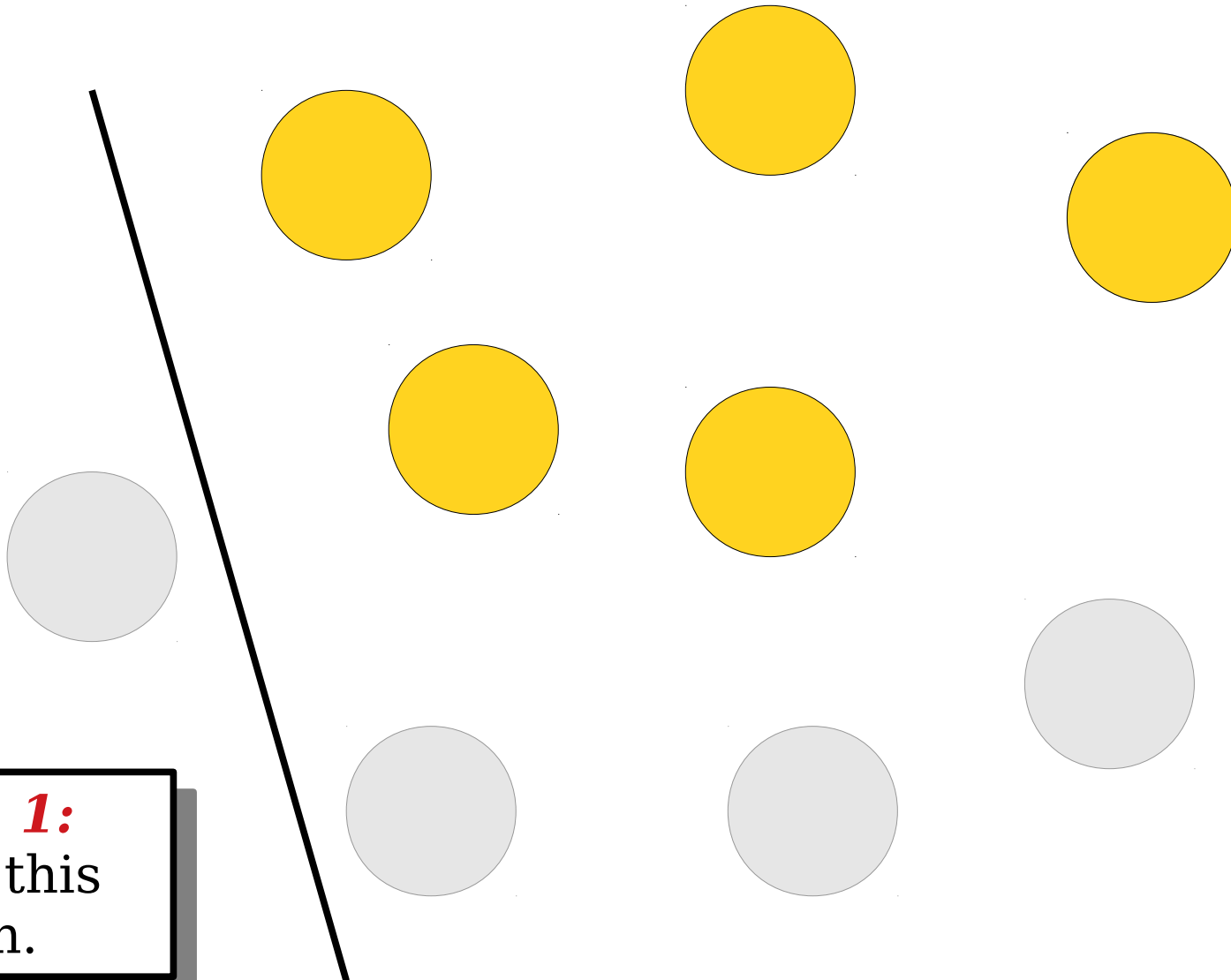
***Option 1:***  
Exclude this  
person.

# Generating Combinations



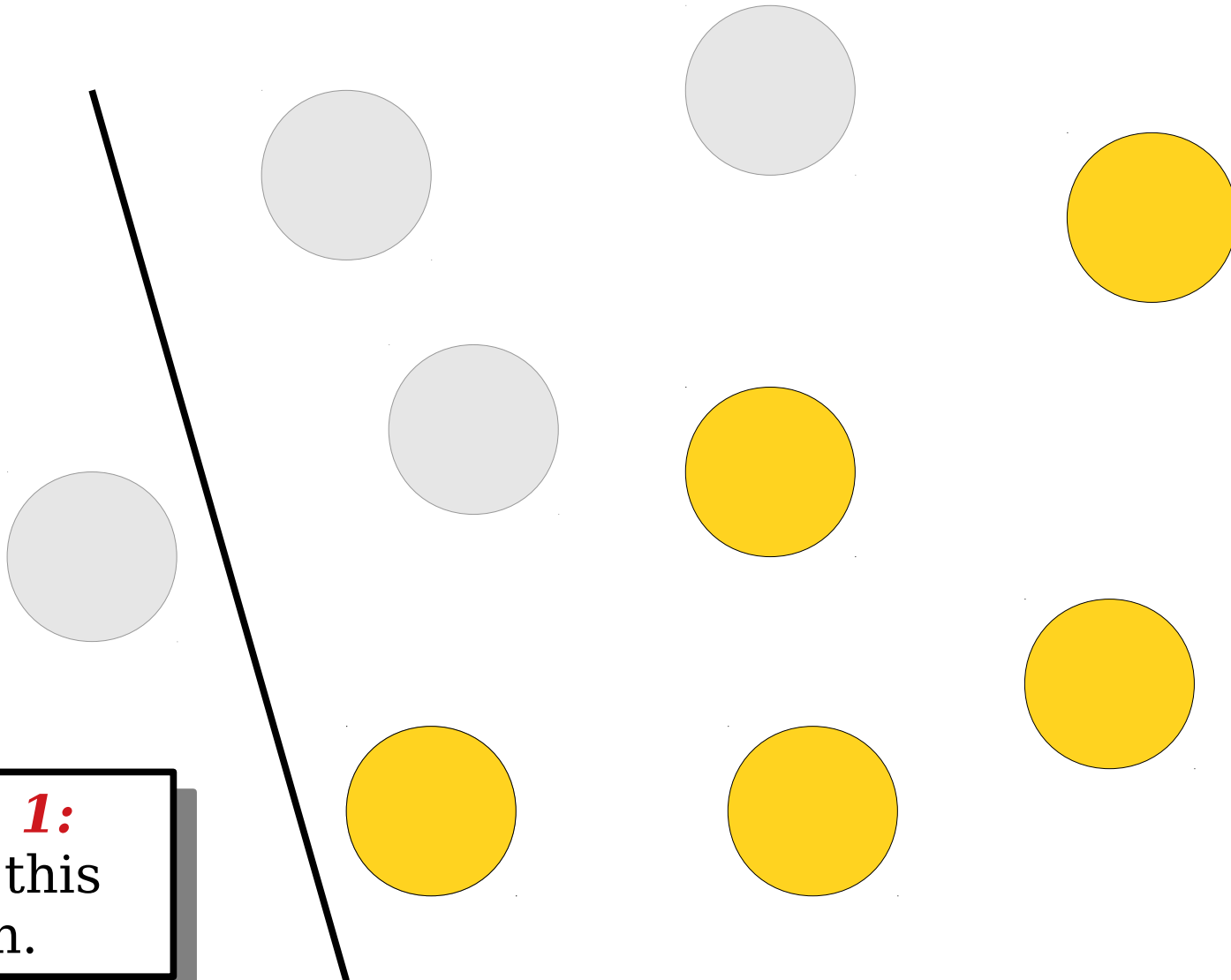
***Option 1:***  
Exclude this  
person.

# Generating Combinations



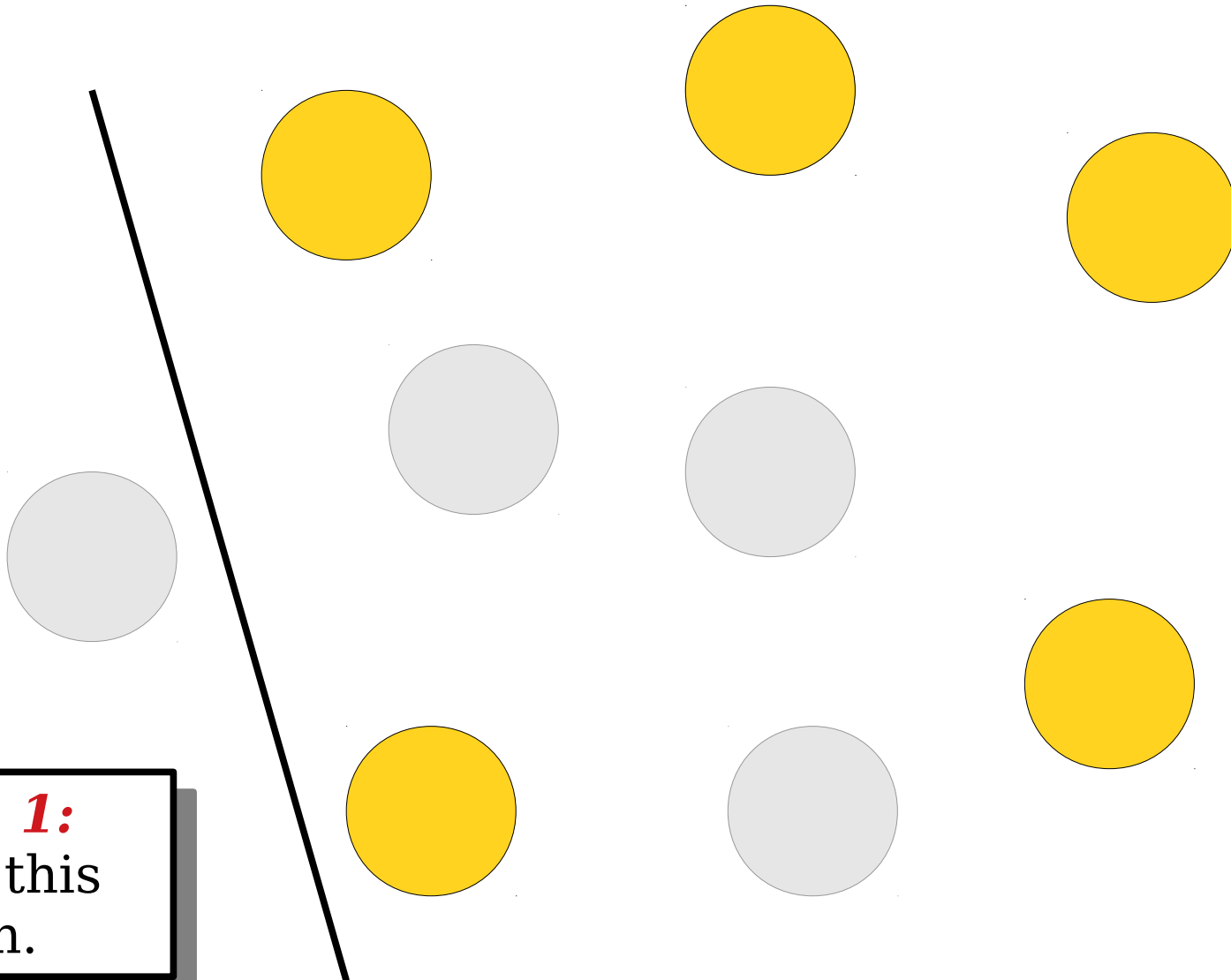
***Option 1:***  
Exclude this  
person.

# Generating Combinations



***Option 1:***  
Exclude this  
person.

# Generating Combinations



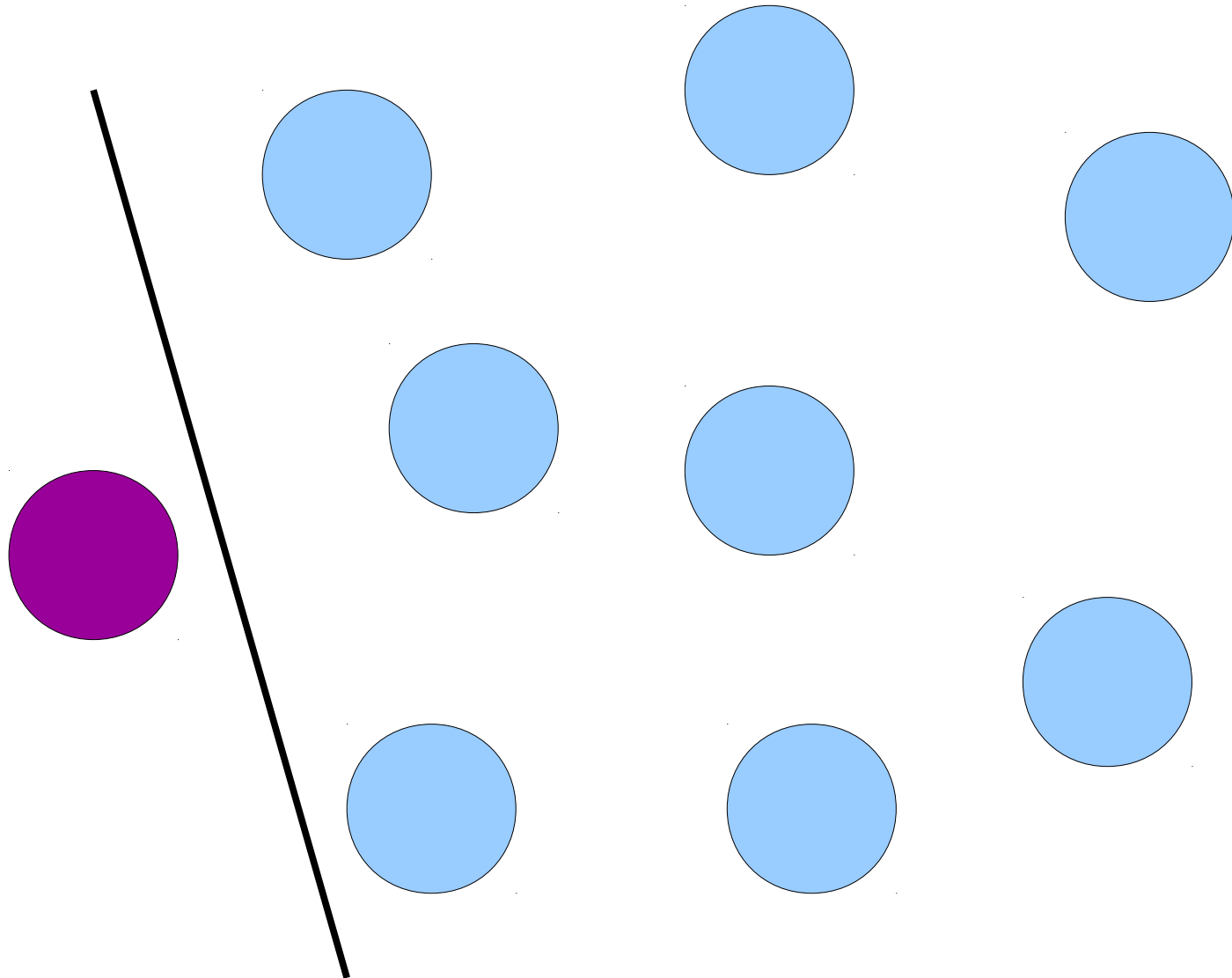
***Option 1:***  
Exclude this  
person.

# Generating Combinations

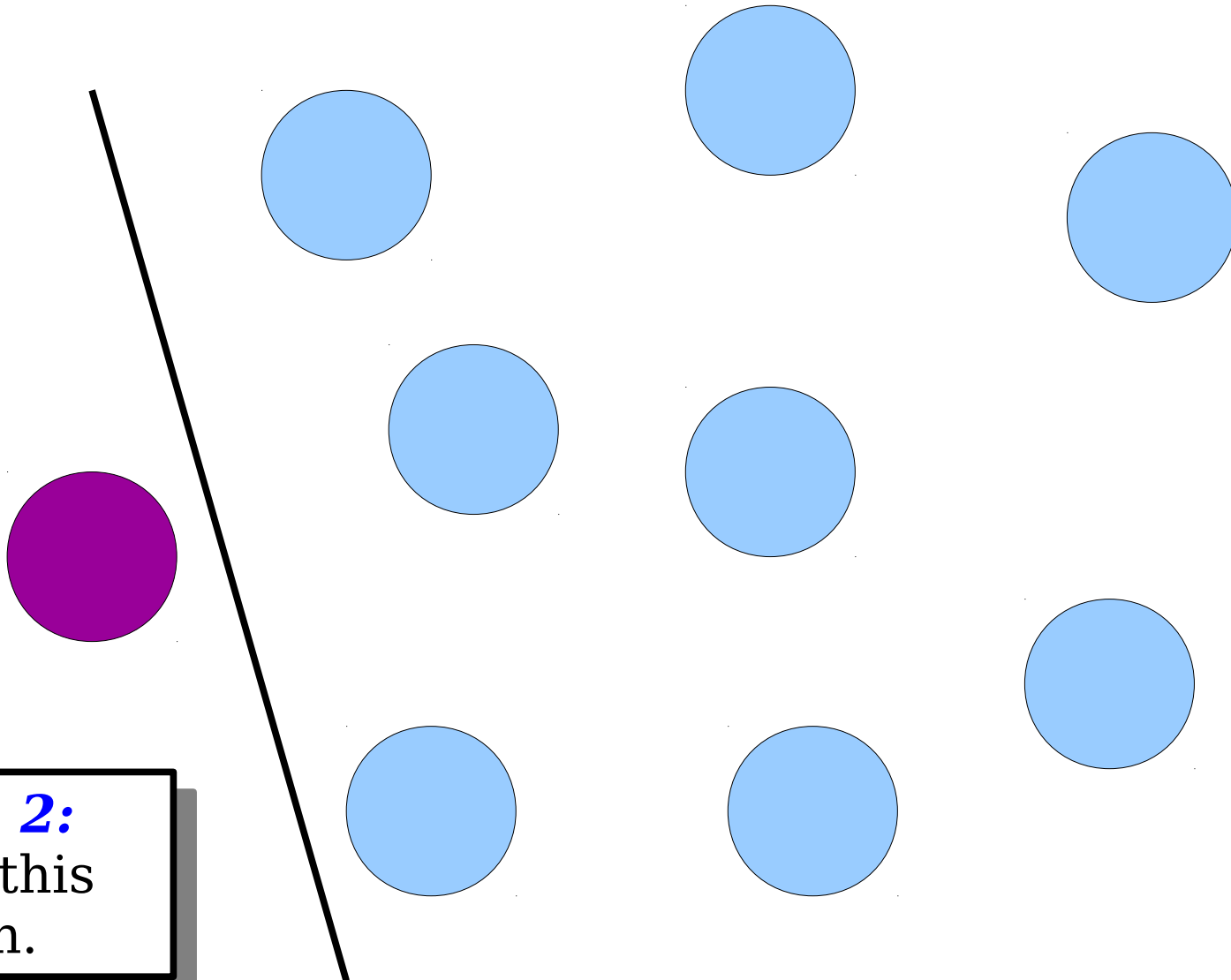
One way to choose **5** elements out of **9** is to exclude the first element, then to choose **5** elements out of the remaining **8**.

***Option 1:***  
Exclude this  
person.

# Generating Combinations



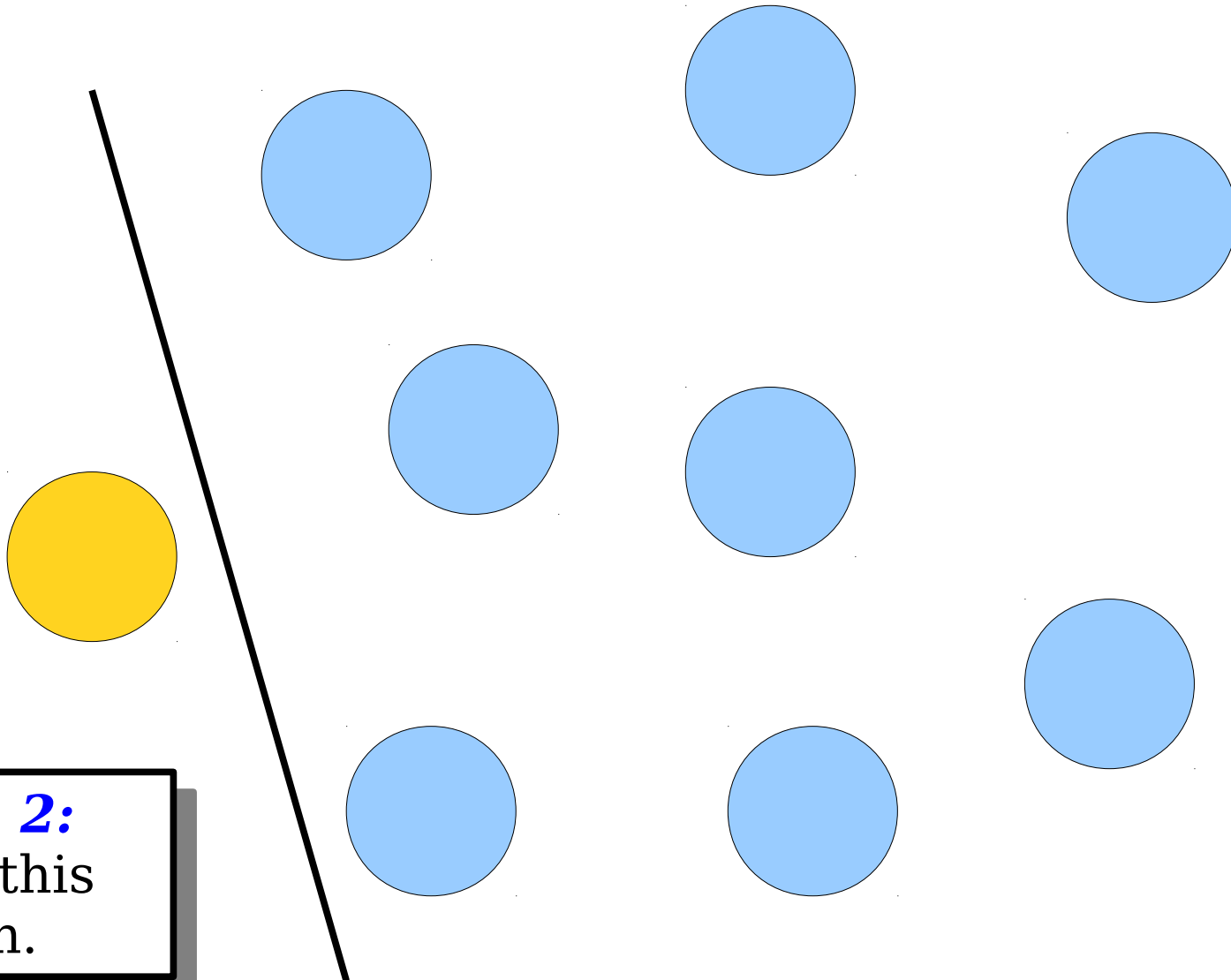
# Generating Combinations



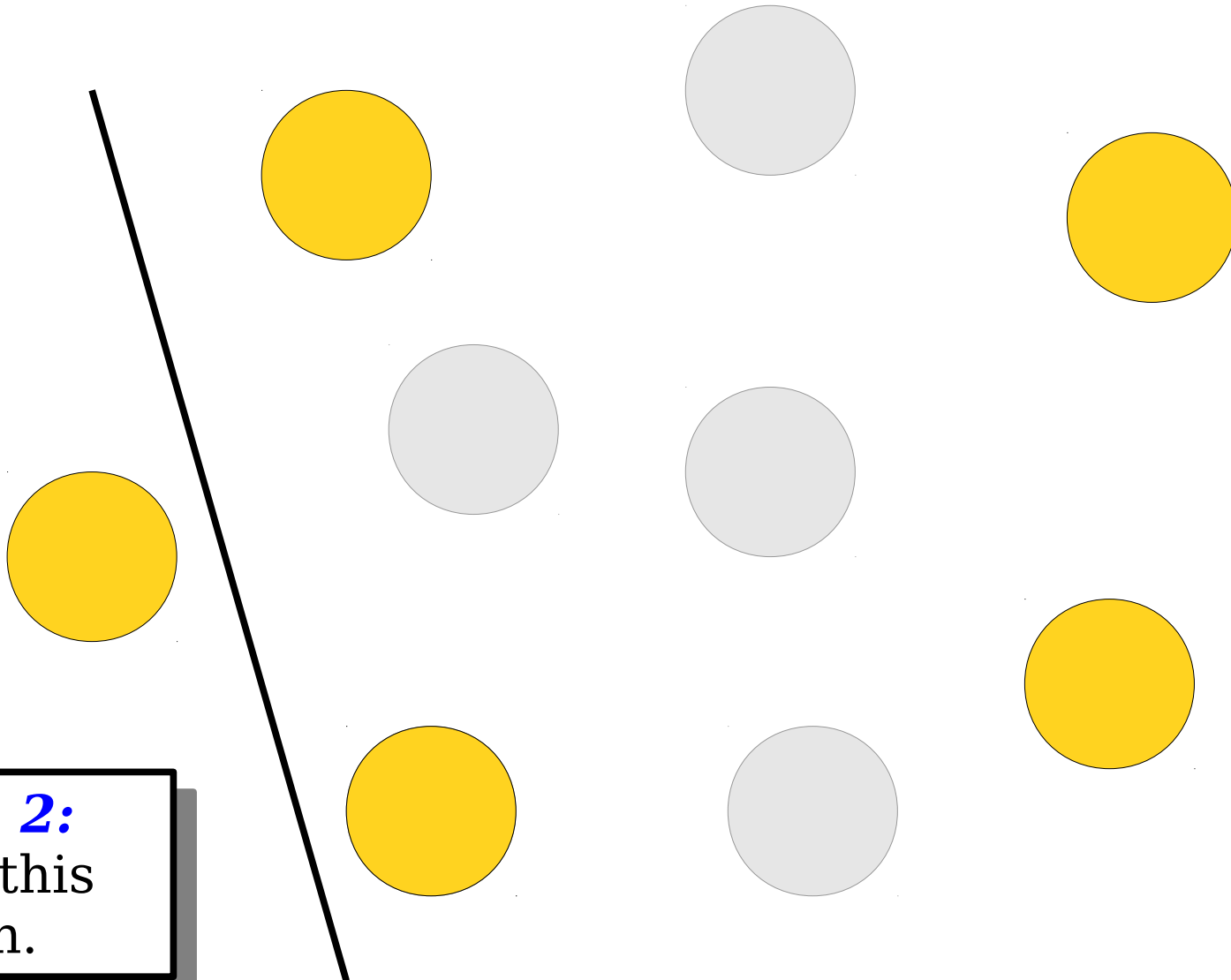
***Option 2:***  
Include this  
person.



# Generating Combinations



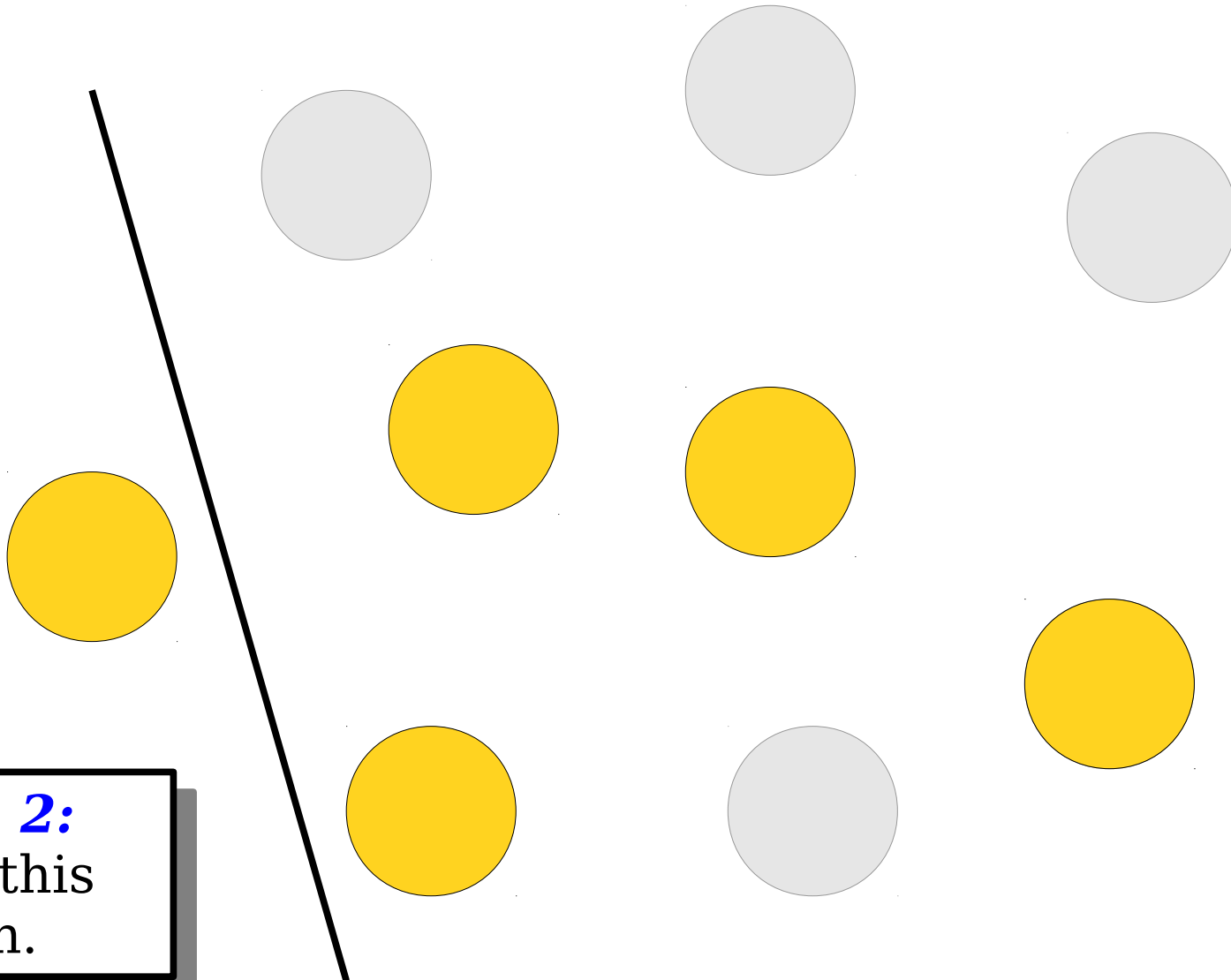
# Generating Combinations



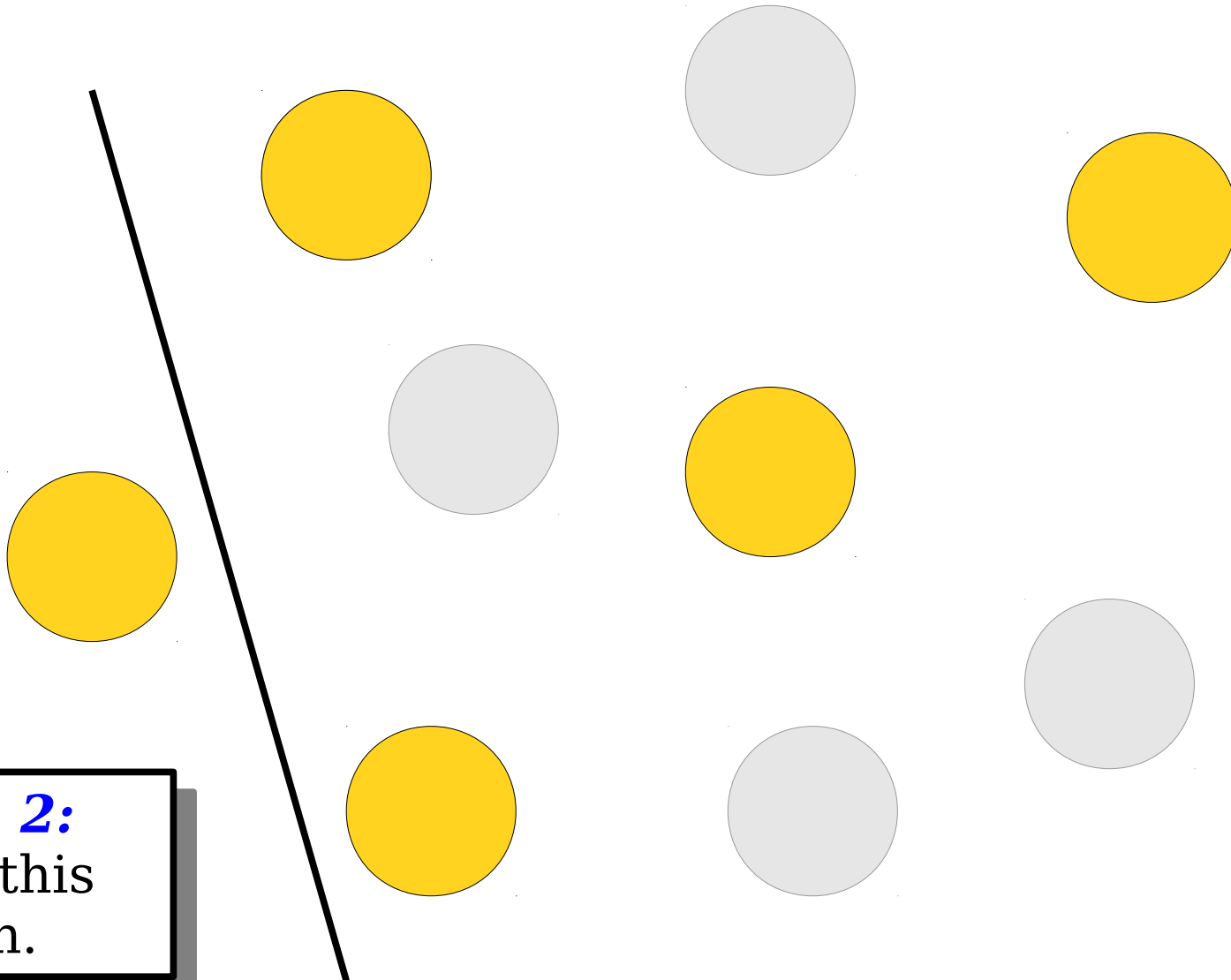
***Option 2:***

Include this  
person.

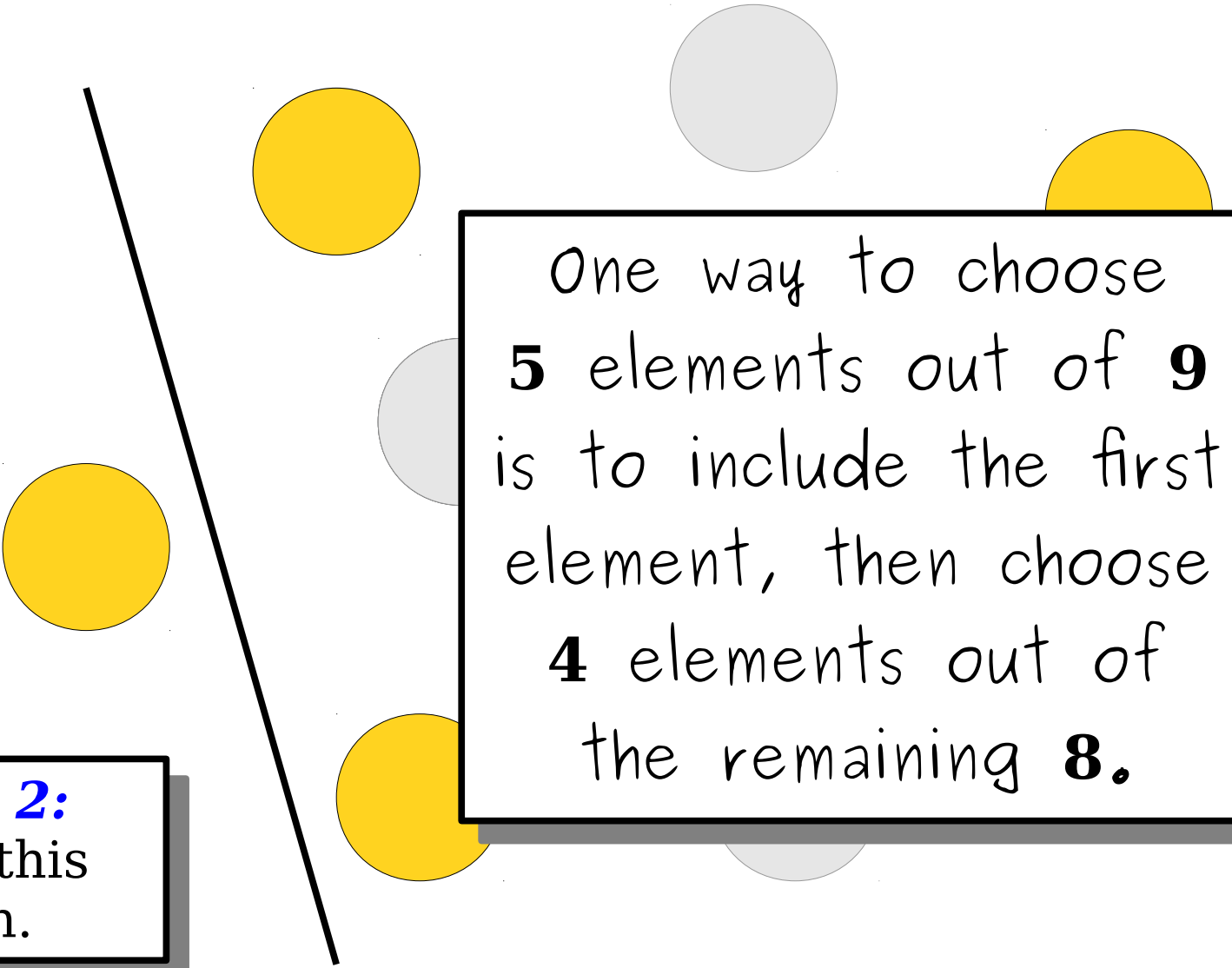
# Generating Combinations



# Generating Combinations



# Generating Combinations



One way to choose  
**5** elements out of **9**  
is to include the first  
element, then choose  
**4** elements out of  
the remaining **8**.

***Option 2:***  
Include this  
person.

# Judicial Decisions

Pick 5 Justices out of  
{Kagan, Breyer, ..., Thomas}

Chosen so far: { }

Include  
Elena Kagan

Exclude  
Elena Kagan

Pick 4 Justices out of  
{ Breyer, ..., Thomas }

Chosen so far: { Kagan }

Pick 5 Justices out of  
{ Breyer, ..., Thomas }

Chosen so far: { }

**Base Case:** No decisions remain.

```
void exploreRec(decisions remaining,  
               decisions already made) {
```

```
  if (no decisions remain) {  
    process decisions made;  
  } else {  
    for (each possible next choice) {  
      exploreRec(all remaining decisions,  
                decisions made + that choice);  
    }  
  }  
}
```

Decisions yet to be made

Decisions already made

**Recursive Case:**  
Try all options for the next decision.

```
void exploreAllTheThings(initial state) {  
  exploreRec(initial state, no decisions made);  
}
```

```

void listCombinationsRec(const Set<int>& remaining, int k,
                        const Set<int>& used) {
    if (k == 0) {
        cout << used << endl;
    } else if (remaining.isEmpty() || k > remaining.size()) {
        return; // Can't succeed.
    } else {
        int elem = remaining.first();

        /* Option 1: Exclude this element. */
        listCombinationsRec(remaining - elem, k, used);

        /* Option 2: Include this element. */
        listCombinationsRec(remaining - elems, k - 1, used + elem);
    }
}

```



# Your Action Items

- ***Finish Assignment 2***. It's due on Monday.
  - Have questions? Stop by the LaIR!
  - Don't forget to run through the Assignment Submission Checklist!
- ***Read Chapter 8 of the textbook***. It's got a lot of goodies about recursion.

# Next Time

- ***Recursive Optimization***
  - We can list all the solutions. How do we choose the best one?
- ***Recursive Backtracking***
  - Finding a needle in a haystack.