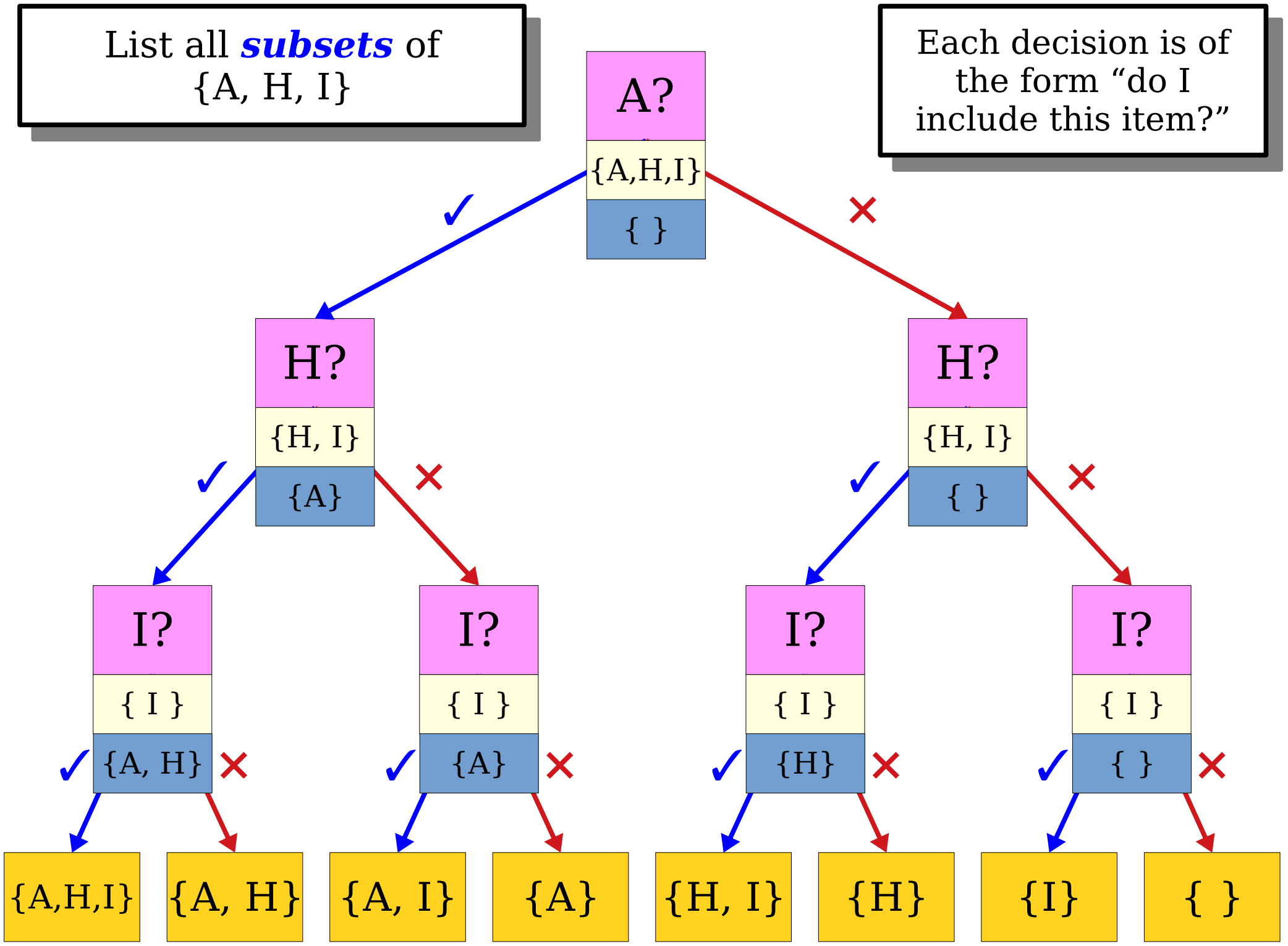
Thinking Recursively

Part IV

Outline for Today

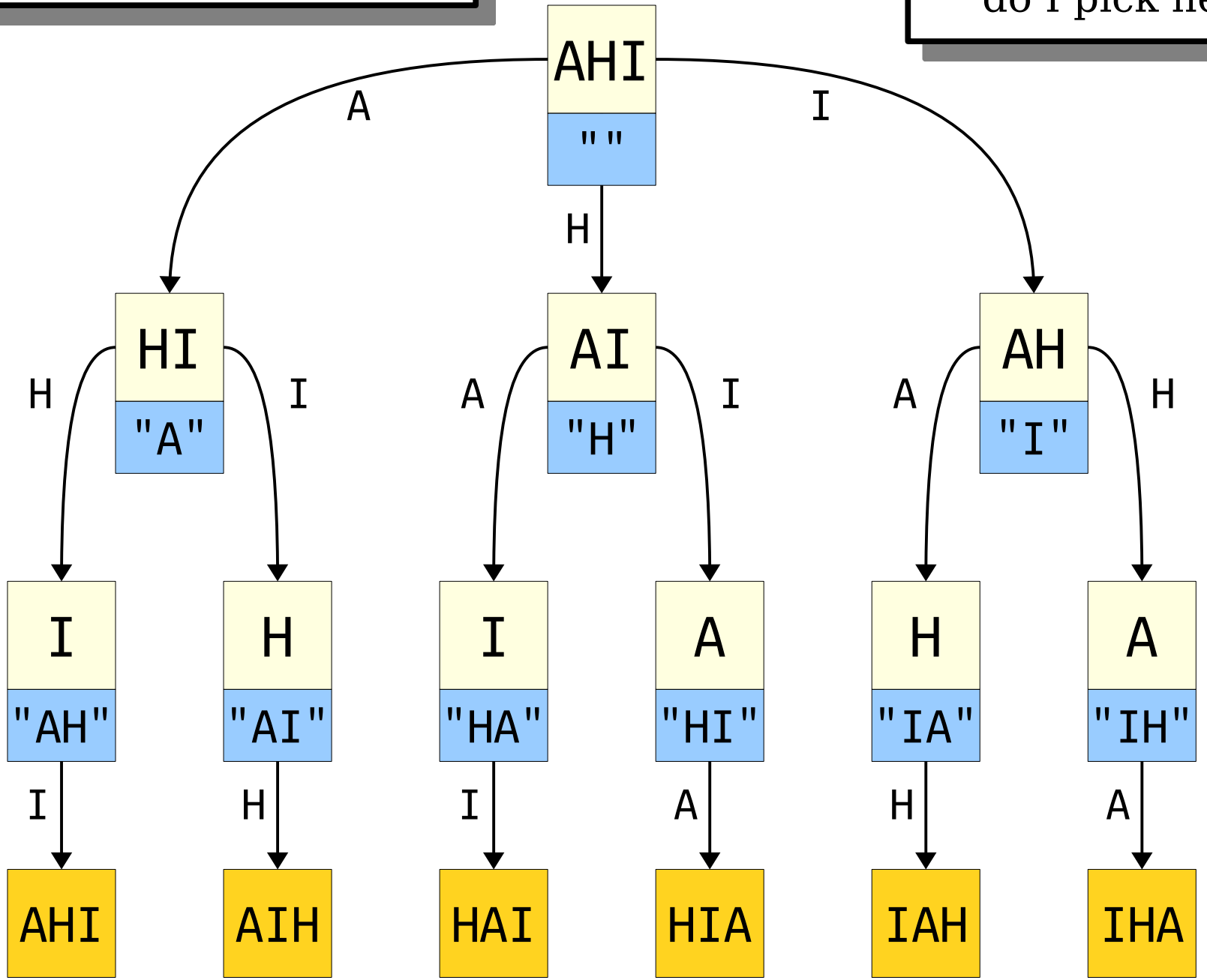
- ***Recap From Last Time***
  - Where are we, again?
- ***Enumerating Combinations***
  - Forming a majority opinion.
- ***Shrinkable Words***
  - A little word puzzle!

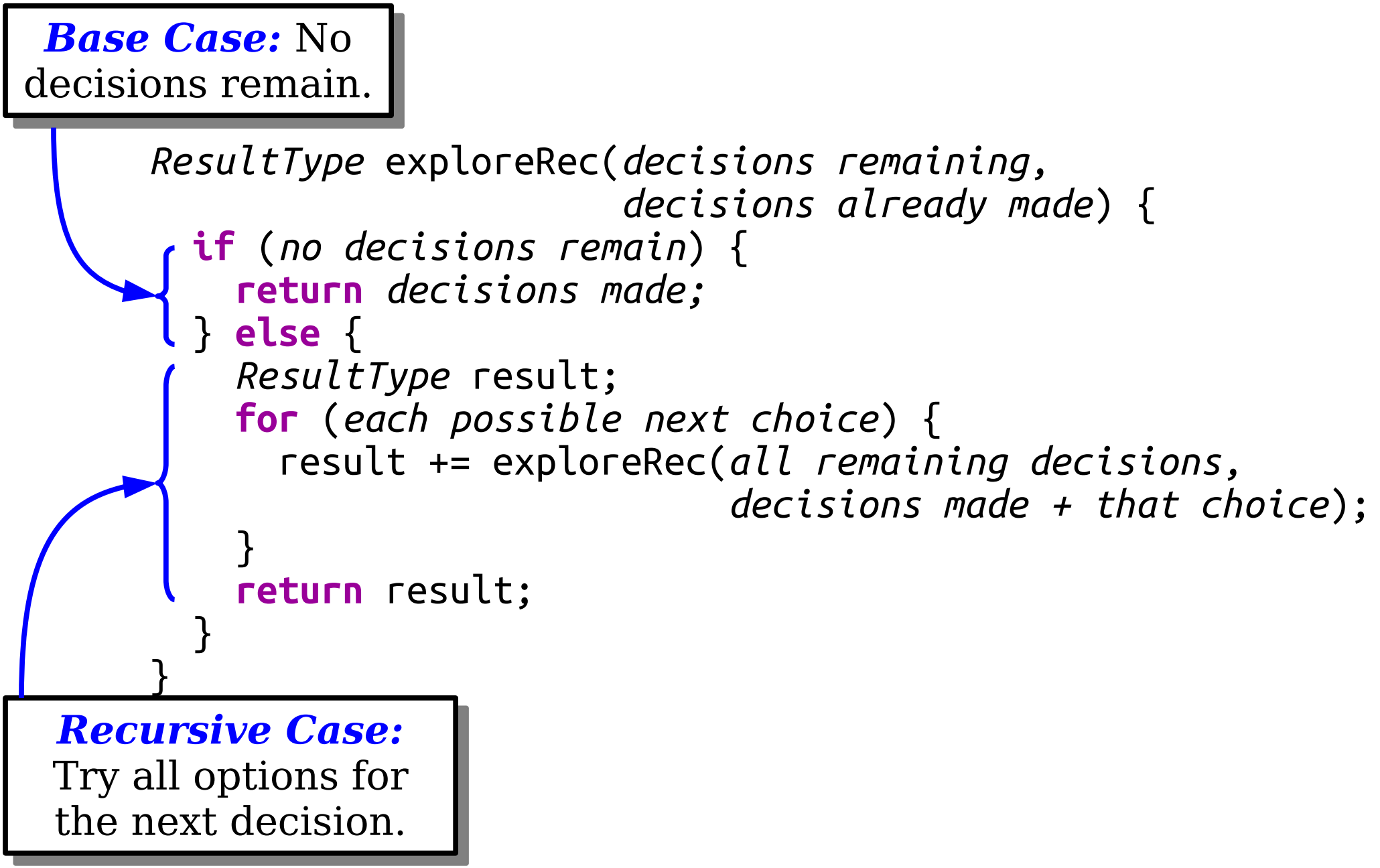
Recap from Last Time


List all *subsets* of {A, H, I}
Each decision is of the form “do I include this item?”
A?
{A,H,I}
{ }
H?
{H, I}
{A}
H?
{H, I}
{ }
I?
{ I }
{A, H}
I?
{ I }
{A}
I?
{ I }
{H}
I?
{ I }
{ }
{A,H,I}
{A, H}
{A, I}
{A}
{H, I}
{H}
{I}
{ }

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

Each decision is of the form “which item do I pick next?”



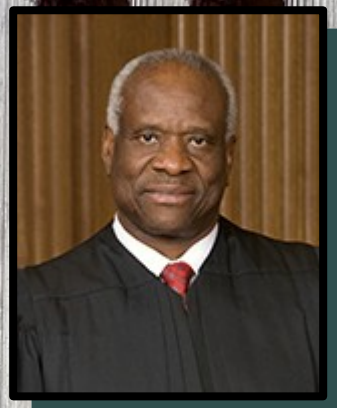

Base Case: No decisions remain.
ResultType exploreRec(decisions remaining,
decisions already made) {
if (no decisions remain) {
return decisions made;
} else {
ResultType result;
for (each possible next choice) {
result += exploreRec(all remaining decisions,
decisions made + that choice);
}
return result;
}
}
Recursive Case: Try all options for the next decision.
ResultType exploreAllTheThings(initial state) {
return exploreRec(initial state, no decisions made);
}

New Stuff!

Enumerating Combinations

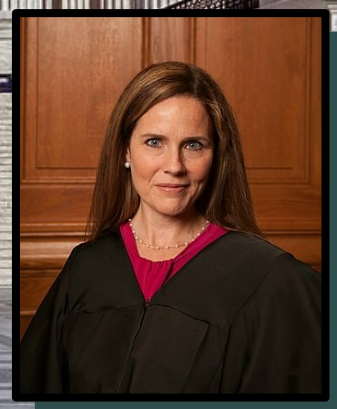
You need at least five US Supreme Court justices to agree to set a precedent.

What are all the ways you can pick five justices from the US Supreme Court?





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

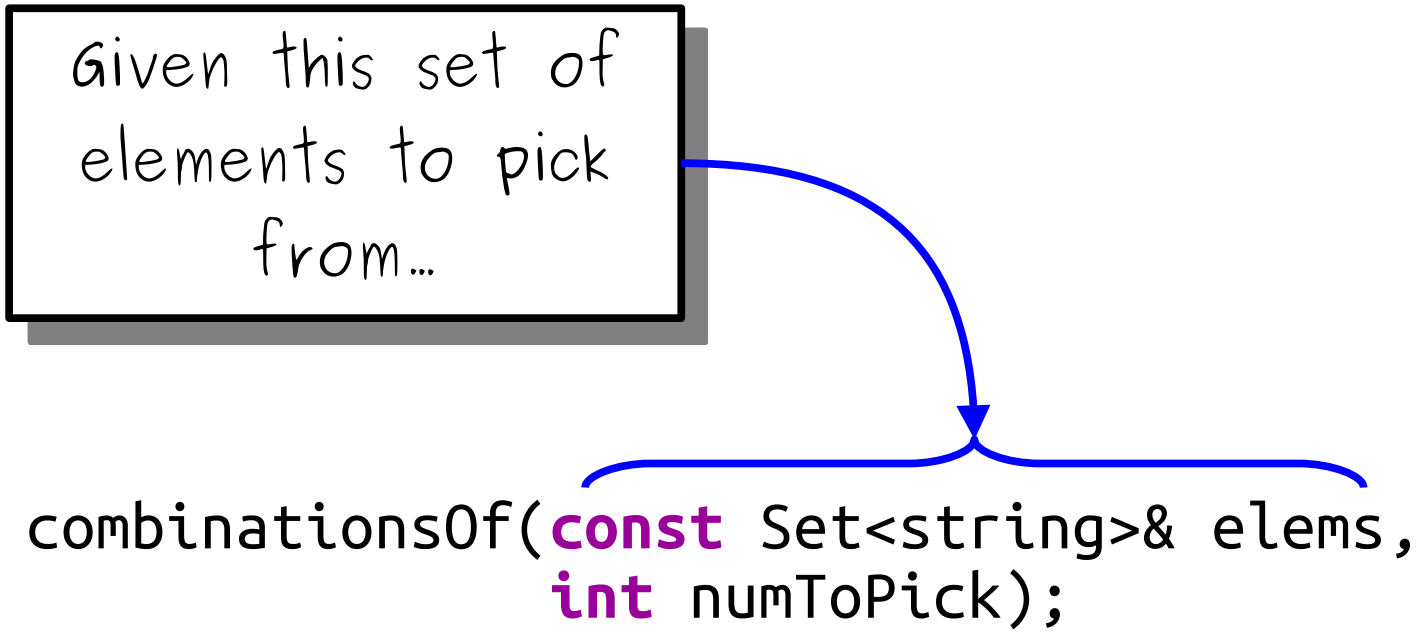
- Suppose that we want to find every way to choose exactly ***three*** elements from a set.
- We could do something like this:

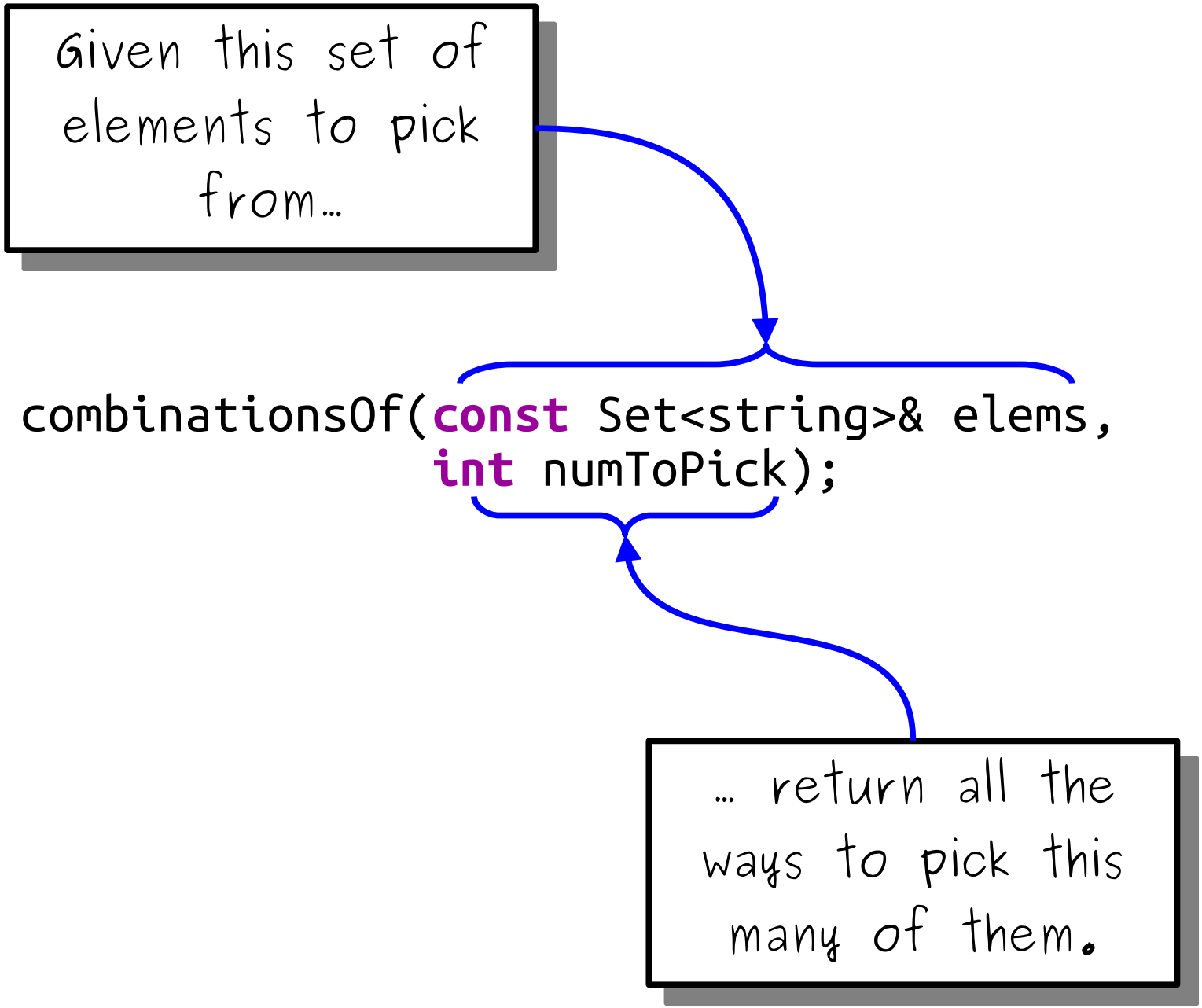
```
for (int x: mySet) {
  for (int y: mySet) {
    for (int z: mySet) {
      if (x != y && x != z && y != z) {
         cout << x << ", " << y << ", " << z << endl;
      }
    }
  }
}
```

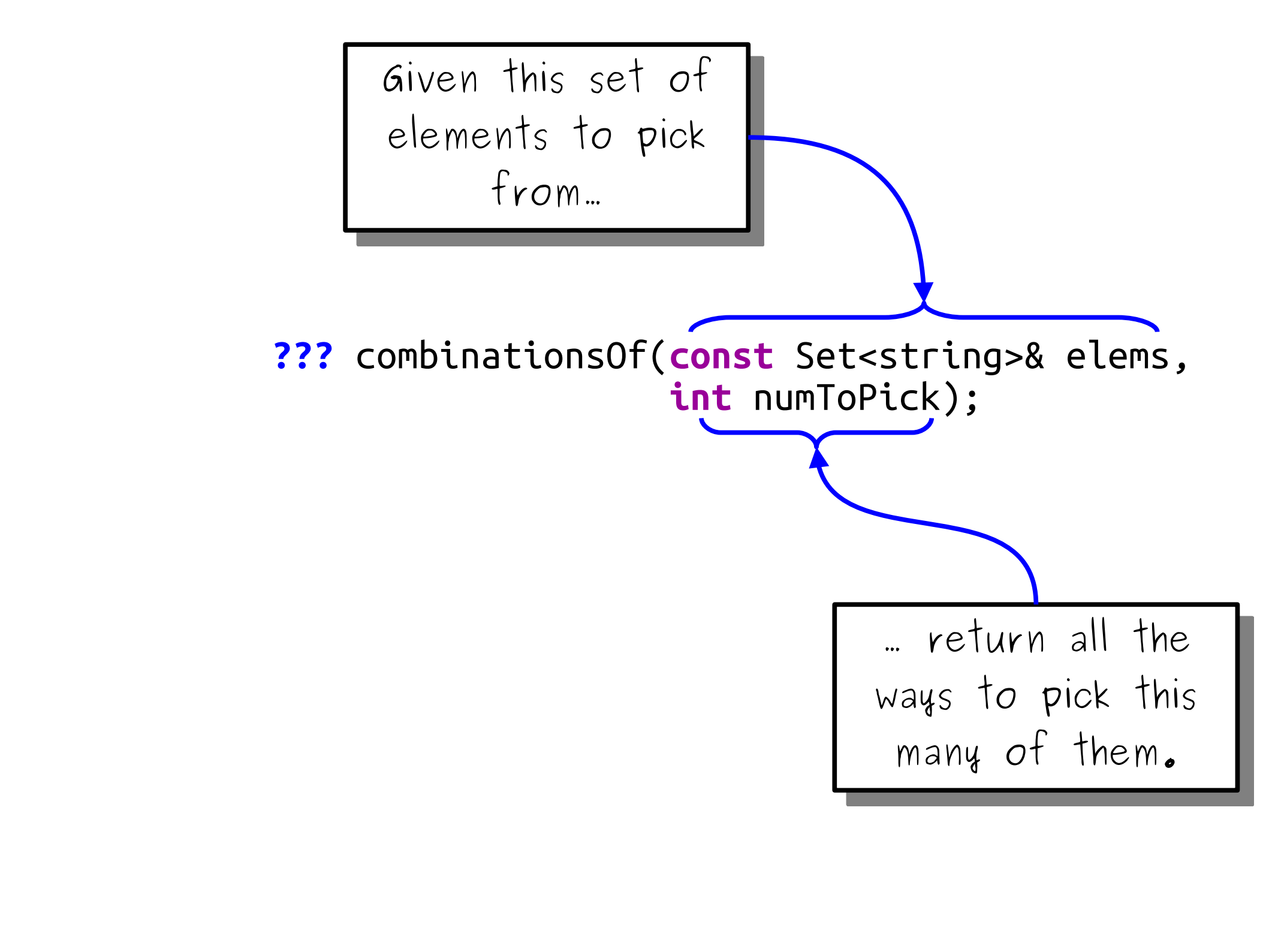
Generating Combinations

- 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 reasonably large number and we don’t want to write a huge number of nested loops and complicated `if` statements?

```
combinationsOf(const Set<string>& elems,
               int numToPick);
```


Given this set of elements to pick from…
combinationsOf(const Set<string>& elems,
int numToPick);


Given this set of elements to pick from...
combinationsOf(const Set<string>& elems,
int numToPick);
... return all the ways to pick this many of them.


Given this set of elements to pick from...
??? combinationsOf(const Set<string>& elems,
int numToPick);
... return all the ways to pick this many of them.

Given this set of elements to pick from…

```
??? combinationsOf(const Set<string>& elems,
                   int numToPick);
```

… return all the ways to pick this many of them.

What should this function's return type be?

Formulate a hypothesis, but ***don't post anything in chat just yet***.

Given this set of elements to pick from…

```
??? combinationsOf(const Set<string>& elems,
                   int numToPick);
```

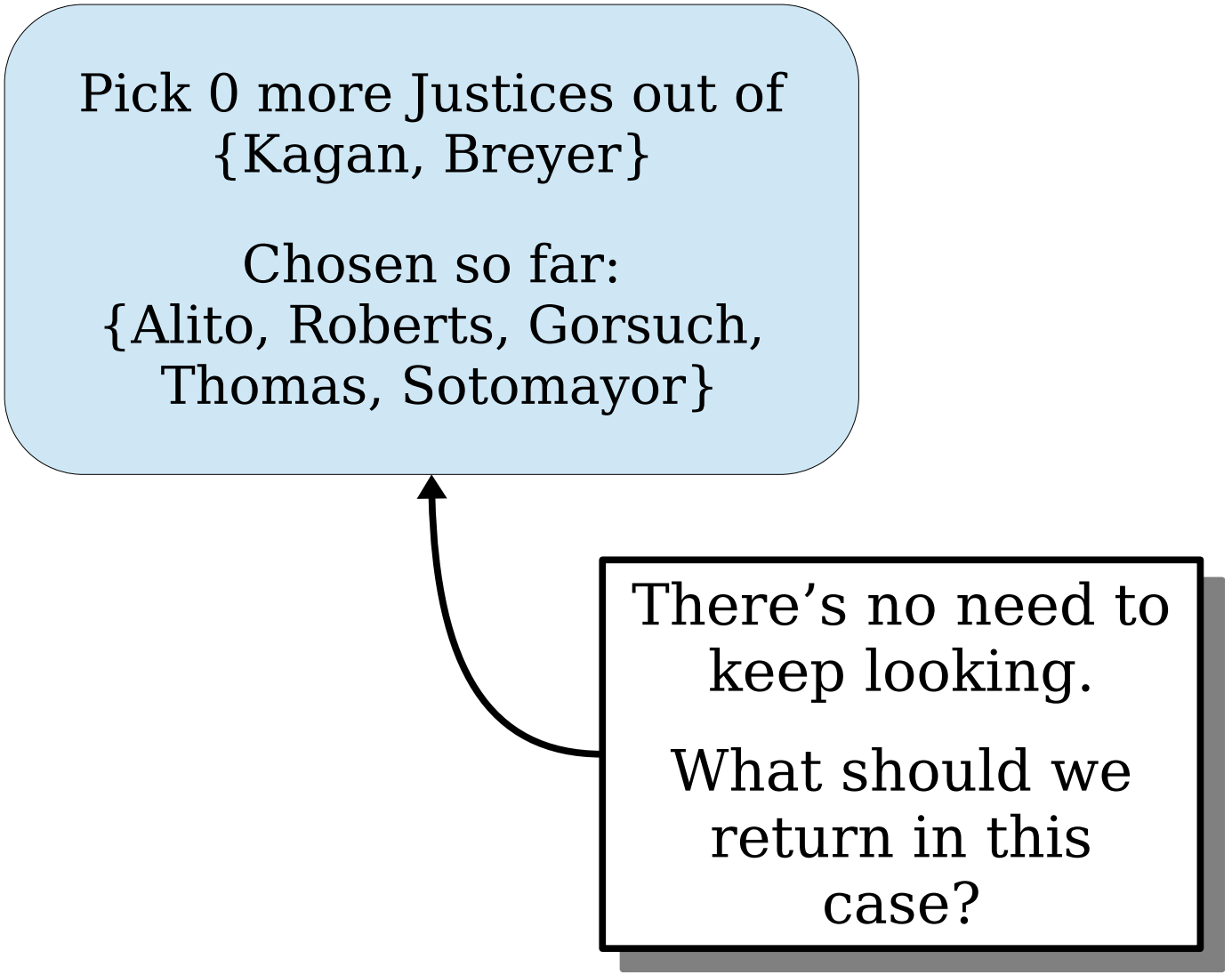
… return all the ways to pick this many of them.

What should this function's return type be?

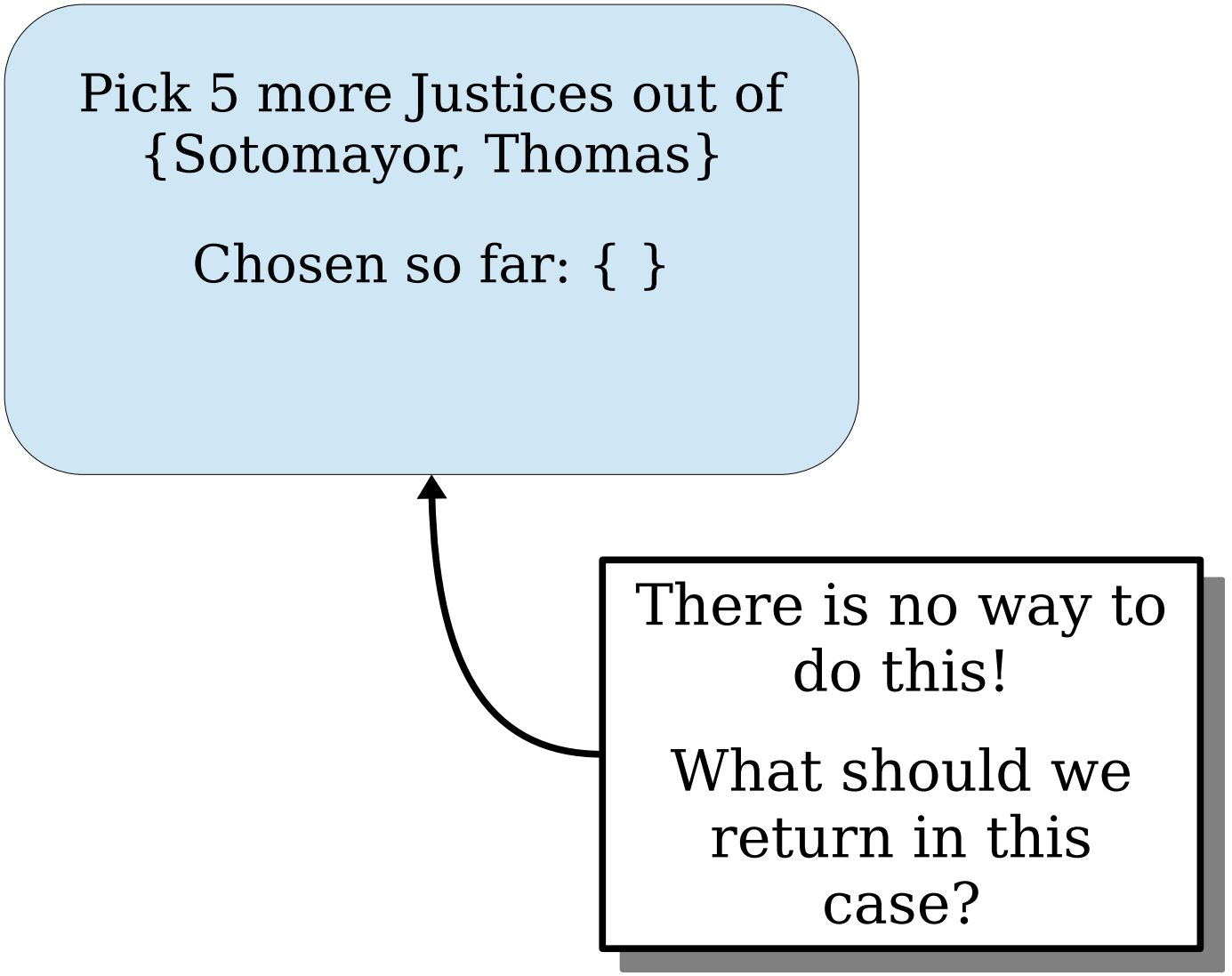
Now, ***private chat me your best guess***. Not sure? Just answer "??"

Implementing Combinations

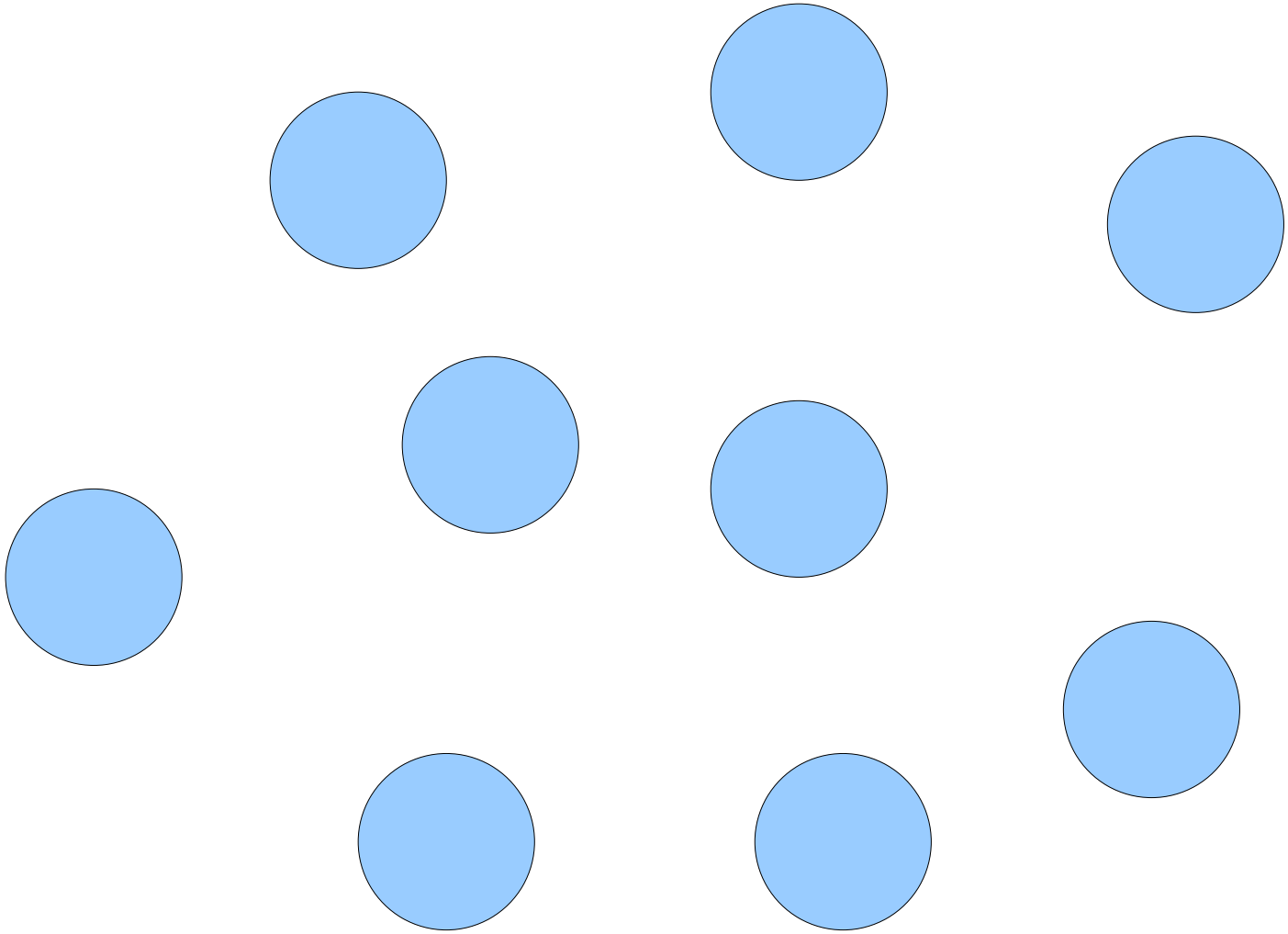
Our Base Case



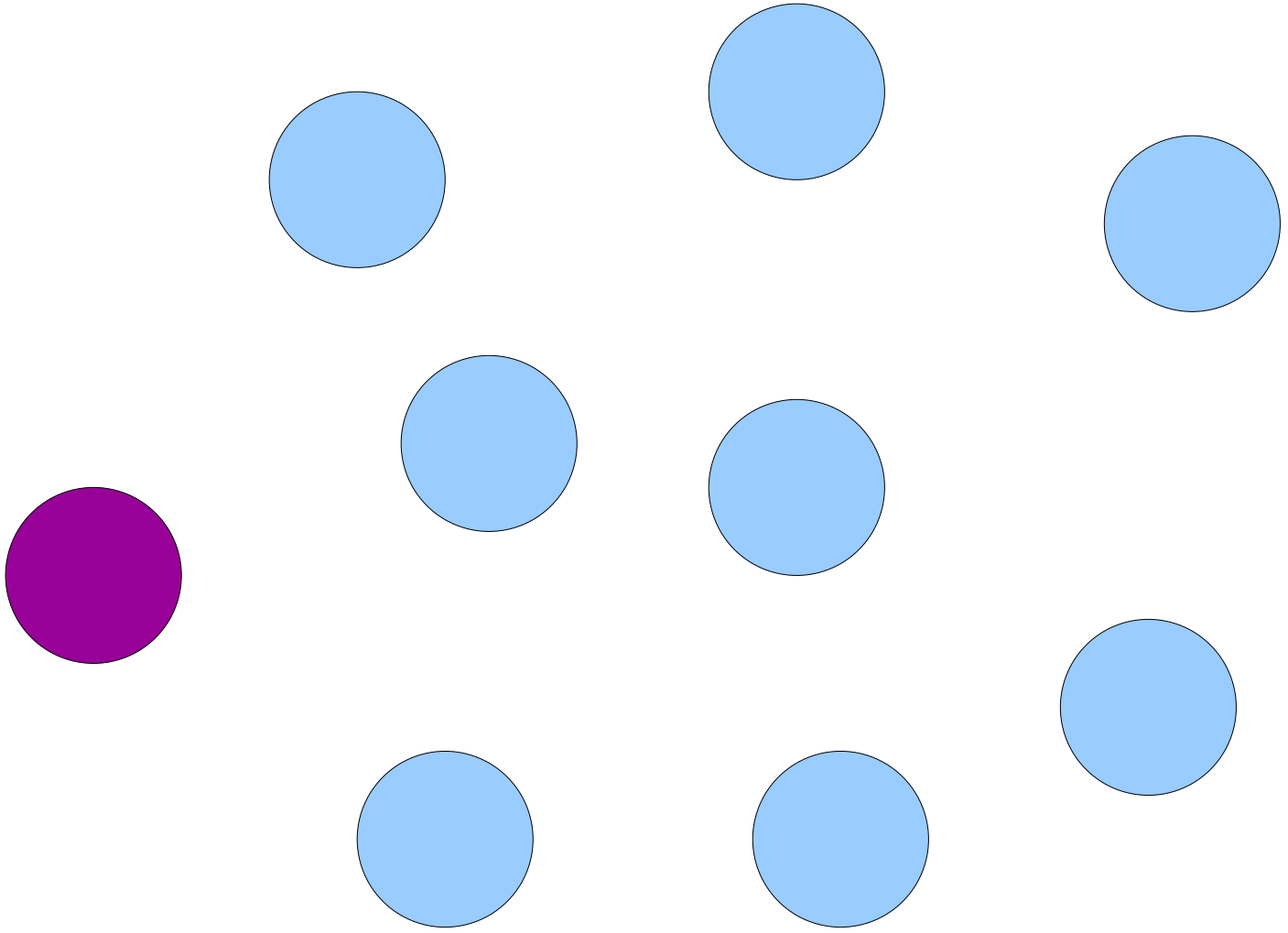
Our Base Case, Part II



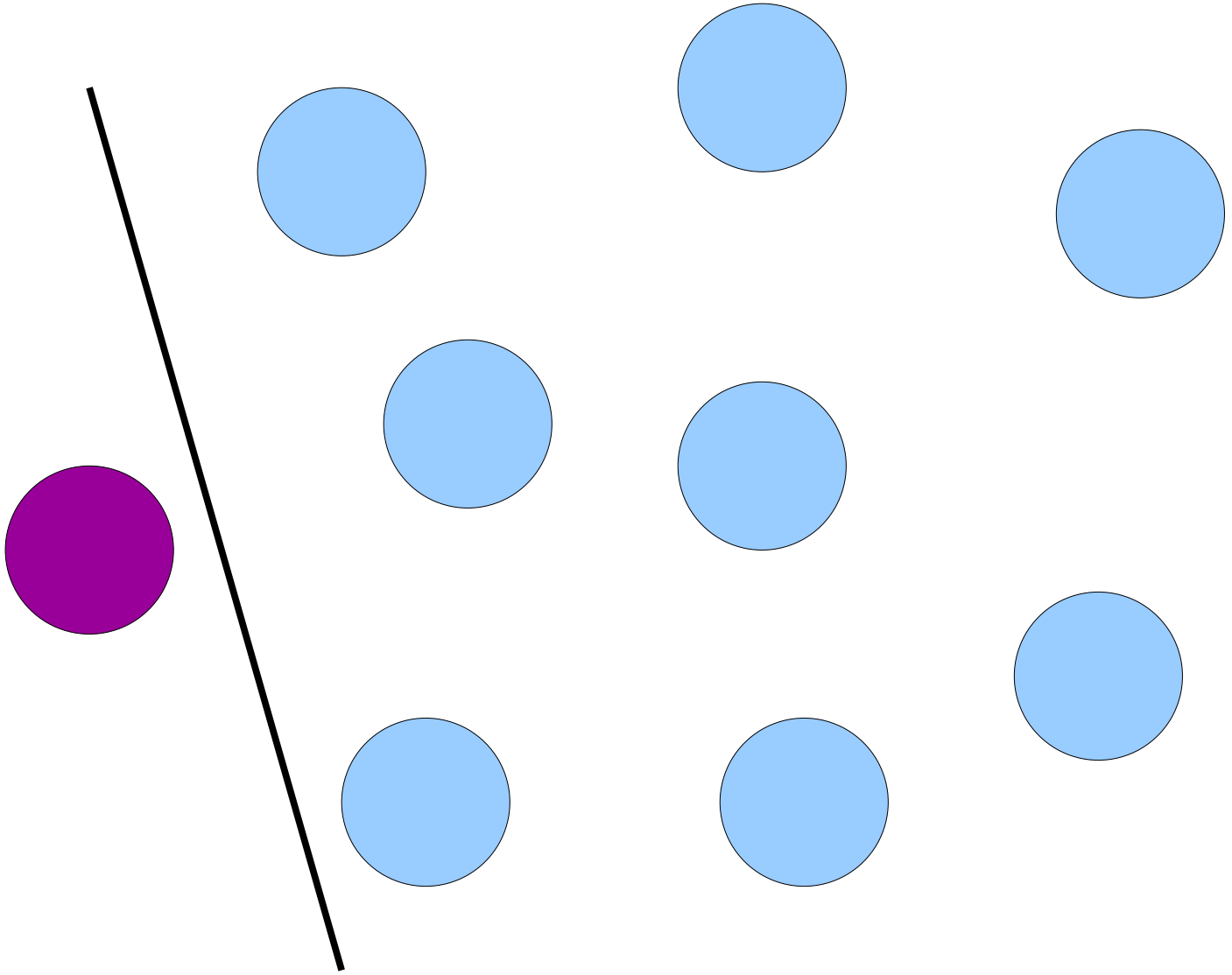
Generating Combinations



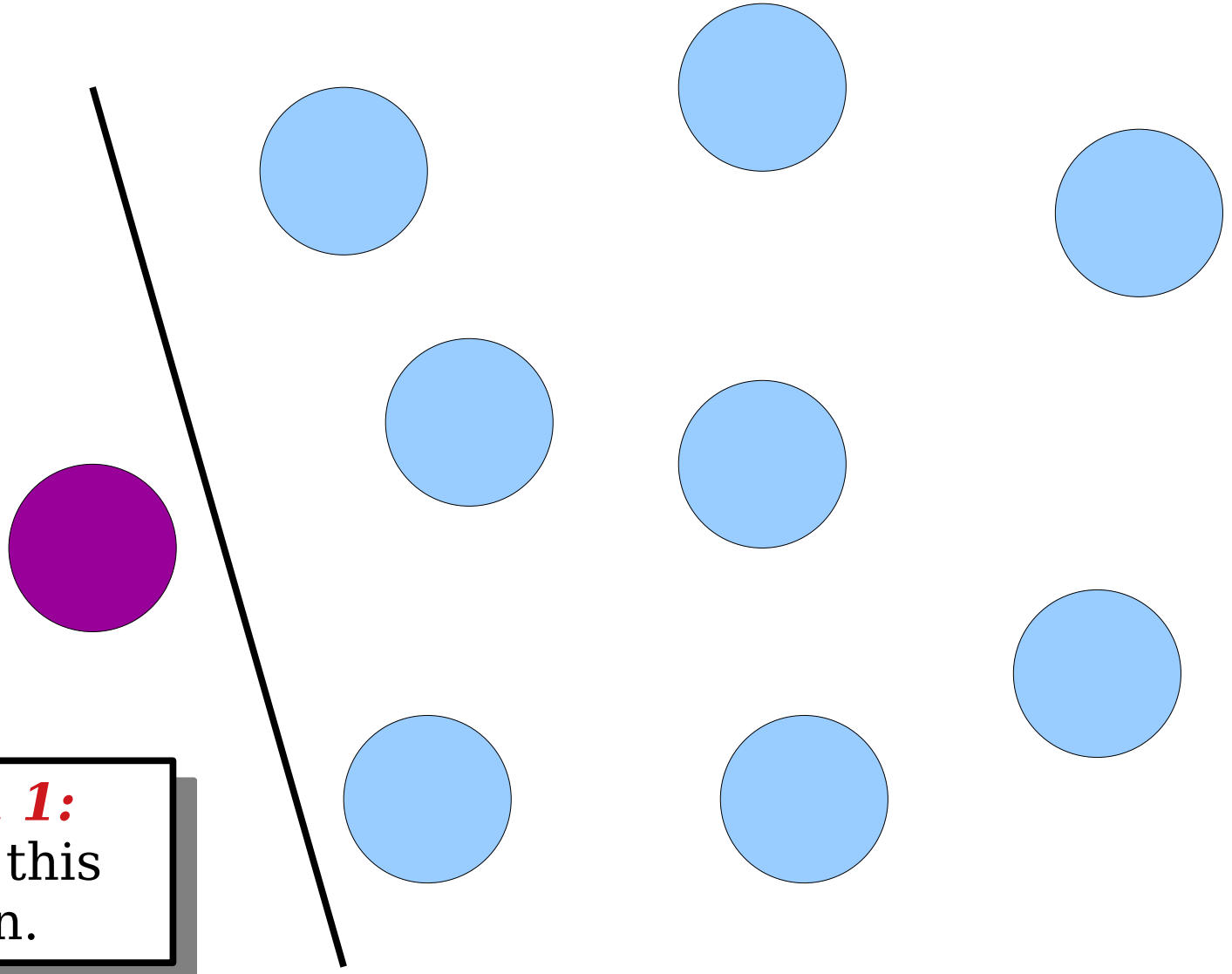
Generating Combinations



Generating Combinations

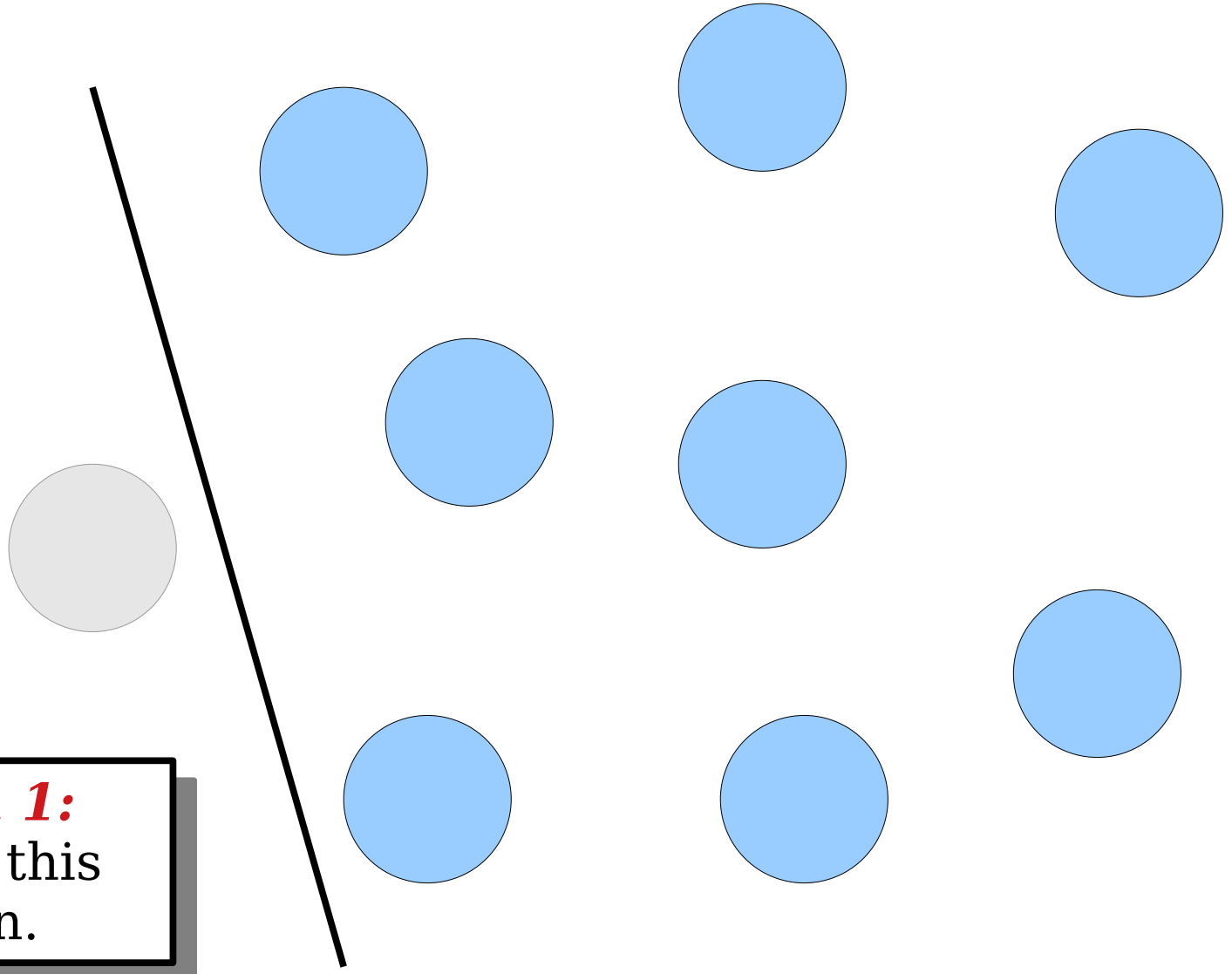


Generating Combinations



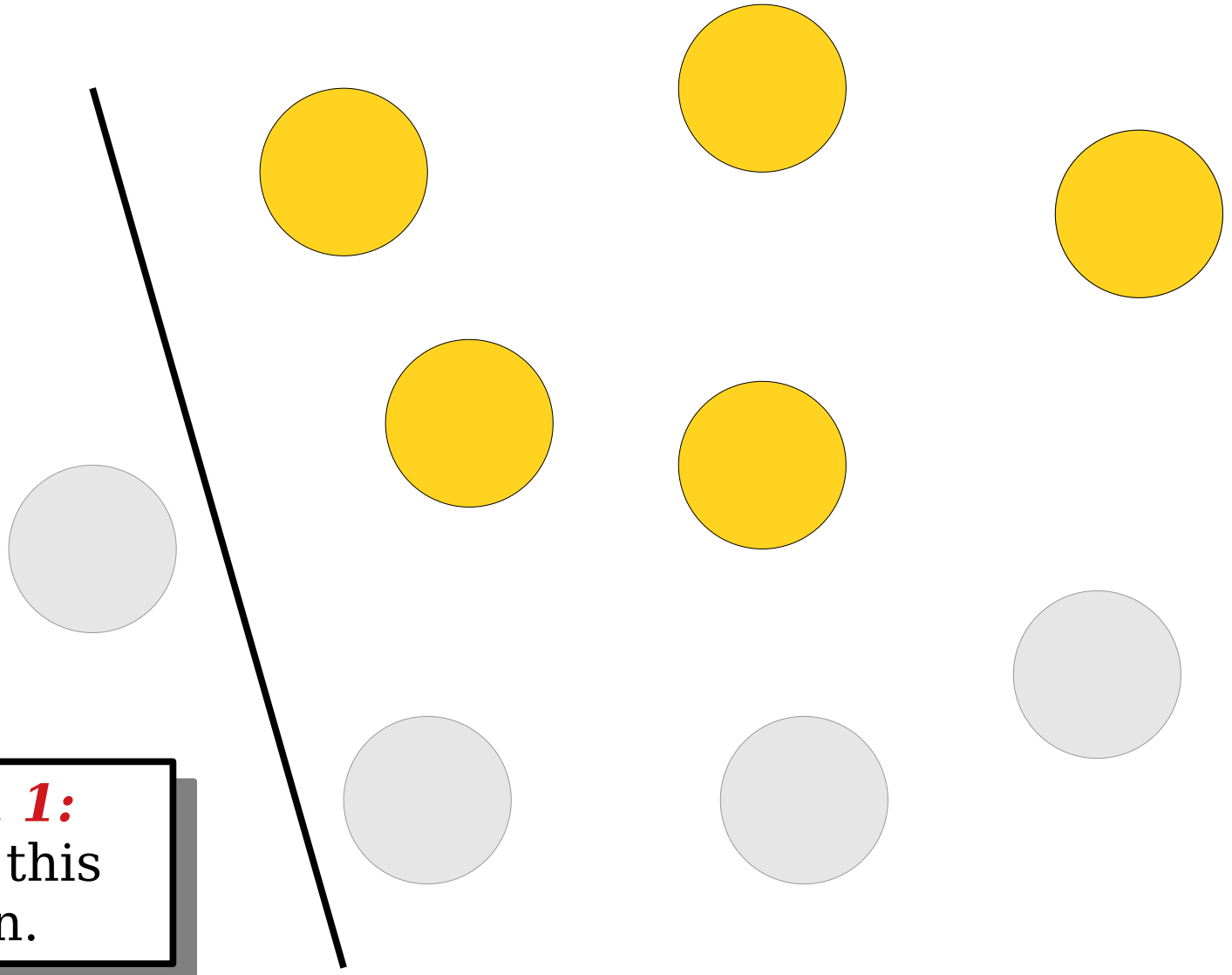
Option 1:
Exclude this person.

Generating Combinations



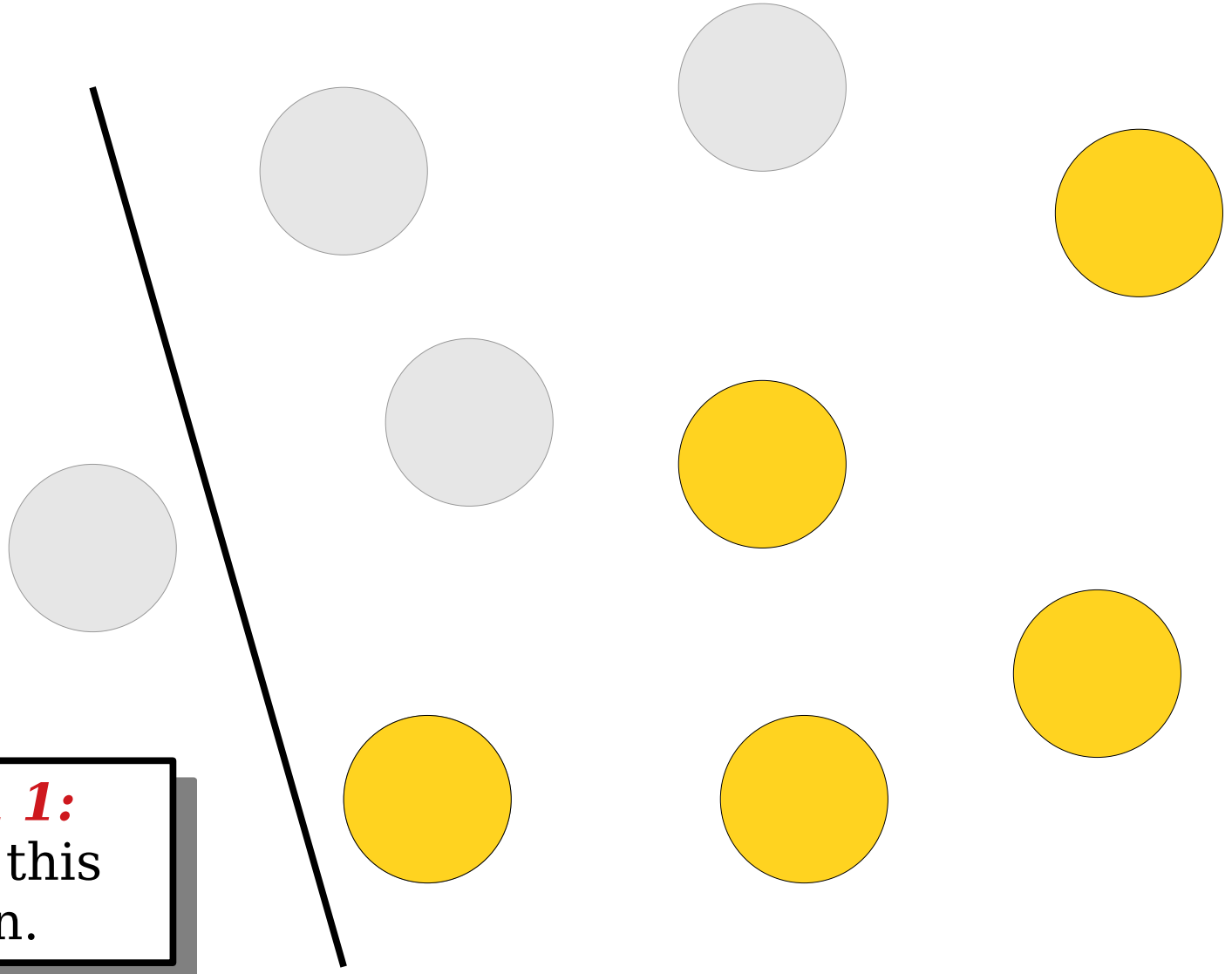
Option 1:
Exclude this person.

Generating Combinations



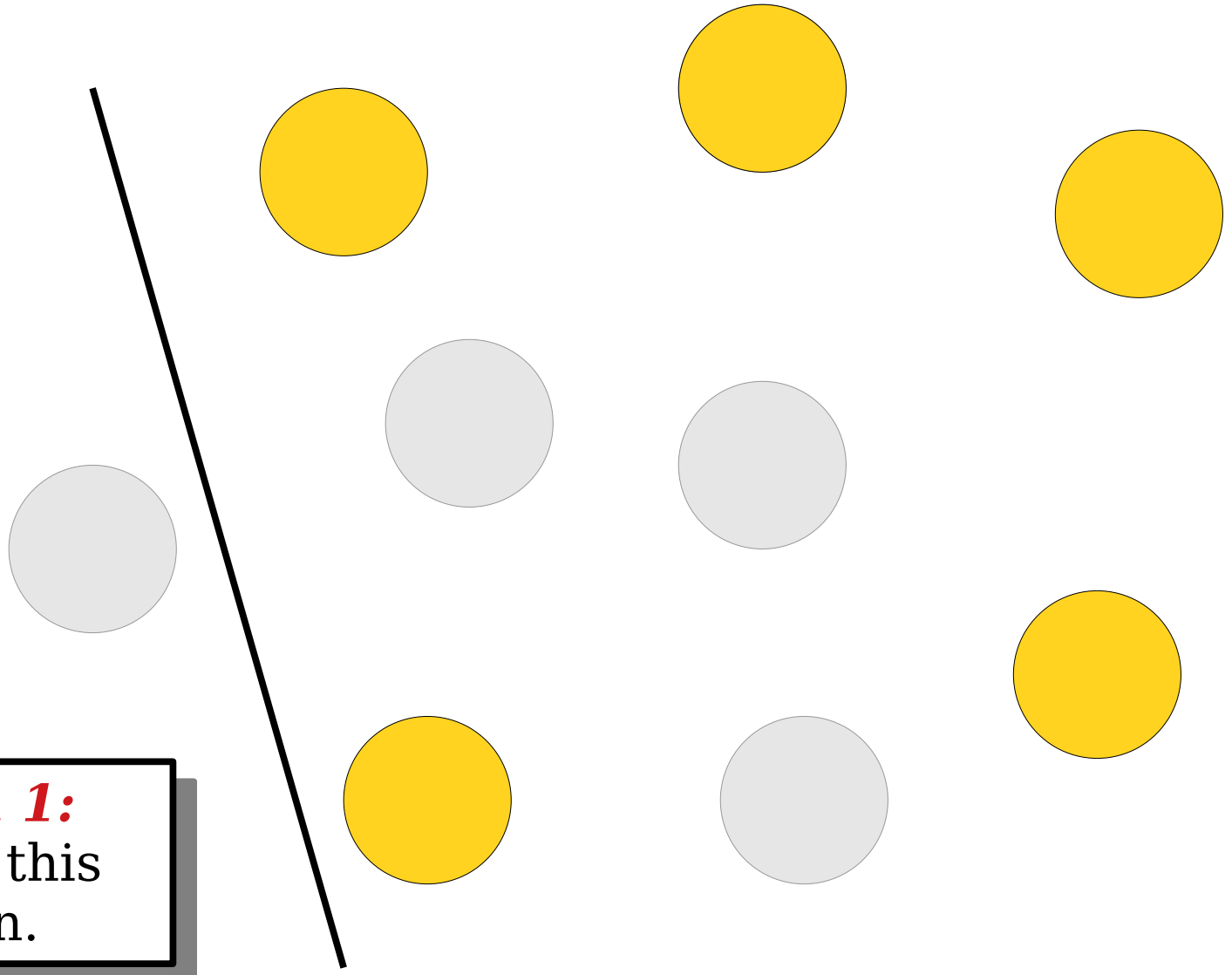
Option 1:
Exclude this person.

Generating Combinations



Option 1:
Exclude this person.

Generating Combinations



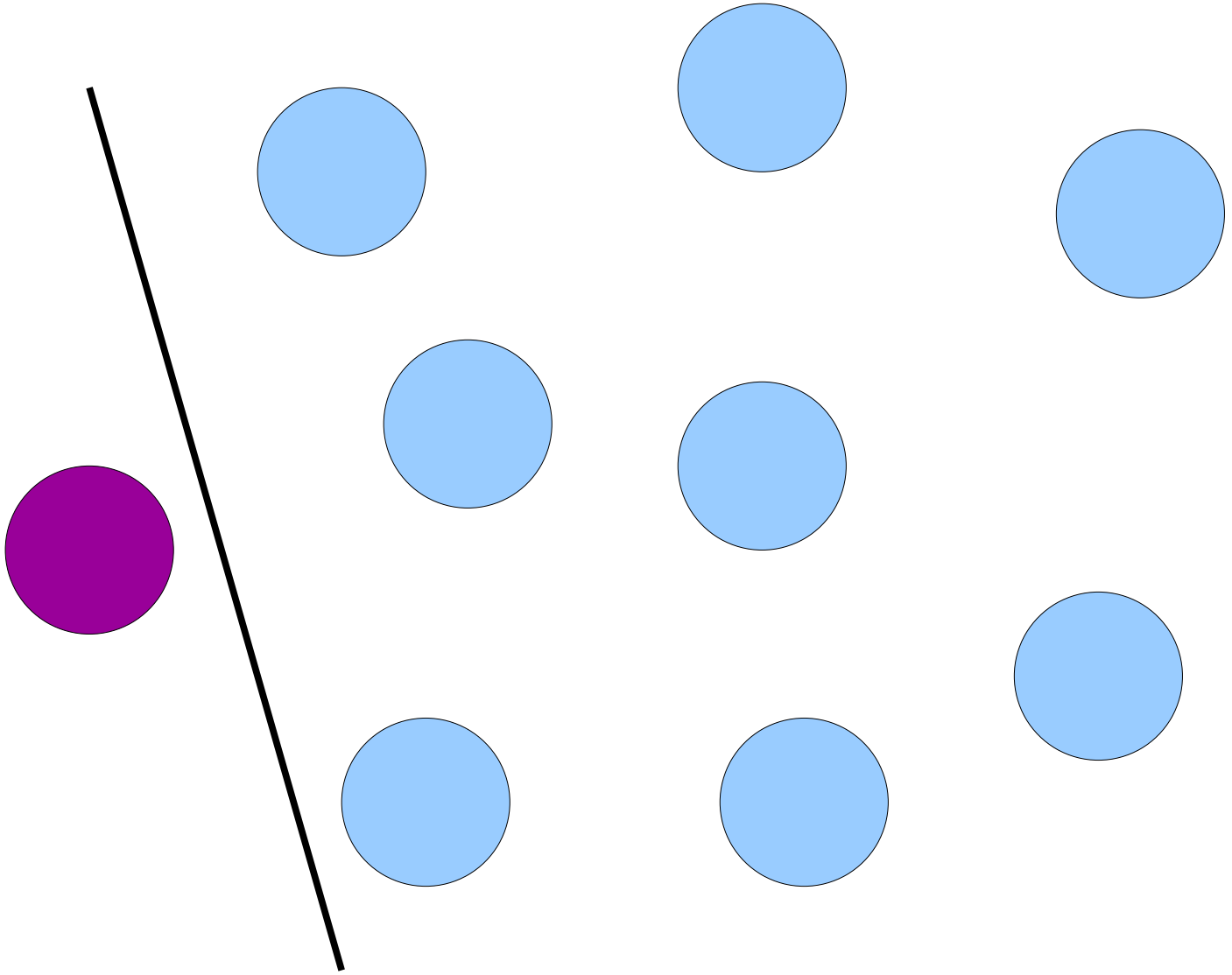
Option 1:
Exclude this person.

Generating Combinations

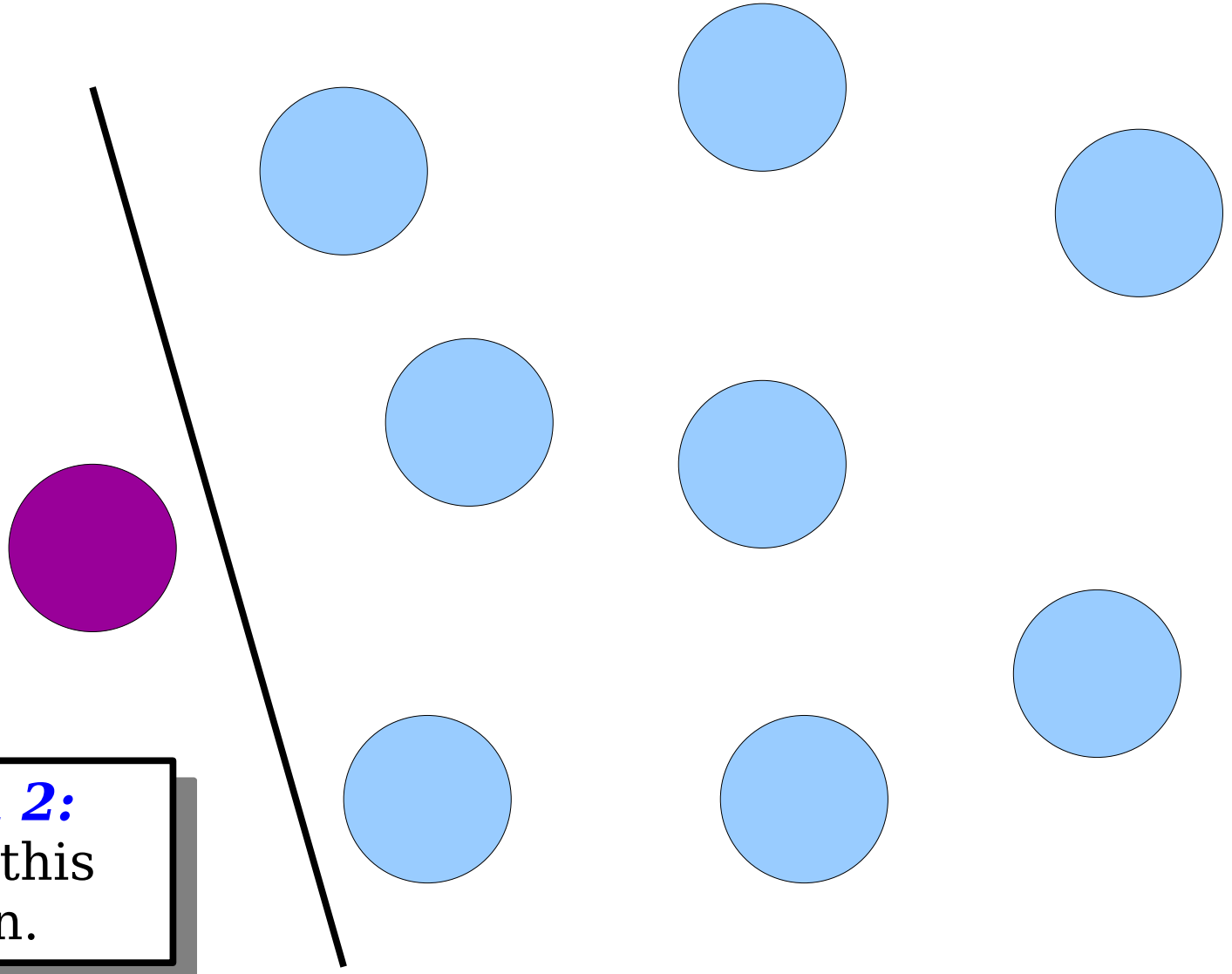
One way to choose **5** elements out of **9** is to exclude the first element, then to choose **5** elements out of the remaining **8.**

Option 1:
Exclude this person.

Generating Combinations

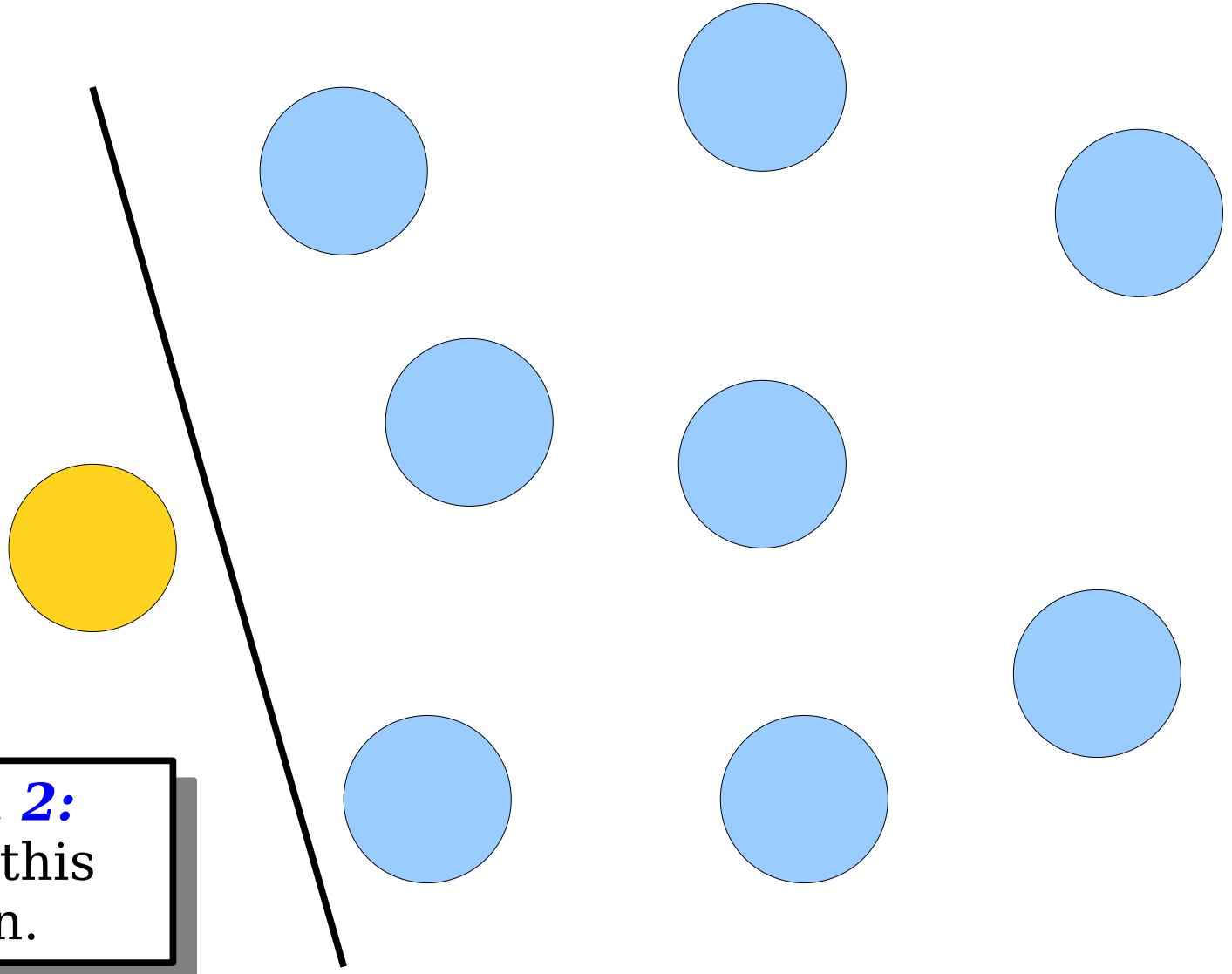


Generating Combinations



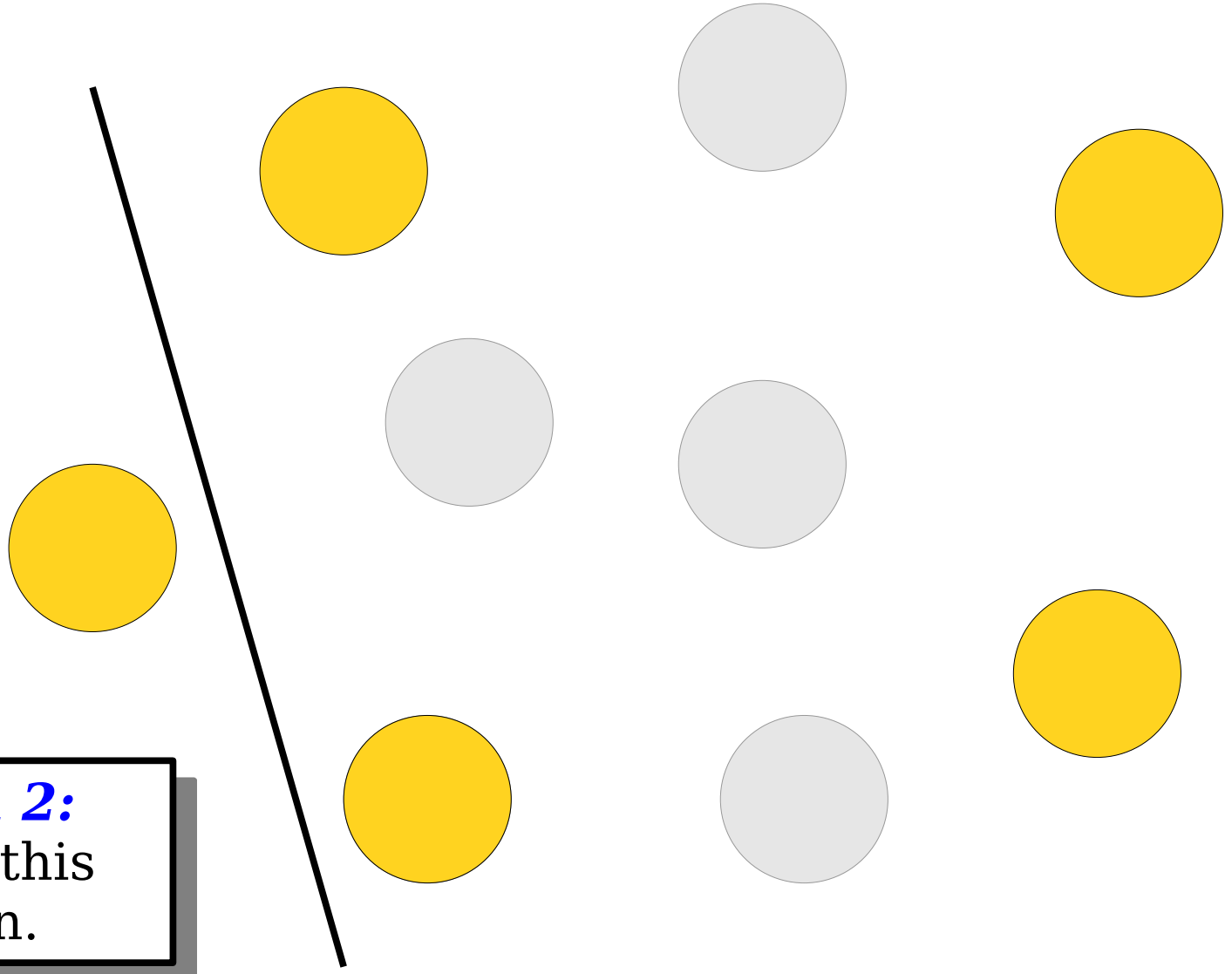
Option 2:
Include this person.

Generating Combinations



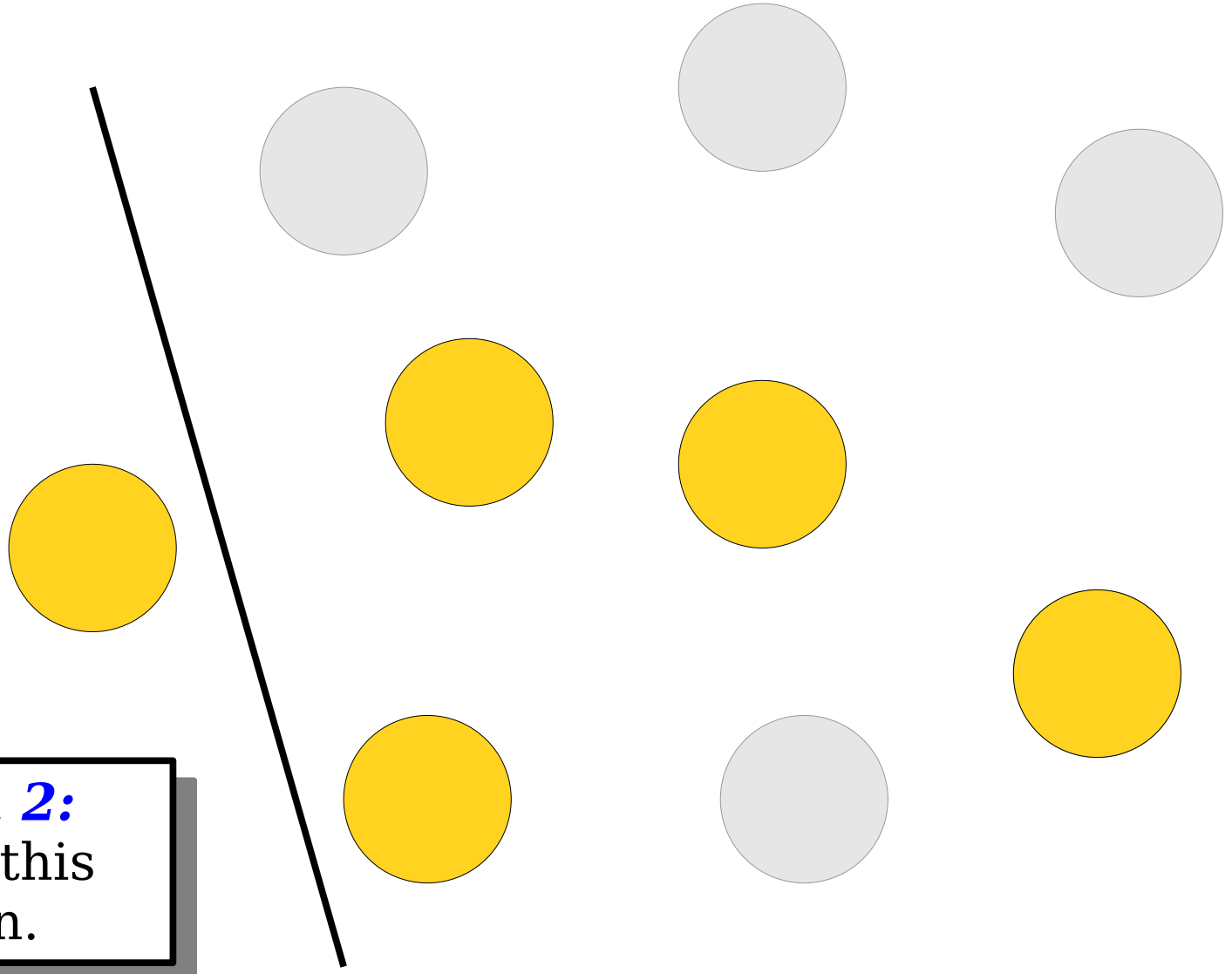
Option 2:
Include this person.

Generating Combinations



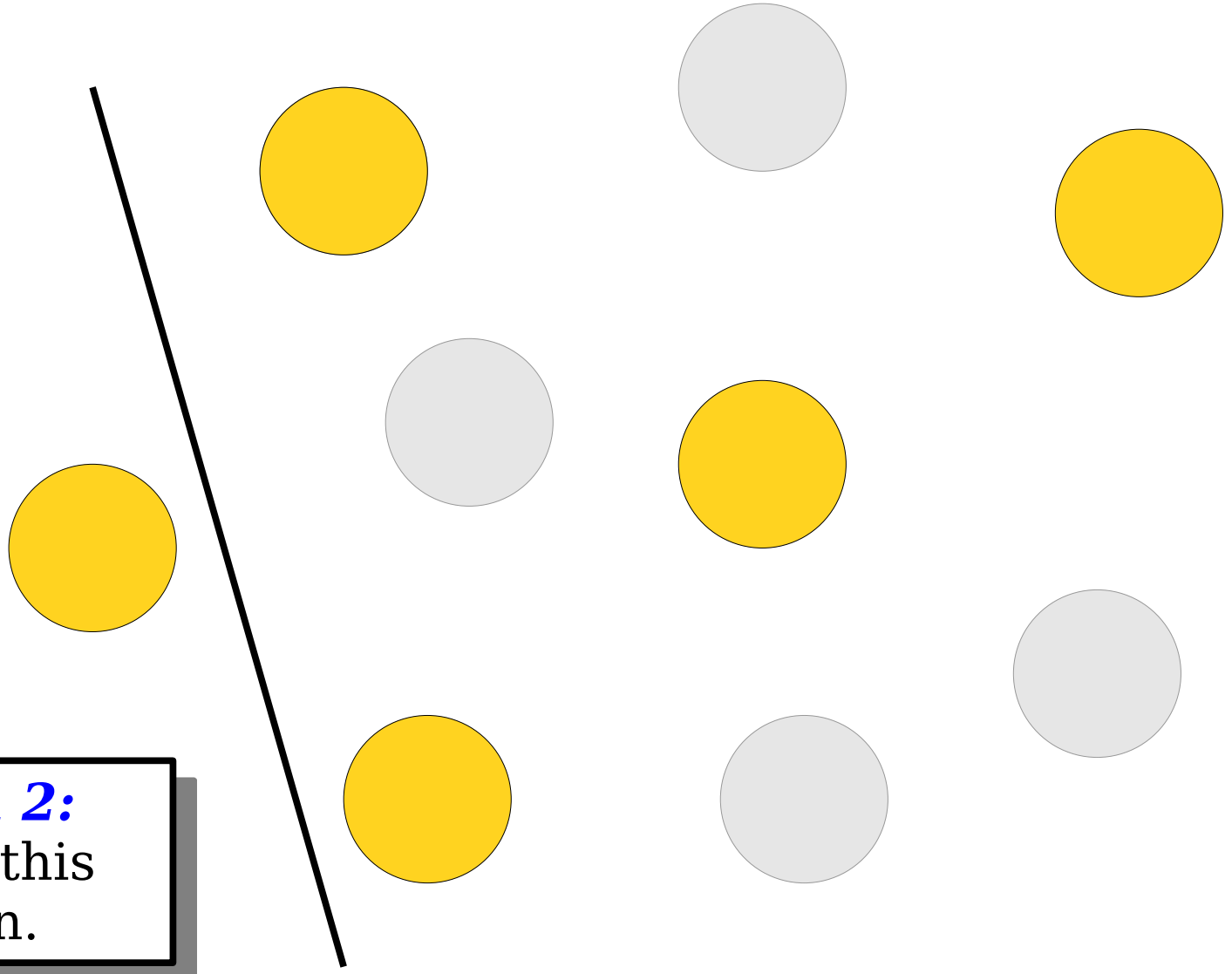
Option 2:
Include this person.

Generating Combinations



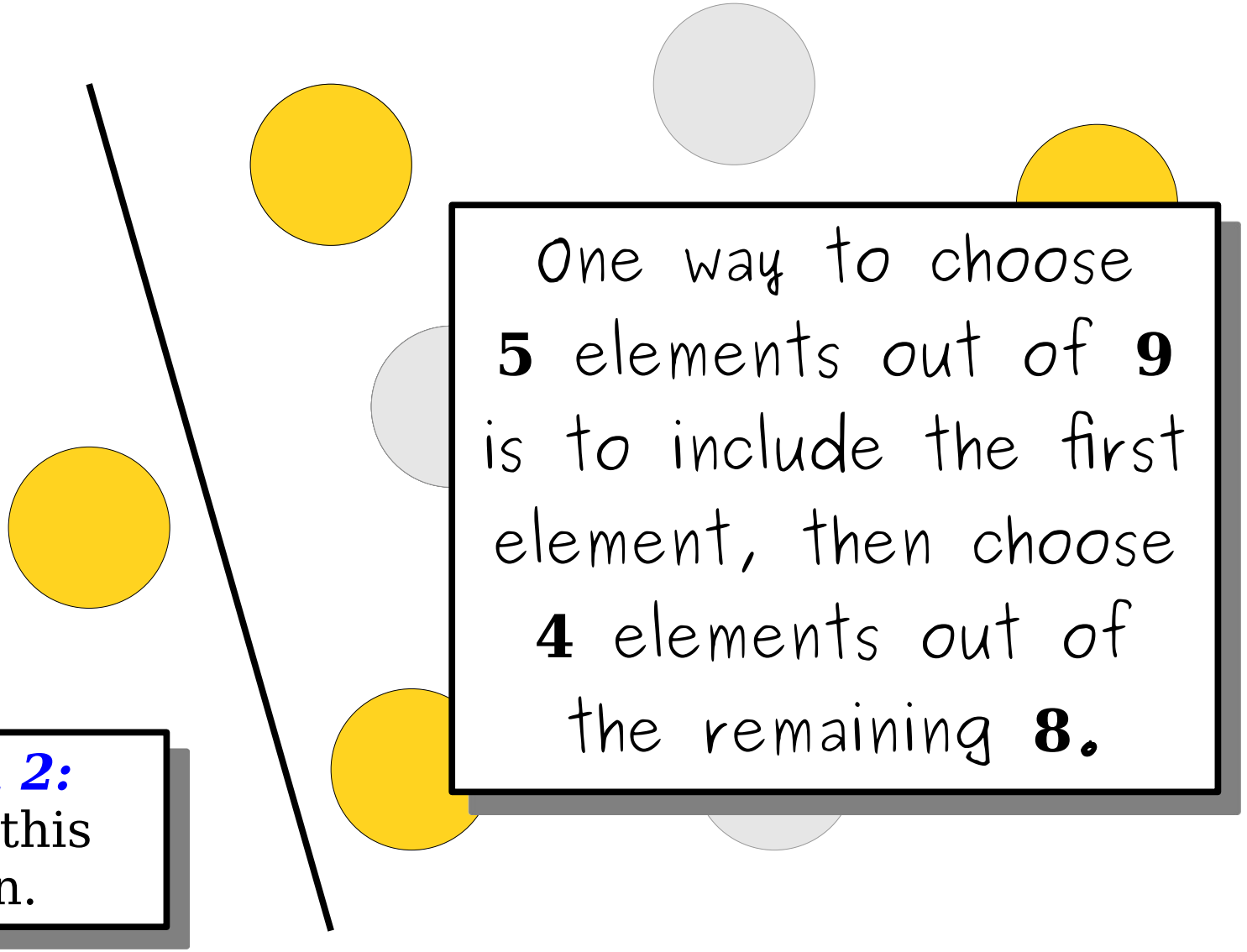
Option 2:
Include this person.

Generating Combinations



Option 2:
Include this person.

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.

A Comment on Types

The Wonderful `auto` Keyword

- There are many cases in which there is exactly one possible type that a variable could have.
- In that case, rather than explicitly writing out the type, you can use the `auto` keyword:

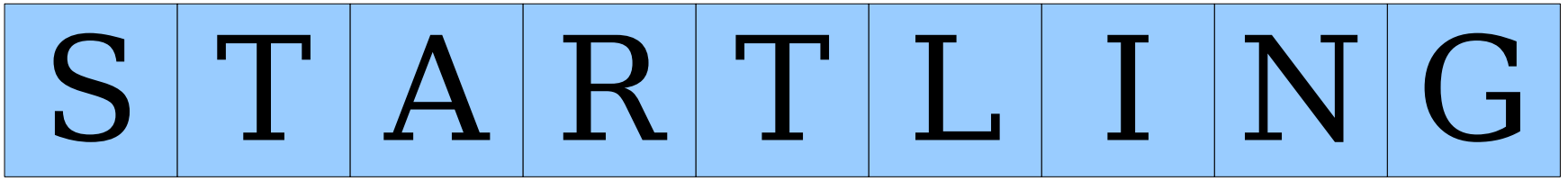
```
auto var = expression;
```

- While in principle you can use this in many places, we recommend just using it to save typing when working with container types.

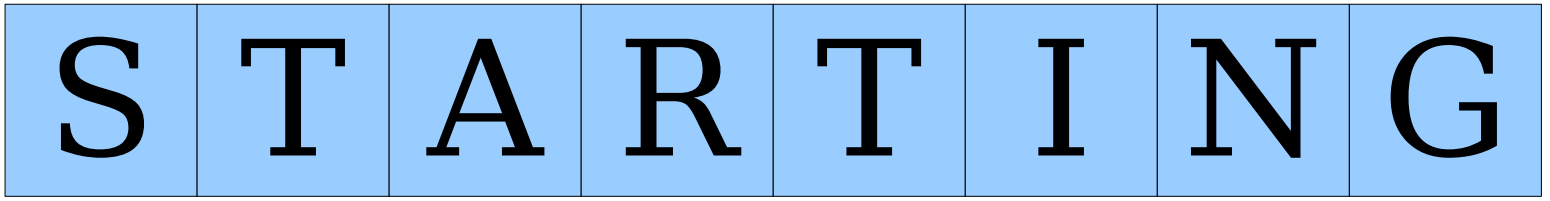
A Little Word Puzzle

“What nine-letter word can be reduced to a single-letter word one letter at a time by removing letters, leaving it a legal word at each step?”

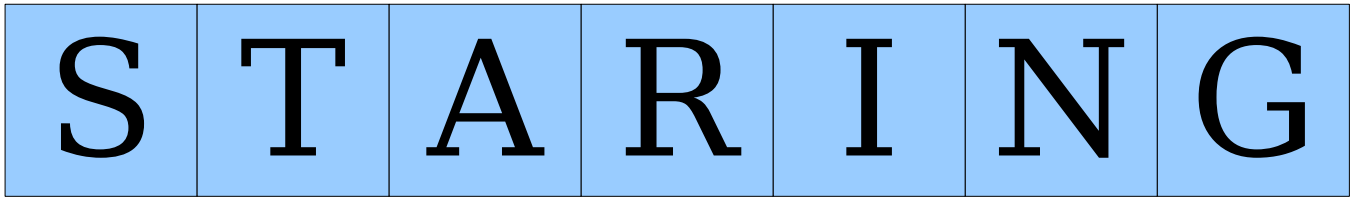
The Startling Truth?



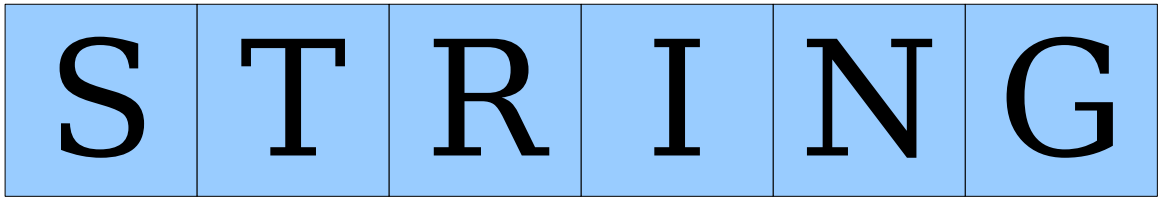
The Startling Truth?



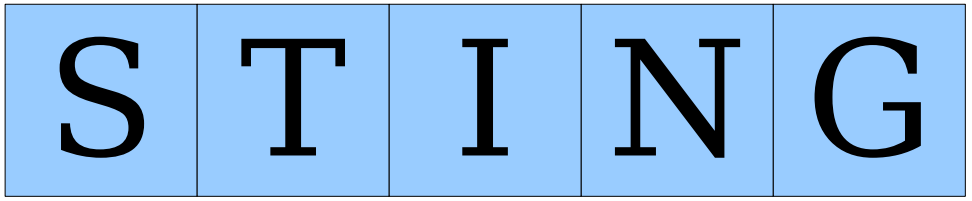
The Startling Truth?



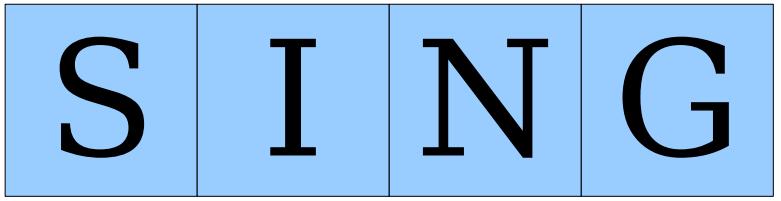
The Startling Truth?



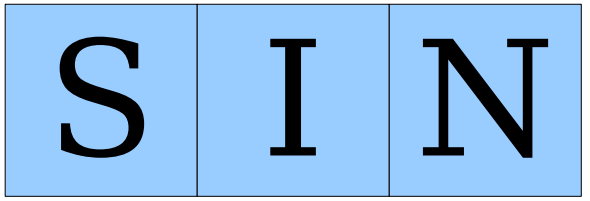
The Startling Truth?



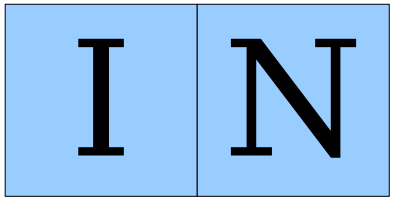
The Startling Truth?



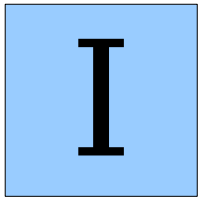
The Startling Truth?



The Startling Truth?

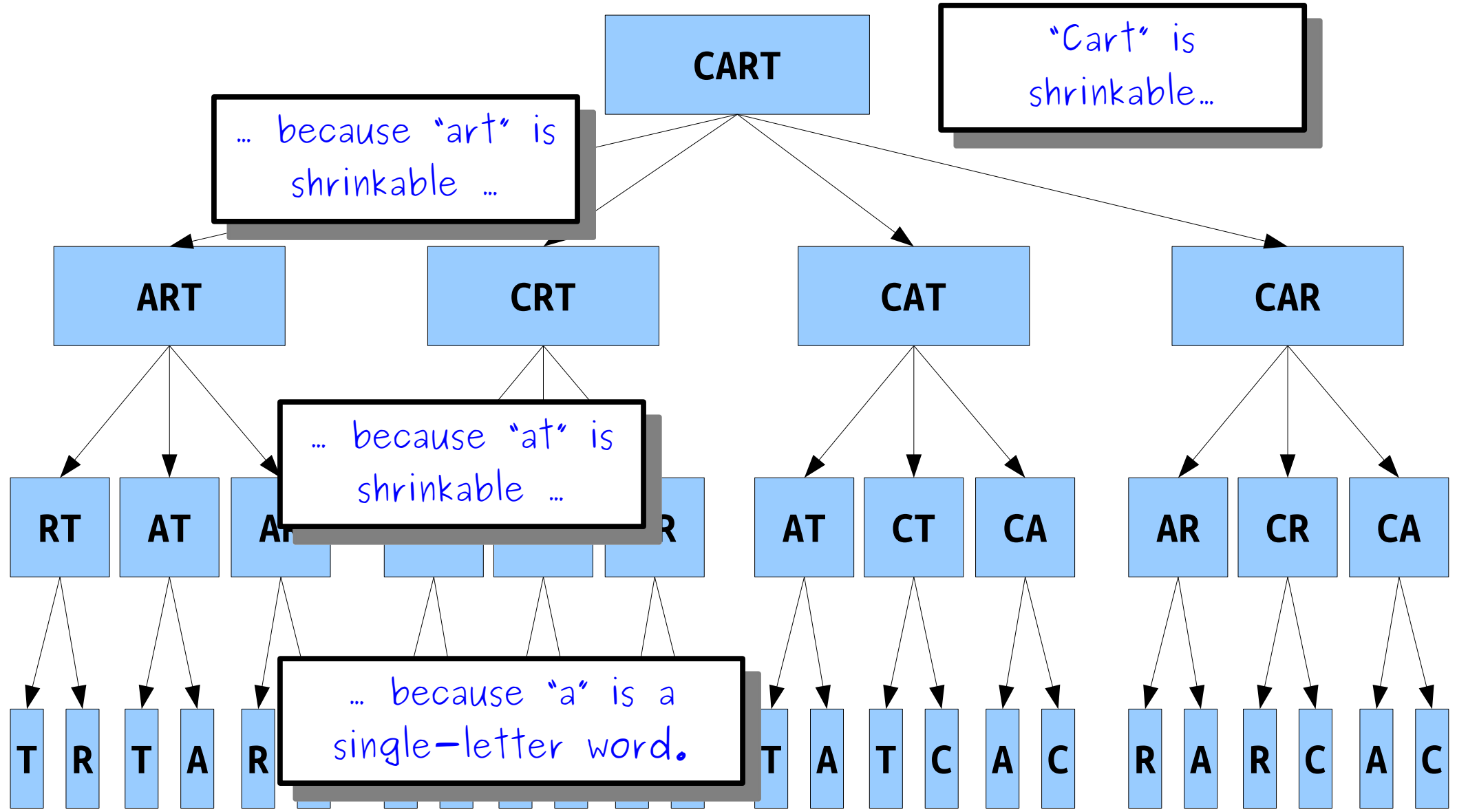


The Startling Truth?

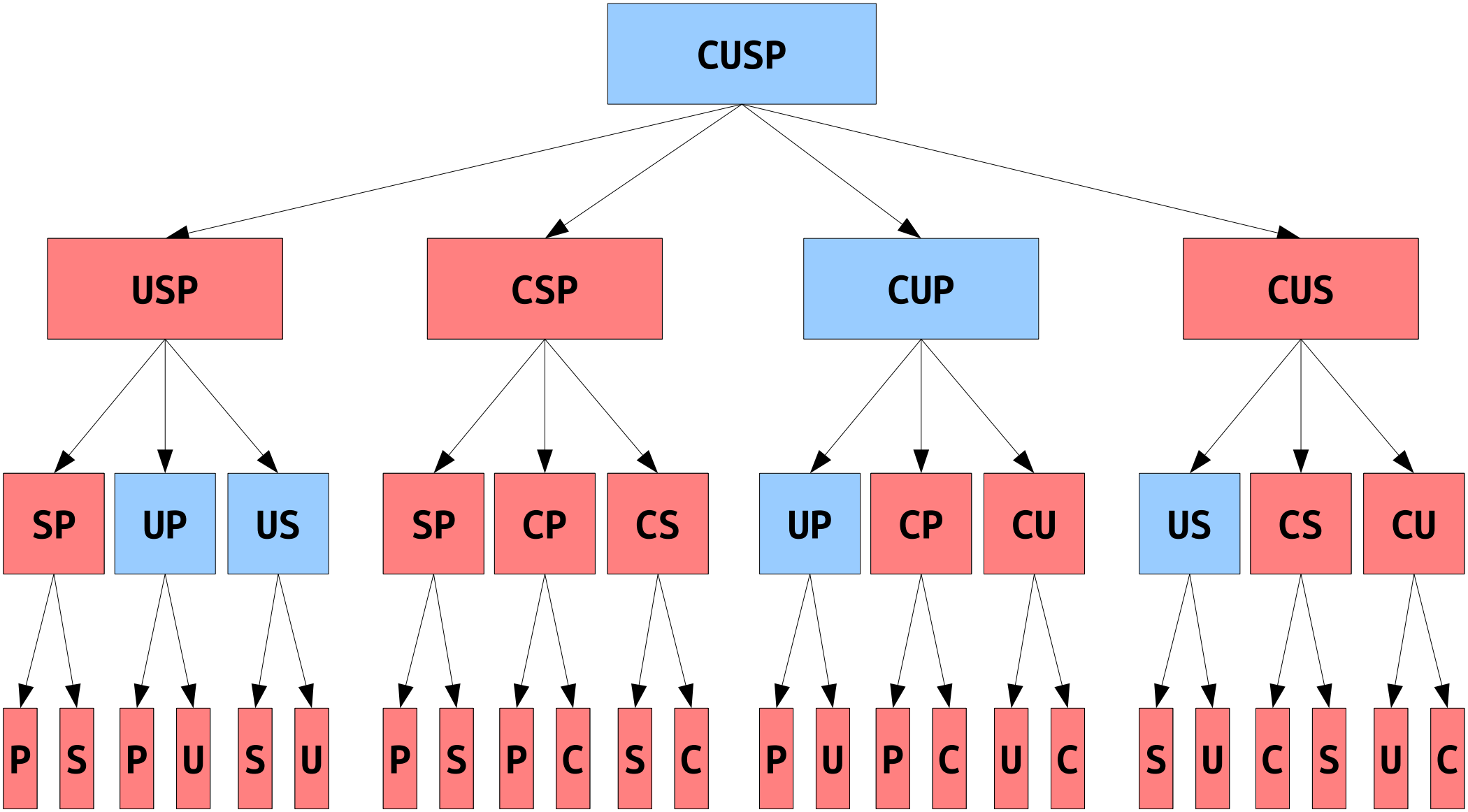


Is there ***really*** just one nine-letter word with this property?

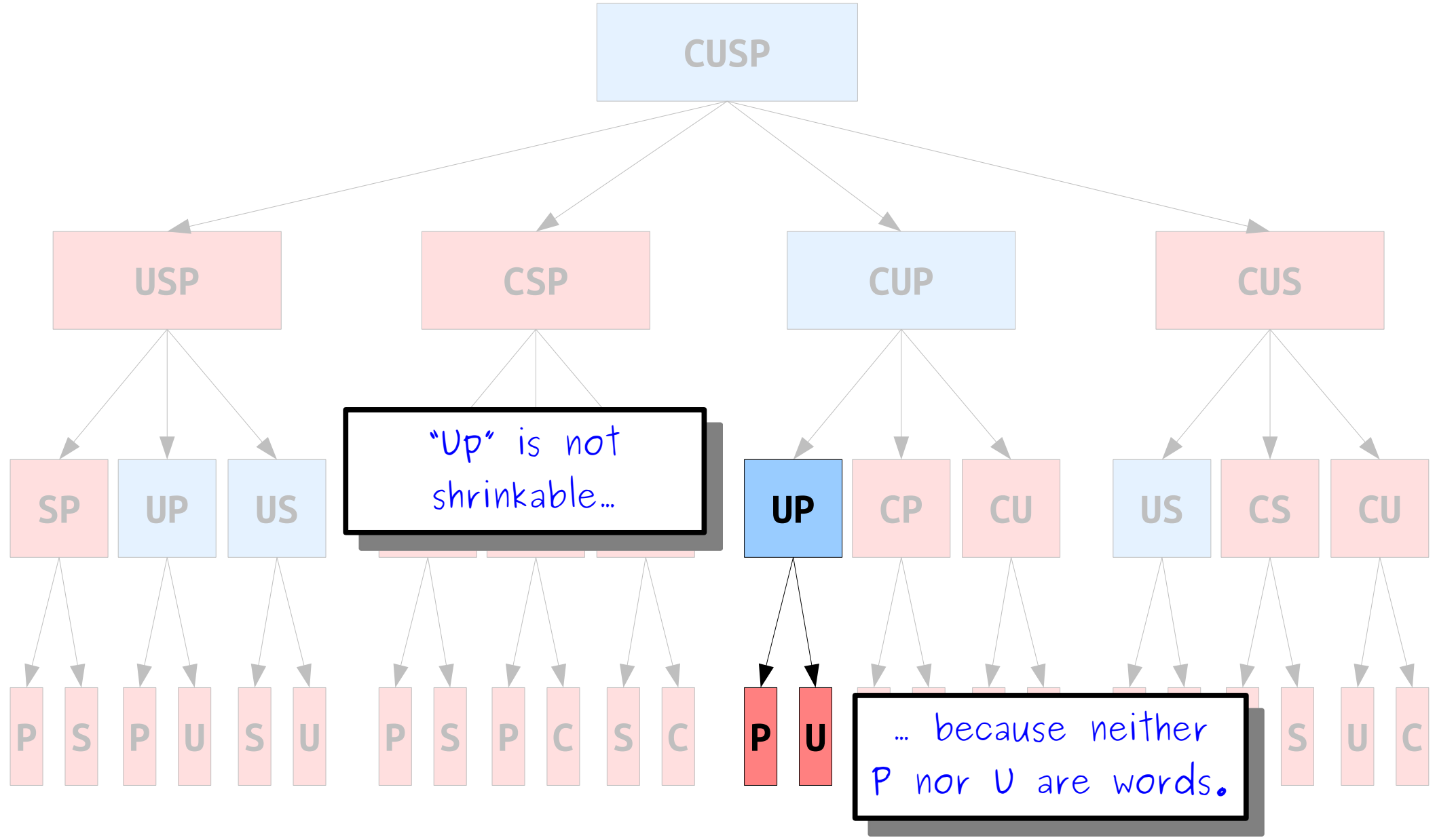
All Possible Paths



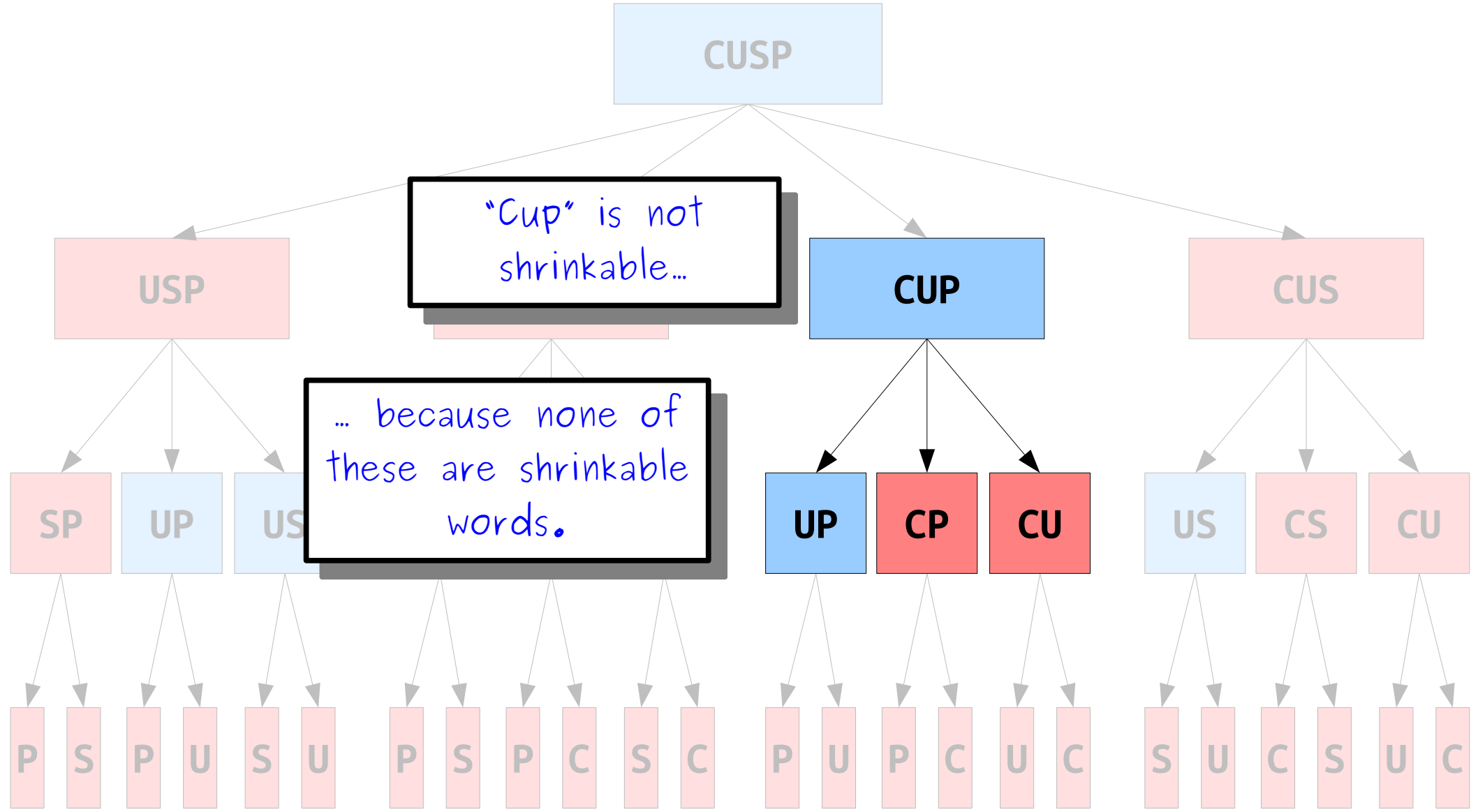
All Possible Paths



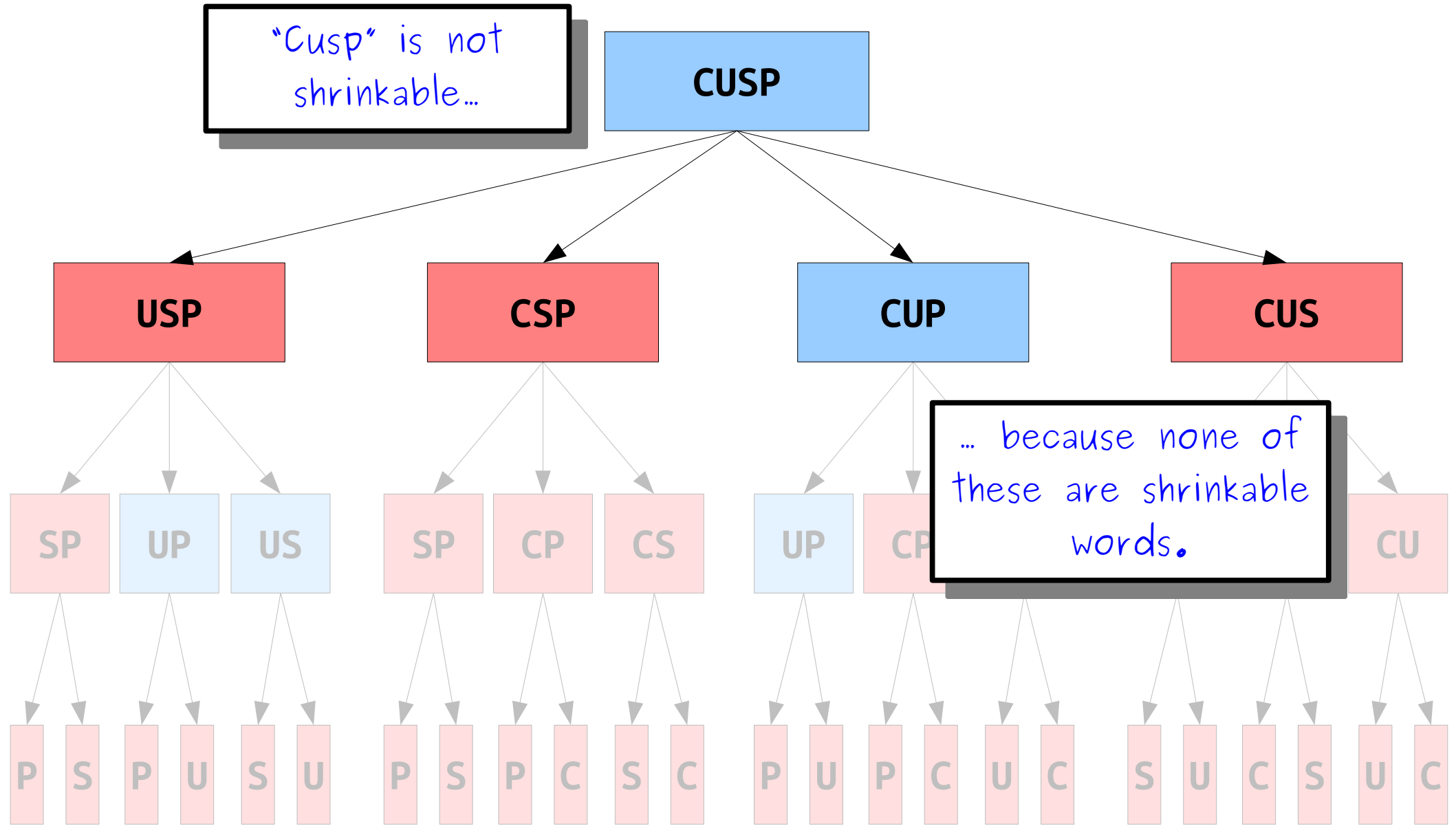
All Possible Paths



All Possible Paths



All Possible Paths



Shrinkable Words

- A ***shrinkable word*** is a word that can be reduced down to one letter by removing one character at a time, leaving a word at each step.
- ***Base Cases*:**
  - A string that is not a word is not a shrinkable word.
  - Any single-letter word is shrinkable (A, I, and O).
- ***Recursive Step*:**
  - A multi-letter word is shrinkable if you can remove a letter to form a shrinkable word.
  - A multi-letter word is not shrinkable if no matter what letter you remove, it’s not shrinkable.

Your Action Items

- ***Read Chapter 9 of the textbook.***
  - There's tons of cool backtracking examples there, and it will help you prep for Friday.
- ***Keep working on Assignment 3.***
  - Ask for help if you need it! That's what we're all here for.

Next Time

- ***Output Parameters***
  - Recovering the solution to a backtracking problem.
- ***More Backtracking***
  - Techniques in searching for feasibility.
- ***Closing Thoughts on Recursion***
  - It’ll come back, but we’re going to focus on other things for a while!