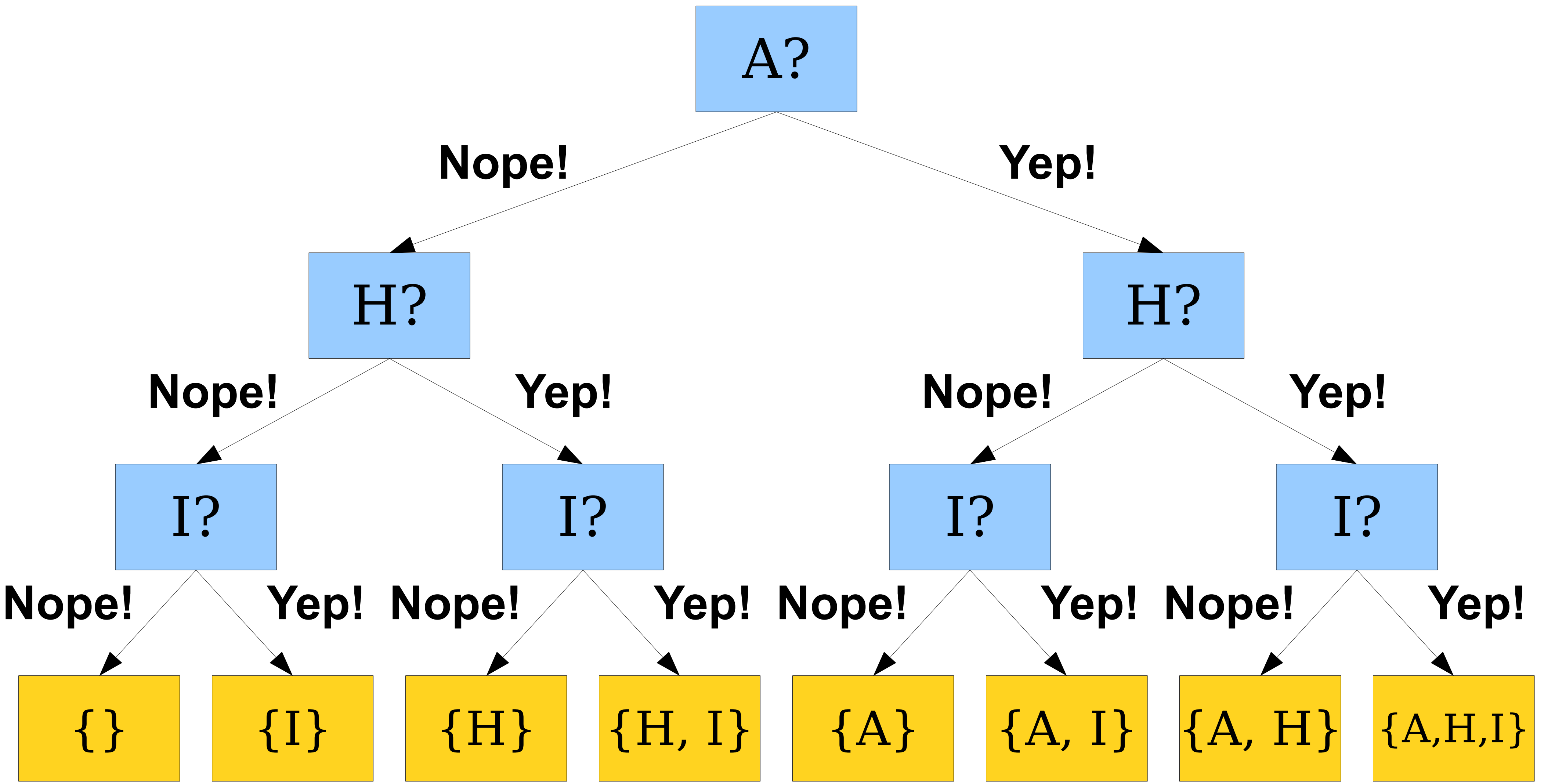
# Thinking Recursively

## Part IV

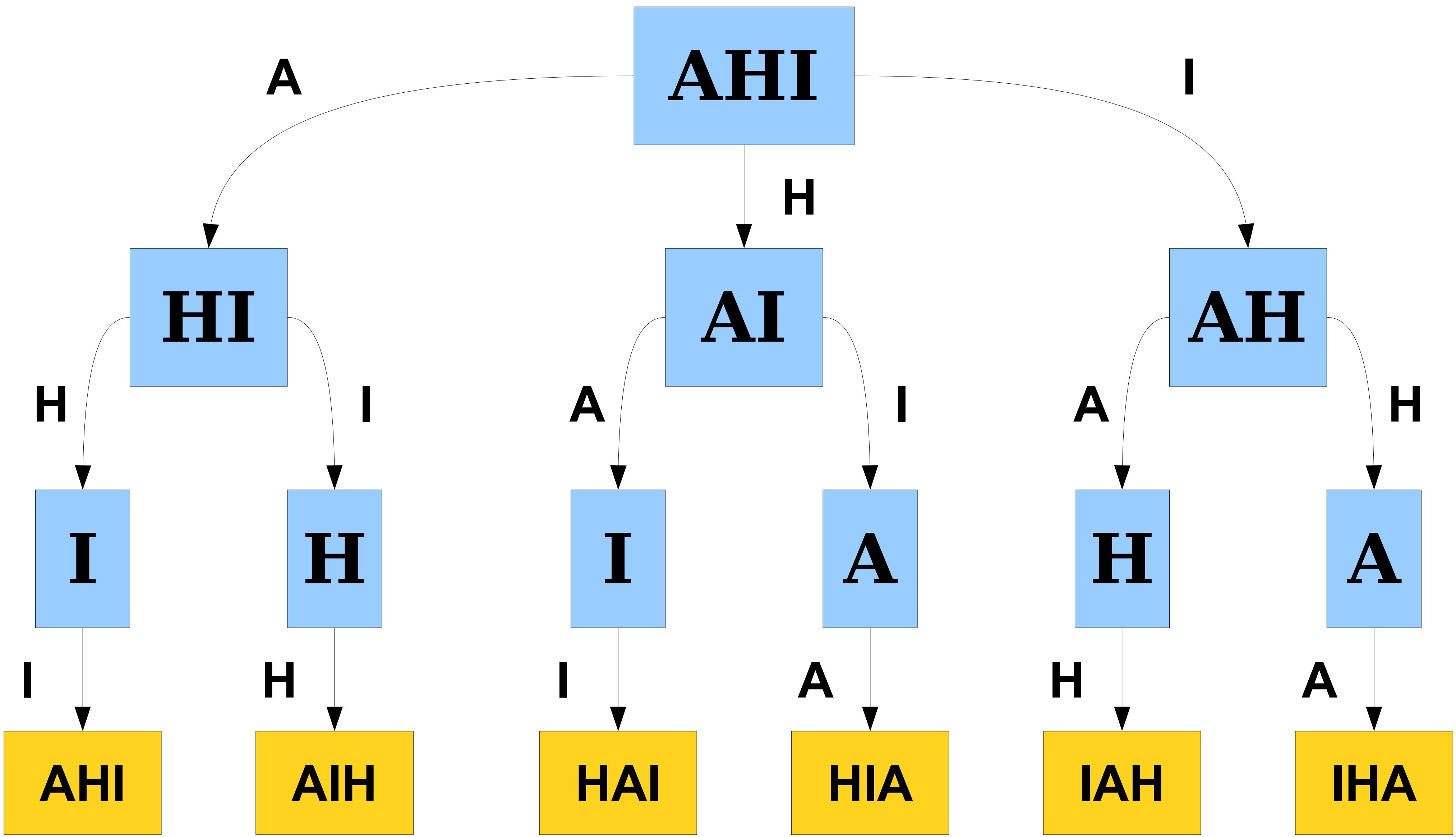
# From Last Time…

Most of the “interesting” exhaustive recursive programs can be reduced to either generating subsets or permutations

# Subset Decision Tree



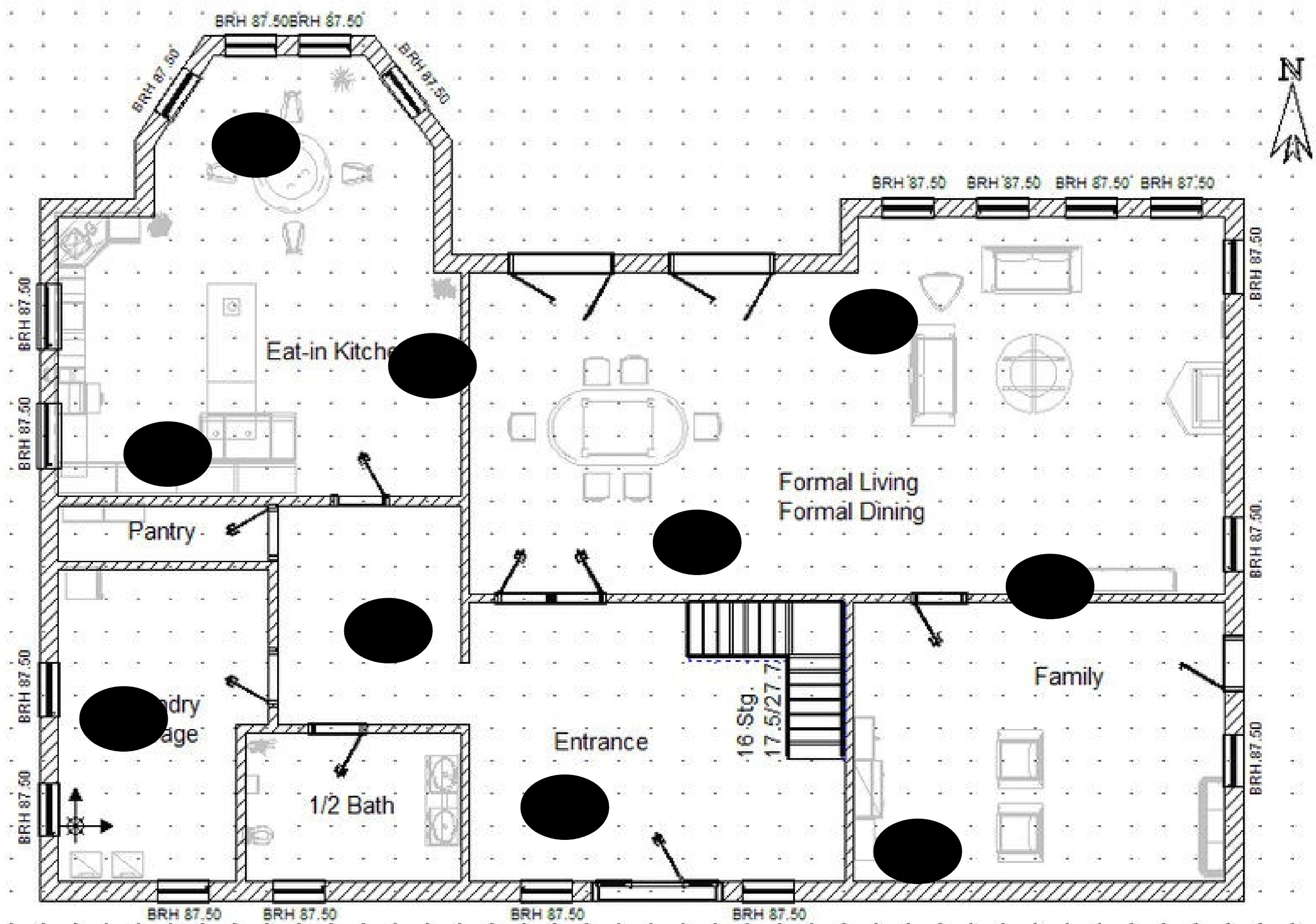
# Permutations Decision Tree



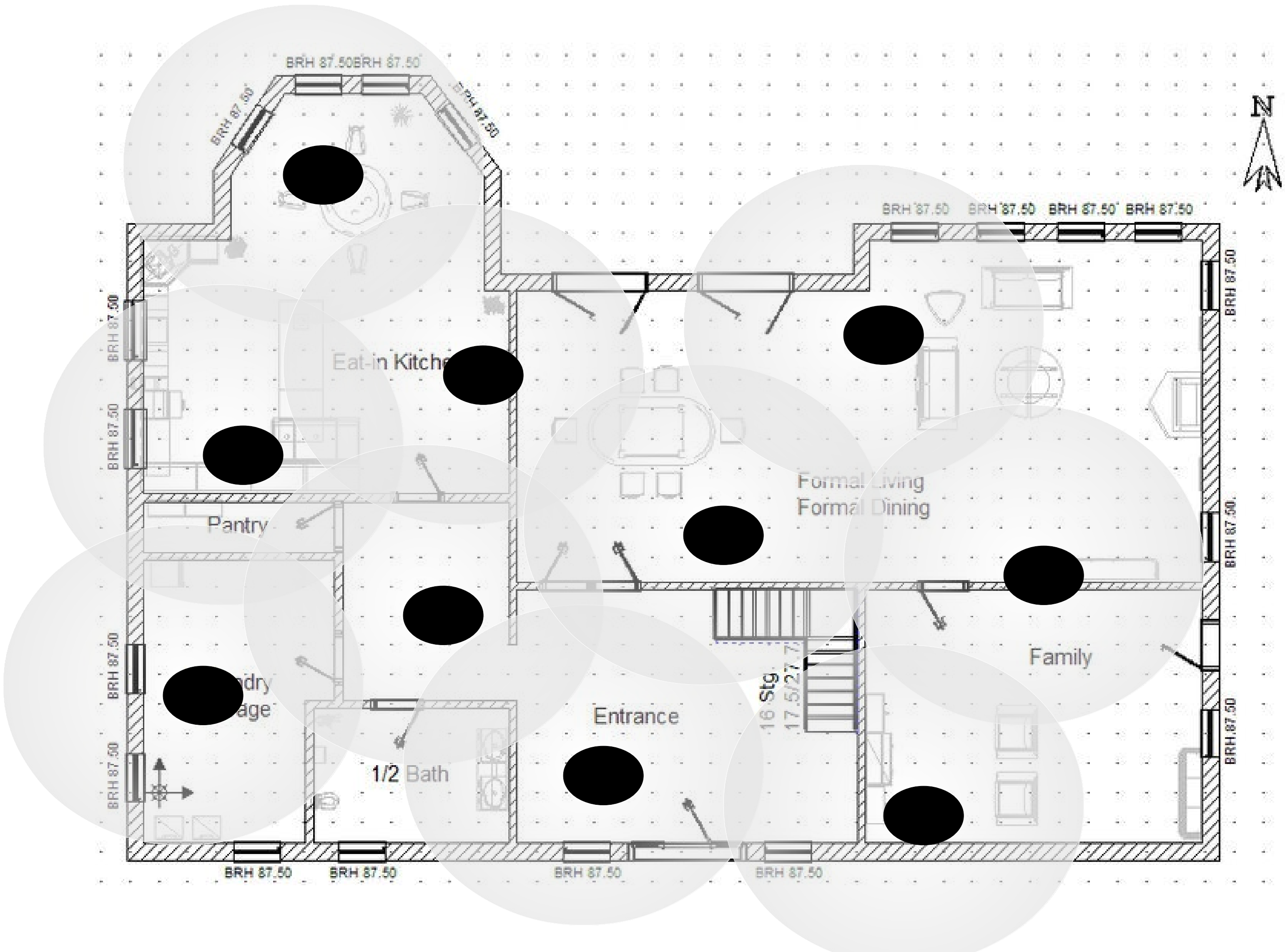
A good first step to solving an exhaustive recursive problem is first determine if it's related to generating subsets or permutations.

# New Stuff…

# Sensor Placement



# Sensor Placement



# Sensor Placement

- Goal is still to maximize covered area
  - New Constraint: Can only pick **k** sensors
- Similar to subset example, **no known** efficient algorithms for solving this problem *perfectly* for arbitrary **k**
- How can we generate all possible choices?

# Generating Combinations

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

```
foreach (int x in 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:

```
foreach (int x in mySet) {
  foreach (int y in 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:

```
foreach (int x in mySet) {
  foreach (int y in mySet) {
    foreach (int z in 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?

# Pascal's Triangle Revisited

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

# Pascal's Triangle Revisited

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | 1 | | | | | |
| | | | | 1 | | 1 | | | | |
| | | | 1 | | 2 | | 1 | | | |
| | | 1 | | 3 | | 3 | | 1 | | |
| | 1 | | 4 | | 6 | | 4 | | 1 | |
| 1 | | 5 | | 10 | | 10 | | 5 | | 1 |

# Pascal's Triangle Revisited

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

# Pascal's Triangle Revisited

(0, 0)

(0, 1) (1, 1)

(0, 2) (1, 2) (2, 2)

(0, 3) (1, 3) (2, 3) (3, 3)

(0, 4) (1, 4) (2, 4) (3, 4) (4, 4)

(0, 5) (1, 5) (2, 5) (3, 5) (4, 5) (5, 5)

# Pascal's Triangle Revisited

(0, 0)

(0, 1) (1, 1)

(0, 2) (1, 2) (2, 2)

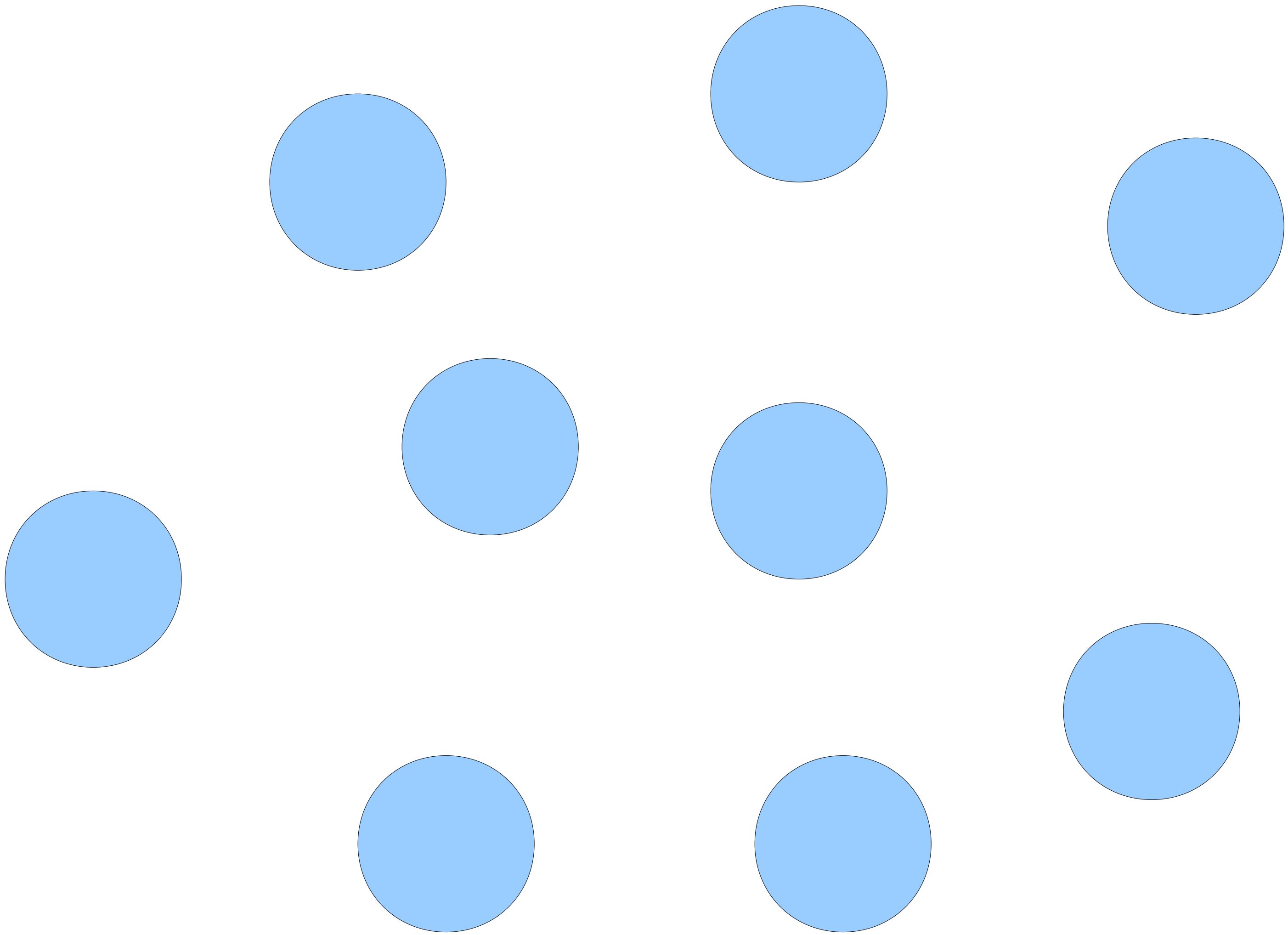
(0, 3) (1, 3) (2, 3) (3, 3)

(0, 4) (1, 4) (2, 4) (3, 4) (4, 4)

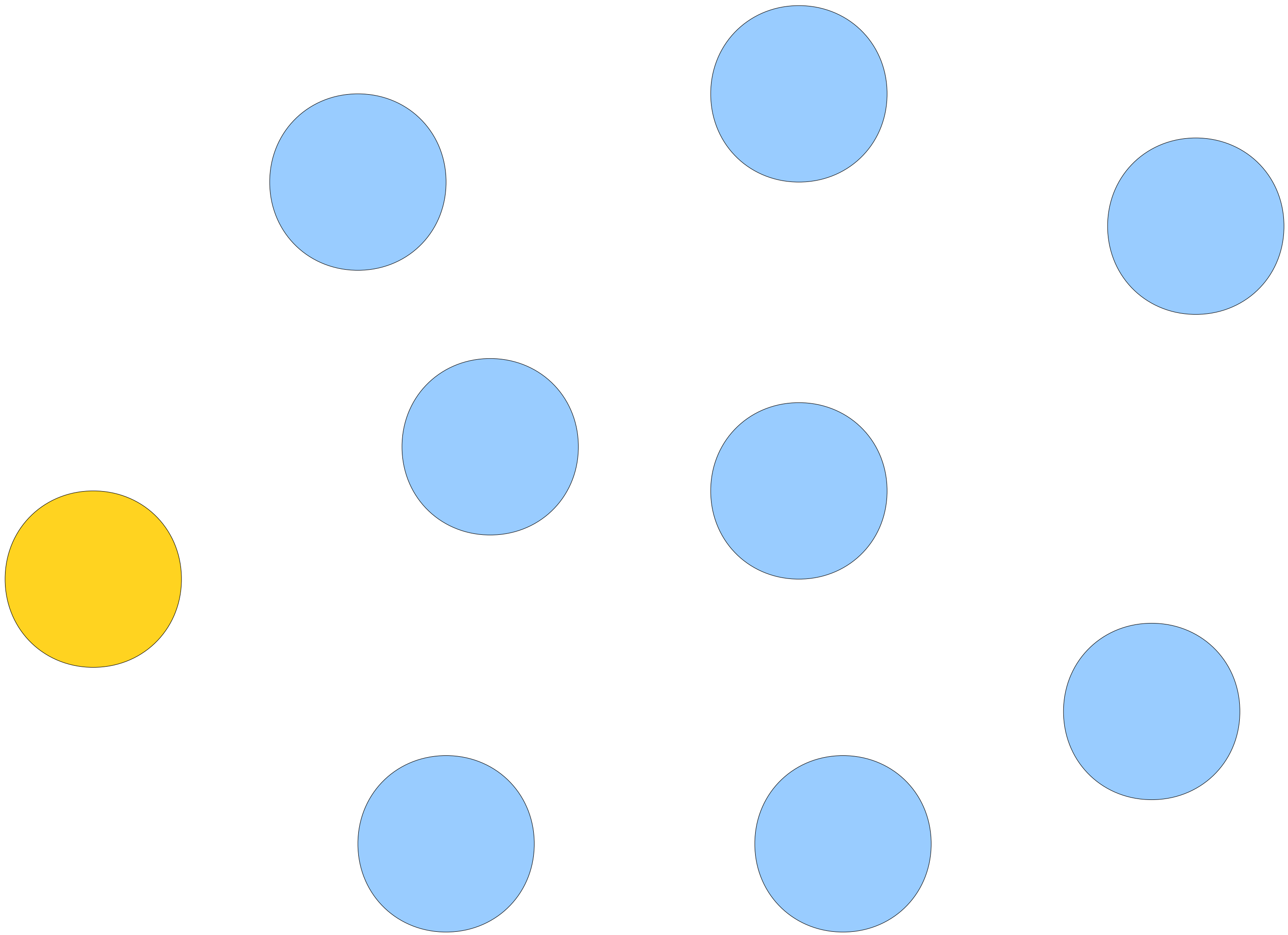
(0, 5) (1, 5) (2, 5) (3, 5) (4, 5) (5, 5)

What's up with that?

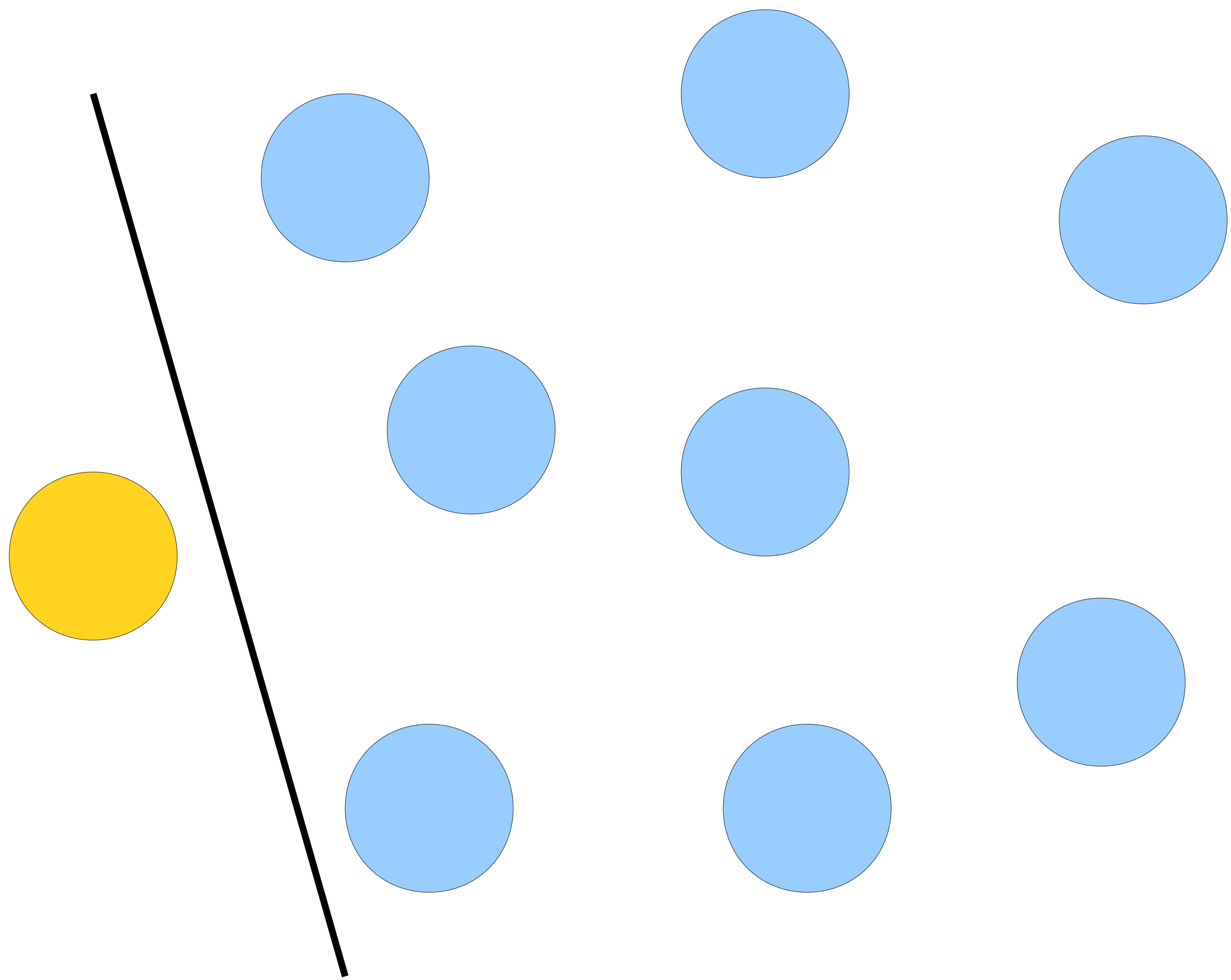
# Generating Combinations



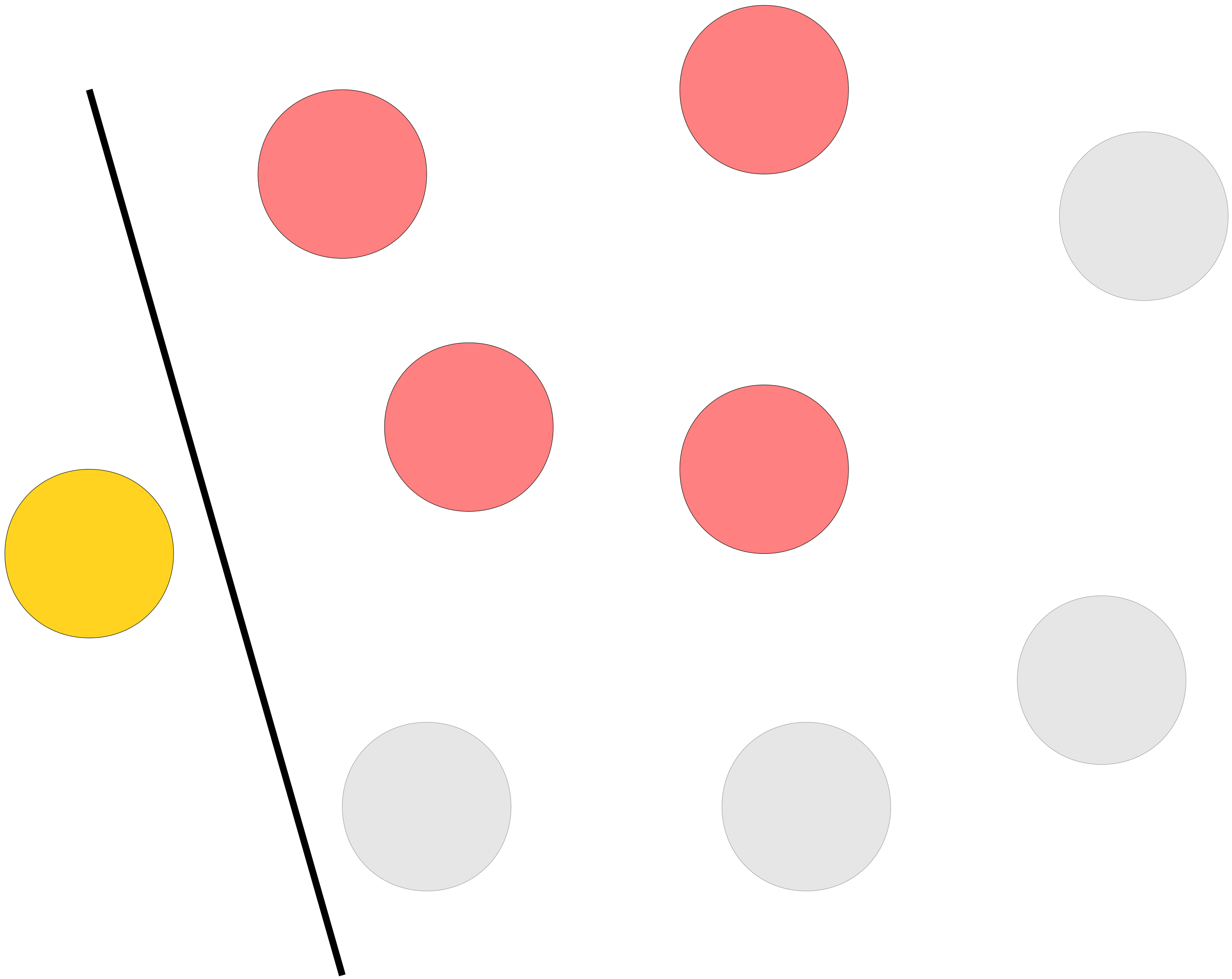
# Generating Combinations



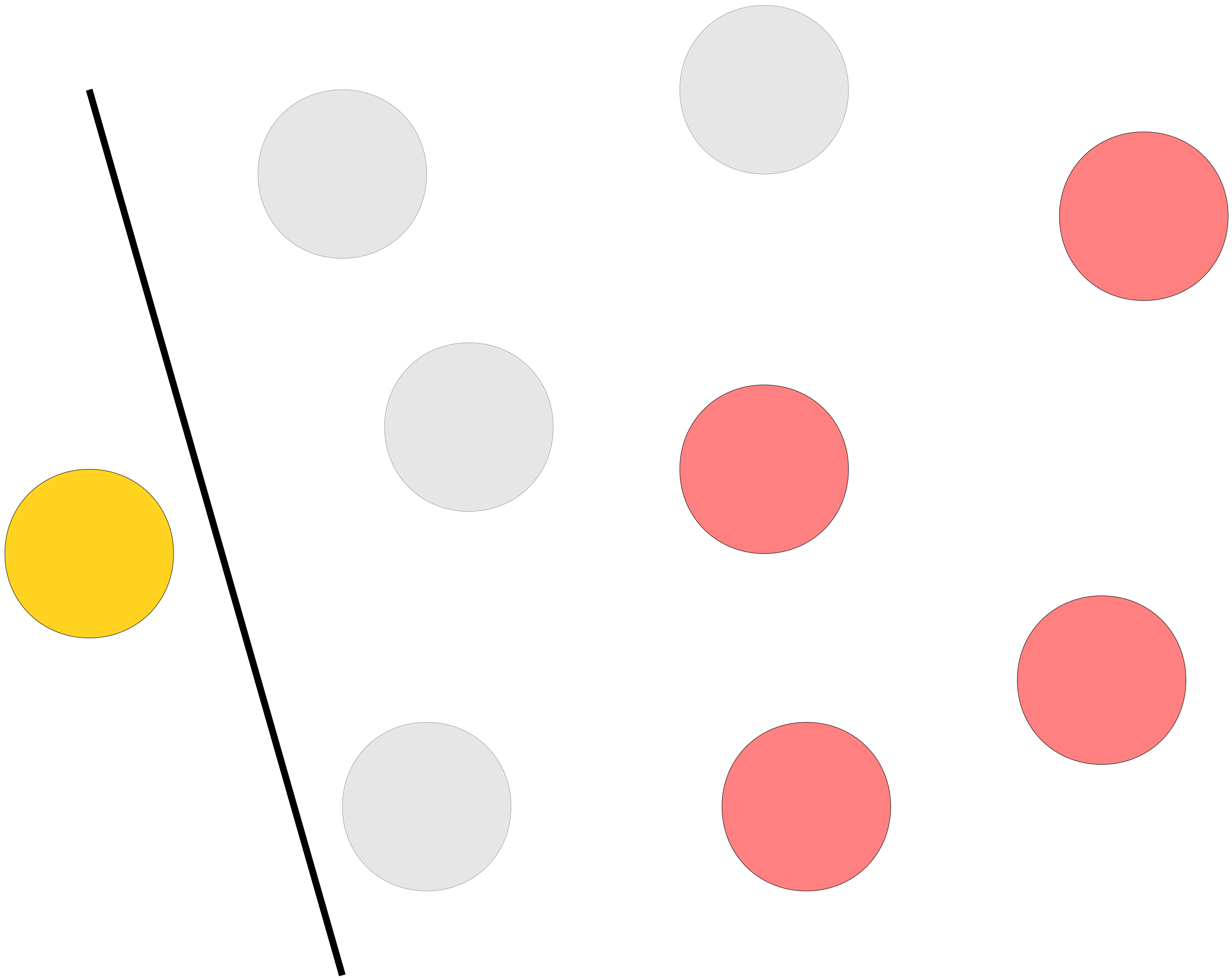
# Generating Combinations



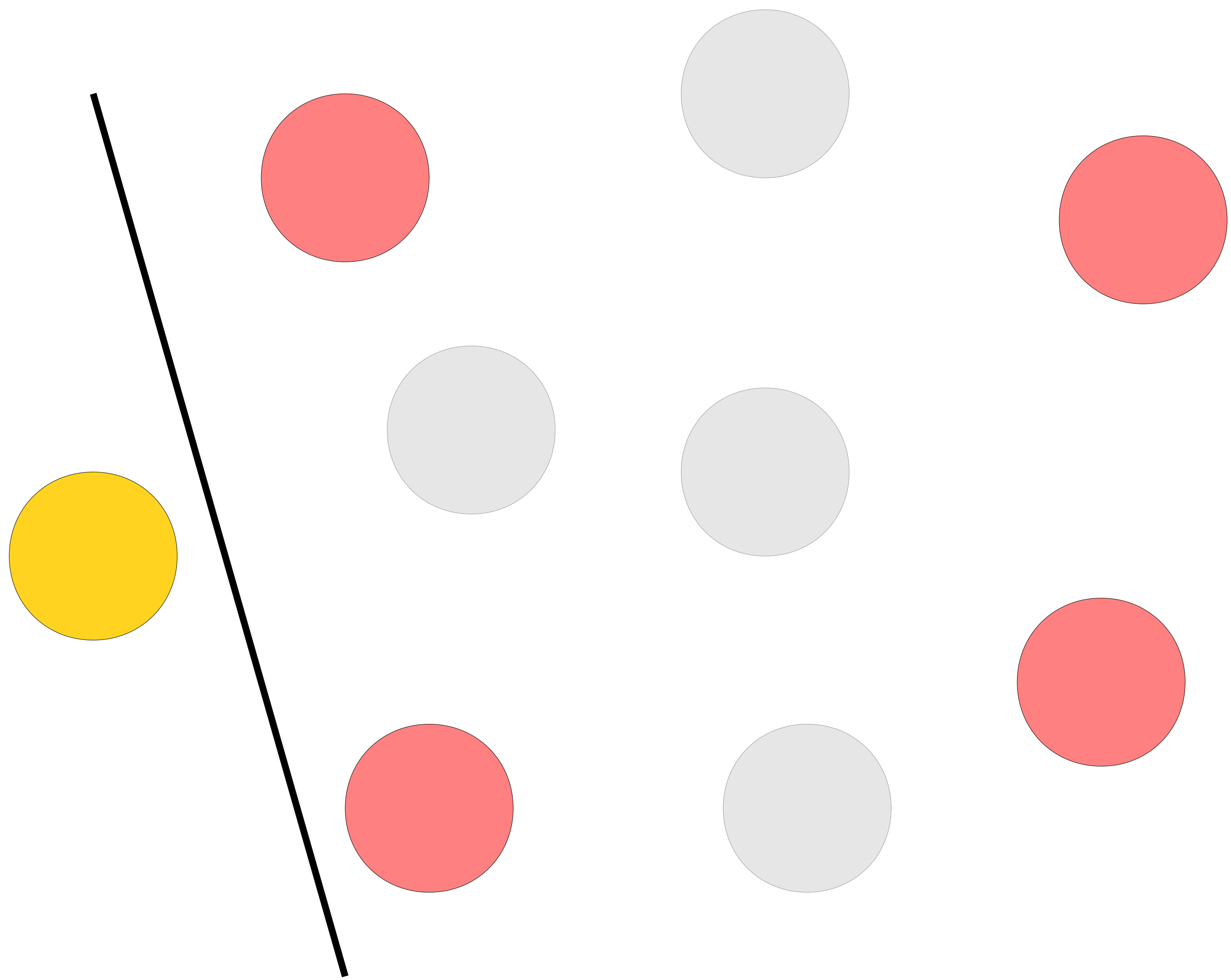
# Generating Combinations



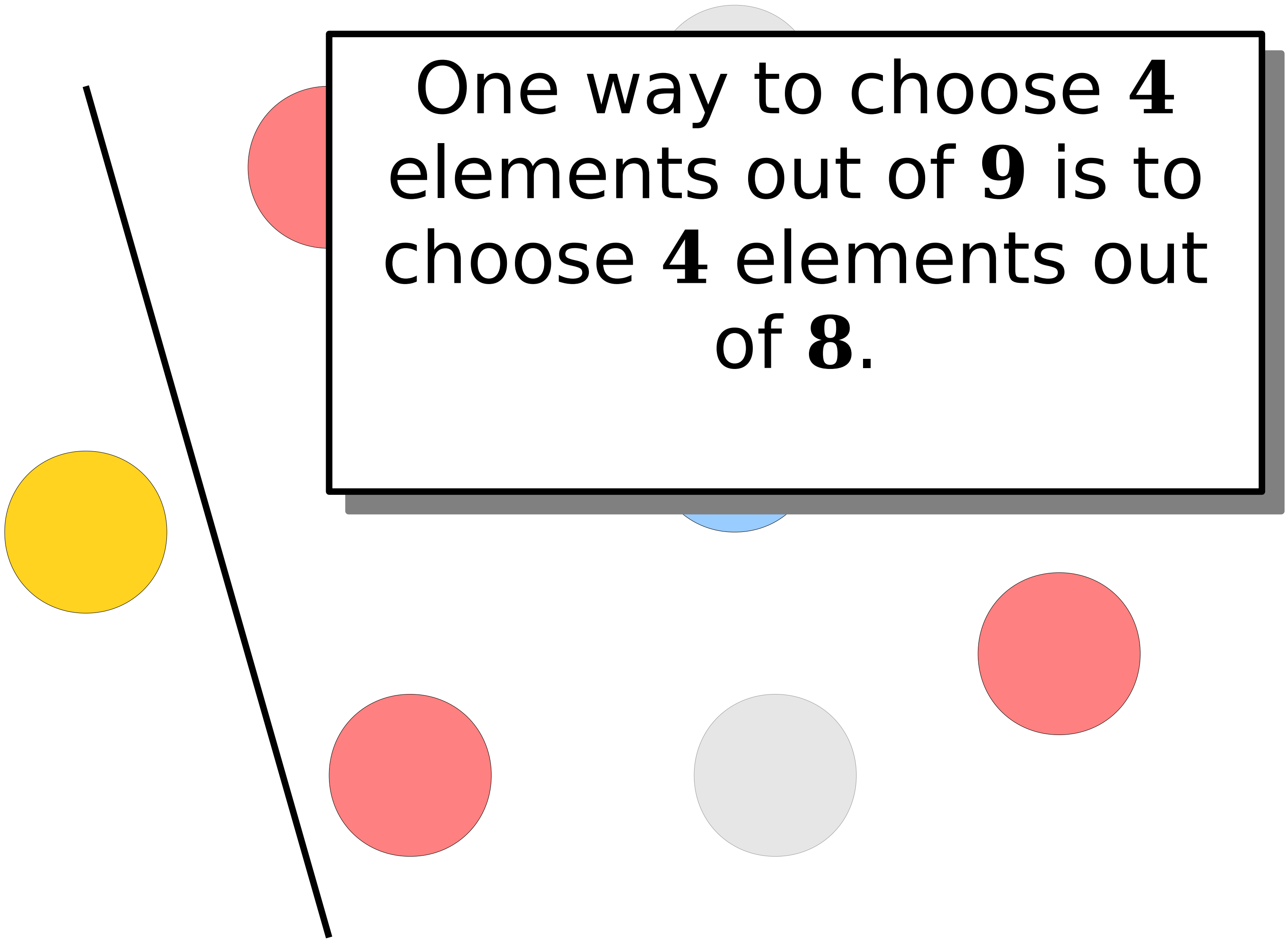
# Generating Combinations



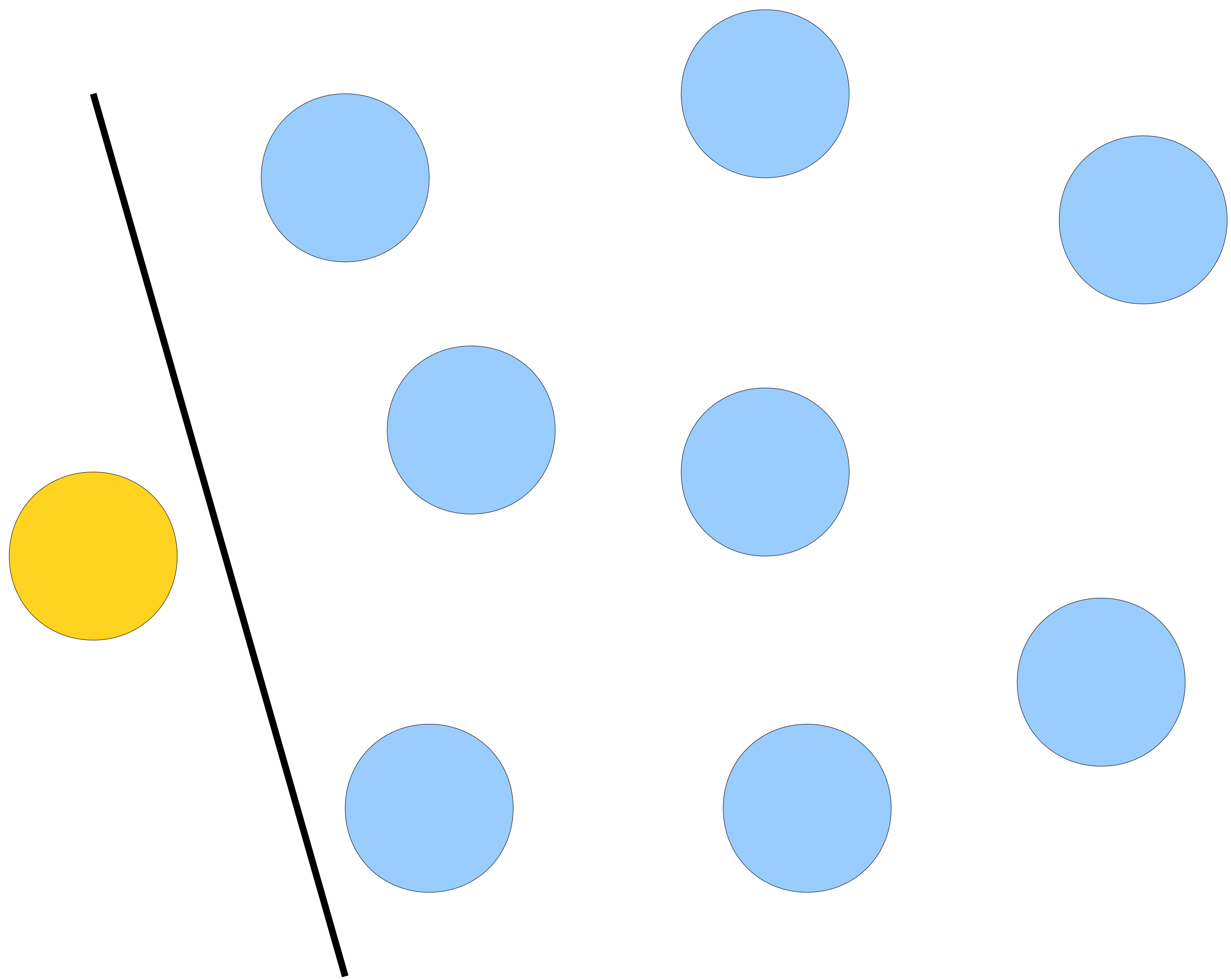
# Generating Combinations



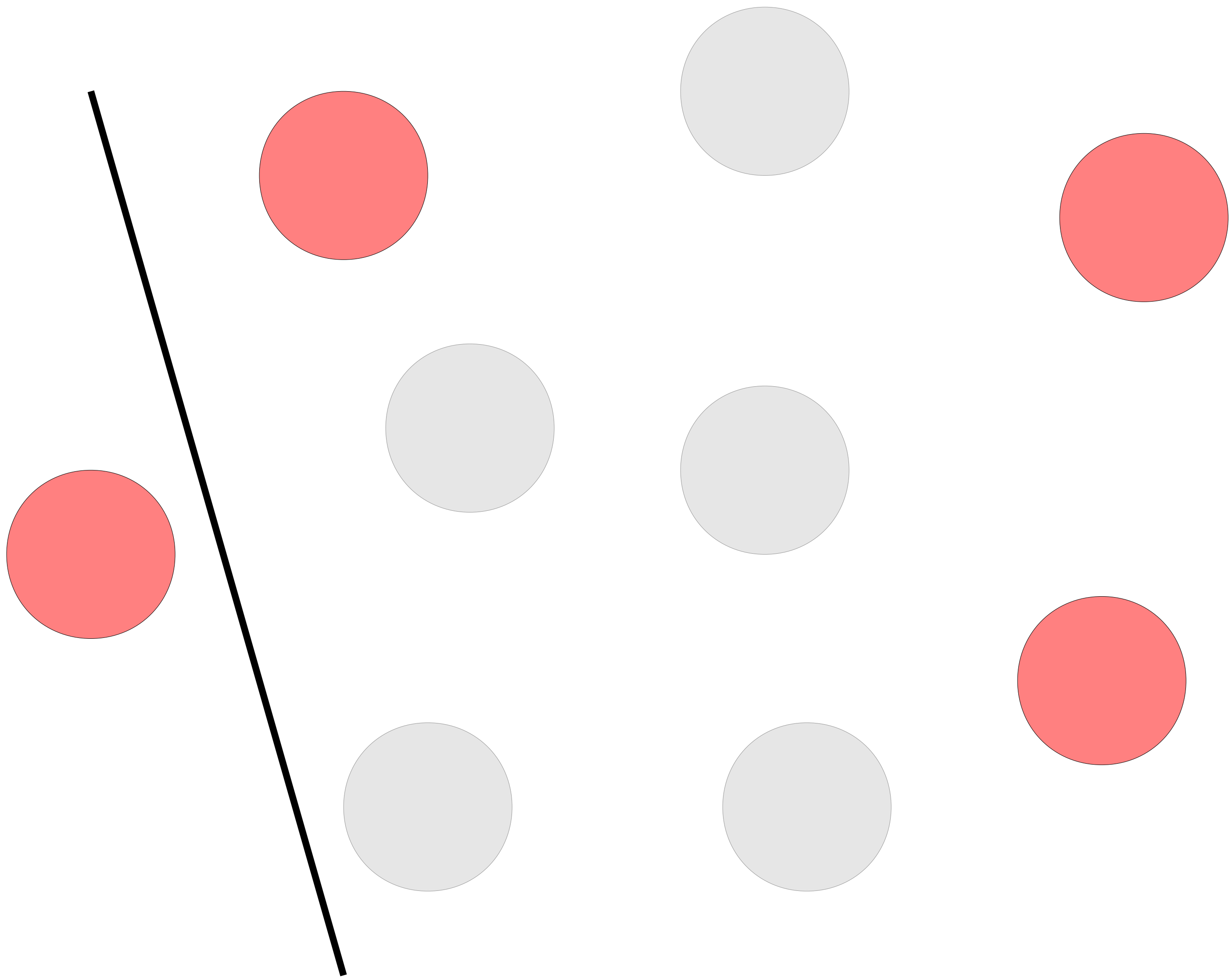
# Generating Combinations



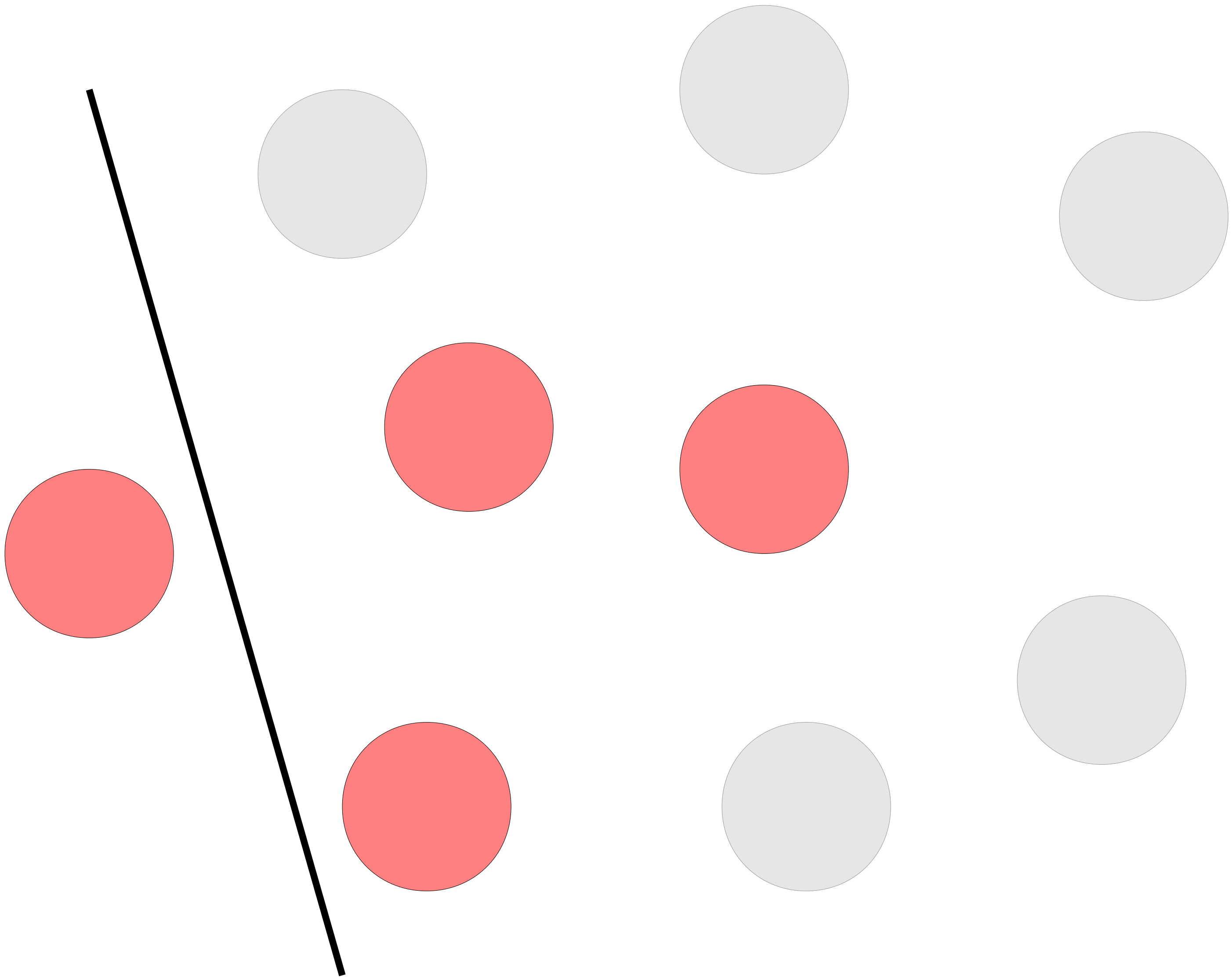
# Generating Combinations



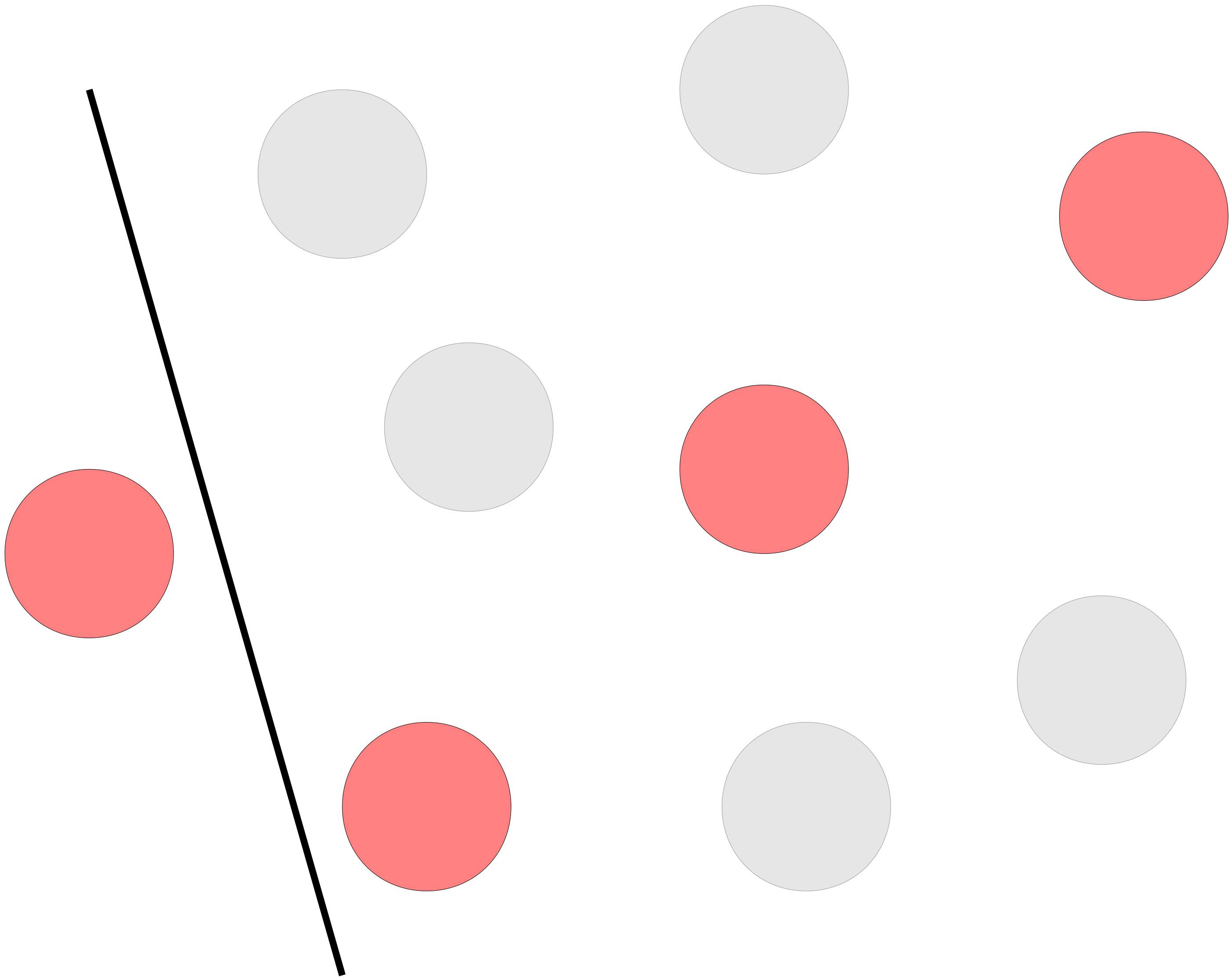
# Generating Combinations



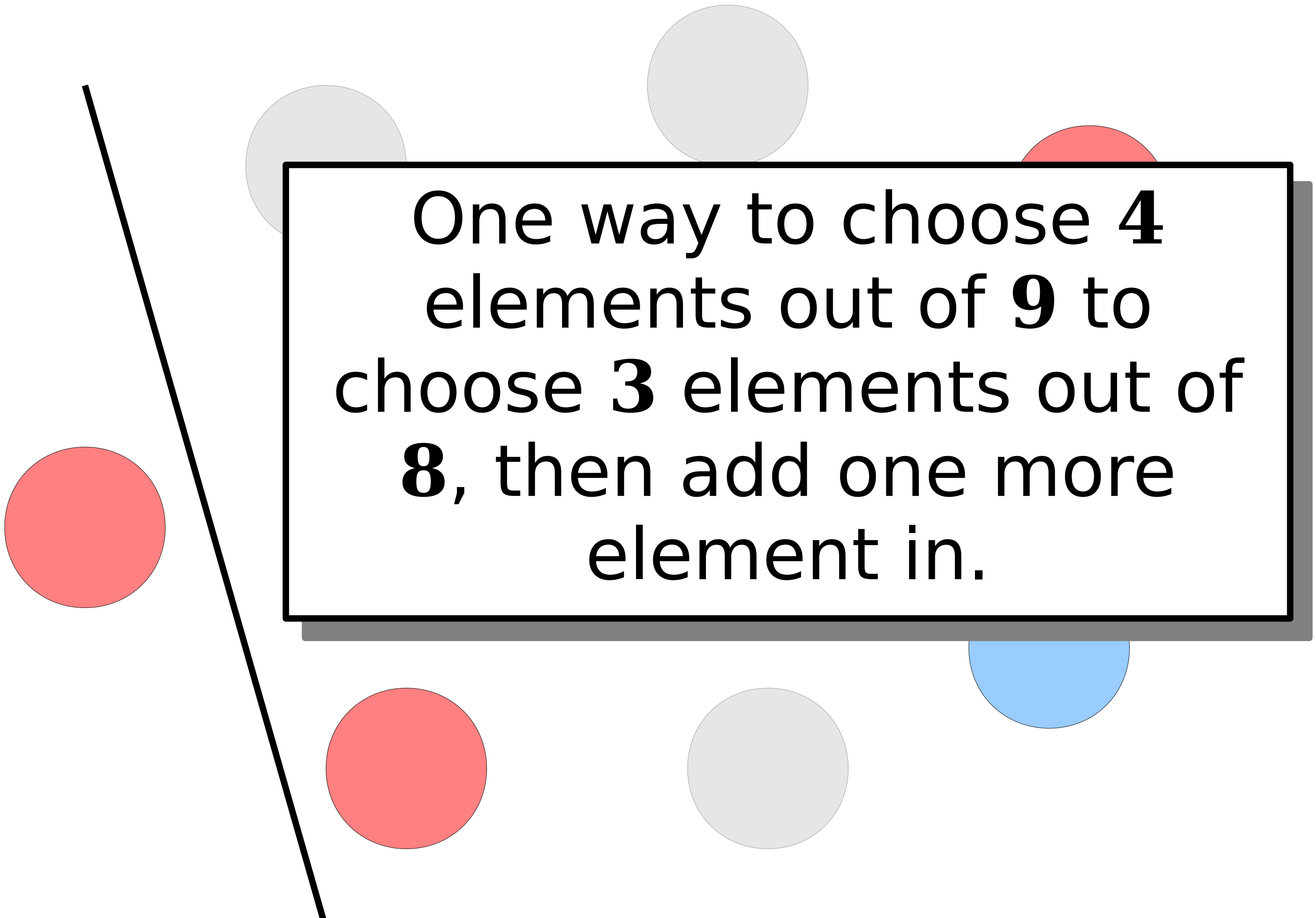
# Generating Combinations



# Generating Combinations



# Generating Combinations



One way to choose **4** elements out of **9** to choose **3** elements out of **8**, then add one more element in.

# Pascal's Triangle Revisited

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | 1 | | | | | |
| | | | | 1 | | 1 | | | | |
| | | | 1 | | 2 | | 1 | | | |
| | | 1 | | 3 | | 3 | | 1 | | |
| | 1 | | 4 | | 6 | | 4 | | 1 | |
| 1 | | 5 | | 10 | | 10 | | 5 | | 1 |

# Pascal's Triangle Revisited

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | 1 | | | | | |
| | | | | 1 | | 1 | | | | |
| | | | 1 | | 2 | | 1 | | | |
| | | 1 | | 3 | | 3 | | 1 | | |
| | 1 | | 4 | | 6 | | 4 | | 1 | |
| 1 | | 5 | | 10 | | 10 | | 5 | | 1 |

# Pascal's Triangle Revisited

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

# Pascal's Triangle Revisited

(0, 0)

(0, 1) (1, 1)

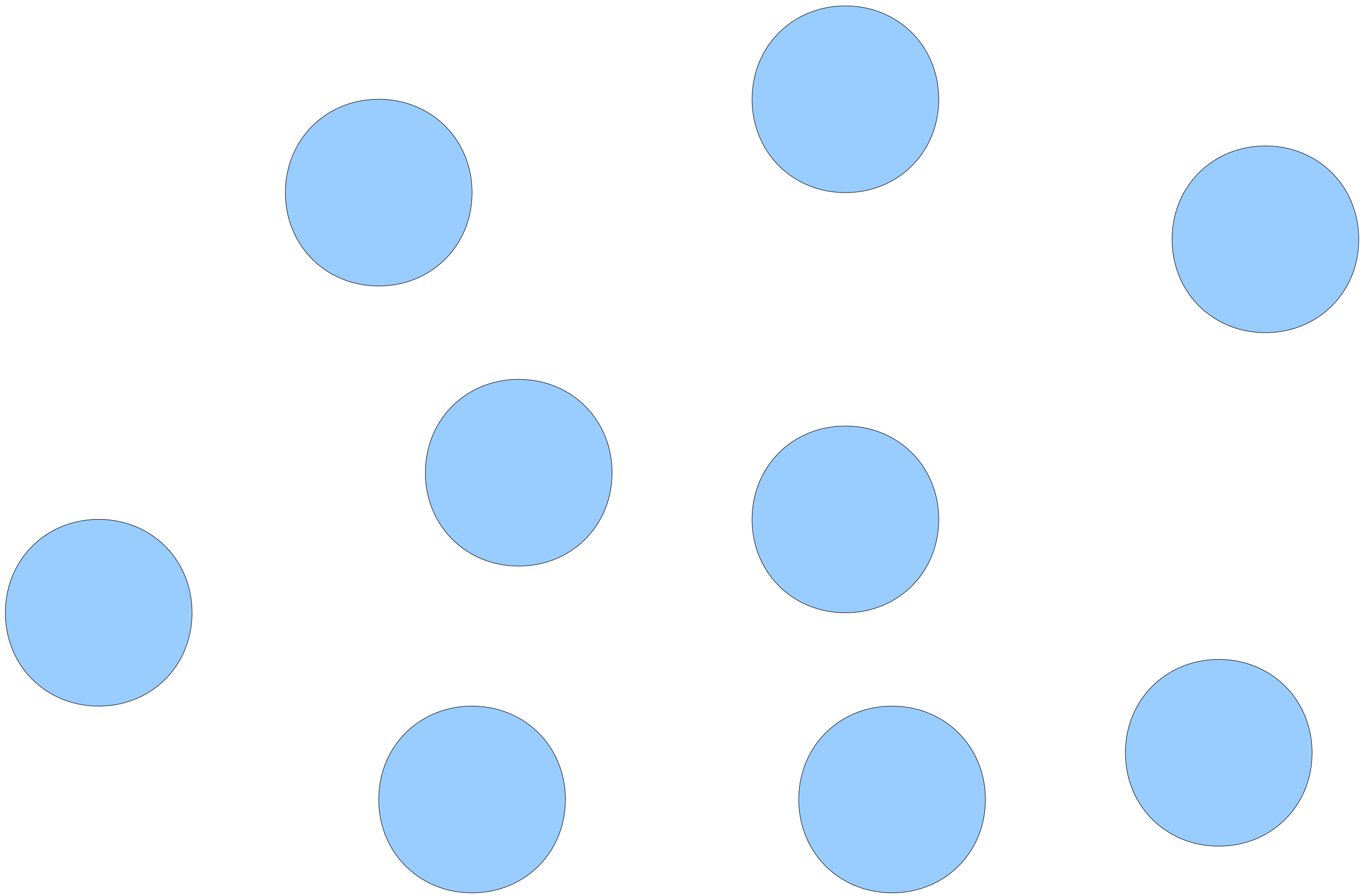
(0, 2) (1, 2) (2, 2)

(0, 3) (1, 3) (2, 3) (3, 3)

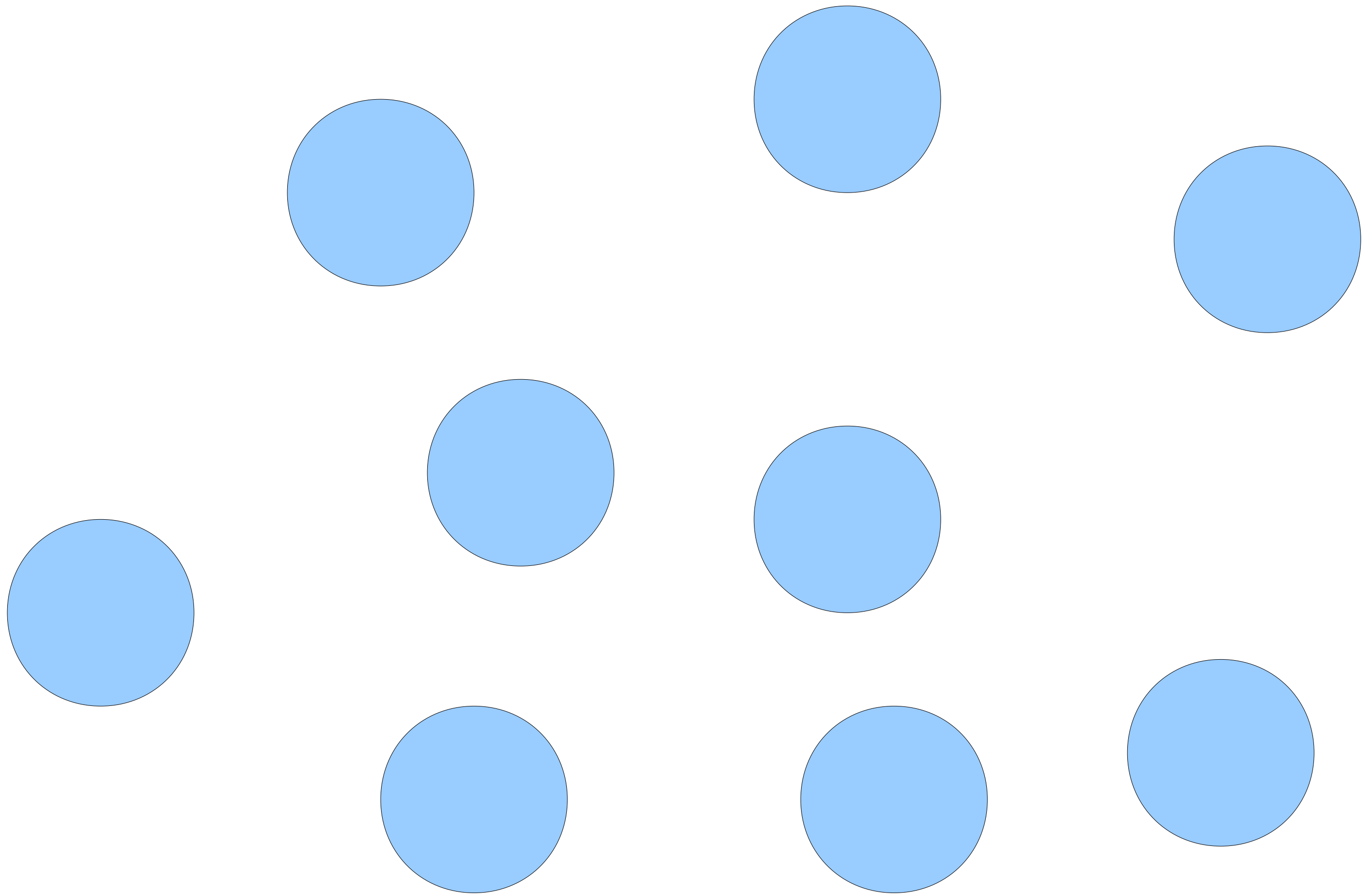
(0, 4) (1, 4) (2, 4) (3, 4) (4, 4)

(0, 5) (1, 5) (2, 5) (3, 5) (4, 5) (5, 5)

# Generating Combinations

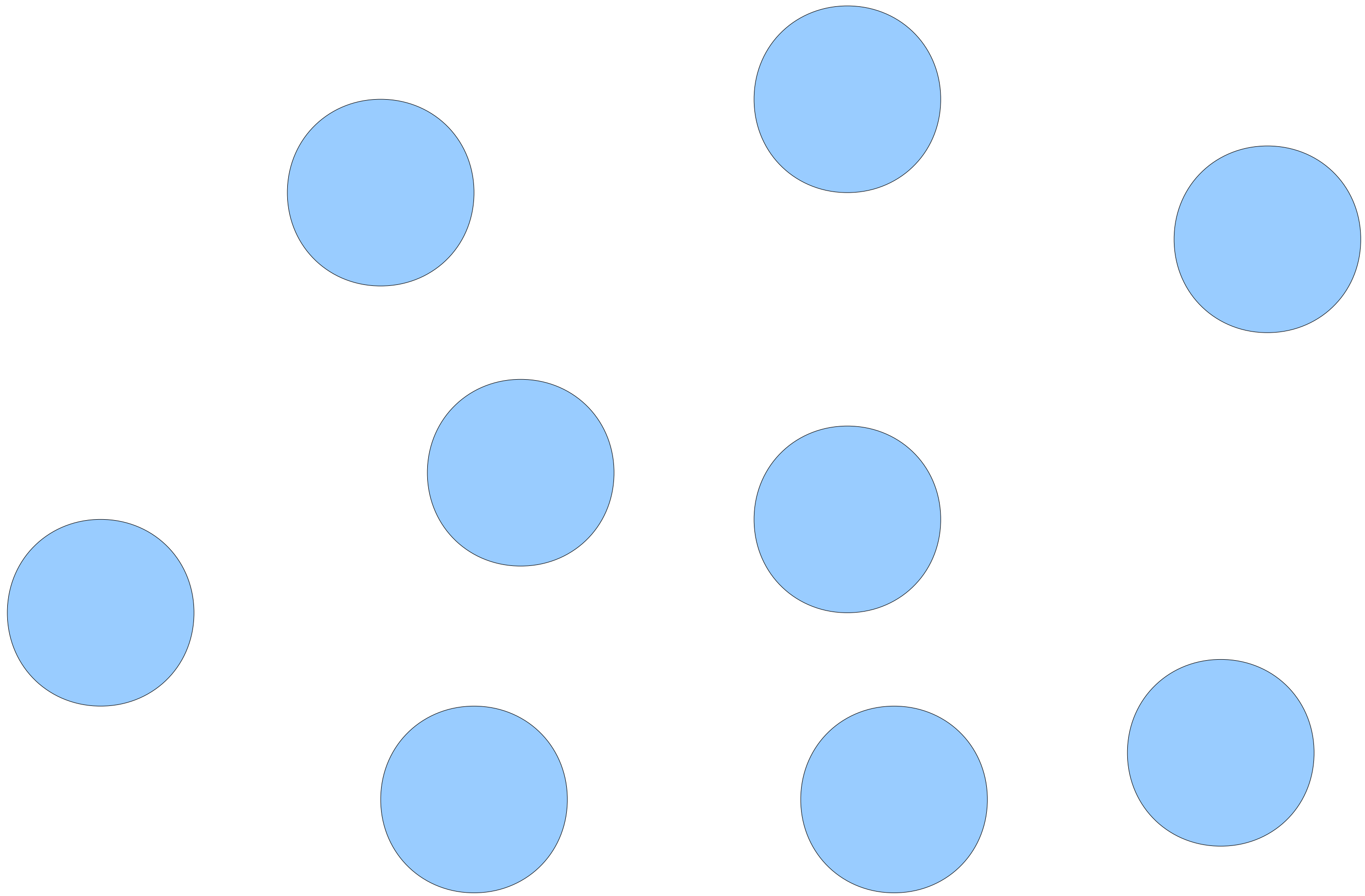


# Generating Combinations



How many ways are there to pick **0** things from this set?

# Generating Combinations



How many ways are there to pick **100** things from this set?

# `combinations` (Pseudocode)

# Combinations, Recursively

- How to pick $k$ elements from a set?
- **Base Cases:**
  - If $k$ is 0, the only option is to pick the empty set.
  - Otherwise, if $k$ is greater than the number of elements of the set, there are no options.
- **Recursive Step:**
  - Pick some element $x$ from the set.
  - Find all ways of picking $k$ elements of what remains.
  - Find all ways of picking $k$ – 1 elements of what remains, then add $x$ back in.

`combinations.cpp`

(Computer)

# Combinations

- Even though a combination is a different mathematical structure, generating combinations is nearly identical to generating subsets
  - All we needed to add was an extra parameter and an extra base case.

# 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

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| S | T | A | R | T | L | I | N | G |

# The Startling Truth

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| S | T | A | R | T | I | N | G |

# The Startling Truth

| | | | | | | |
|---|---|---|---|---|---|---|
| S | T | A | R | I | N | G |

# The Startling Truth

| S | T | R | I | N | G |
|---|---|---|---|---|---|

# The Startling Truth

| | | | | |
|---|---|---|---|---|
| S | T | I | N | G |

# The Startling Truth

| | | | |
|---|---|---|---|
| S | I | N | G |

# The Startling Truth

| | | |
|---|---|---|
| S | I | N |

# The Startling Truth

| | |
|---|---|
| I | N |

# The Startling Truth

I

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

# Shrinkable Words

- Let's define a **shrinkable word** as 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:**
  - Any string that is not a word cannot be a shrinkable word.
  - Any single-letter word is shrinkable.
    - A, I, O
- **Recursive Step:**
  - Any multi-letter word is shrinkable if you can remove a letter to form a shrinkable word.

# `shrinkable-words.cpp`
# (Pseudocode)

# `shrinkable-words.cpp`
# (Computer)

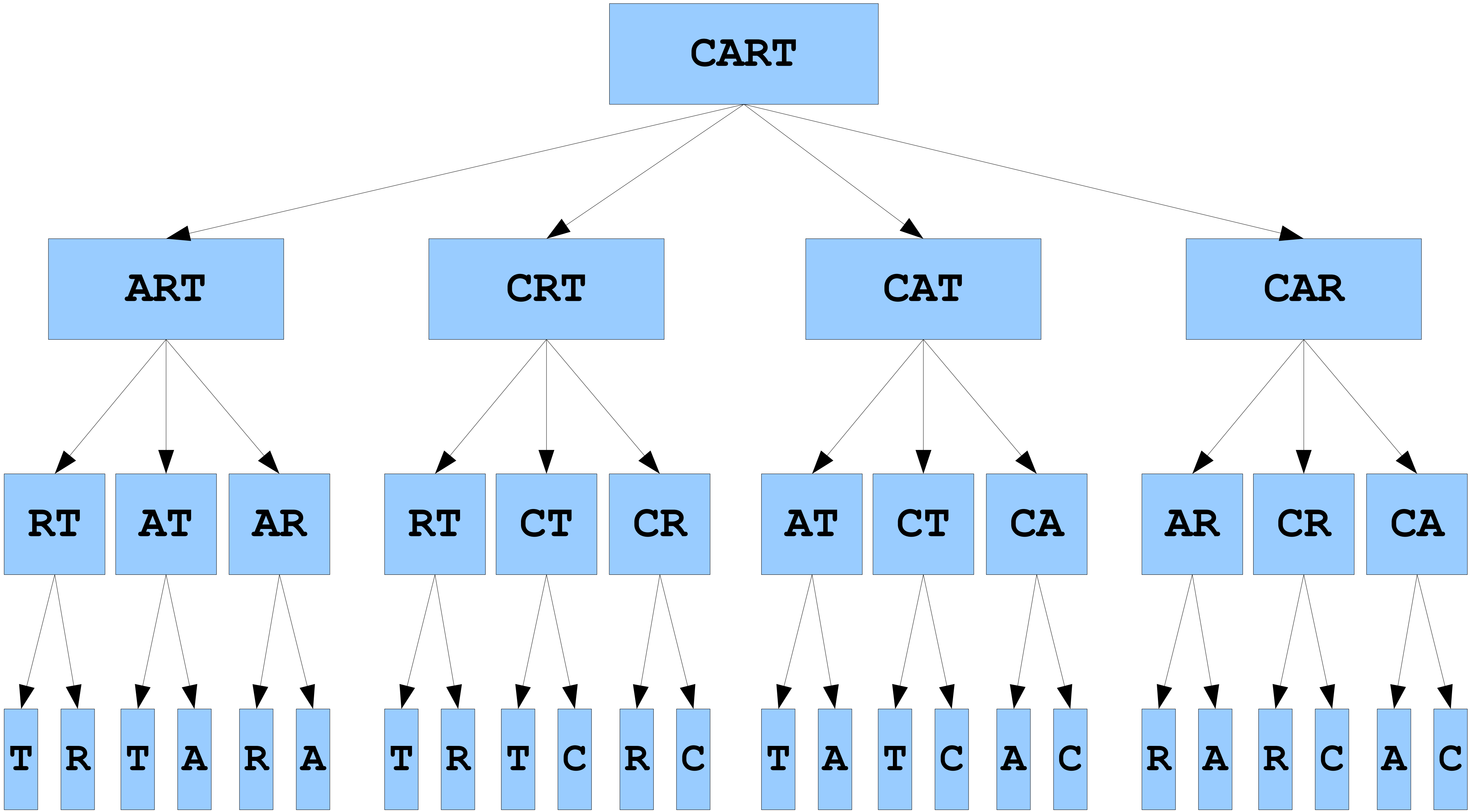
# Recursive Backtracking

- The function we have just written is an example of **recursive backtracking**.
- At each step, we try one of many possible options.
- If *any* option succeeds, that's great! We're done.
- If *none* of the options succeed, then this particular problem can't be solved.
- In recursive backtracking we care about finding “one thing” instead of “generating all things”

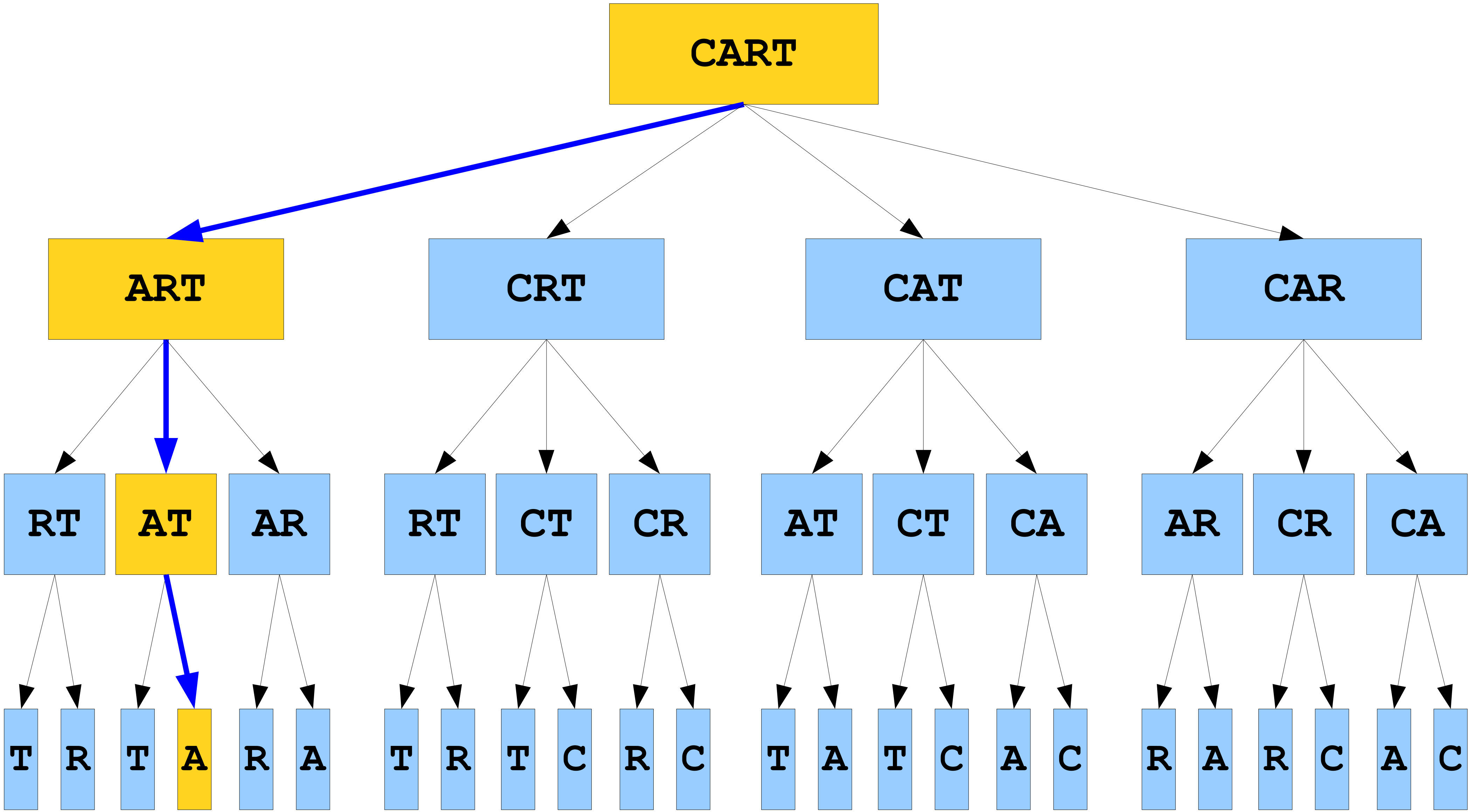
# Recursive Backtracking

- I claimed that most exhaustive recursive problems can be reduced to generating permutations or subsets.
- Is shrinkable words a subsets or permutations problem?
  - Like permutations, we are computing an ordering: the order in which we remove characters.
  - Instead of *adding* characters to a string we are *removing* characters from a string.

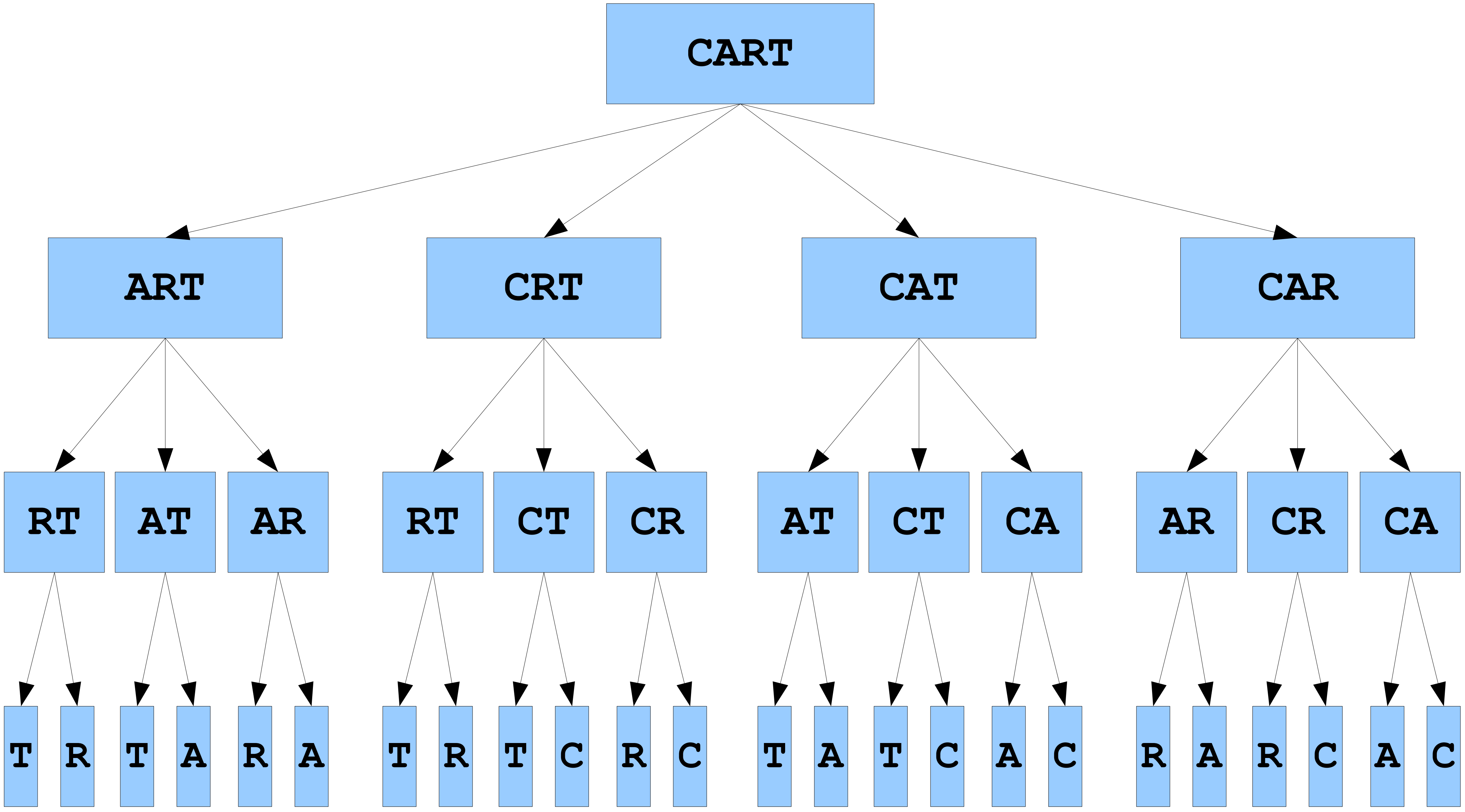
# Decision Tree



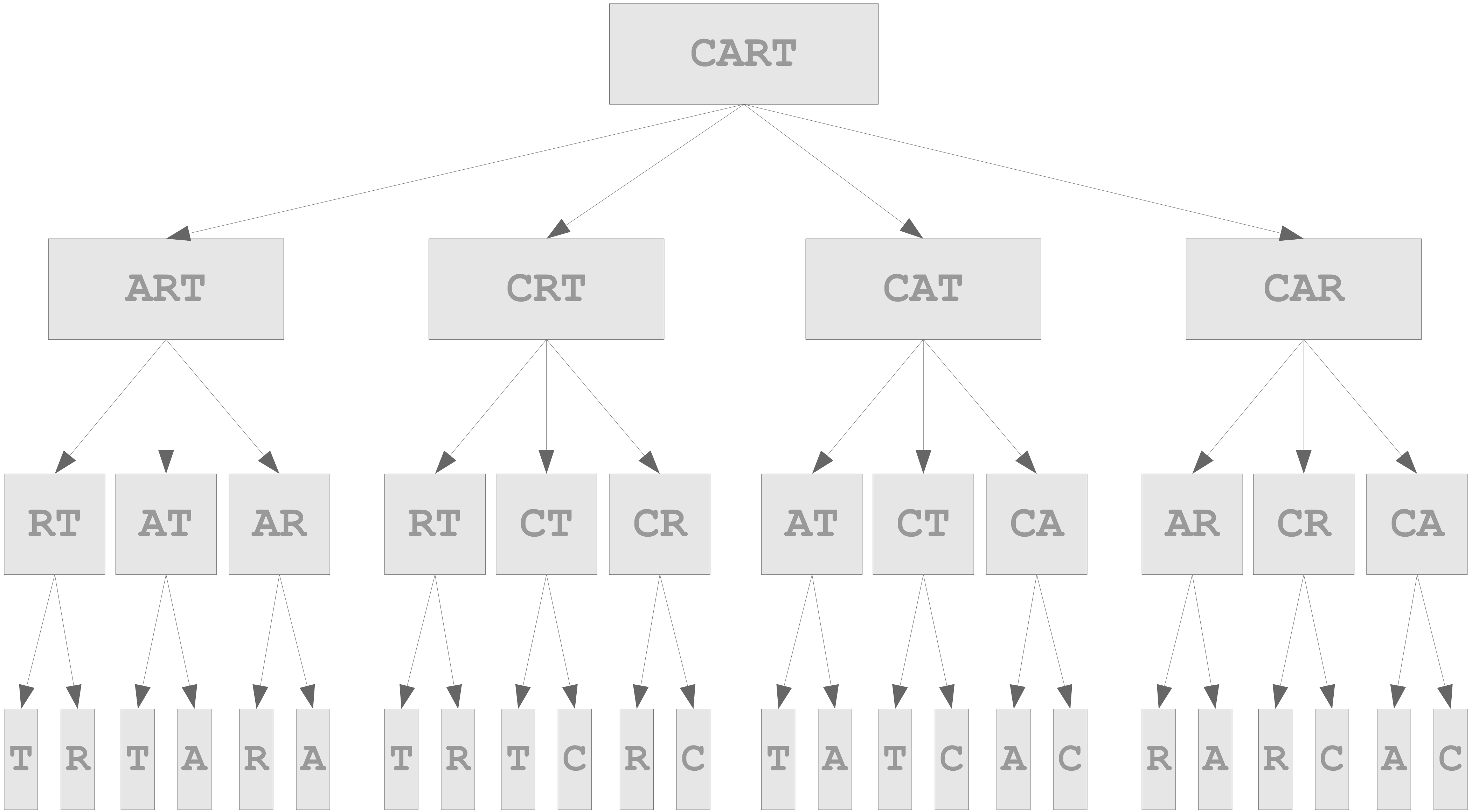
# Decision Tree



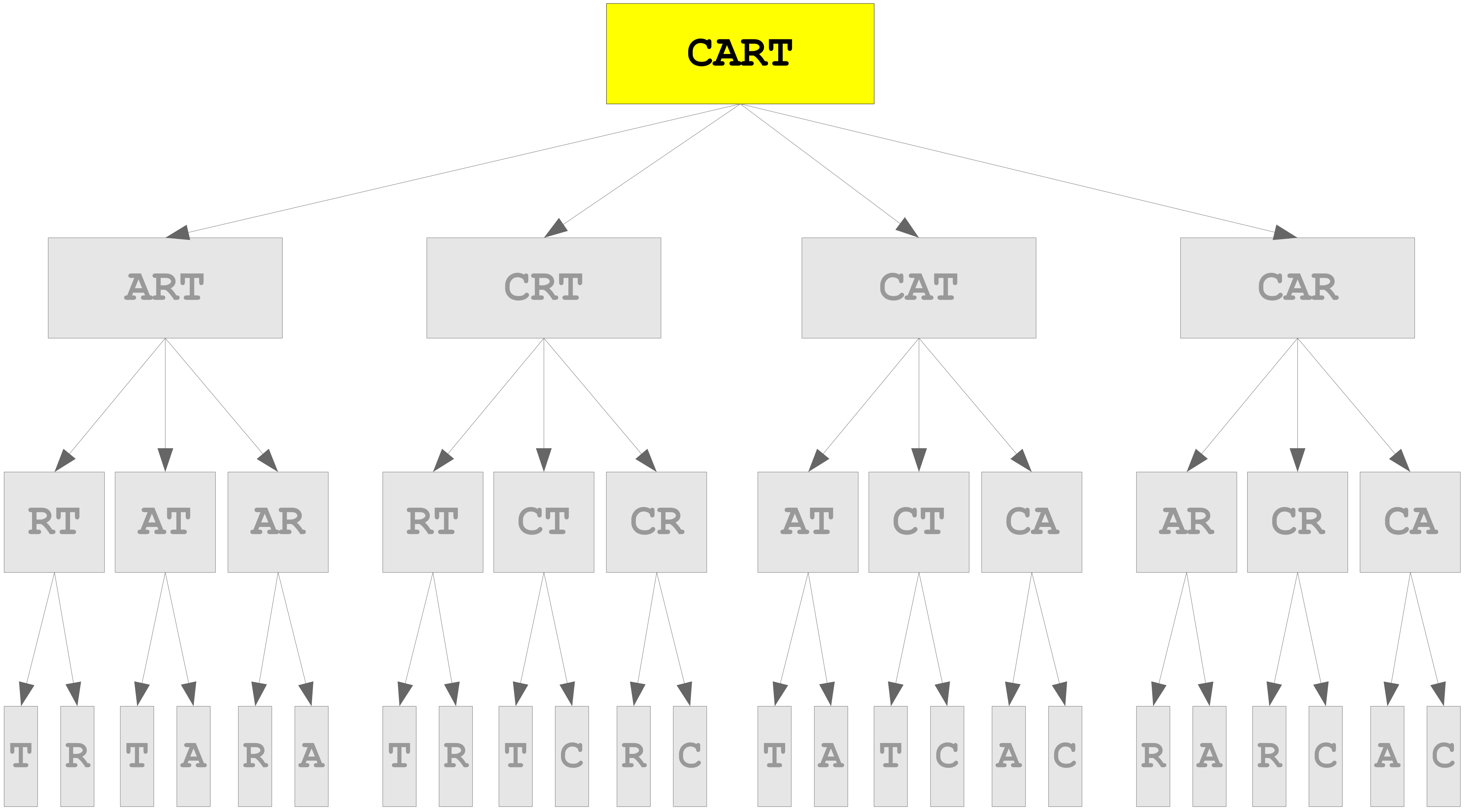
# Recursive Backtracking



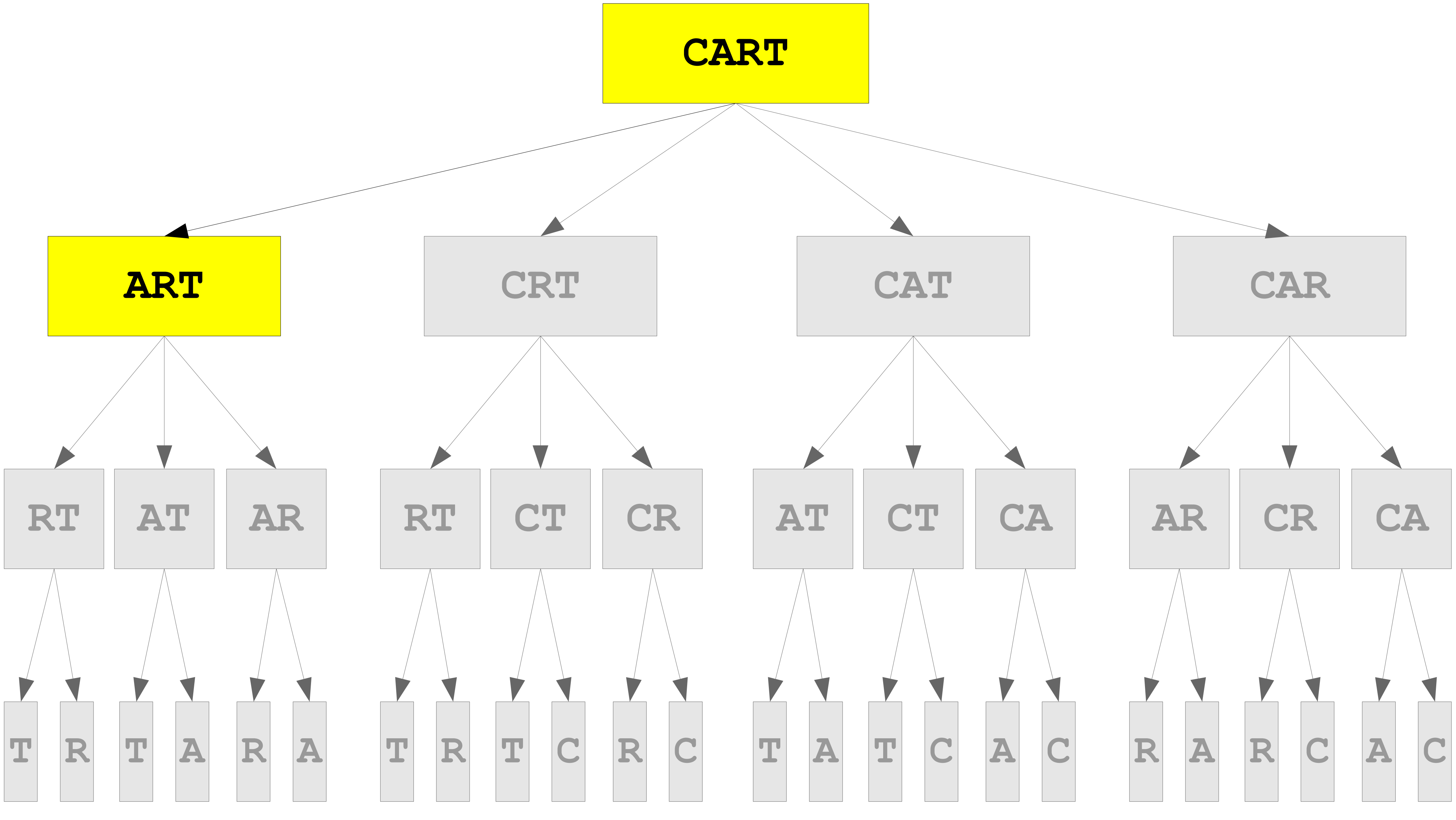
# Recursive Backtracking



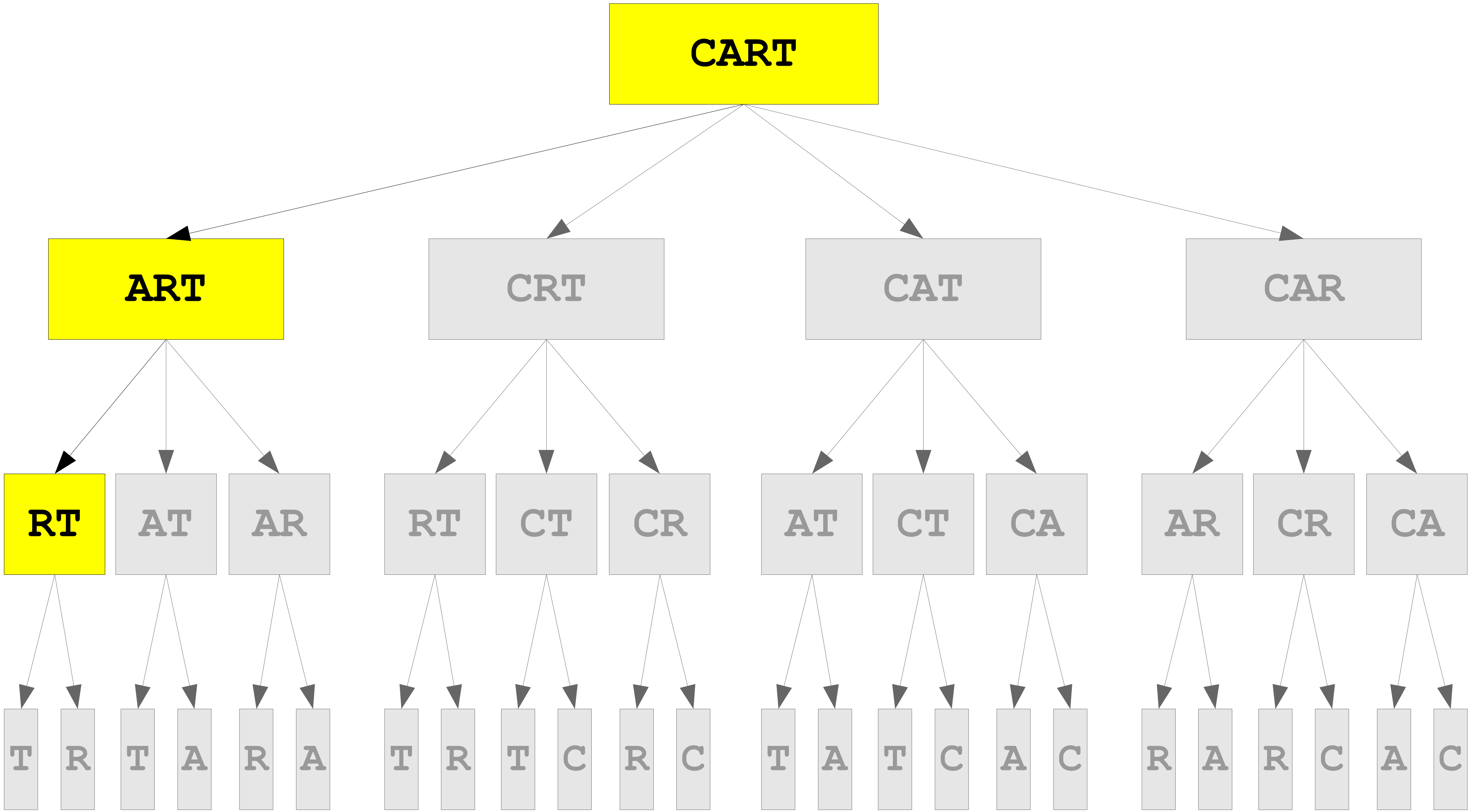
# Recursive Backtracking



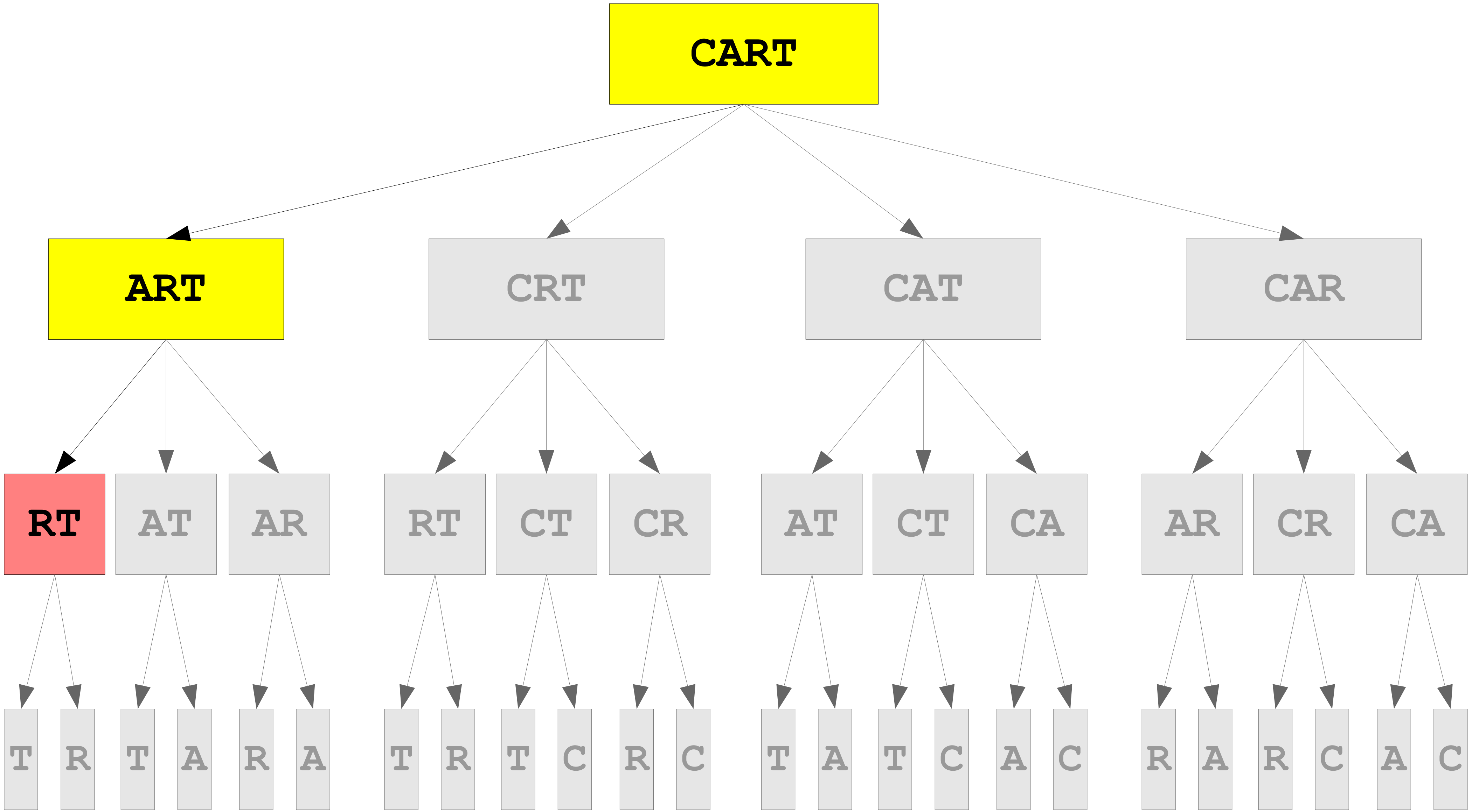
# Recursive Backtracking



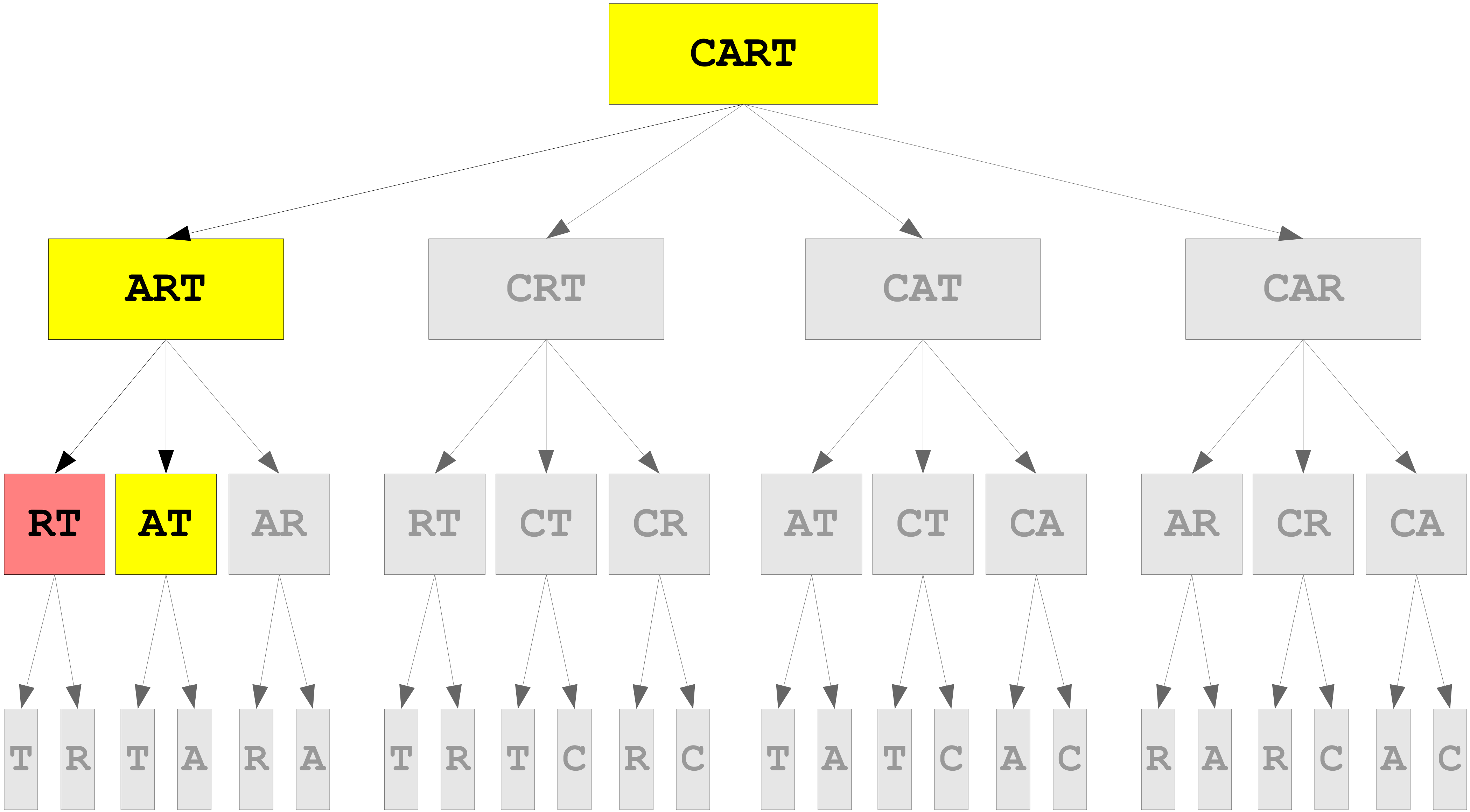
# Recursive Backtracking



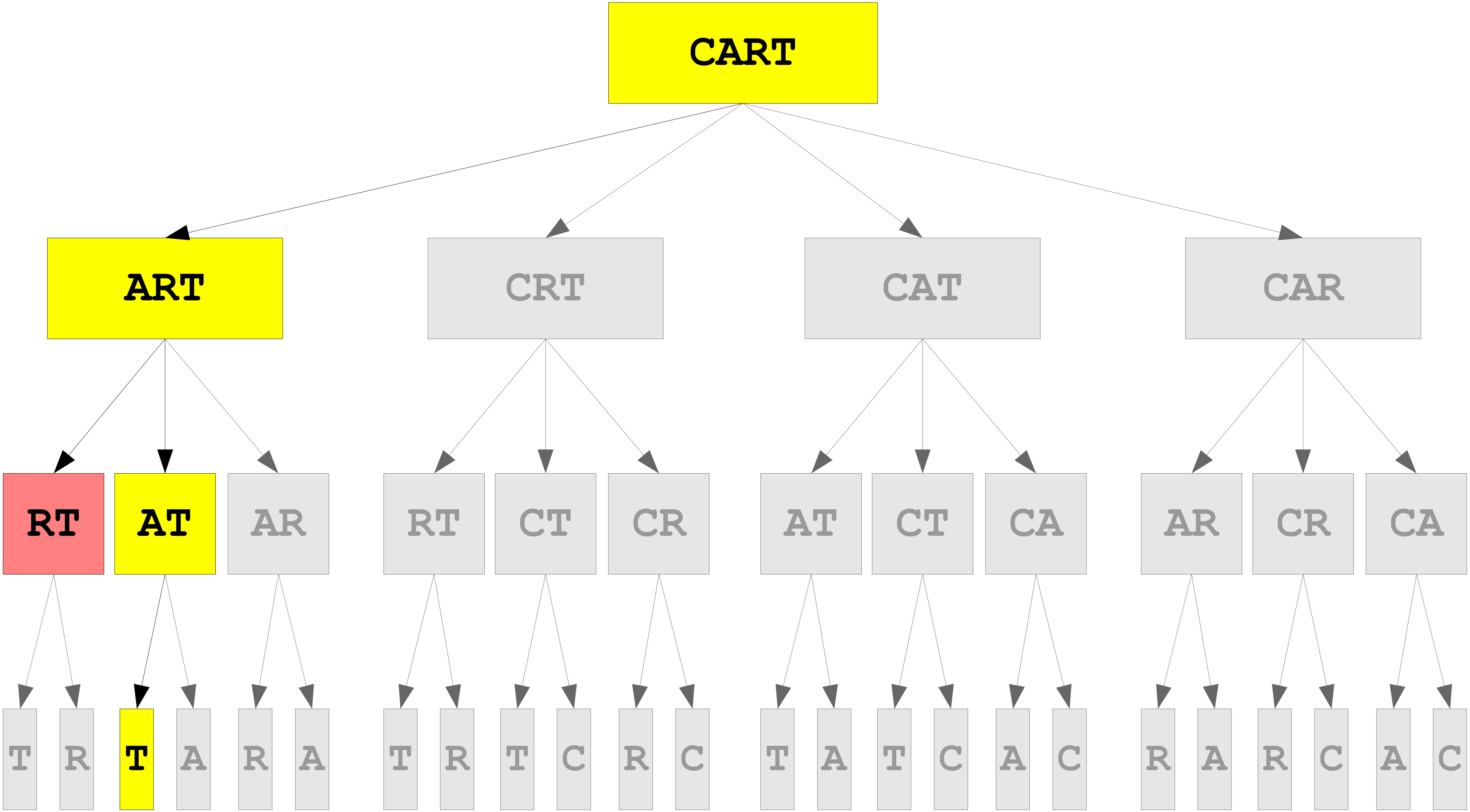
# Recursive Backtracking



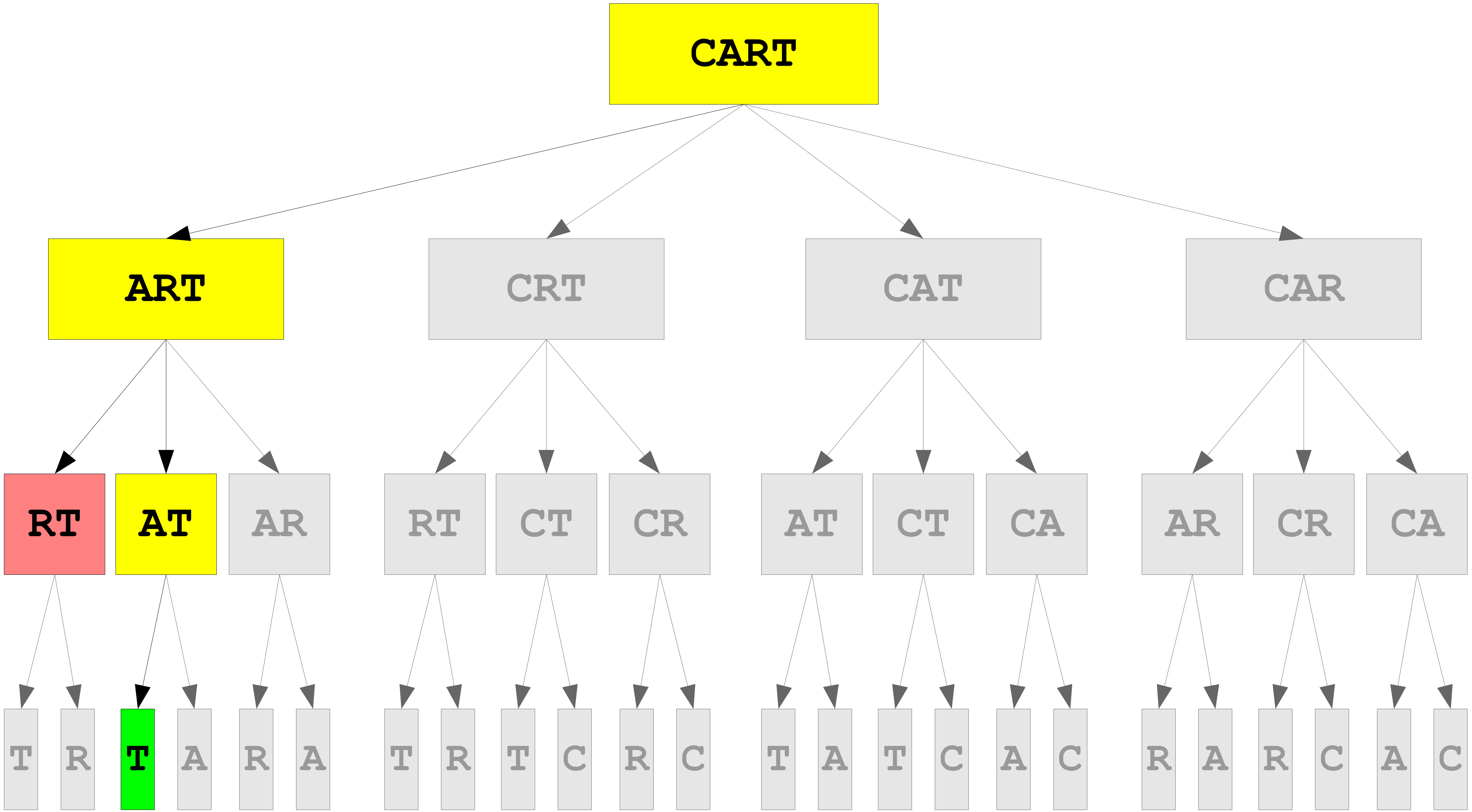
# Recursive Backtracking



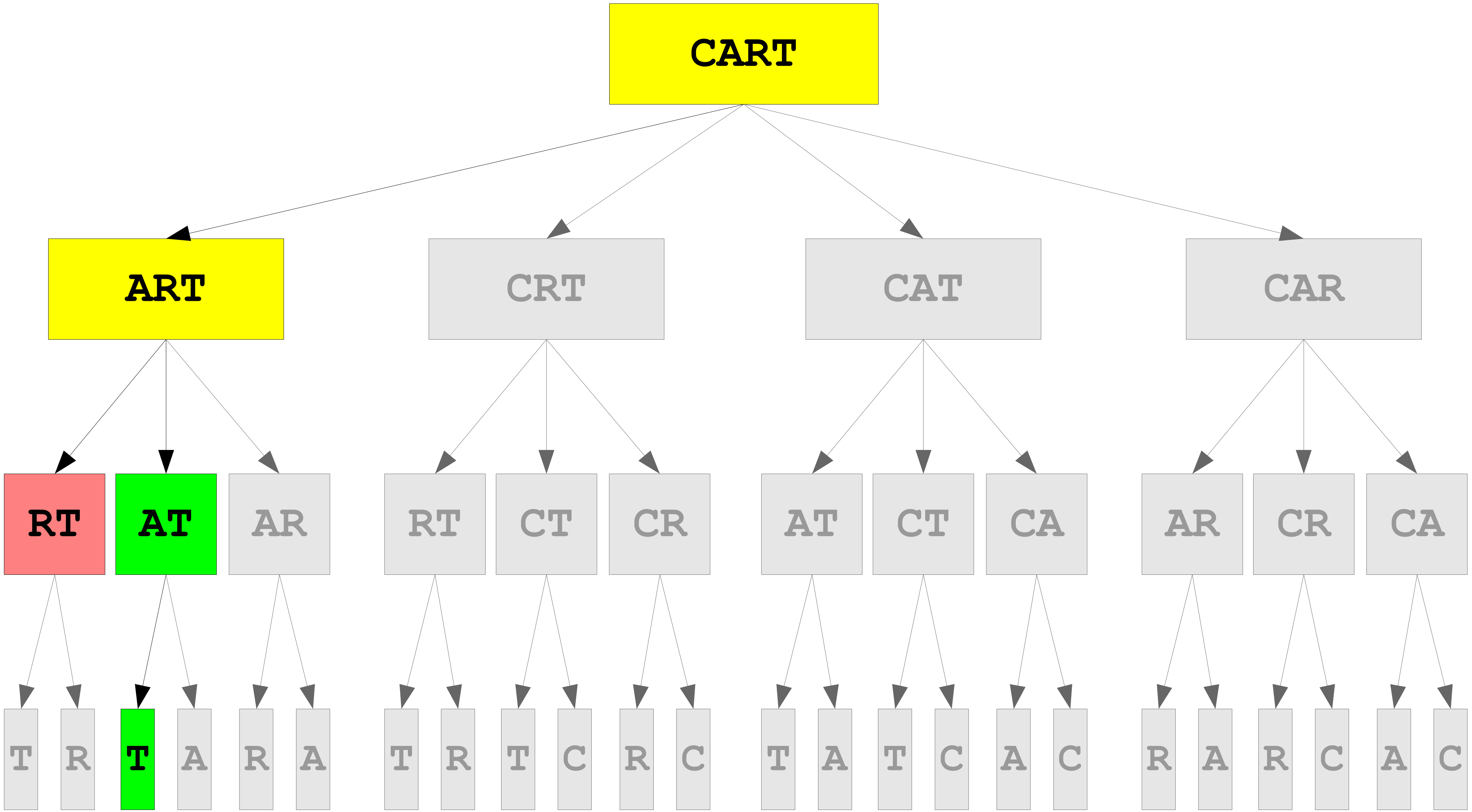
# Recursive Backtracking



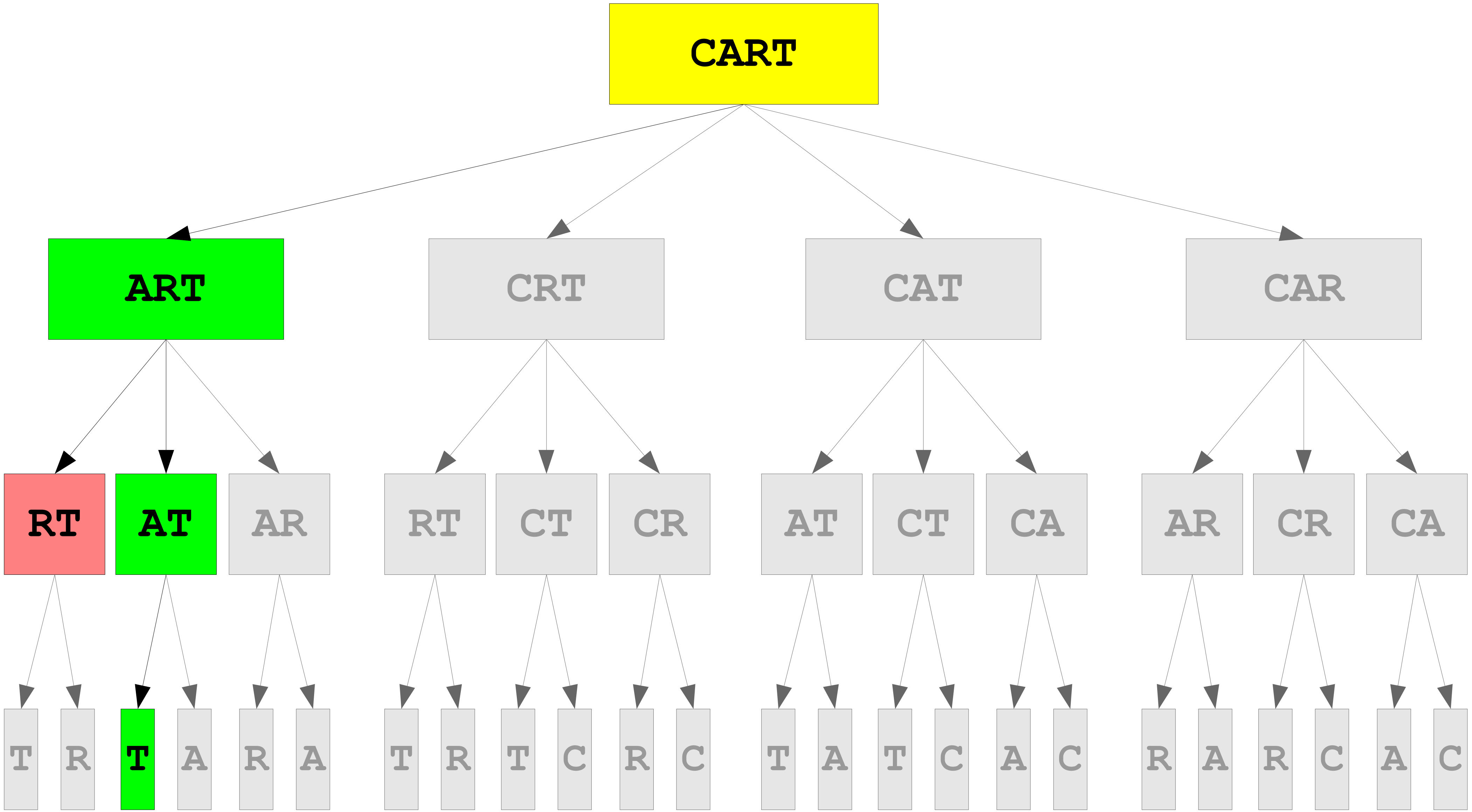
# Recursive Backtracking



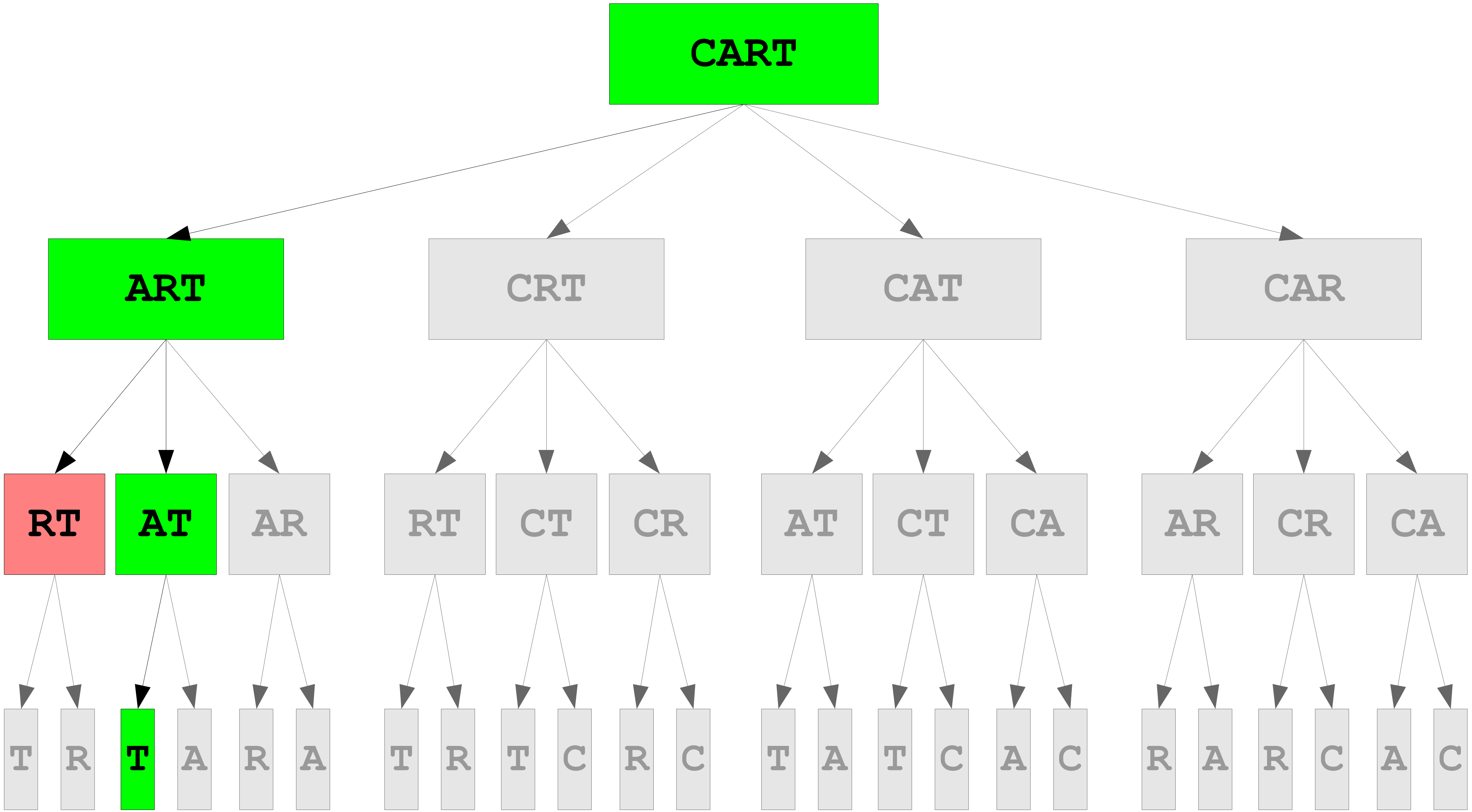
# Recursive Backtracking



# Recursive Backtracking



# Recursive Backtracking



# Failure in Backtracking

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| S | T | A | R | T | L | I | N | G |

# Failure in Backtracking

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| S | T | A | R | T | L | I | N | G |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| T | A | R | T | L | I | N | G |

# Failure in Backtracking

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| S | T | A | R | T | L | I | N | G |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| T | A | R | T | L | I | N | G |

# Failure in Backtracking

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| S | T | A | R | T | L | I | N | G |

# Failure in Backtracking

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| S | T | A | R | T | L | I | N | G |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| S | A | R | T | L | I | N | G |

# Failure in Backtracking

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| S | T | A | R | T | L | I | N | G |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| S | A | R | T | L | I | N | G |

# Failure in Backtracking

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| S | T | A | R | T | L | I | N | G |

# Failure in Backtracking

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| S | T | A | R | T | L | I | N | G |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| S | T | R | T | L | I | N | G |

# Failure in Backtracking

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| S | T | A | R | T | L | I | N | G |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| S | T | R | T | L | I | N | G |

# Failure in Backtracking

| S | T | A | R | T | L | I | N | G |
|---|---|---|---|---|---|---|---|---|

# Failure in Backtracking

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| S | T | A | R | T | L | I | N | G |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| S | T | A | T | L | I | N | G |

# Failure in Backtracking

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| S | T | A | R | T | L | I | N | G |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| S | T | A | T | L | I | N | G |

# Failure in Backtracking

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| S | T | A | R | T | L | I | N | G |

# Failure in Backtracking

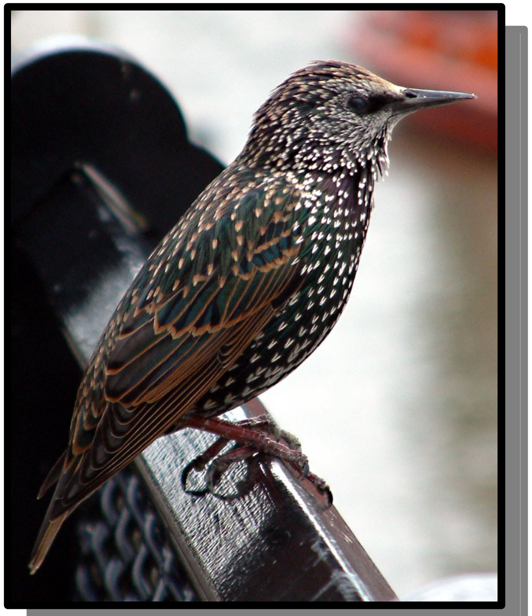
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| S | T | A | R | T | L | I | N | G |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| S | T | A | R | L | I | N | G |

# Failure in Backtracking

| S | T | A | R | T | L | I | N | G |
|---|---|---|---|---|---|---|---|---|

| S | T | A | R | L | I | N | G |
|---|---|---|---|---|---|---|---|



Sources: http://www.rubyinside.com/wp-content/uploads/2008/07/starling.jpg an
http://2.bp.blogspot.com/-hCzVof1xqWo/TvaFXmpJAmI/AAAAAAAAIdk/QzUg1sK mmons.jpg

# Failure in Backtracking

| S | T | A | R | T | L | I | N | G |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |

| S | T | A | R | L | I | N | G |
| --- | --- | --- | --- | --- | --- | --- | --- |

| T | A | R | L | I | N | G |
| --- | --- | --- | --- | --- | --- | --- |

# Failure in Backtracking

| S | T | A | R | T | L | I | N | G |
|---|---|---|---|---|---|---|---|---|

| S | T | A | R | L | I | N | G |
|---|---|---|---|---|---|---|---|

| T | A | R | L | I | N | G |
|---|---|---|---|---|---|---|

# Failure in Backtracking

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| S | T | A | R | T | L | I | N | G |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| S | T | A | R | L | I | N | G |

# Failure in Backtracking

| S | T | A | R | T | L | I | N | G |
|---|---|---|---|---|---|---|---|---|

| S | T | A | R | L | I | N | G |
|---|---|---|---|---|---|---|---|

| S | A | R | L | I | N | G |
|---|---|---|---|---|---|---|

# Failure in Backtracking

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| S | T | A | R | T | L | I | N | G |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| S | T | A | R | L | I | N | G |

| | | | | | | |
|---|---|---|---|---|---|---|
| S | A | R | L | I | N | G |

# Recursive Backtracking

```
if (problem is sufficiently simple) {
    return whether or not the problem is solvable
} else {
    for (each choice) {
        try out that choice.
        if (that choice leads to success) {
            return success
        }
    }
    return failure
}
```

# Recursive Backtracking

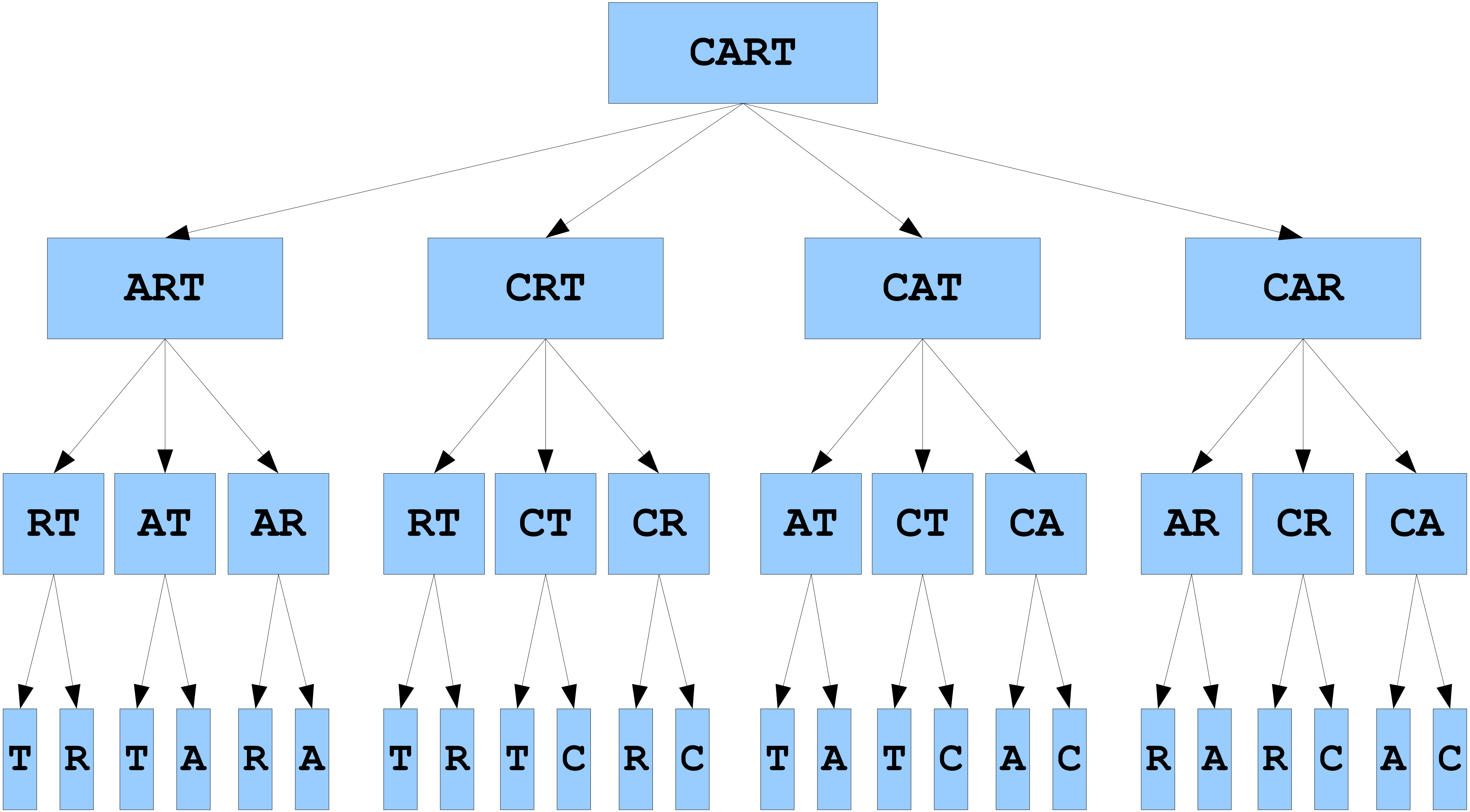
```
if (problem is sufficiently simple) {
    return whether or not the problem is solvable
} else {
    for (each choice) {
        try out that choice.
        if (that choice leads to success) {
            return success;
        }
    }
    return failure
}
```

Note that ***if*** it succeeds, ***then*** we return success. If it doesn't succeed, that doesn't mean we've failed – it just means we need to try out the next option.

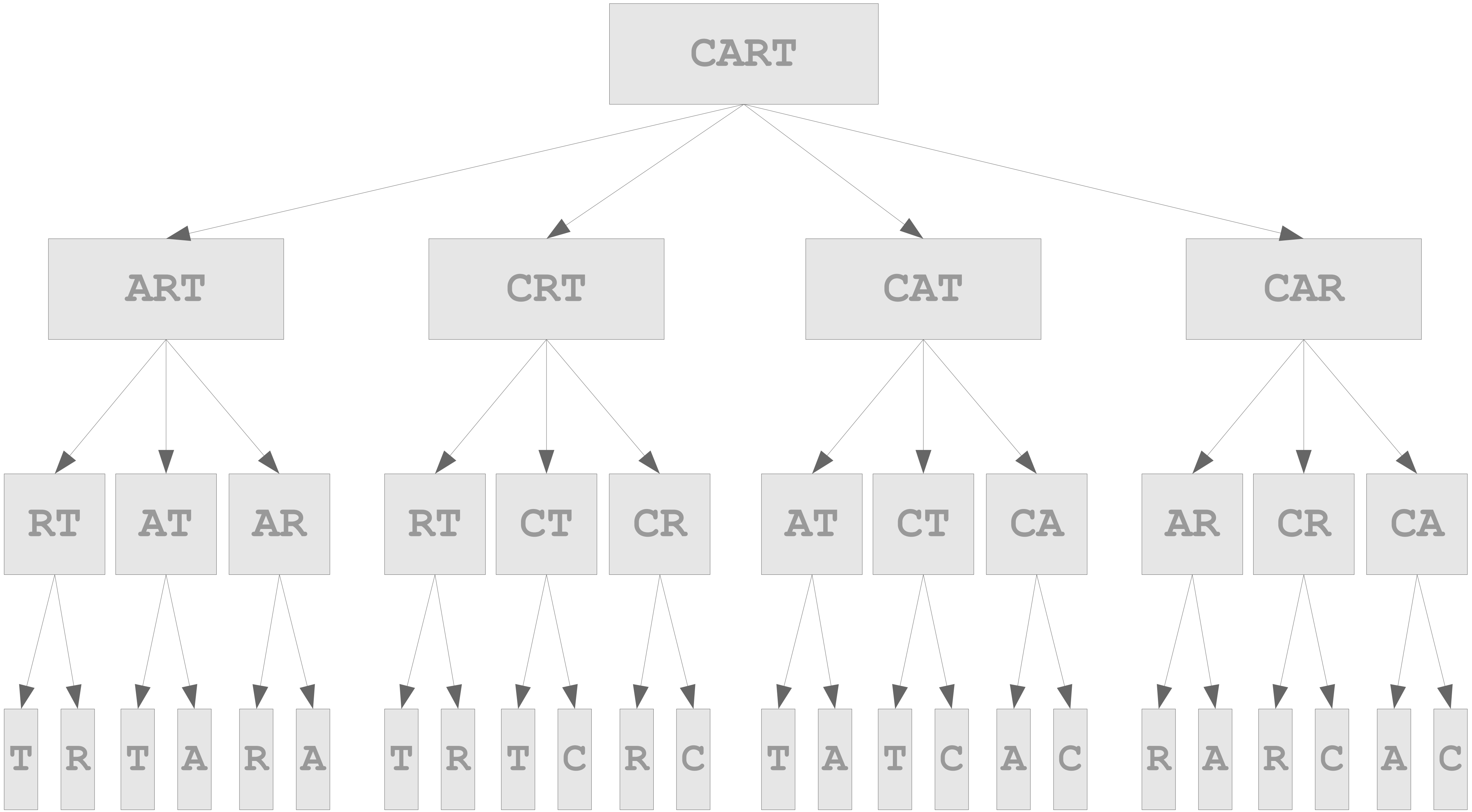
# Failure in Backtracking

- Returning false in recursive backtracking does ***not*** mean that the entire problem is unsolvable!
- Instead, it just means that the current subproblem is unsolvable.
- Whoever made the call to this function can then try other options.
- Only when all options are exhausted can we know that the problem is unsolvable.

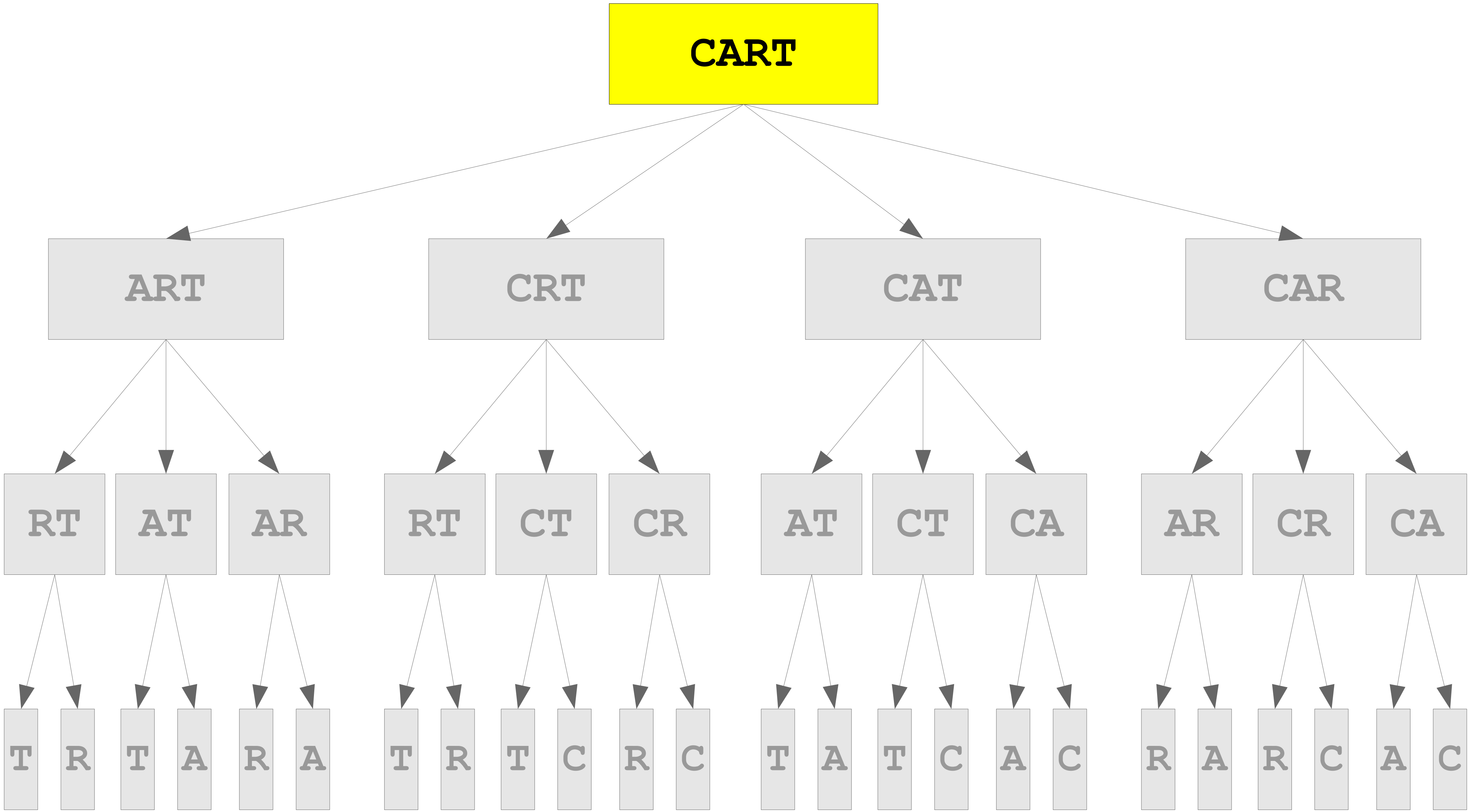
# Ur Doin It Rong!



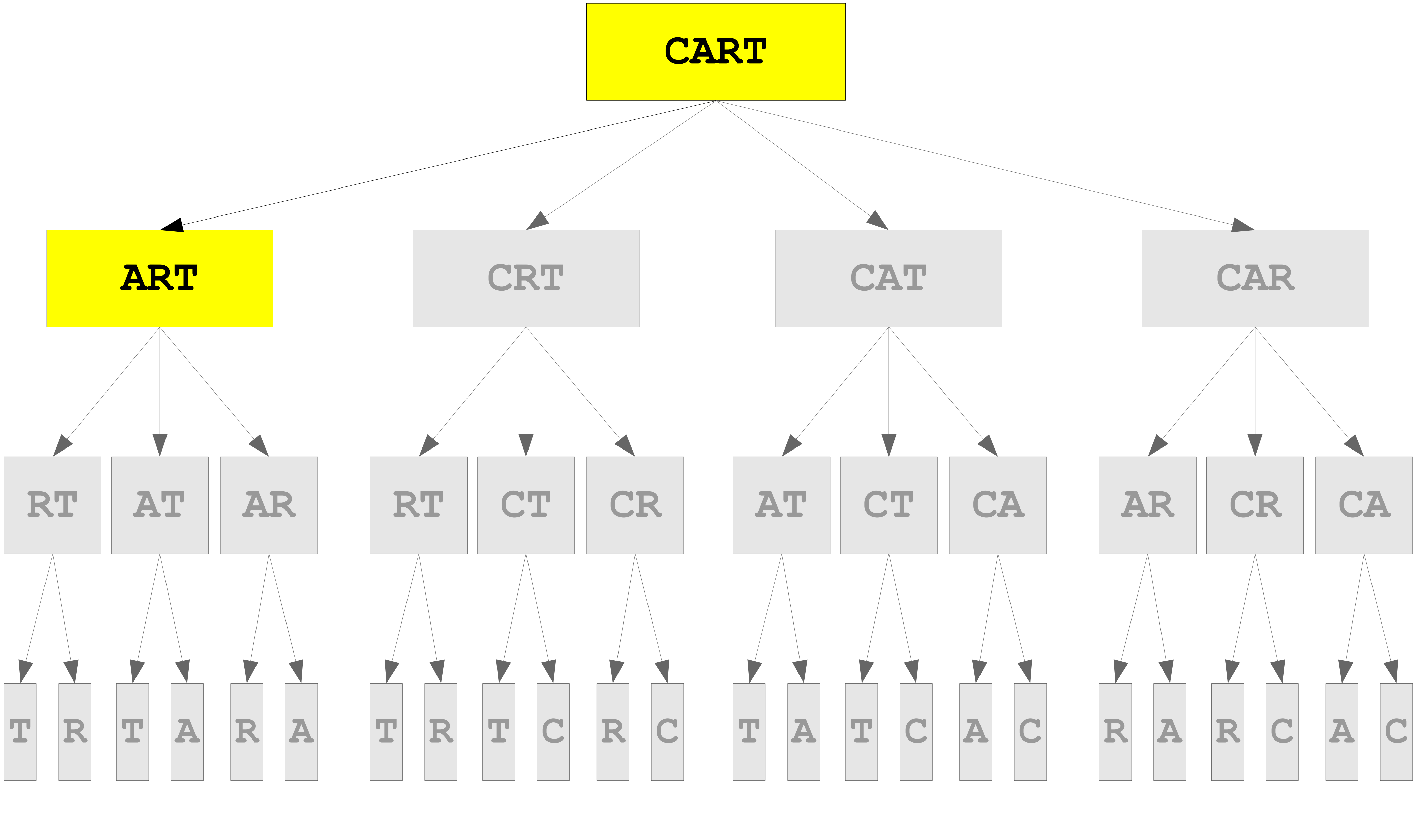
# Ur Doin It Rong!



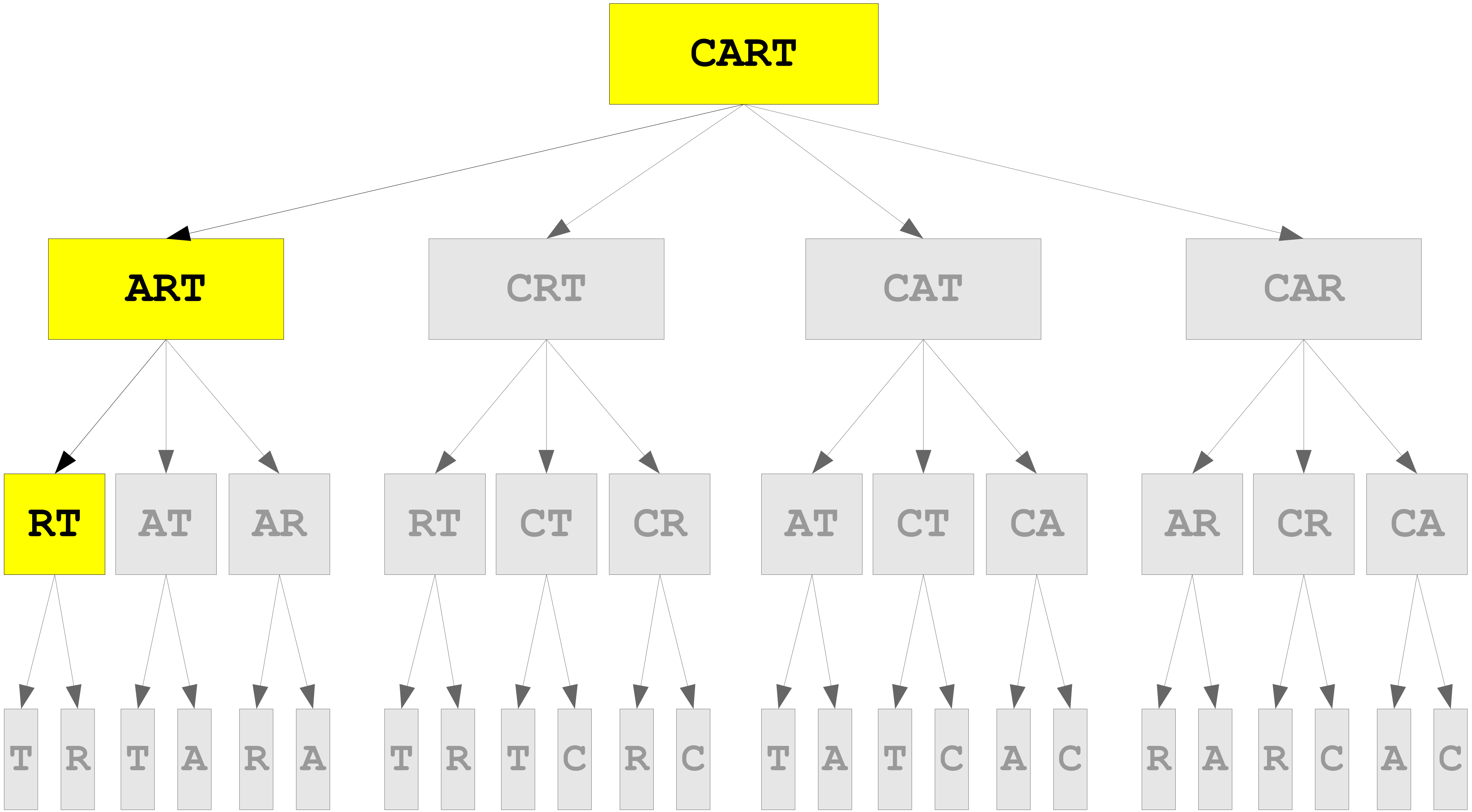
# Ur Doin It Rong!



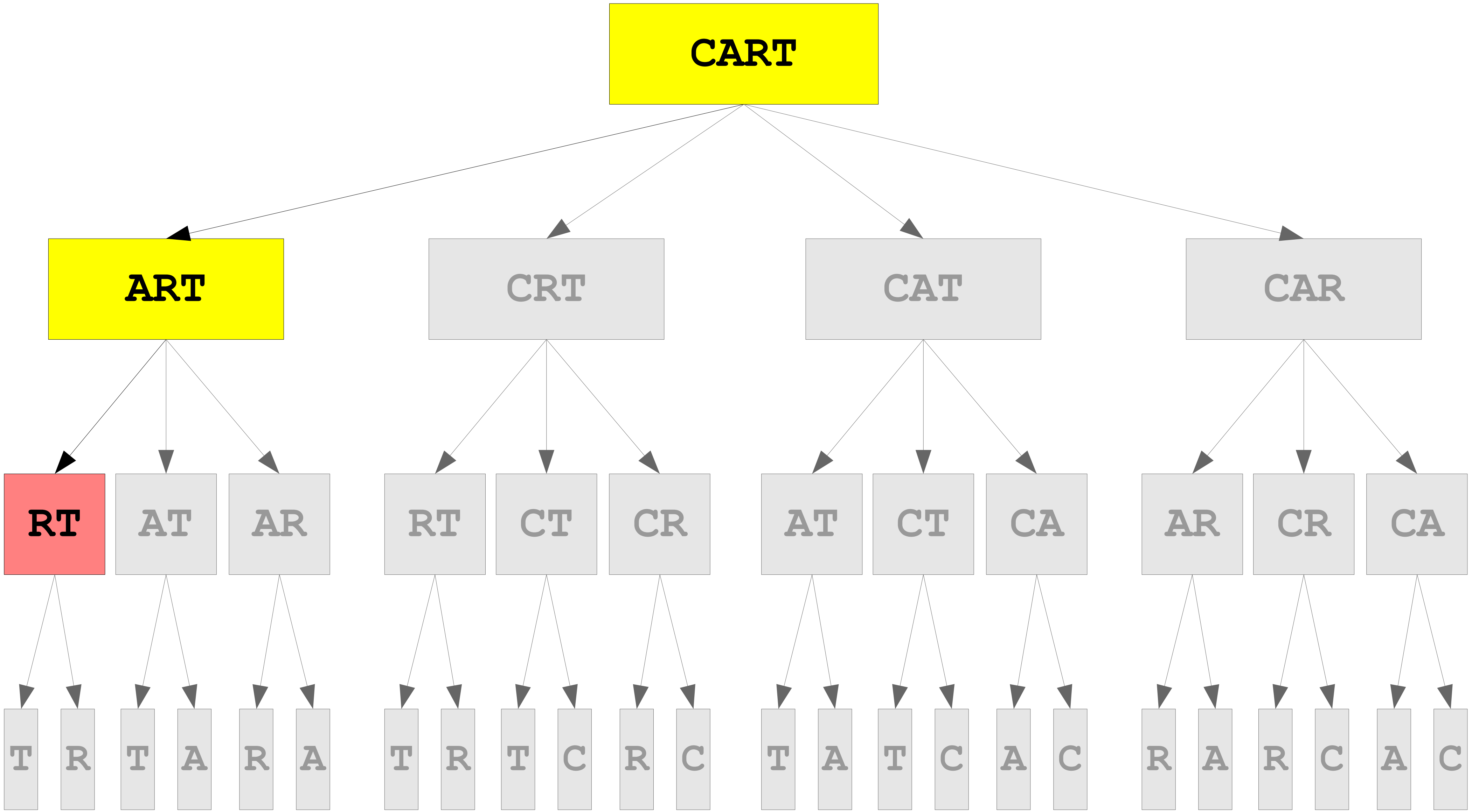
# Ur Doin It Rong!



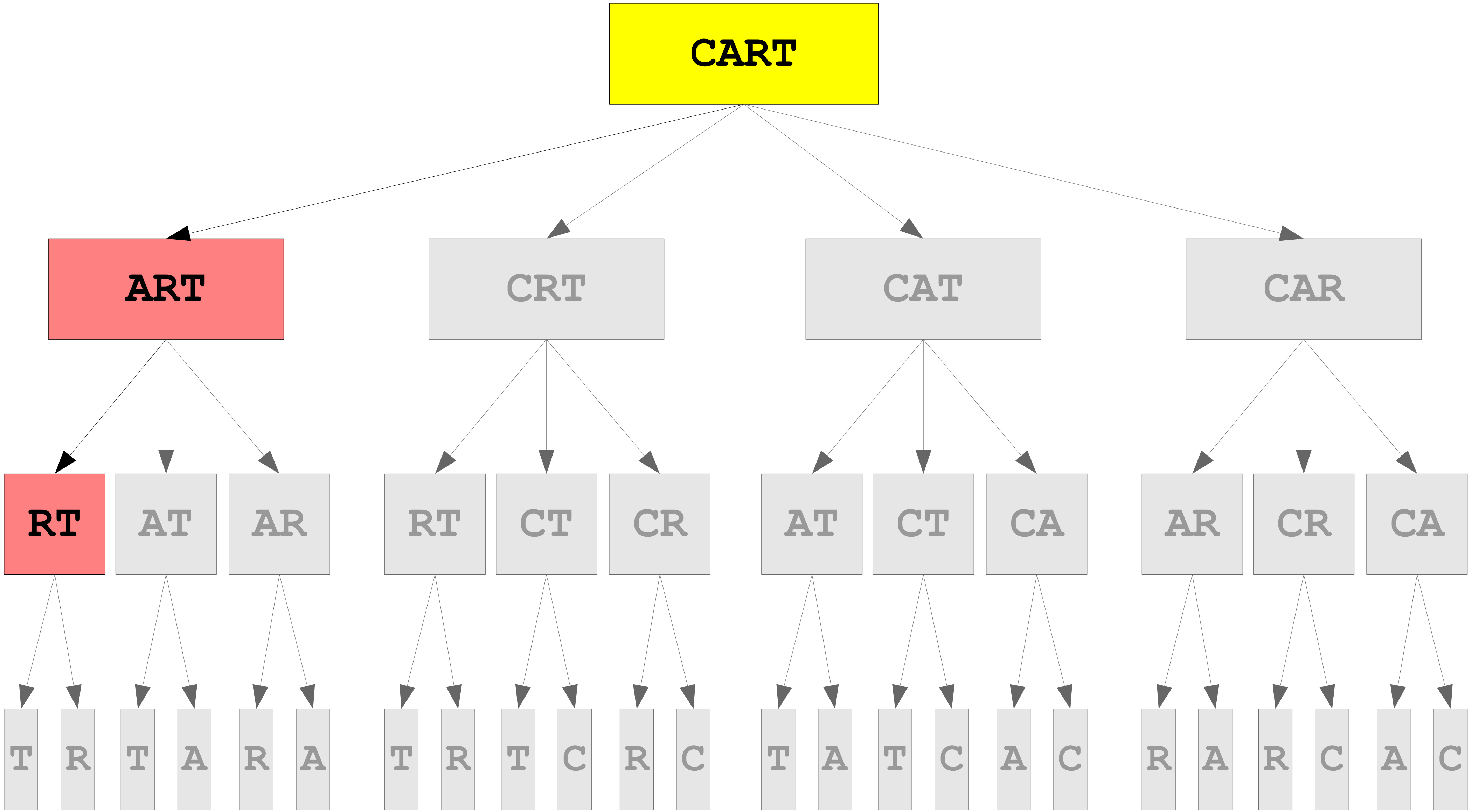
# Ur Doin It Rong!



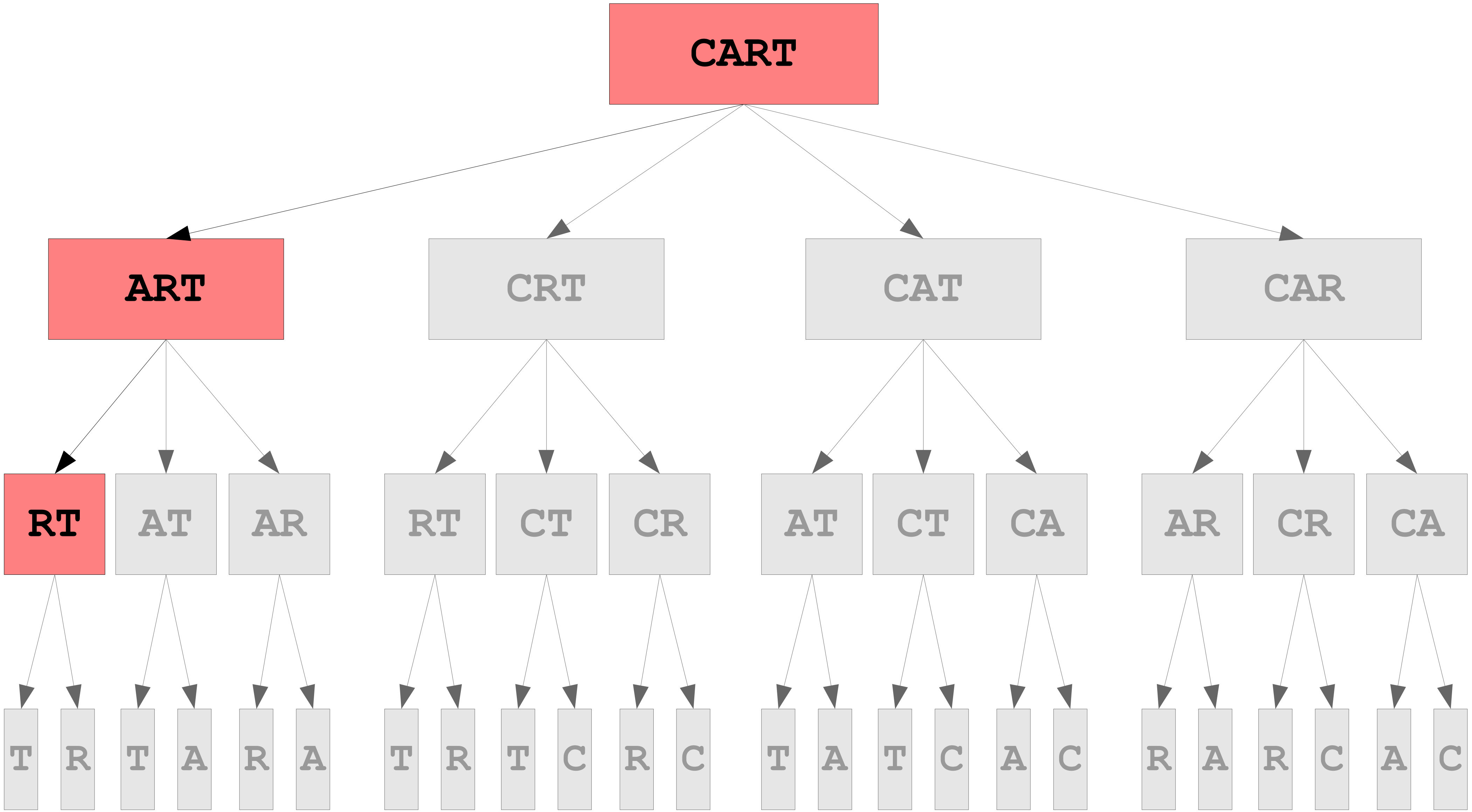
# Ur Doin It Rong!



# Ur Doin It Rong!



# Ur Doin It Rong!



# Next Week

- **Algorithmic Efficiency**
  - How can we compare the speed of two different algorithms?
- **Sorting Algorithms**
- **Implementing Collections Classes**