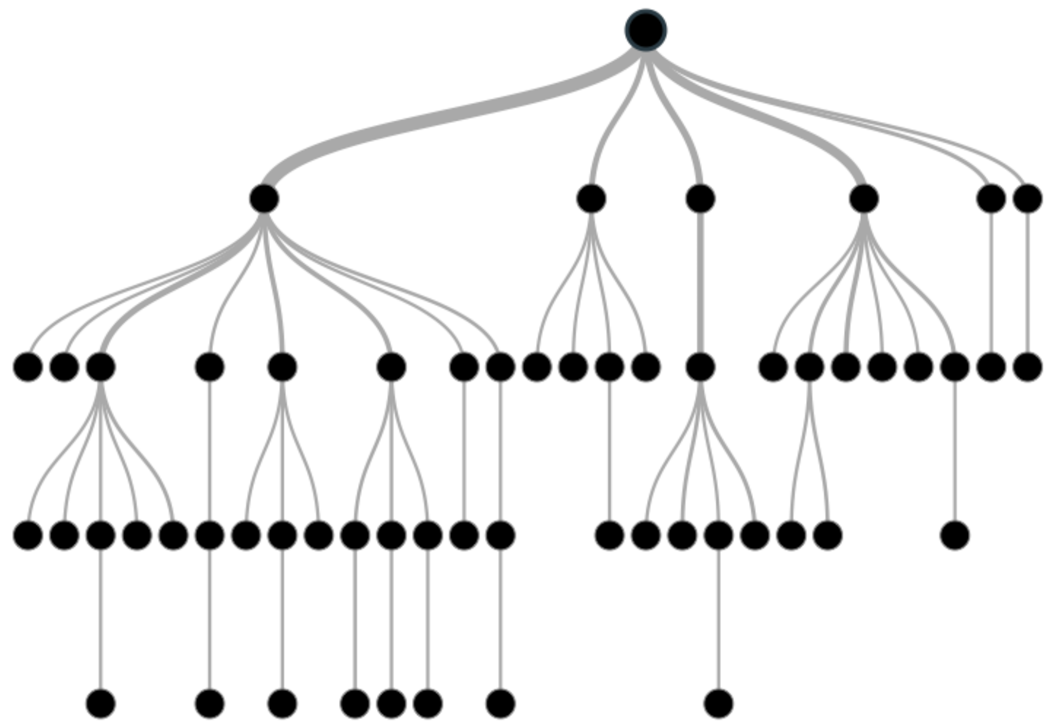
CS 106X

Lecture 9: Recursive Backtracking 1: Decision Trees

Monday, January 30, 2017



Programming Abstractions (Accelerated)
Winter 2017
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Chapter 8.2-8.3

Today's Topics

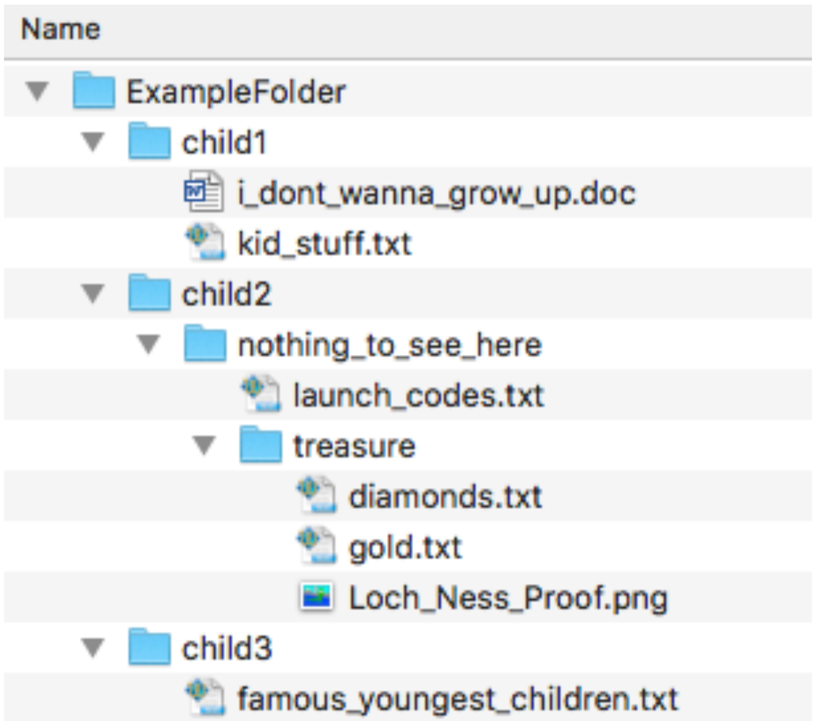
- Logistics:
  - Due dates...
  - All midterm accommodations: let us know **by today**.

- Recursion and Decision Trees
  - Folders and Directories
  - Reducible Words

- Recursive Backtracking: Exhaustive Search
  - Permutations

Recursion and Decision Trees

- The following is a graphical depiction of the files in a folder on my computer:



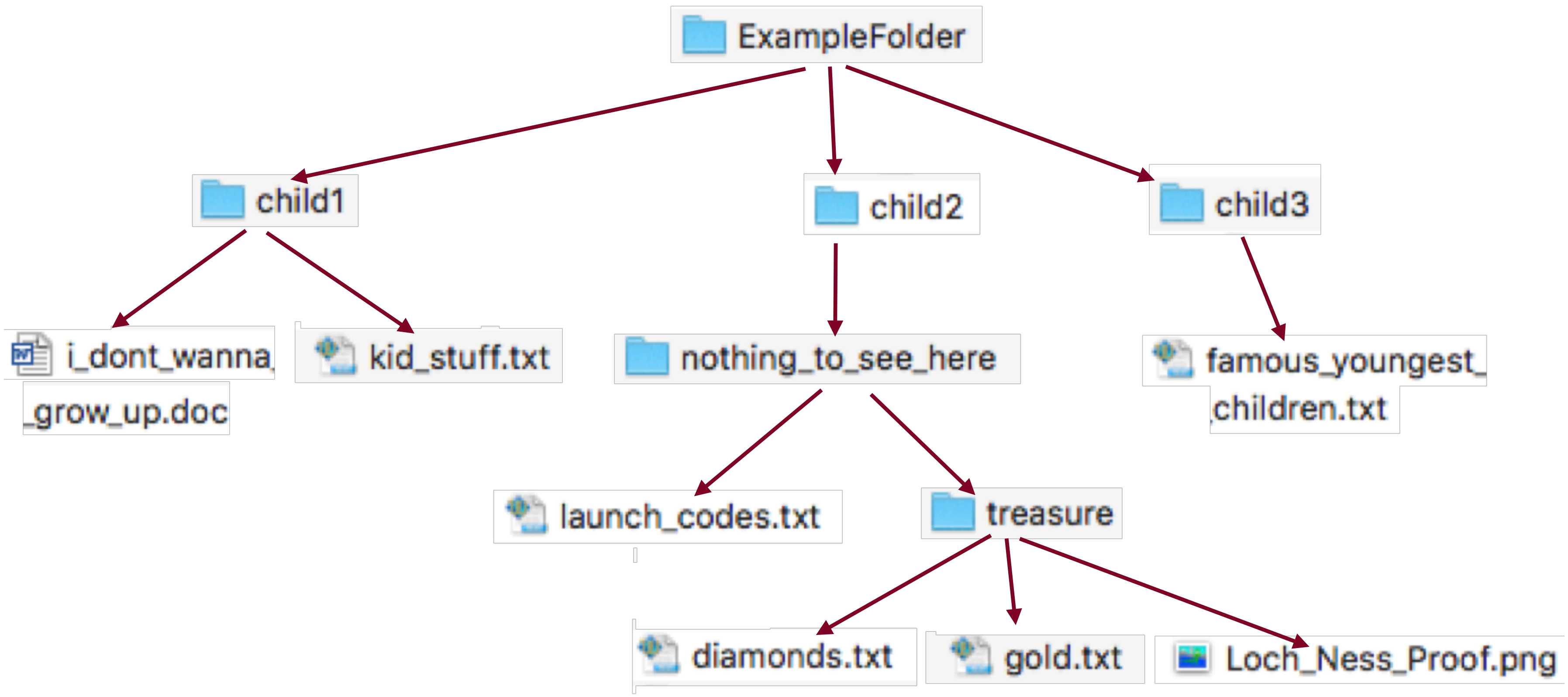
- The top-level folder is called "**ExampleFolder**", and it has three children folders, called "**child1**", "**child2**", and "**child3**".

- **child1** has two files, "**i_dont_wanna_grow_up.doc**" and "**kid_stuff.txt**"

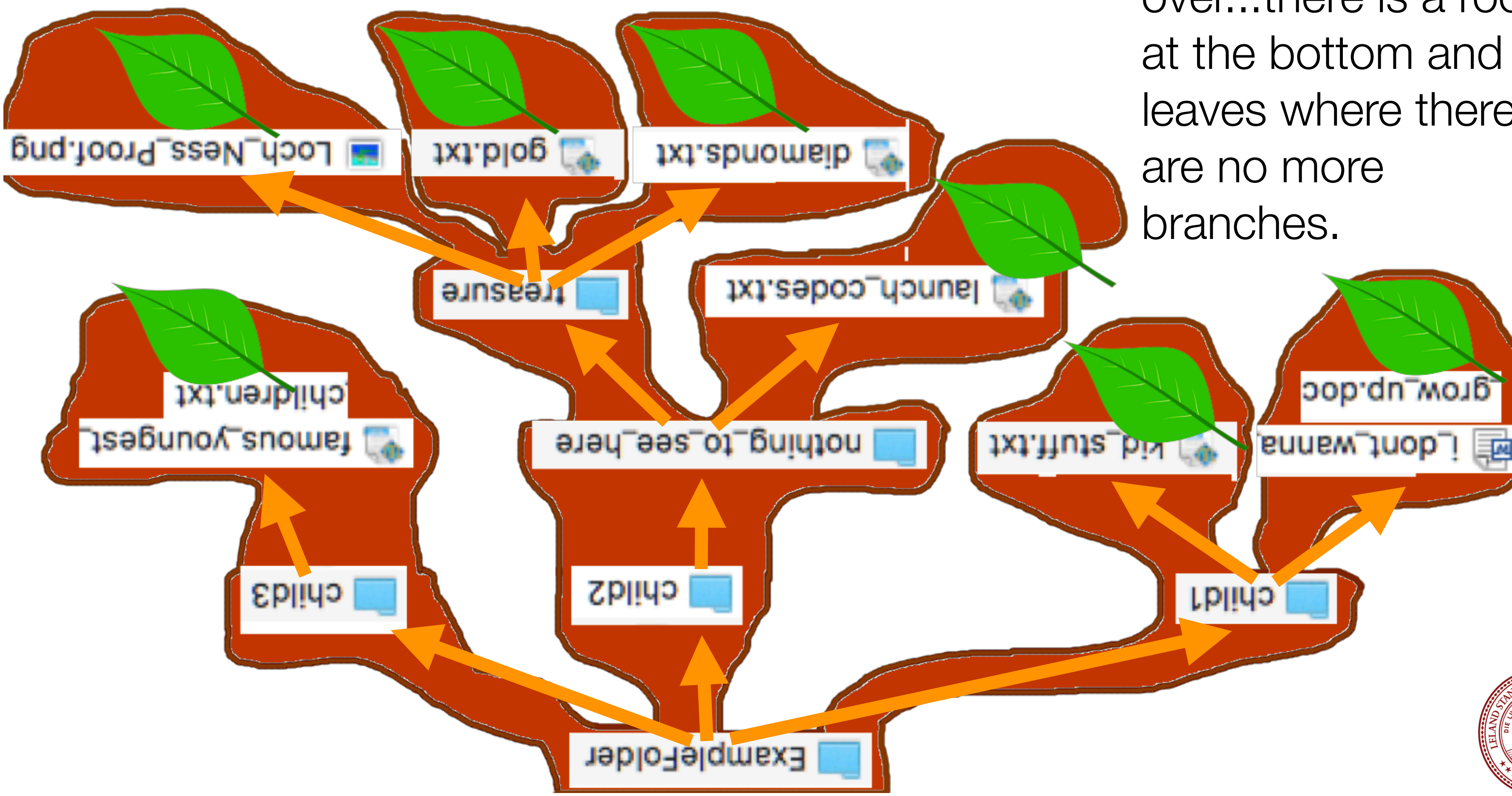
- etc.

Recursion and Decision Trees

- Let's re-draw that structure a bit, into a "tree" format.

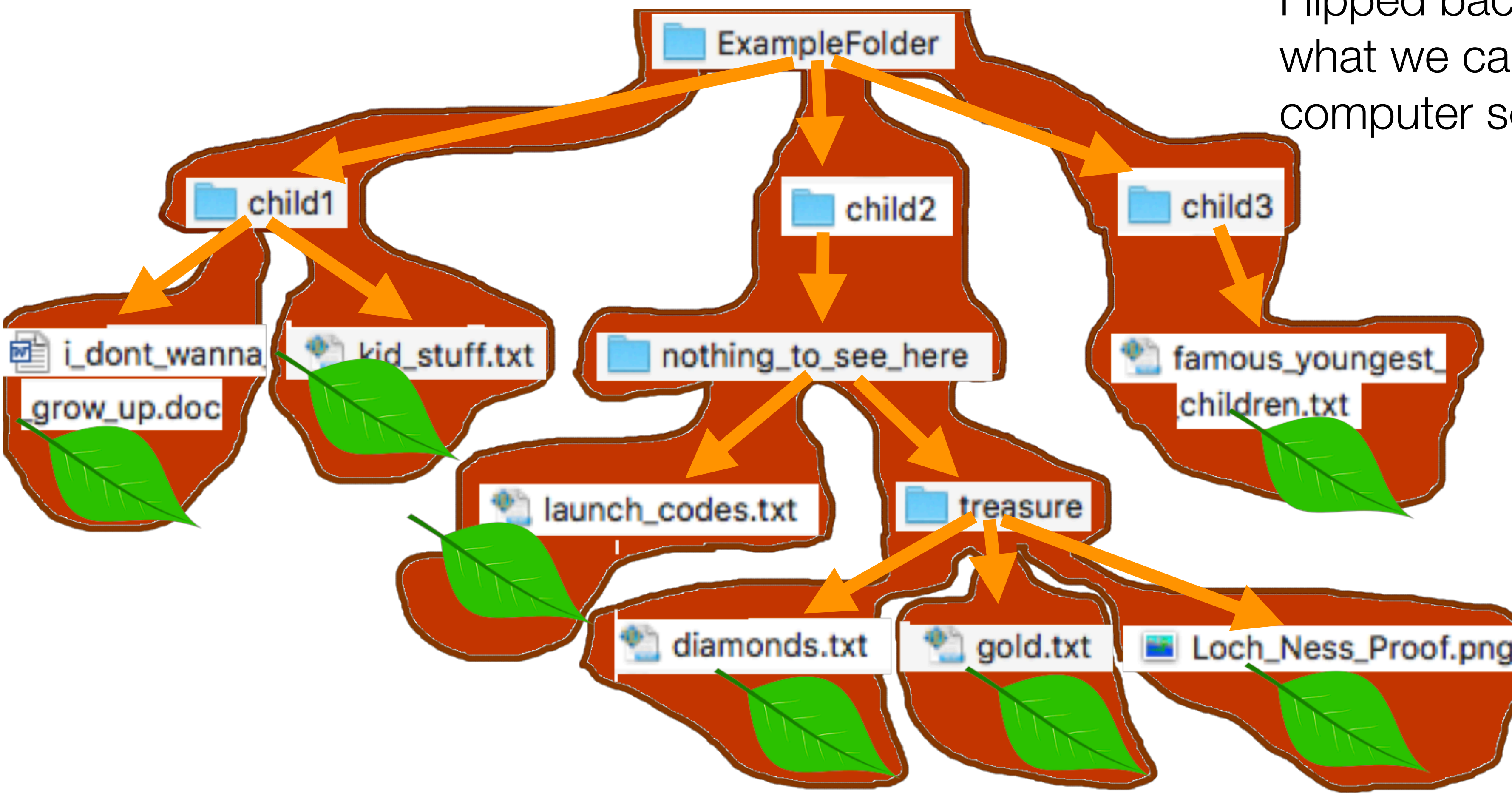


Recursion and Decision Trees



If we flip it over...there is a root at the bottom and leaves where there are no more branches.

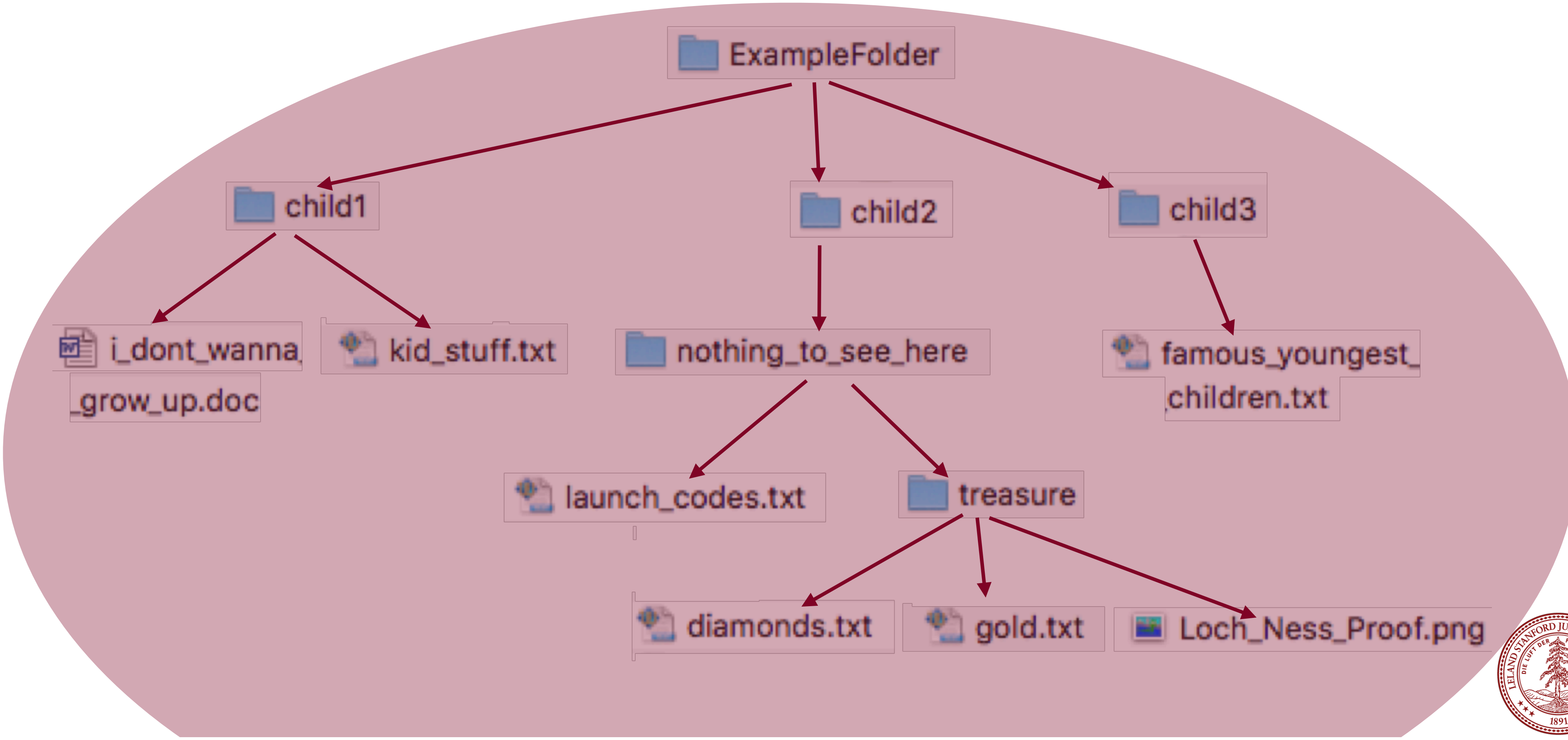
Recursion and Decision Trees



Flipped back, this is what we call a tree in computer science.

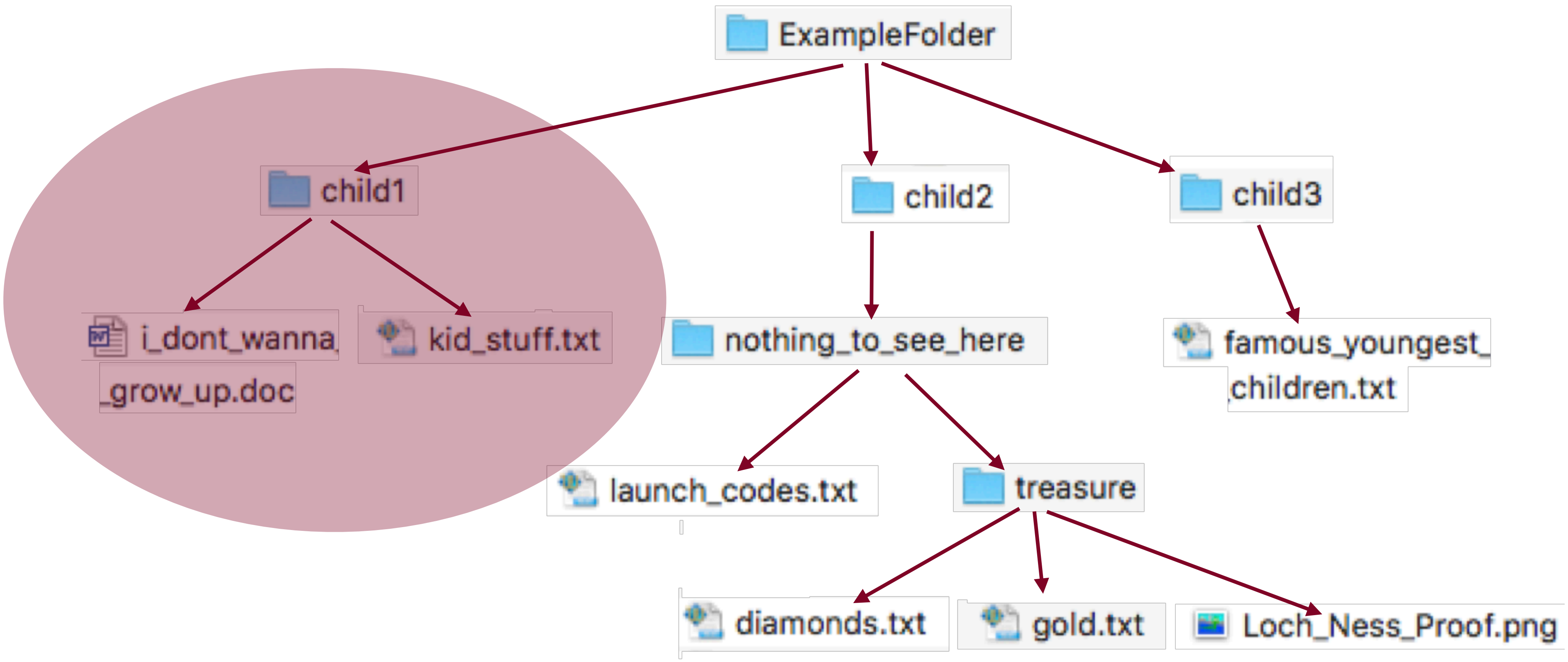
A folder is just a recursive container!

- A folder is a tree!



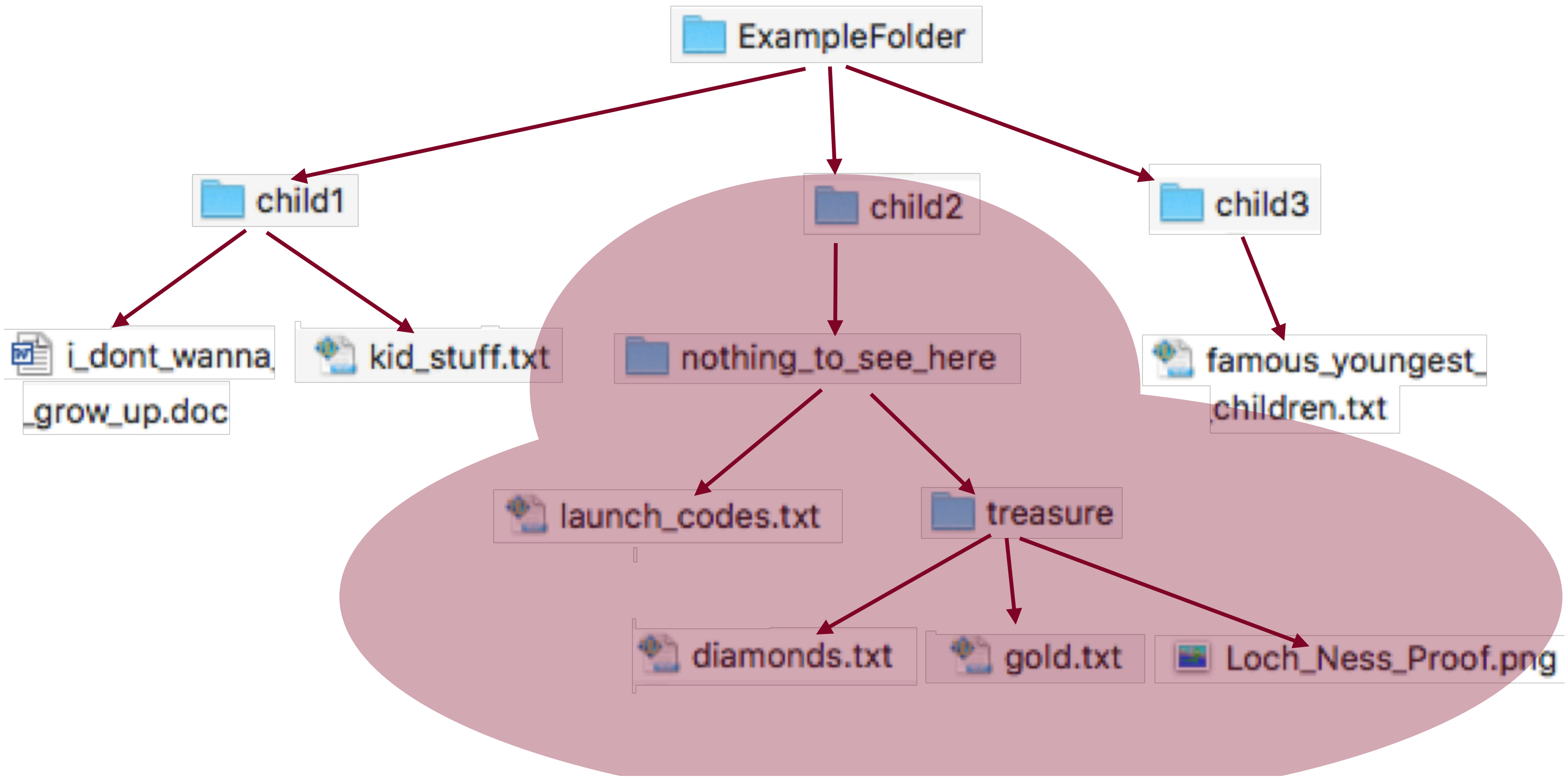
A folder is just a recursive container!

- All children are also complete trees!



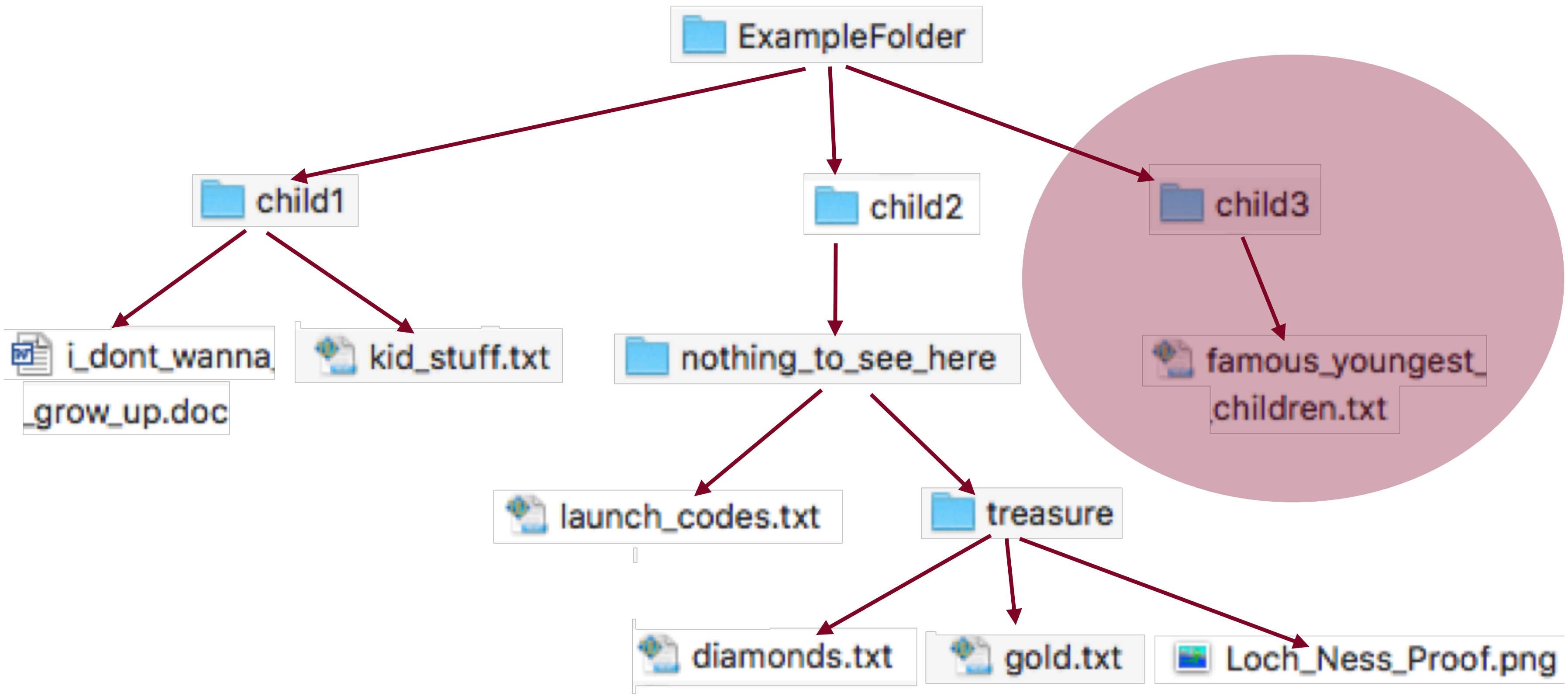
A folder is just a recursive container!

- All children are also complete trees!



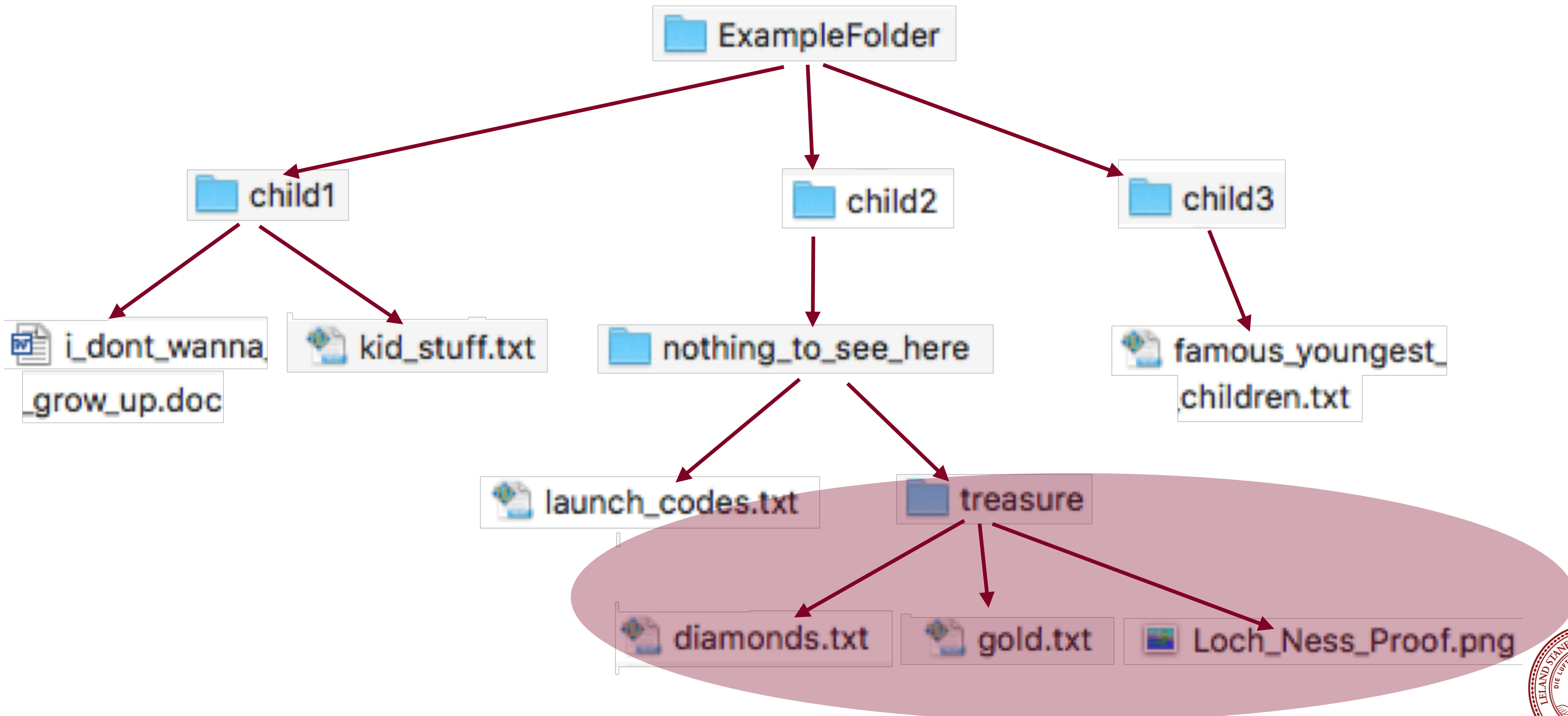
A folder is just a recursive container!

- All children are also complete trees!



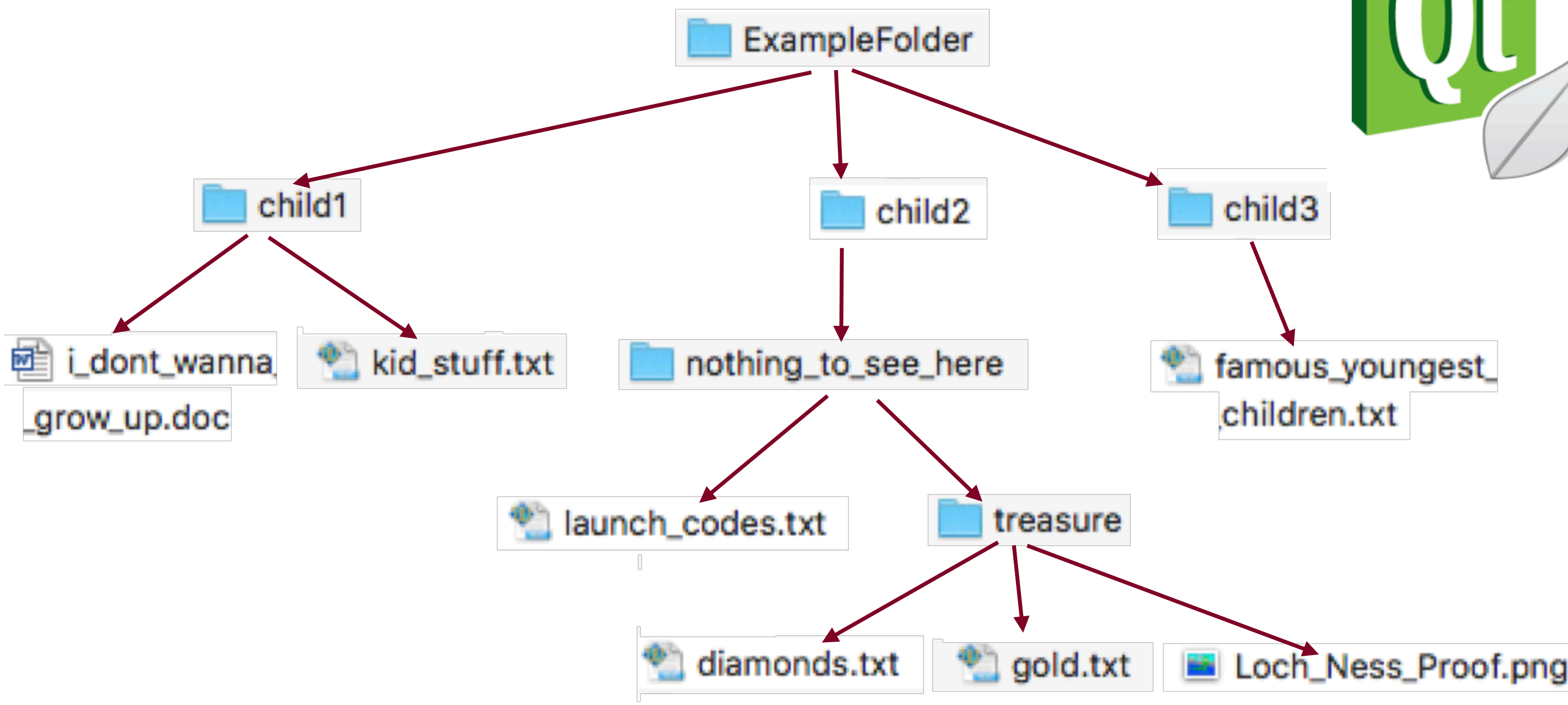
A folder is just a recursive container!

- All children are also complete trees!



Let's write a program to output all files in a folder

- All children are also complete trees!



Another Example: Reducible Words

Here is a word puzzle: "Is there a nine-letter English word that can be reduced to a single-letter word one letter at at time by removing letters, leaving a legal word at each step?

Another Example: Reducible Words

4-letter example:

cart ☞ art ☞ at ☞ a

can you think of a nine letter word?

Another Example: Reducible Words

startling

startling ☞ starling ☞ staring ☞ string ☞sting ☞sing ☞sin ☞in ☞i

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

All Reducible 9-letter words

can we do this iteratively?

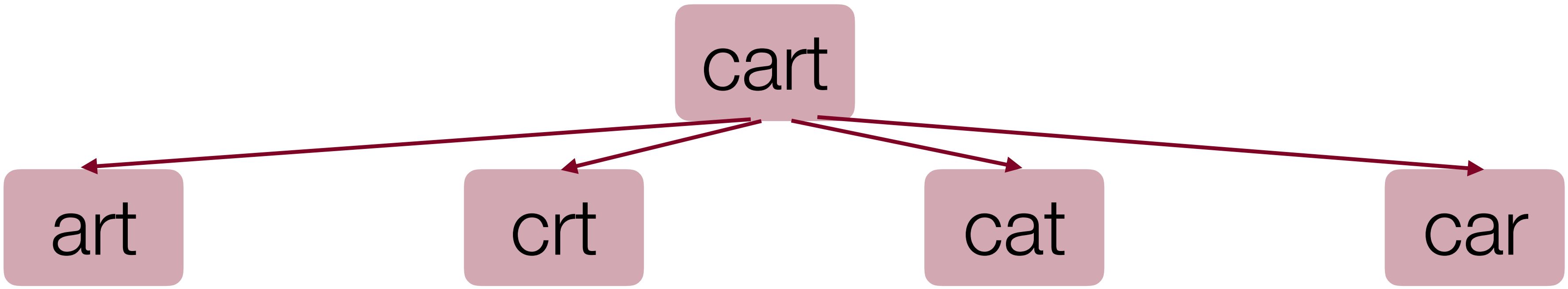
it would be very messy!

All Reducible 9-letter words

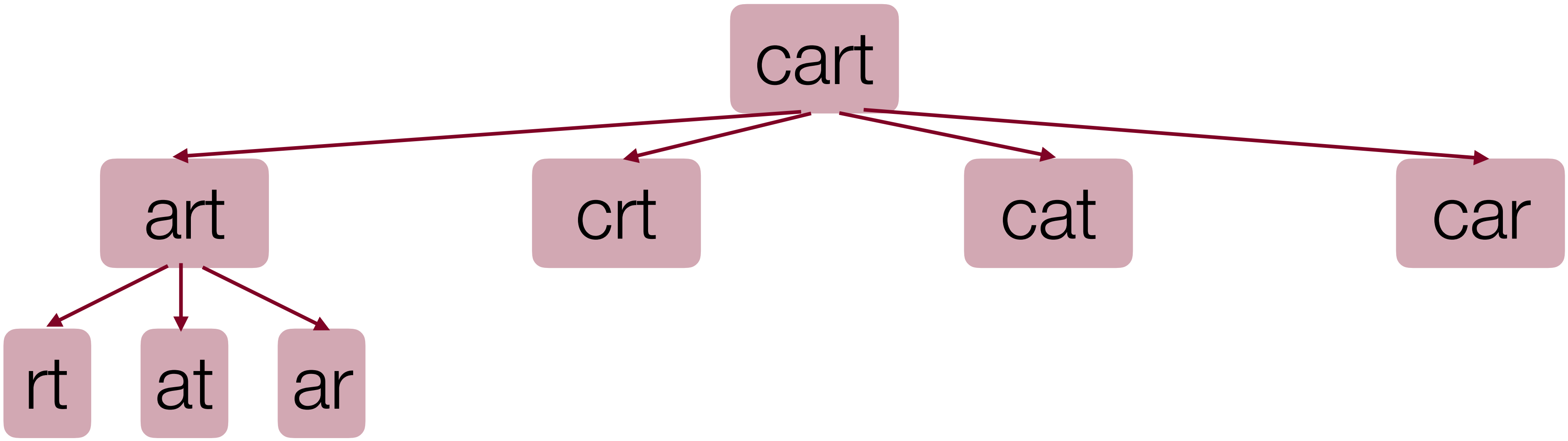
can we do this recursively?

yes!
what is the decision tree?

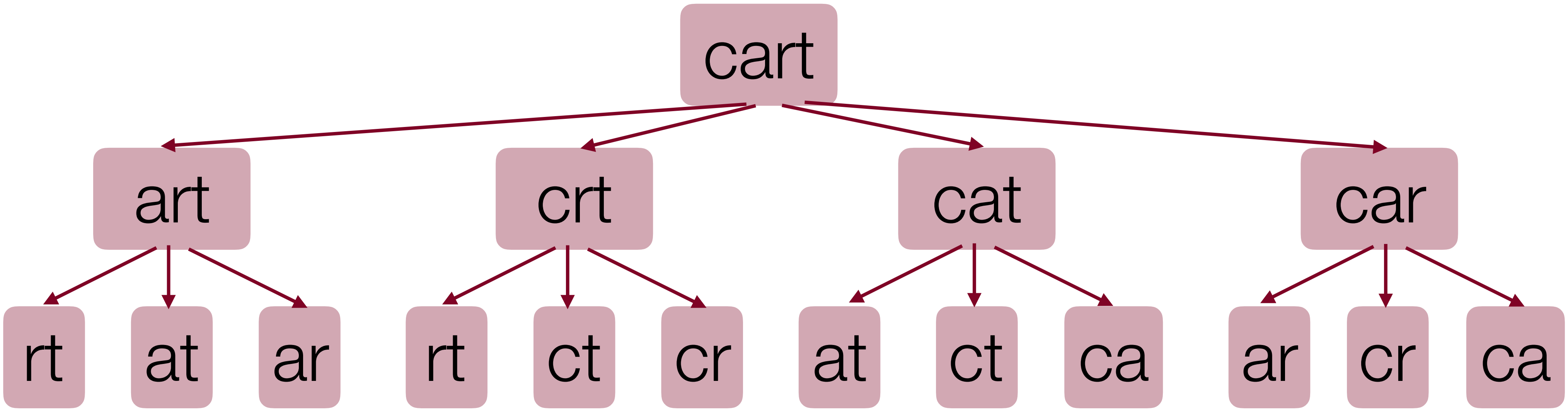
Reducability Decision Tree



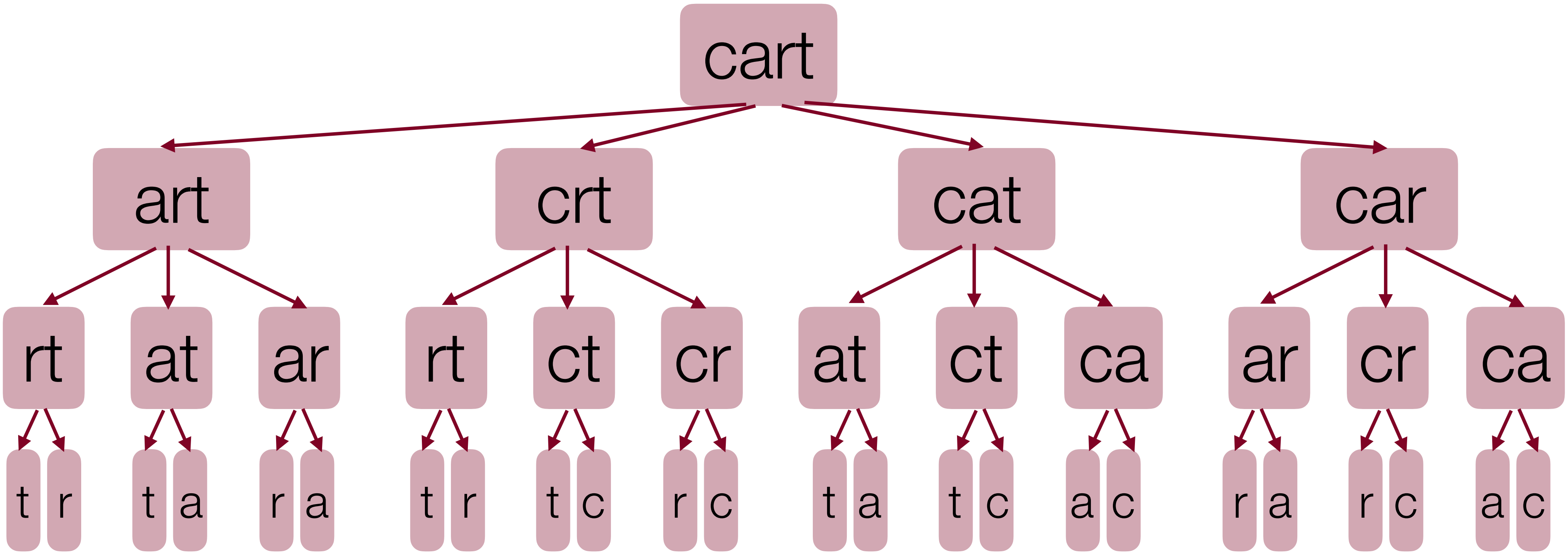
Reducability Decision Tree



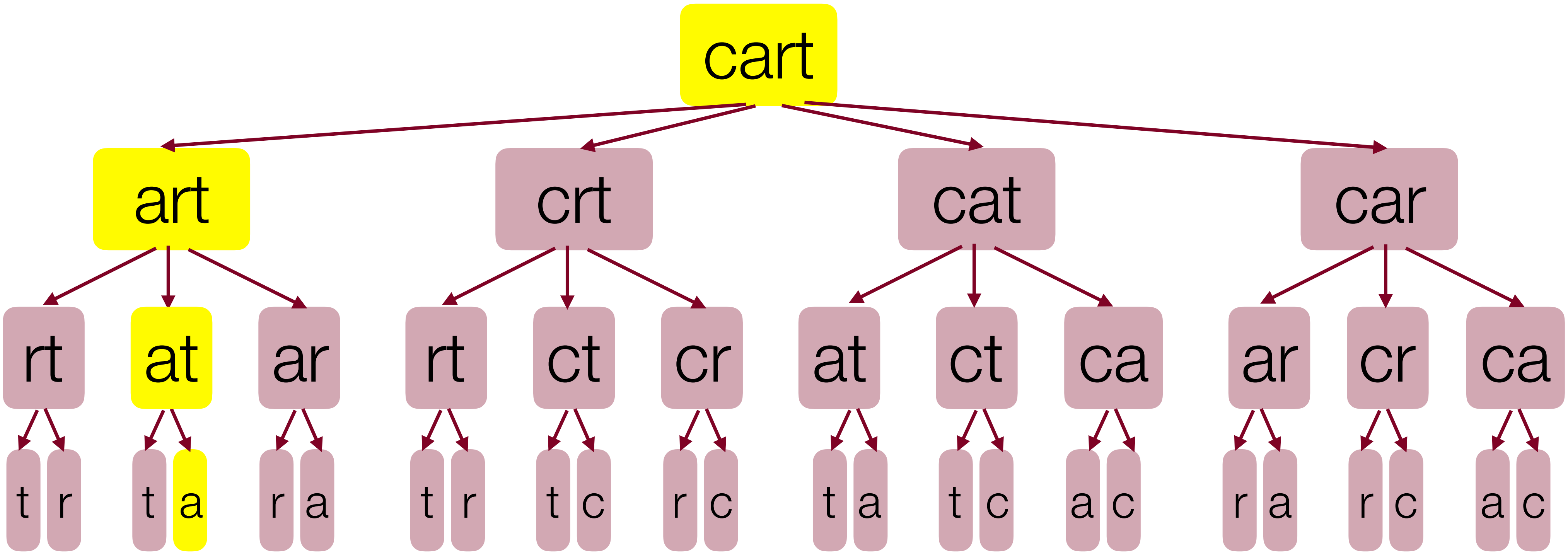
Reducability Decision Tree



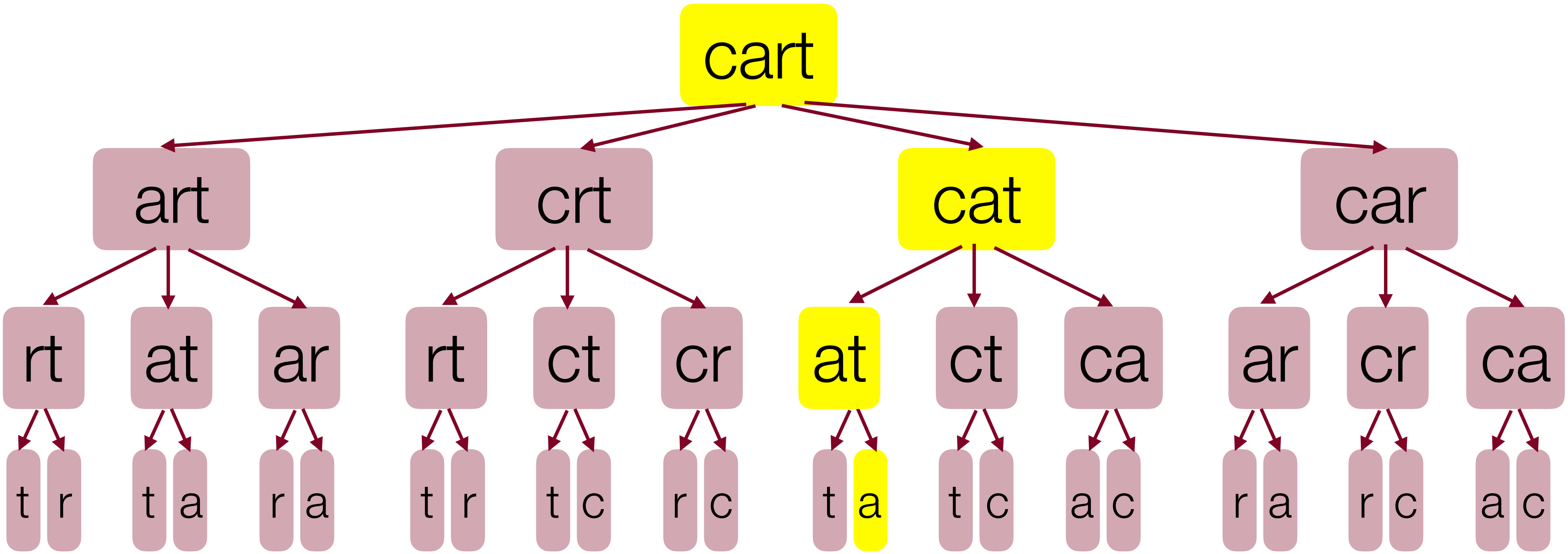
Reducability Decision Tree



Reducability Decision Tree



Reducability Decision Tree



Decision Tree Search Template

```
bool search(currentState) {
    if (isSolution(currentState)) {
        return true;
    } else {
        for (option : moves from currentState) {
            nextState = takeOption(curr, option);
            if (search(nextState)) {
                return true;
            }
        }
        return false;
    }
}
```

Reducible Word

Let's define a reducible 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:
  - The empty string

- Recursive Step:
  - Any multi-letter word is reducible if you can remove a letter (legal move) to form a shrinkable word.

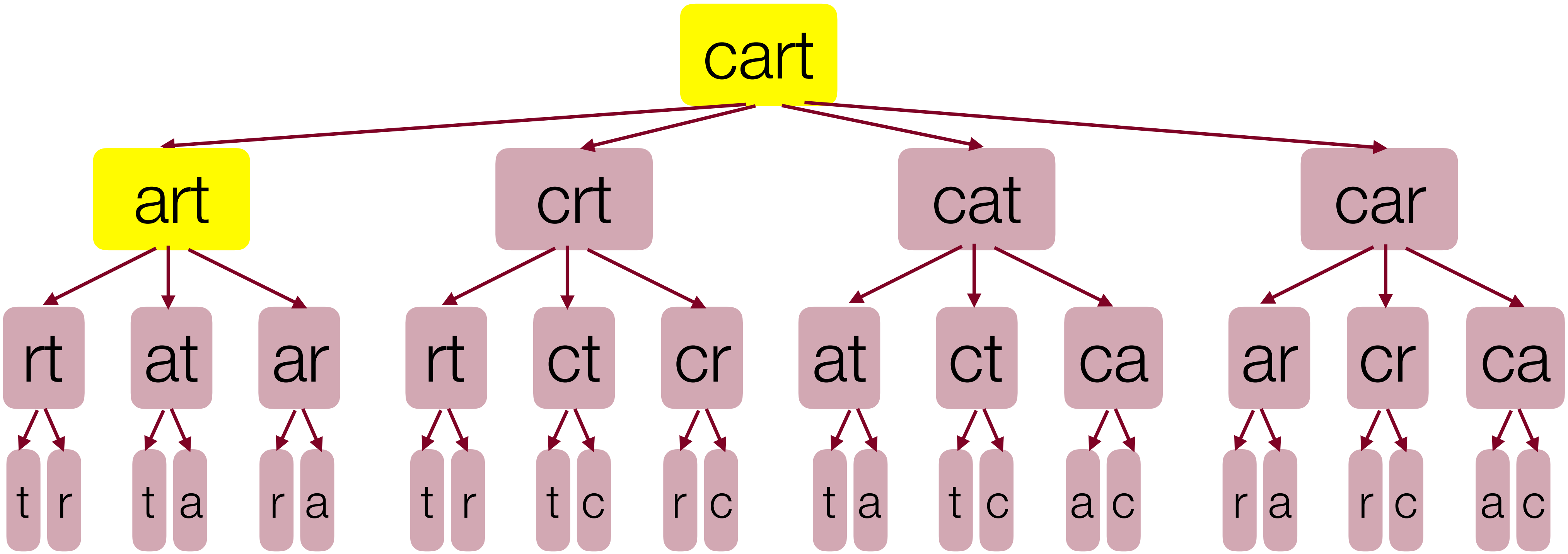
Reducible Word

Let's define a reducible 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:
  - The empty string

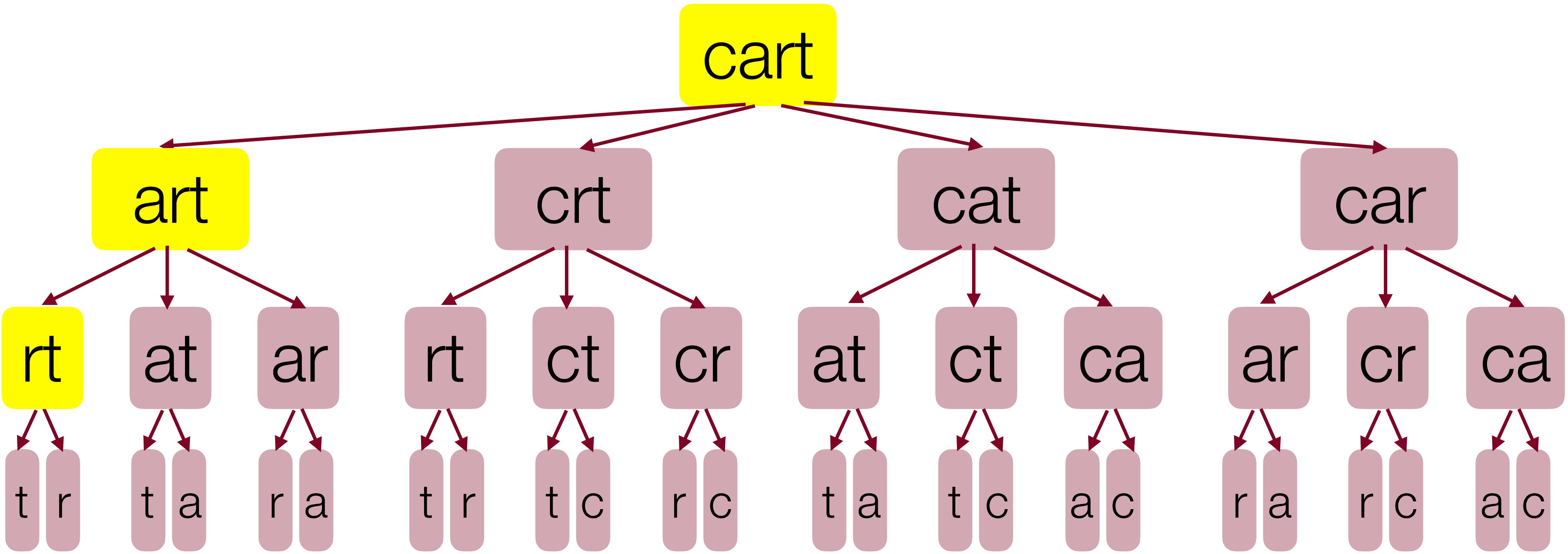
- Recursive Step:
  - Any multi-letter word is reducible if you can remove a letter (legal move) to form a shrinkable word.

How the algorithm works



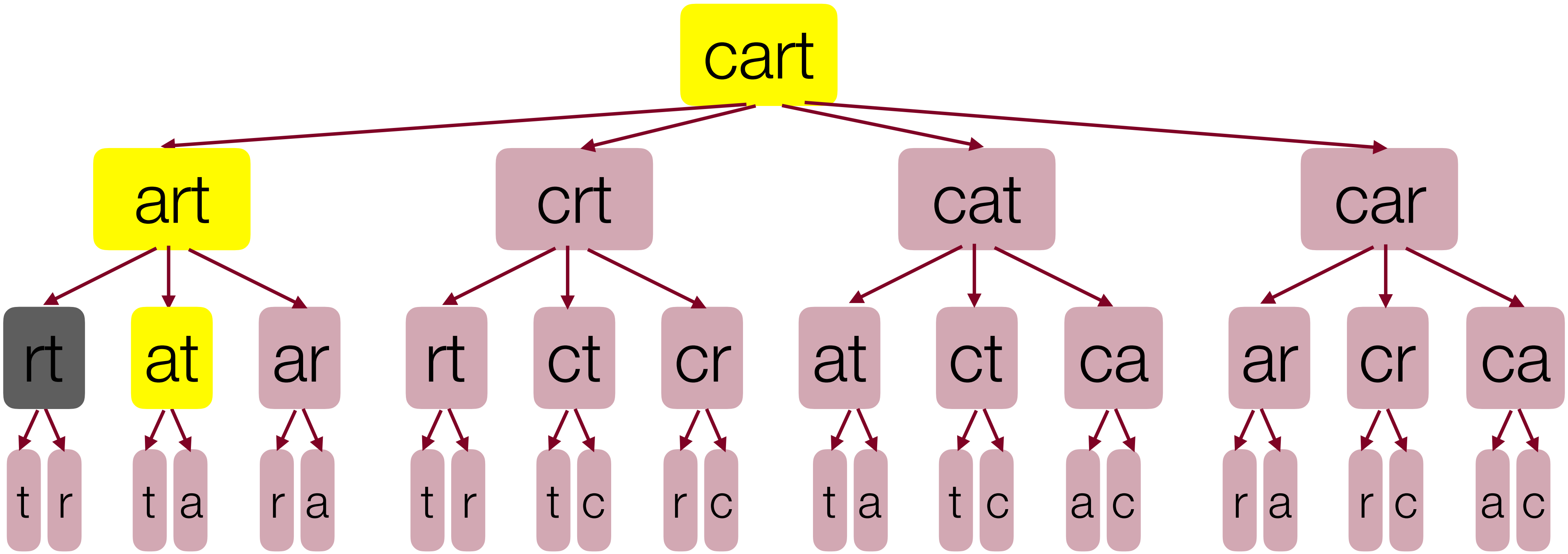
art: is a word

How the algorithm works



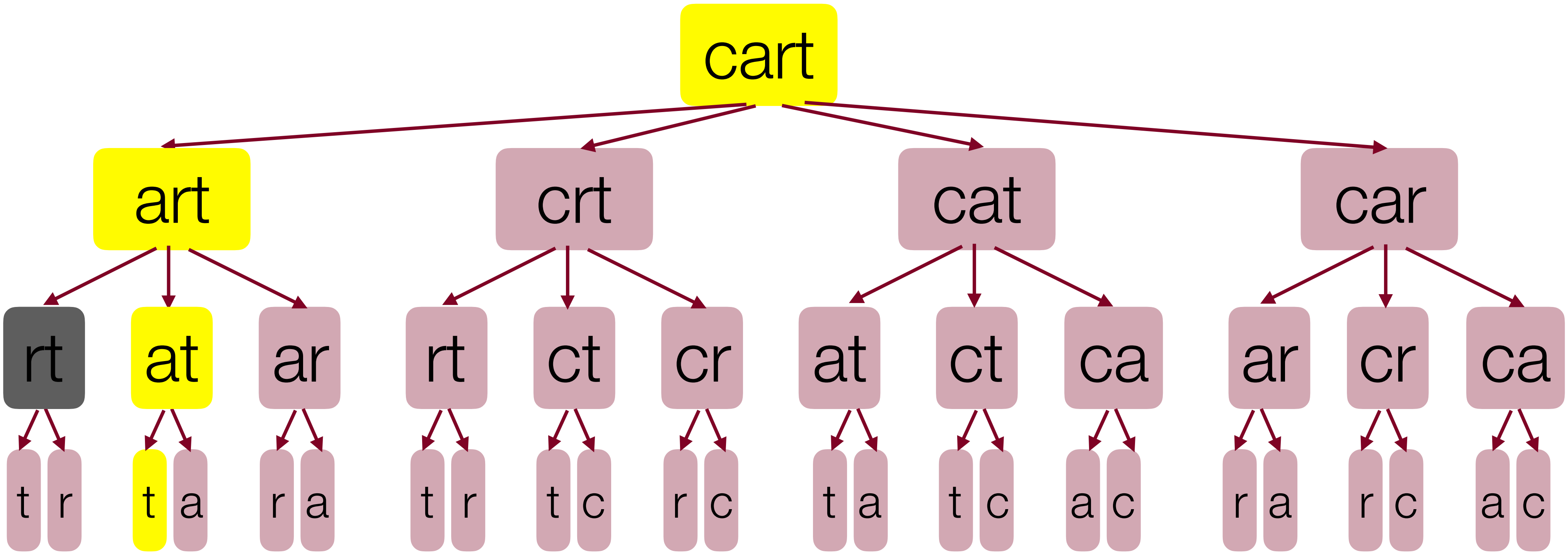
rt: not a word

How the algorithm works



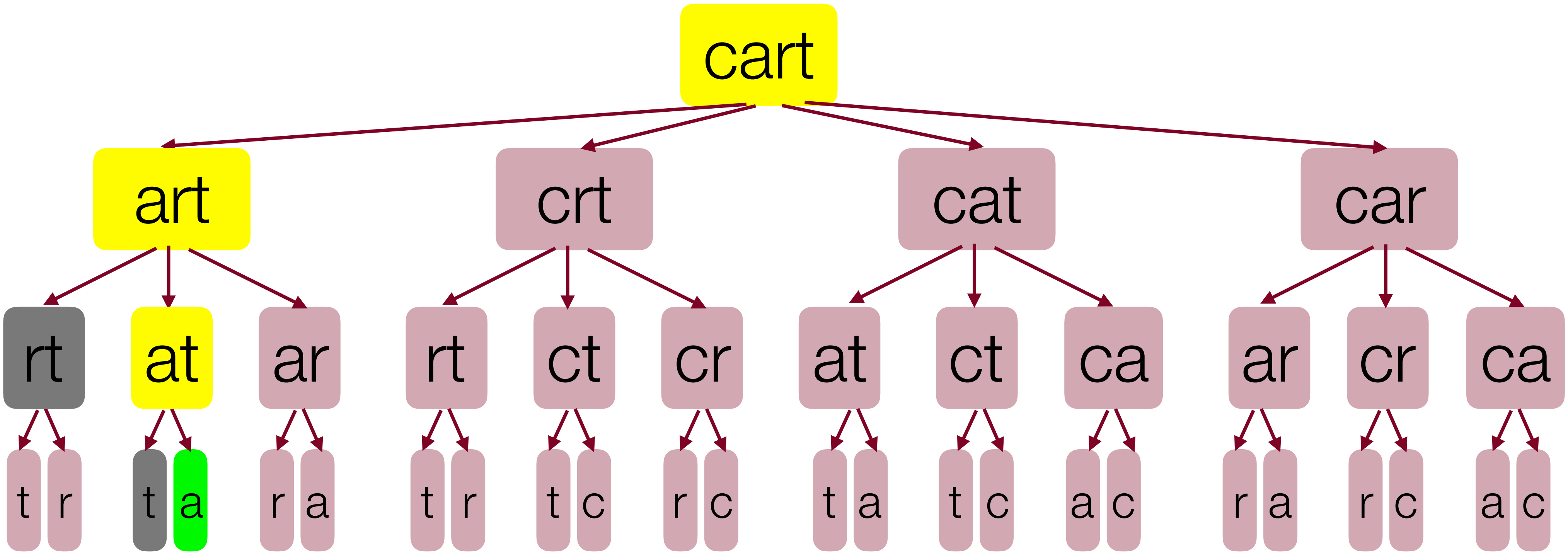
at: is a word

How the algorithm works



t: not a word

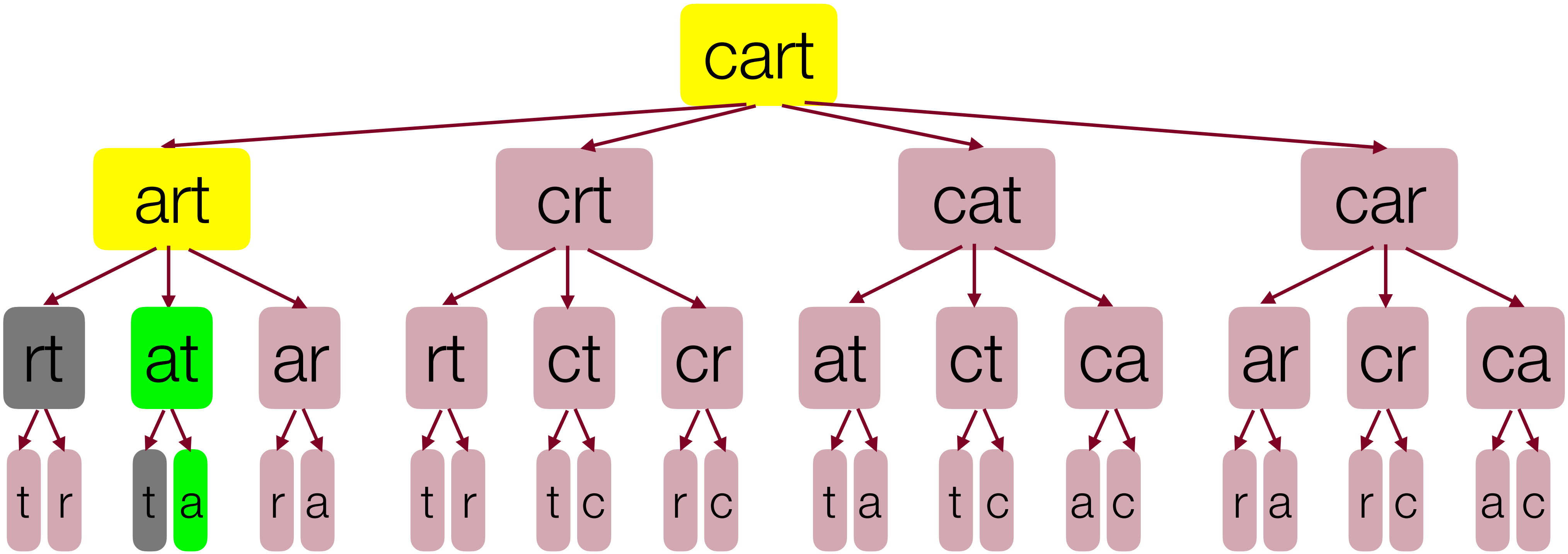
How the algorithm works



a: is a word

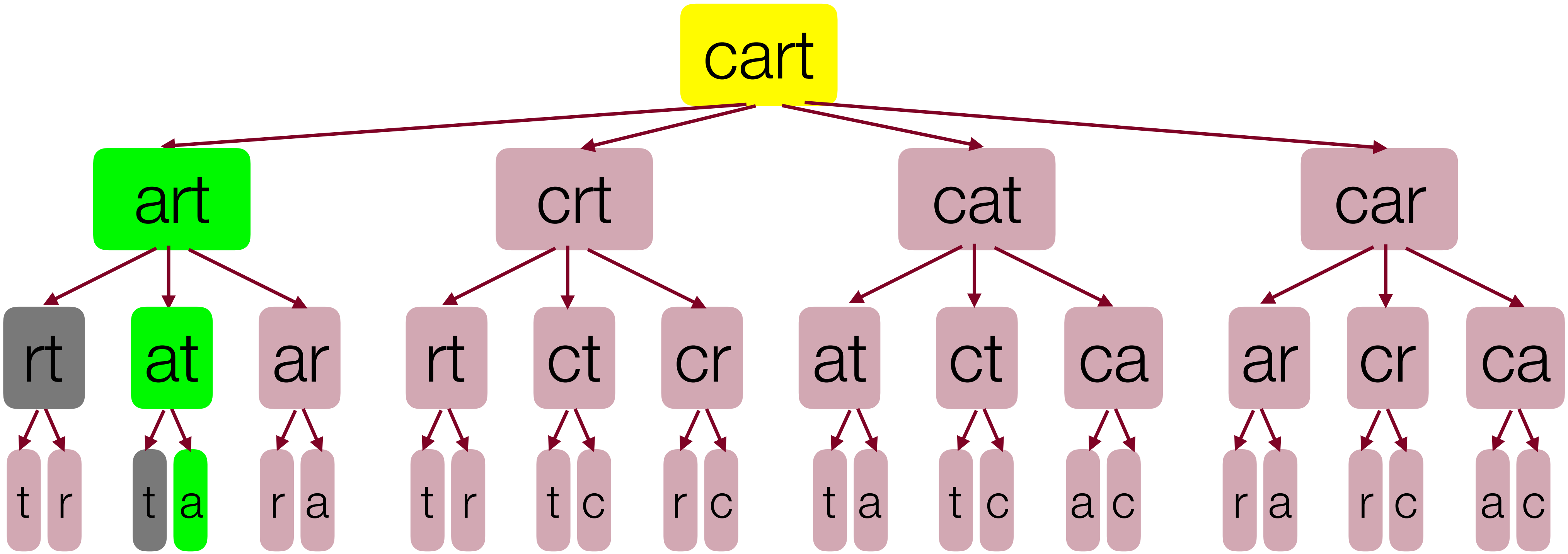
there is a solution!

How the algorithm works



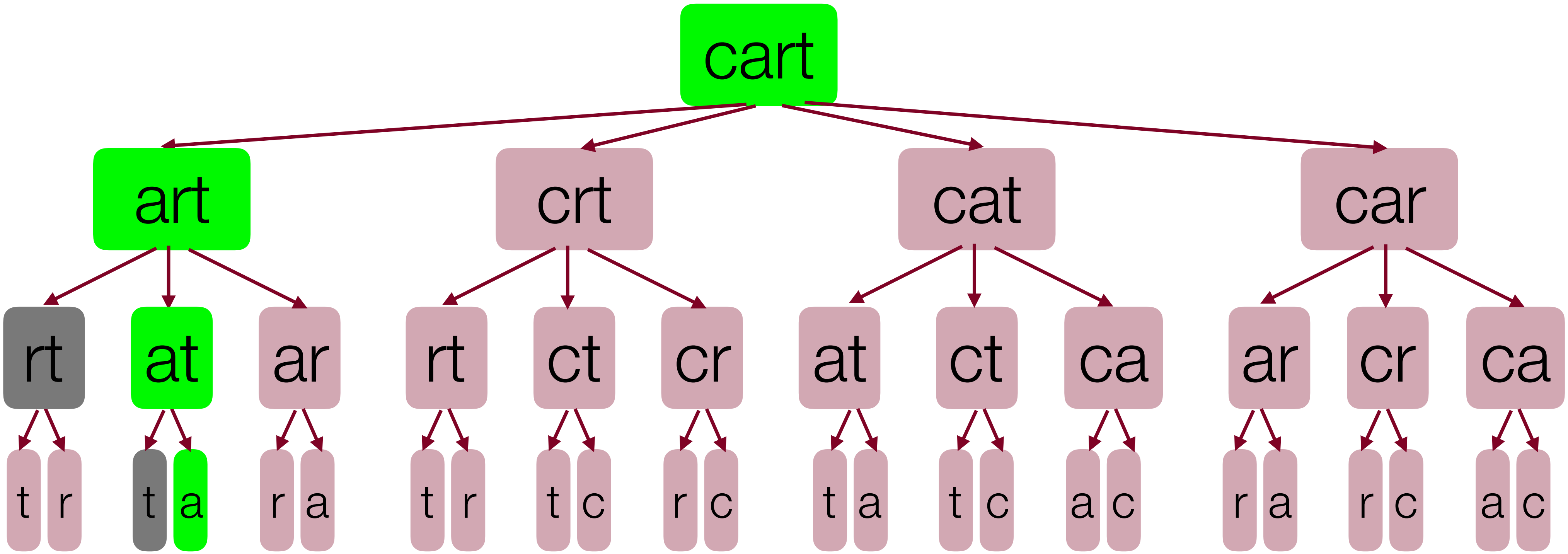
a: is a word
there is a solution!

How the algorithm works



a: is a word
there is a solution!

How the algorithm works



a: is a word
there is a solution!

Reducible Word

Is there really just one nine-letter word?

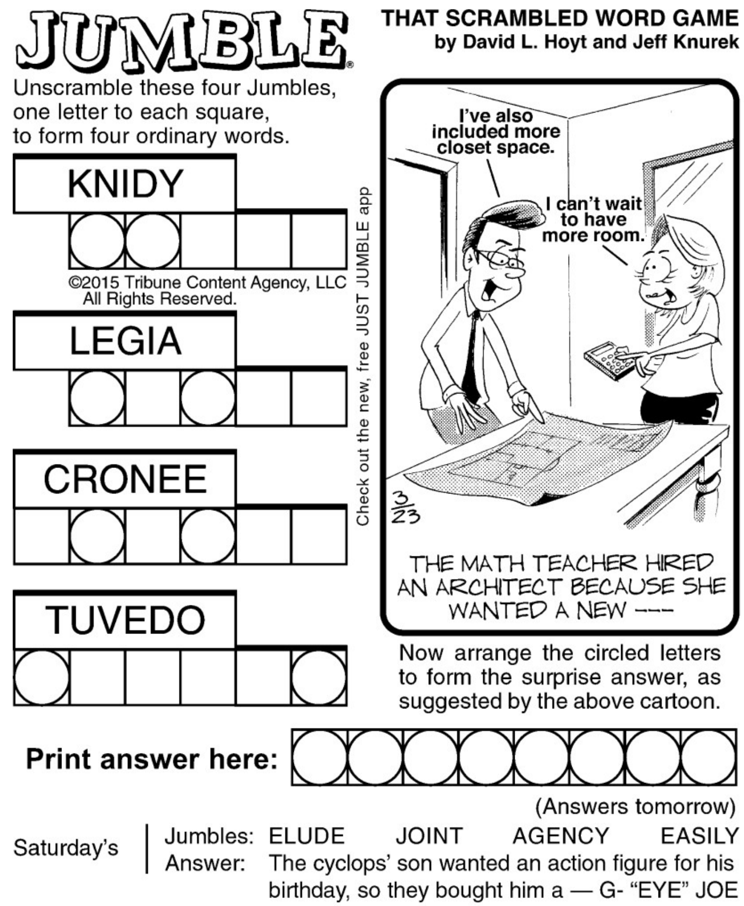
Recursive Backtracking: Templates

There are basically five different problems you might see that will require recursive backtracking:

- Determine whether a solution exists
- Find a solution
- Find the best solution
- Count the number of solutions
- Print/find all the solutions

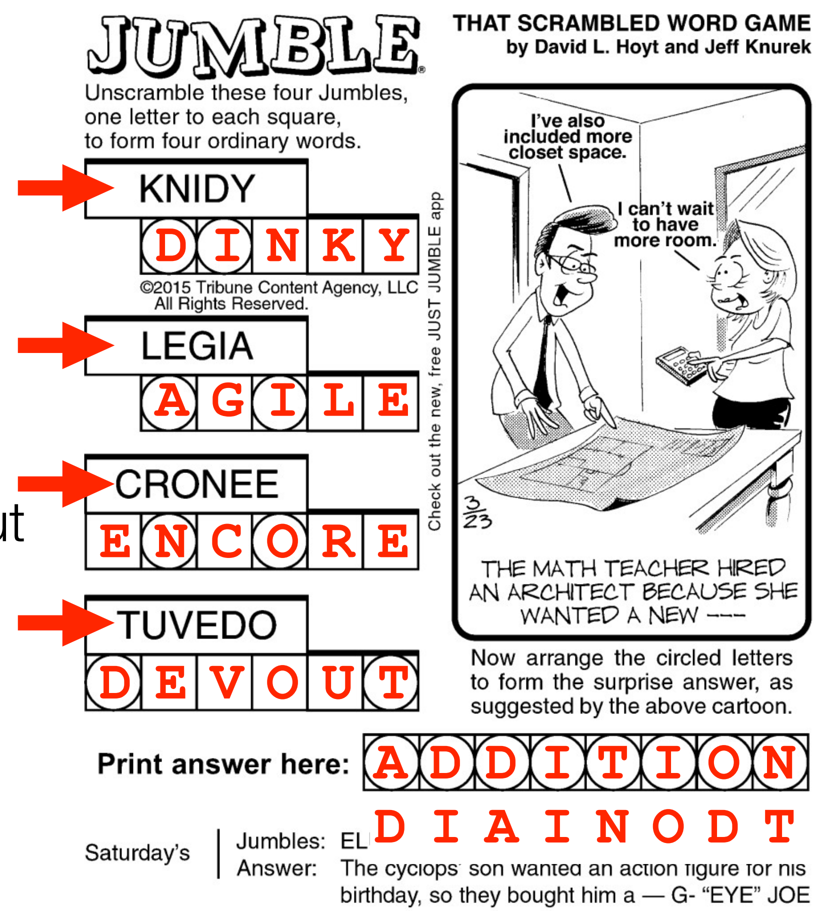
Jumble

- Since 1954, the *JUMBLE* has been a staple in newspapers.
- The basic idea is to unscramble the anagrams for the words on the left, and then use the letters in the circles as another anagram to unscramble to answer the pun in the comic.
- As a kid, I played the puzzle every day, but some days I just couldn't descramble the words. Six letter words have 6! == 720 combinations, which can be tricky!
- I figured I would write a computer program to print out all the permutations!



Jumble

- Since 1954, the *JUMBLE* has been a staple in newspapers.
- The basic idea is to unscramble the anagrams for the words on the left, and then use the letters in the circles as another anagram to unscramble to answer the pun in the comic.
- As a kid, I played the puzzle every day, but some days I just couldn't descramble the words. Six letter words have 6! == 720 combinations, which can be tricky!
- I figured I would write a computer program to print out all the permutations!



Permutations

My original function to print out all permutations of four letters:

```
void permute4(string s) {
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4 ; j++) {
            if (j == i) {
                continue; // ignore
            }
            for (int k = 0; k < 4; k++) {
                if (k == j || k == i) {
                    continue; // ignore
                }
                for (int w = 0; w < 4; w++) {
                    if (w == k || w == j || w == i) {
                        continue; // ignore
                    }
                    cout << s[i] << s[j] << s[k] << s[w] << endl;
                }
            }
        }
    }
}
```

Permutations

I also had a permute5() function…

```
void permute5(string s) {
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5 ; j++) {
            if (j == i) {
                continue; // ignore
            }
            for (int k = 0; k < 5; k++) {
                if (k == j || k == i) {
                    continue; // ignore
                }
                for (int w = 0; w < 5; w++) {
                    if (w == k || w == j || w == i) {
                        continue; // ignore
                    }
                    for (int x = 0; x < 5; x++) {
                        if (x == k || x == j || x == i || x == w) {
                            continue;
                        }
                        cout << "  " << s[i] << s[j] << s[k] << s[w] << s[x] << endl;
                    }
                }
            }
        }
    }
}
```

Permutations

And a permute6() function…

```
void permute6(string s) {
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5 ; j++) {
            if (j == i) {
                continue; // ignore
            }
            for (int k = 0; k < 5; k++) {
                if (k == j || k == i) {
                    continue; // ignore
                }
                for (int w = 0; w < 5; w++) {
                    if (w == k || w == j || w == i) {
                        continue; // ignore
                    }
                    for (int x = 0; x < 5; x++) {
                        if (x == k || x == j || x == i || x == w) {
                            continue;
                        }
                        for (int y = 0; y < 6; y++) {
                            if (y == k || y == j || y == i || y == w || y == x) {
                                continue;
                            }
                            cout << "  " << s[i] << s[j] << s[k] << s[w] << s[x] << s[y] << endl;
                        }
                    }
                }
            }
        }
    }
}
```

What has been seen
cannot be un-seen

This is not tenable!

Tree Framework — Permutations

- Permutations do not lend themselves well to iterative looping because we are really *rearranging* the letters, which doesn't follow an iterative pattern.
- Instead, we can look at a recursive method to do the rearranging, called an *exhaustive algorithm*. We want to investigate all possible solutions. We don't need to know how many letters there are in advance!

- In pseudocode:

```
If you have no more characters left to rearrange, print current permutation
for (every possible choice among the characters left to rearrange) {
    Make a choice and add that character to the permutation so far
    Use recursion to rearrange the remaining letters
}
```

- In English:
  - The permutation starts with zero characters, as we have all the letters in the original string to arrange. The base case is that there are no more letters to arrange.
  - Take one letter from the letters left, add it to the current permutation, and recursively continue the process, decreasing the characters left by one.

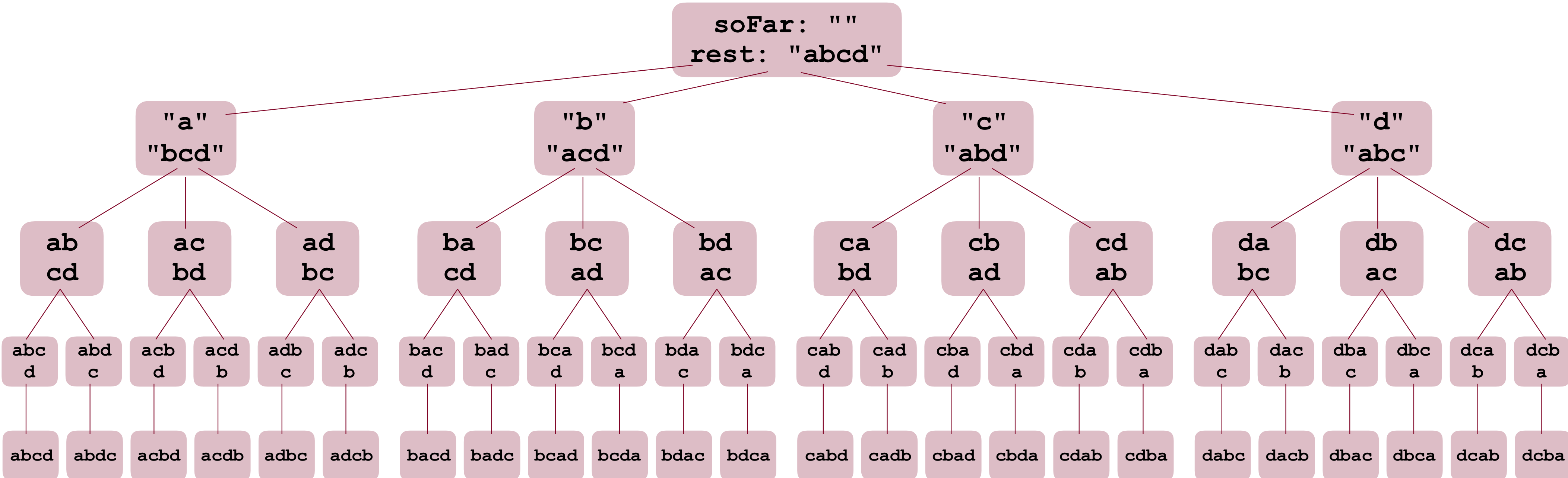
Tree Framework — Permutations

- The algorithm in C++:

```
void permute(string soFar, string rest) {
    if (rest == "") {
        cout << soFar << endl;
    } else {
        for (int i = 0; i < rest.length(); i++) {
            string remaining = rest.substr(0, i) + rest.substr(i+1);
            permute(soFar + rest[i], remaining);
        }
    }
}
```

- Example call:
  - **`recPermute("","abcd");`**

Tree Framework — Permutations



This is a tree!

- ✓ Exhaustive
- ✓ Works for any length string
- ✓ N! different results
- ✓ Can think of this as a "call tree" or a "decision tree"

Tree Framework — Helper functions

- Here is the algorithm again:

```
void permute(string soFar, string rest) {
    if (rest == "") {
        cout << soFar << endl;
    } else {
        for (int i = 0; i < rest.length(); i++) {
            string remaining = rest.substr(0, i) + rest.substr(i+1);
            permute(soFar + rest[i], remaining);
        }
    }
}
```

- Some might argue that this isn't a particularly good function, because it requires the user to always start the algorithm with the empty string for the **`soFar`** parameter. It's ugly, and it exposes our internal parameter.
- What we really want is a **`permute(string s)`** function that is cleaner.
- We can re-name the function above **`permuteHelper()`** (and change the inner call, as well!), and have a cleaner permute function that calls this one.

Tree Framework — Helper functions

- The cleaner interface:

```
void permuteHelper(string soFar, string rest) {
    if (rest == "") {
        cout << soFar << endl;
    } else {
        for (int i = 0; i < rest.length(); i++) {
            string remaining = rest.substr(0, i) + rest.substr(i+1);
            permuteHelper(soFar + rest[i], remaining);
        }
    }
}

void permute(string s) {
    permuteHelper("", s);
}
```

- Now, a user only has to call **`permute("tuvedo")`**, which hides the helper recursion parameter.

References and Advanced Reading

- **References:**
  - Understanding permutations: http://stackoverflow.com/questions/7537791/understanding-recursion-to-generate-permutations
  - Maze algorithms: https://en.wikipedia.org/wiki/Maze_solving_algorithm

- **Advanced Reading:**
  - Exhaustive recursive backtracking: https://see.stanford.edu/materials/icspacs106b/h19-recbacktrackexamples.pdf
  - Backtracking: https://en.wikipedia.org/wiki/Backtracking

Extra Slides