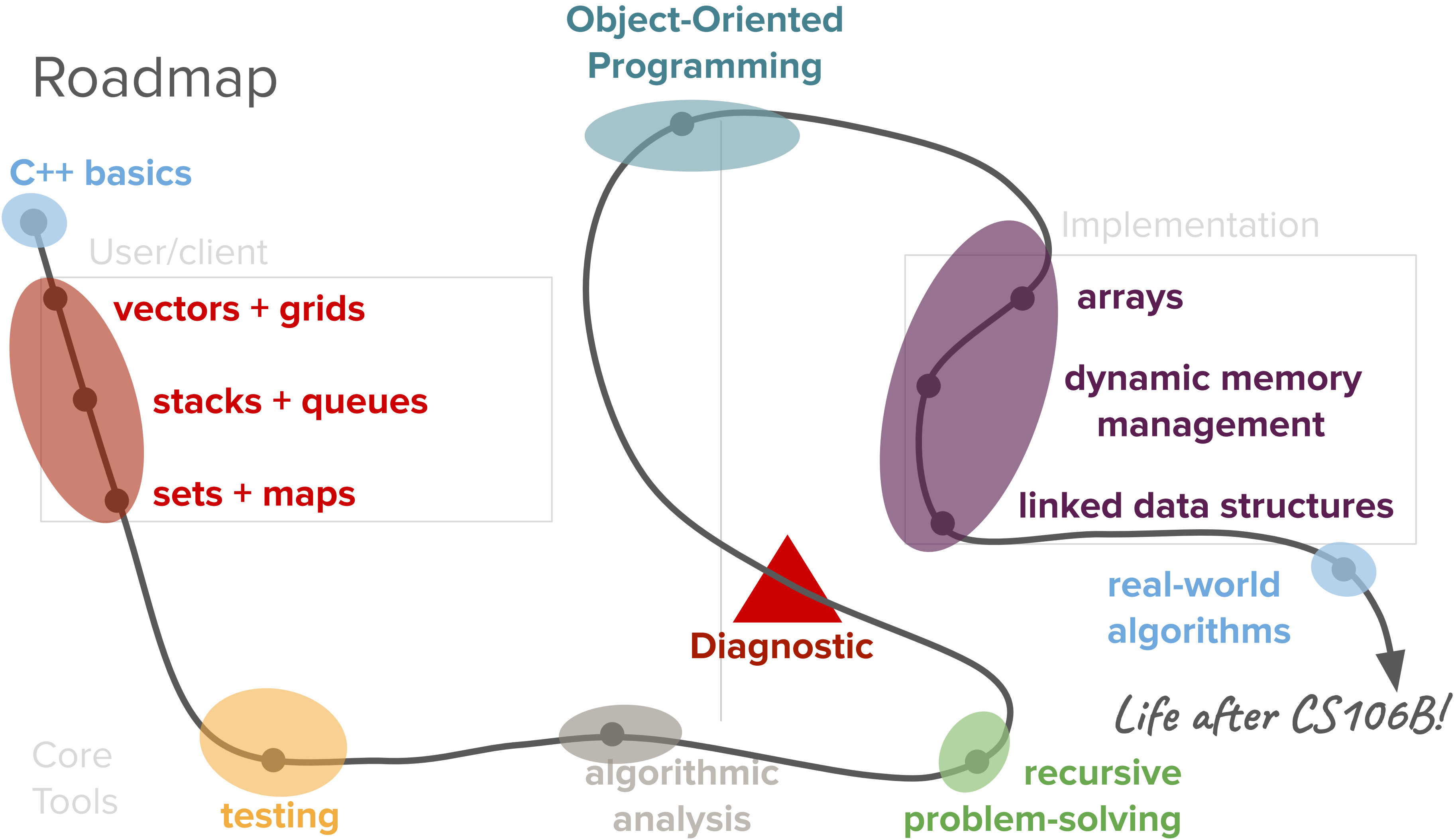
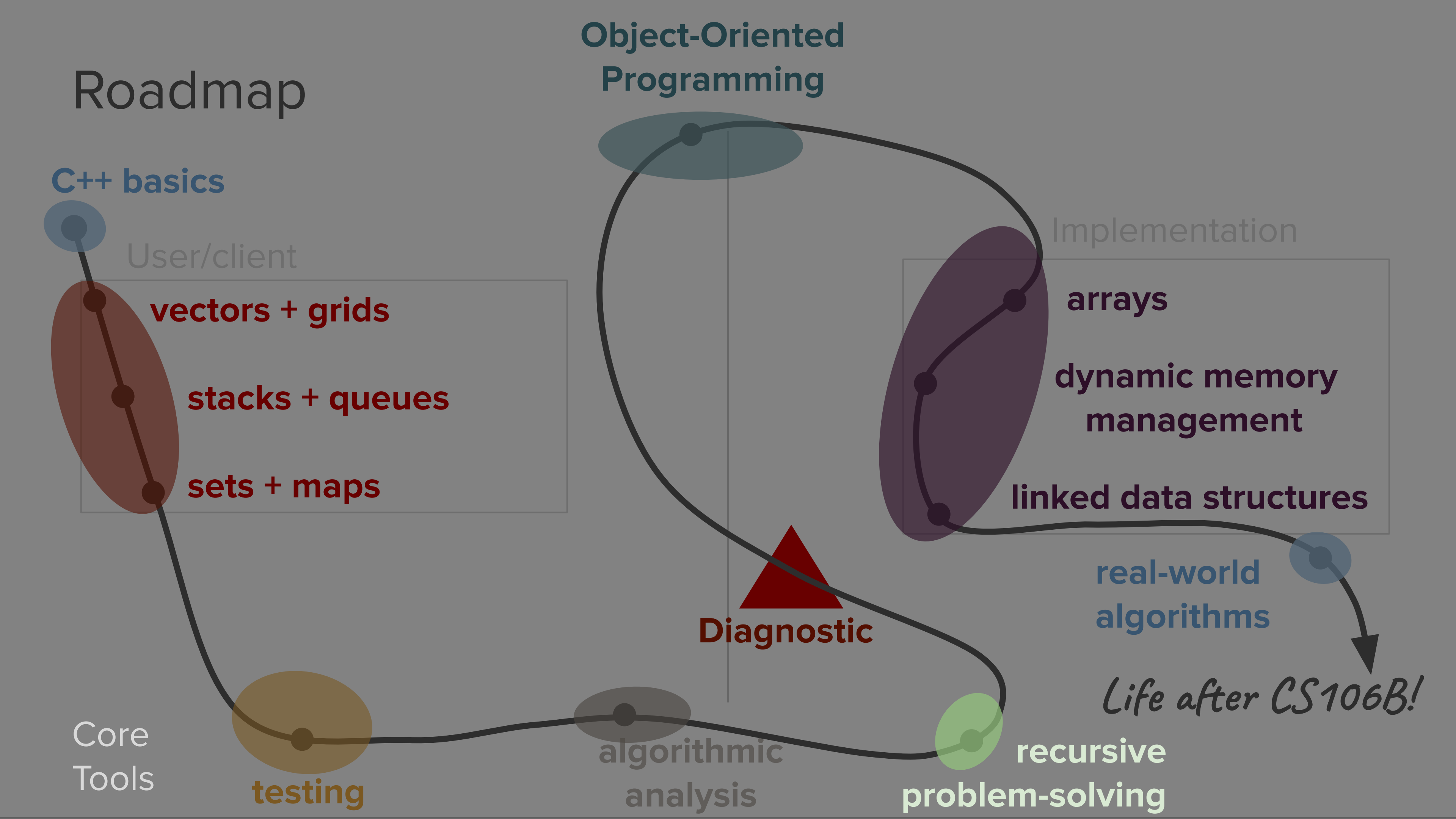
Recursive Backtracking and Enumeration

What is an example of a game that would be easy to play if you had the ability to quickly think of all possible moves/plays?

(put your answers the chat)

Roadmap


Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving


Roadmap
Object-Oriented
Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory
management
linked data structures
real-world
algorithms
Diagnostic
Life after CS106B!
Core
Tools
testing
algorithmic
analysis
recursive
problem-solving

Today's question

How can we leverage backtracking recursion to solve interesting problems?

Today's topics

1. Review
2. Word Scramble
3. Shrinkable Words
4. Generating Subsets

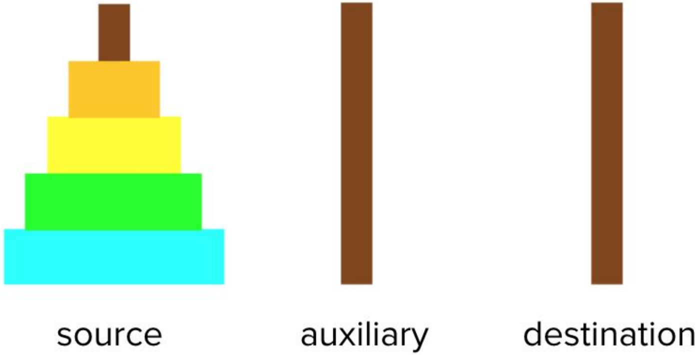
Review

(advanced recursion patterns)

Why do we use recursion?

- Elegance
  - Allows us to solve problems with very clean and concise code
- Efficiency
  - Allows us to accomplish better runtimes when solving problems
- Dynamic
  - Allows us to solve problems that are hard to solve iteratively

Elegance (Towers of Hanoi)




```
void findSolution(int n, char source, char dest,
char aux) {
    if (n == 1) {
        moveSingleDisk(source, dest);
    } else {
        findSolution(n - 1, source, aux, dest);
        moveSingleDisk(source, dest);
        findSolution(n - 1, aux, dest, source);
    }
}
```

```
void findSolutionIterative(int n, char source, char dest, char aux) {
   int numMoves = pow(2, n) - 1; // total number of moves necessary

   // if number of disks is even, swap dest and aux posts
   if (n % 2 == 0) {
       char temp = dest;
       dest = aux;
       aux = temp;
   }

   Stack<int> srcStack;
   for (int i = n; i > 0; i--) {
       srcStack.push(i);
   }
   cout << srcStack << endl;
   Stack<int> destStack;
   Stack<int> auxStack;

   // Determine next move based on how many moves have been made so far
   for (int i = 1; i <= numMoves; i++) {
       switch (i % 3) {
           case 1:
               if (srcStack.isEmpty() ||
                       (!destStack.isEmpty() && srcStack.peek() > destStack.peek())) {
                   srcStack.push(destStack.pop());
                   moveSingleDisk(dest, source);
               } else {
                   destStack.push(srcStack.pop());
                   moveSingleDisk(source, dest);
               }
               break;
           case 2:
               if (srcStack.isEmpty() ||
                       (!auxStack.isEmpty() && srcStack.peek() > auxStack.peek())) {
                   srcStack.push(auxStack.pop());
                   moveSingleDisk(aux, source);
               } else {
                   auxStack.push(srcStack.pop());
                   moveSingleDisk(source, aux);
               }
               break;
           case 0:
               if (destStack.isEmpty() ||
                       (!auxStack.isEmpty() && destStack.peek() > auxStack.peek())) {
                   destStack.push(auxStack.pop());
                   moveSingleDisk(aux, dest);
               } else {
                   auxStack.push(destStack.pop());
                   moveSingleDisk(dest, aux);
               }
               break;
       }
   }
}
```

Efficiency (Binary Search)

- Leverage the structure in sorted data to **eliminate half of the search space every time** when searching for an element
  - Only do a direct comparison with the middle element in the list
  - Recursively search the left half if the element is less than the middle
  - Recursively search the right half if the element is greater than the middle

- Binary search has logarithmic Big-O: `O(log N)`
  - Enables efficient performance of sets and maps

Binary Search

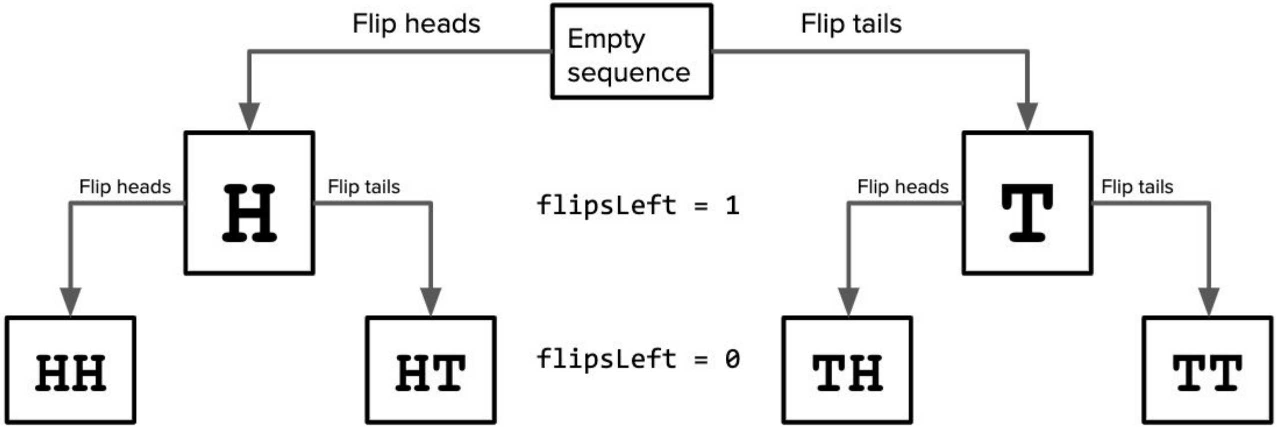
Input Size	Runtime (s)
1000000	0.064
2000000	0.072
4000000	0.082
8000000	0.097
16000000	0.111
32000000	0.121
64000000	0.14

Linear Search

Input Size	Runtime (s)
10000	0.096
20000	0.189
40000	0.368
8000000	0.767
160000	1.387
320000	2.746
640000	6.154

Dynamic (Coin Sequences + Decision Trees)

- The **height** of the tree corresponds to the **number of decisions** we have to make. The **width** at each decision point corresponds to the **number of options at each decision**.
- To exhaustively explore the entire search space, we must **try every possible option for every possible decision**.



Two types of recursion

Basic recursion

- One repeated task that builds up a solution as you come back up the call stack
- The final base case defines the initial seed of the solution and each call contributes a little bit to the solution
- Initial call to recursive function produces final solution

Backtracking recursion

- Build up many possible solutions through multiple recursive calls at each step
- Seed the initial recursive call with an “empty” solution
- At each base case, you have a potential solution

How can we leverage backtracking recursion to solve interesting problems?

Using backtracking recursion

- There are 3 main categories of problems that we can solve by using backtracking recursion:
  - We can generate all possible solutions to a problem or count the total number of possible solutions to a problem
  - We can find one specific solution to a problem or prove that one exists
  - We can find the best possible solution to a given problem

Using backtracking recursion

- There are 3 main categories of problems that we can solve by using backtracking recursion:
  - We can generate all possible solutions to a problem or count the total number of possible solutions to a problem
  - We can find one specific solution to a problem or prove that one exists
  - We can find the best possible solution to a given problem
- There are many, many examples of specific problems that we can solve, including
  - Generating permutations
  - Generating subsets
  - Generating combinations
  - And many, many more

Using backtracking recursion

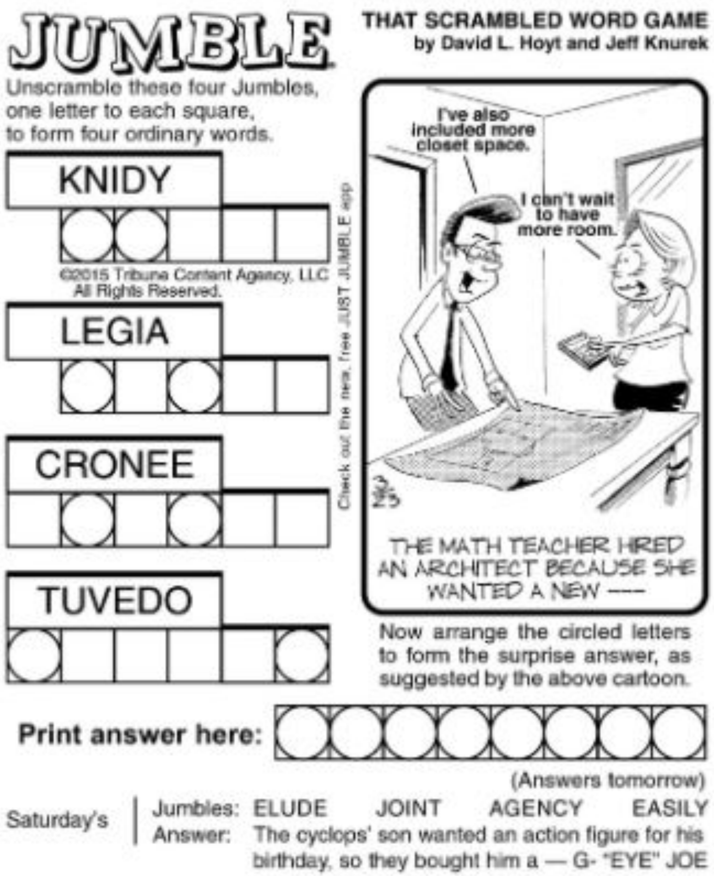
- There are 3 main categories of problems that we can solve by using backtracking recursion:
  - **We can generate all possible solutions to a problem or count the total number of possible solutions to a problem**
  - **We can find one specific solution to a problem or prove that one exists**
  - We can find the best possible solution to a given problem
- There are many, many examples of specific problems that we can solve, including
  - **Generating permutations**
  - **Generating subsets**
  - Generating combinations
  - And many, many more

Word Scramble

Jumble

- Since 1954, the JUMBLE word puzzle has been a staple in newspapers.

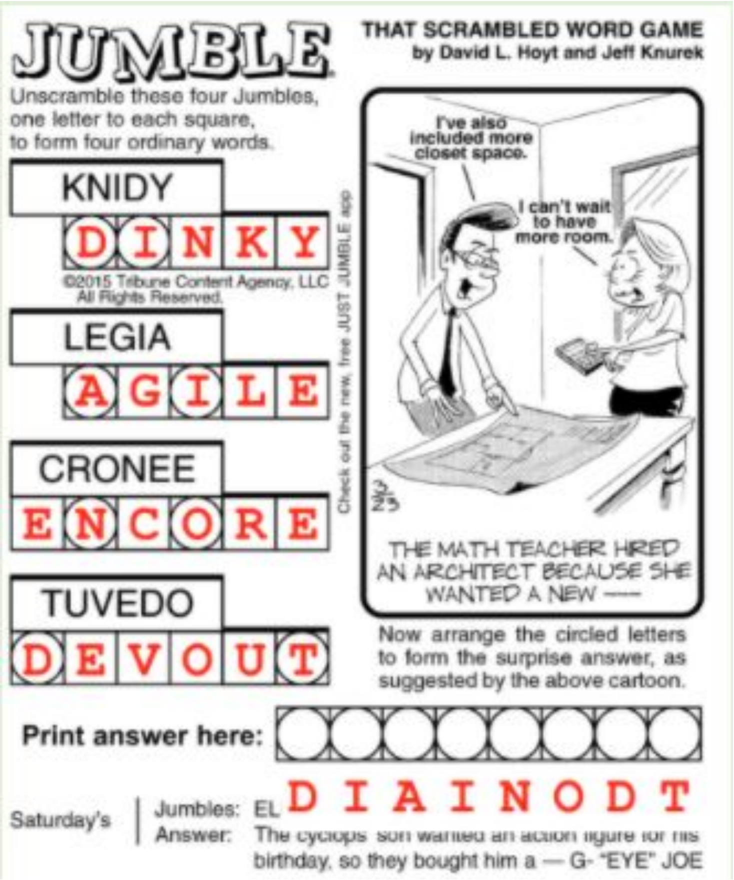
- The basic idea is to unscramble the provided letters to make the words on the left, and then use the letters in the circles as another set of letters to unscramble to answer the pun in the comic.



Jumble

- Since 1954, the JUMBLE word puzzle has been a staple in newspapers.

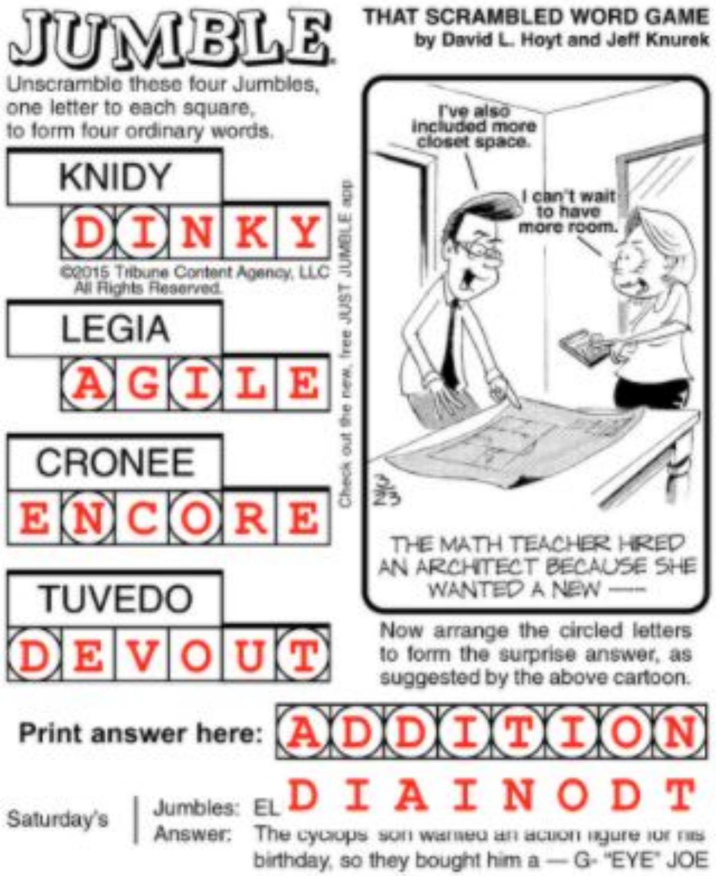
- The basic idea is to unscramble the provided letters to make the words on the left, and then use the letters in the circles as another set of letters to unscramble to answer the pun in the comic.



Jumble

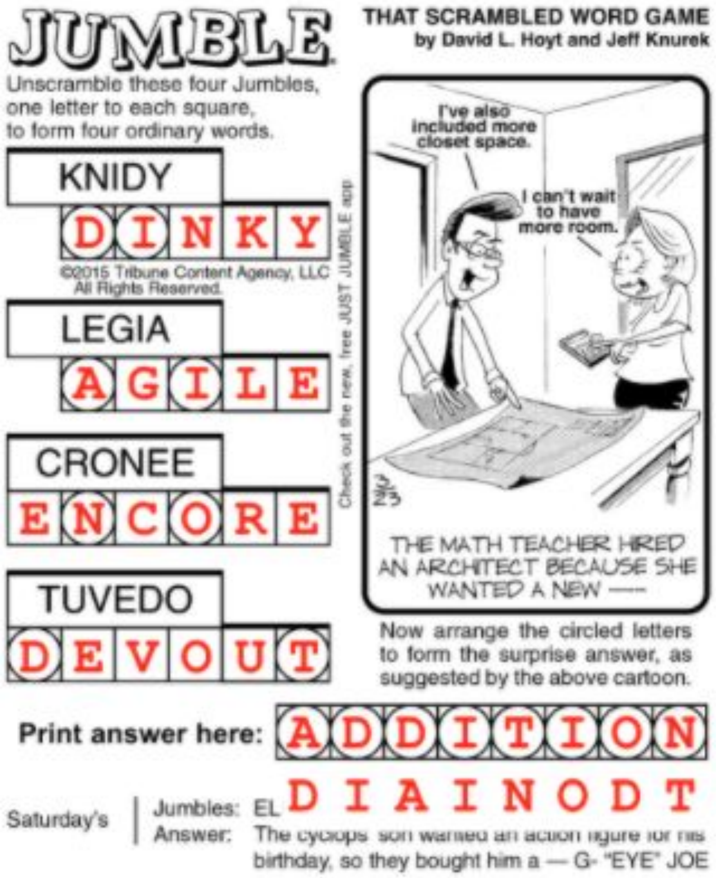
- Since 1954, the JUMBLE word puzzle has been a staple in newspapers.

- The basic idea is to unscramble the provided letters to make the words on the left, and then use the letters in the circles as another set of letters to unscramble to answer the pun in the comic.



Jumble

- For some people solving puzzles like this comes pretty easily, but this is actually a pretty challenging problem!
  - For a 6-letter word, there are 6! = 720 possible arrangements of the letters
- Can we write a program to print out all the combinations to help us solve this puzzle?



Permutations

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, permutations of the words in the motto "E Pluribus Unum" would be:
  - E Pluribus Unum
  - E Unum Pluribus
  - Pluribus E Unum
  - Pluribus Unum E
  - Unum E Pluribus
  - Unum Pluribus E

Permutations

- A **permutation** of a sequence is a sequence with the same elements, though possibly in a different order.
- We can think of permutations as an extension of the coin flip sequences we generated yesterday.
    - Rather than having 2 fixed options (heads and tails), the components of our original sequence define the options we can use to build our new sequence.

Discuss in breakouts:

What are the possible permutations of the string "saki"?

What potential recursive insights about generating permutations can you glean from this example?

[Time-permitting] Can you come up with a base case and recursive case for generating permutations?

Common question from lecture yesterday

- Can you solve all backtracking recursion problems with equivalent iterative solutions?
- Answer:

Common question from lecture yesterday

- Can you solve all backtracking recursion problems with equivalent iterative solutions?
- Answer:

```
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;
              }
          }
      }
  }
}
```

Common question from lecture yesterday

- Can you solve a … valent iterative solutions?
- Answer:

```
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;
                  }
              }
          }
      }
  }
}
```

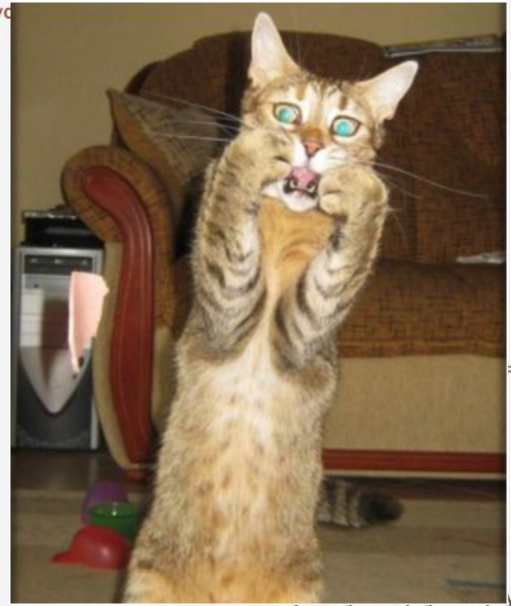
Common c

- Can you sc ... t iterative solutions?
- Answer:

```
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;
                      }
                  }
              }
          }
      }
  }
}
```

Common c

- Can you so t iterative solutions?
- Answer:



```
vo
                    = i) {

                      {
                      x == i || x == w) {

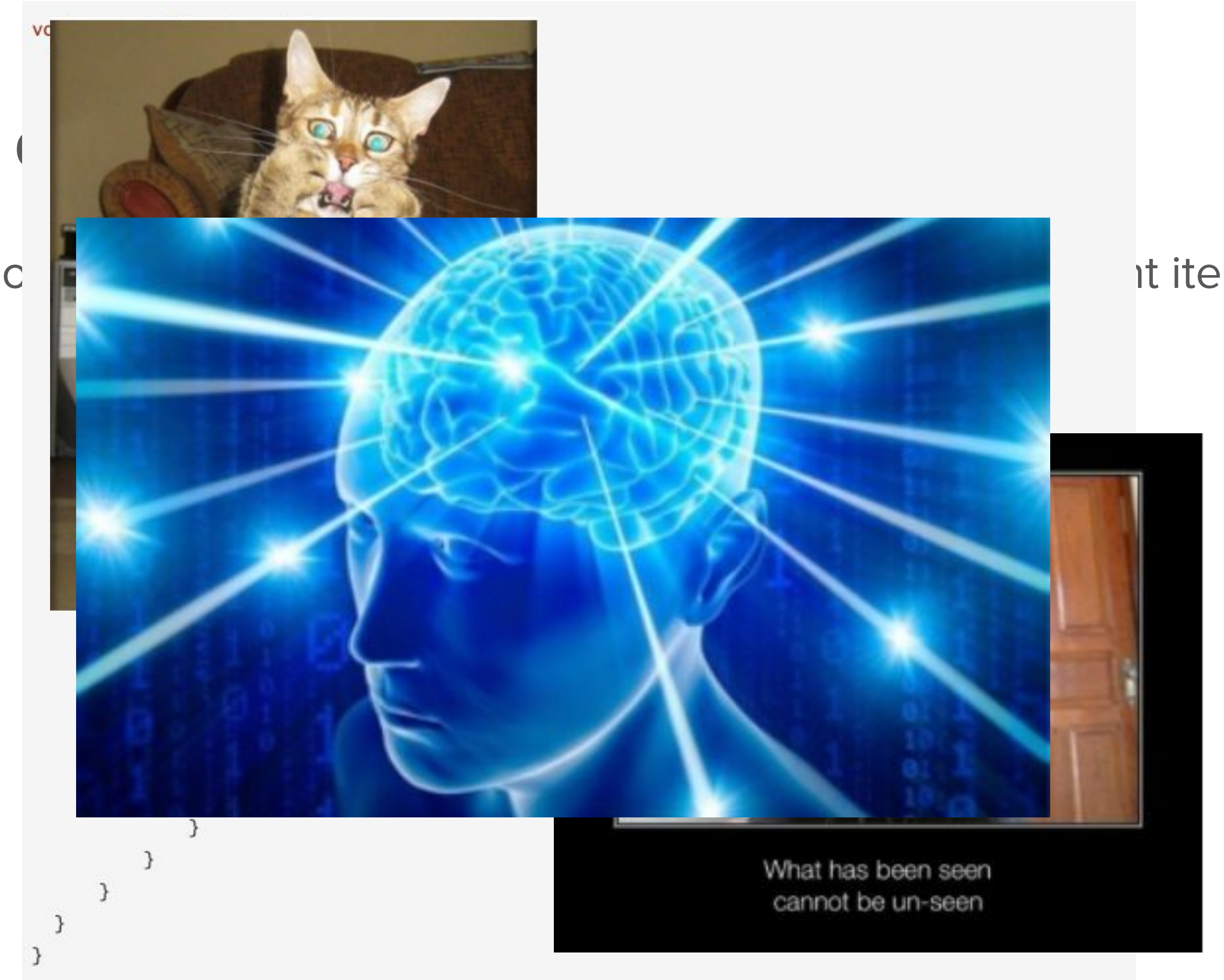
                      y++) {
                if (y == k || y == j || y == i || y == w || y == x) {
                    continue;
                }
                cout << "   " << s[i] << s[j] << s[k] << s[w] << s[x] << s[y] << endl;
            }
          }
        }
      }
    }
  }
}
```

Common

- Can you sc … t iterative solutions?
- Answer:

Common

- Can you sc t iterative solutions?
- Answer:



Permutations Intuition

What are all the permutations of the string "saki"?

- "saki"
- "saik"
- "skai"
- "skia"
- "sika"
- "siak"
- "aski"
- "asik"
- "aksi"
- "akis"
- "aisk"
- "aiks"
- "ksai"
- "ksia"
- "kasi"
- "kais"
- "kias"
- "kisa"
- "ikas"
- "iksa"
- "iaks"
- "lask"
- "iska"
- "isak"

Permutations Intuition

What are all the permutations of the string "saki"?

- **"saki"**
- **"saik"**
- **"skai"**
- **"skia"**
- **"sika"**
- **"siak"**
- "aski"
- "asik"
- "aksi"
- "akis"
- "aisk"
- "aiks"
- "ksai"
- "ksia"
- "kasi"
- "kais"
- "kias"
- "kisa"
- "ikas"
- "iksa"
- "iaks"
- "Iask"
- "iska"
- "isak"

A quarter of the permutations start with "s", followed by all the permutations of "aki"

Permutations Intuition

What are all the permutations of the string "saki"?

- "saki"
- "saik"
- "skai"
- "skia"
- "sika"
- "siak"
- **"aski"**
- **"asik"**
- **"aksi"**
- **"akis"**
- **"aisk"**
- **"aiks"**

A quarter of the permutations start with "a", followed by all the permutations of "ski"

- "ksai"
- "ksia"
- "kasi"
- "kais"
- "kias"
- "kisa"
- "ikas"
- "iksa"
- "iaks"
- "Iask"
- "iska"
- "isak"

Permutations Intuition

What are all the permutations of the string "saki"?

- "saki"
- "saik"
- "skai"
- "skia"
- "sika"
- "siak"
- "aski"
- "asik"
- "aksi"
- "akis"
- "aisk"
- "aiks"
- **"ksai"**
- **"ksia"**
- **"kasi"**
- **"kais"**
- **"kias"**
- **"kisa"**
- "ikas"
- "iksa"
- "iaks"
- "Iask"
- "iska"
- "isak"

A quarter of the permutations start with "k", followed by all the permutations of "sai"

Permutations Intuition

What are all the permutations of the string "saki"?

- "saki"
- "saik"
- "skai"
- "skia"
- "sika"
- "siak"
- "aski"
- "asik"
- "aksi"
- "akis"
- "aisk"
- "aiks"
- "ksai"
- "ksia"
- "kasi"
- "kais"
- "kias"
- "kisa"
- **"ikas"**
- **"iksa"**
- **"iaks"**
- **"iask"**
- **"iska"**
- **"isak"**

A quarter of the permutations start with "i", followed by all the permutations of "sak"

Permutations Intuition

What are all the permutations of the string "saki"?

- "saki"
- "saik"
- "skai"
- "skia"
- "sika"
- "siak"
- "aski"
- "asik"
- "aksi"
- "akis"
- "aisk"
- "aiks"
- "ksai"
- "ksia"
- "kasi"
- "kais"
- "kias"
- "kisa"
- "ikas"
- "iksa"
- "iaks"
- "iask"
- "iska"
- "isak"

Can we formalize this intuition in a decision tree?

What defines our permutations decision tree?

What defines our permutations decision tree?

- **Decision** at each step (each level of the tree):
    - What is the next letter that is going to get added to the permutation?

What defines our permutations decision tree?

- **Decision** at each step (each level of the tree):
  - What is the next letter that is going to get added to the permutation?

- **Options** at each decision (branches from each node):
  - One option for every remaining element that hasn't been selected yet
  - **Note: The number of options will be different at each level of the tree!**

What defines our permutations decision tree?

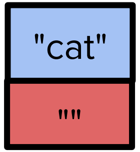
- **Decision** at each step (each level of the tree):
    - What is the next letter that is going to get added to the permutation?

- **Options** at each decision (branches from each node):
    - One option for every remaining element that hasn't been selected yet
    - **Note: The number of options will be different at each level of the tree!**

- Information we need to store along the way:
    - The permutation you’ve built so far
    - The remaining elements in the original sequence

Decisions yet to be made
Decisions made so far

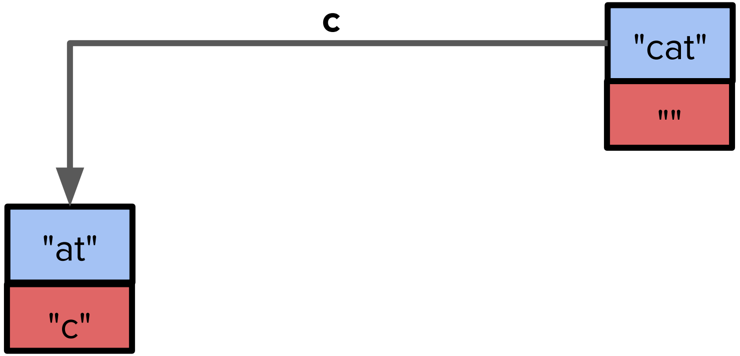
Decision tree: Find all permutations of "cat"



Decisions yet to be made

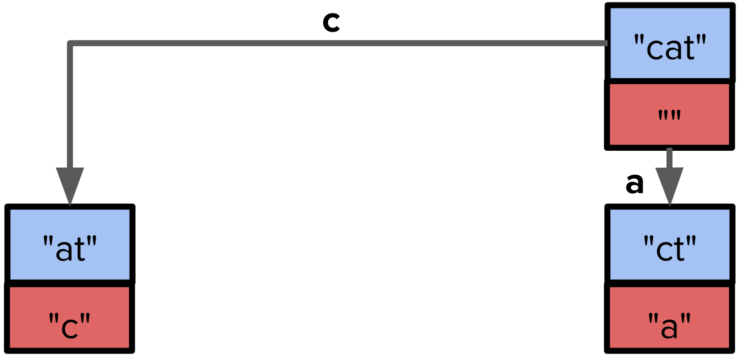
Decisions made so far

Decision tree: Find all permutations of "cat"



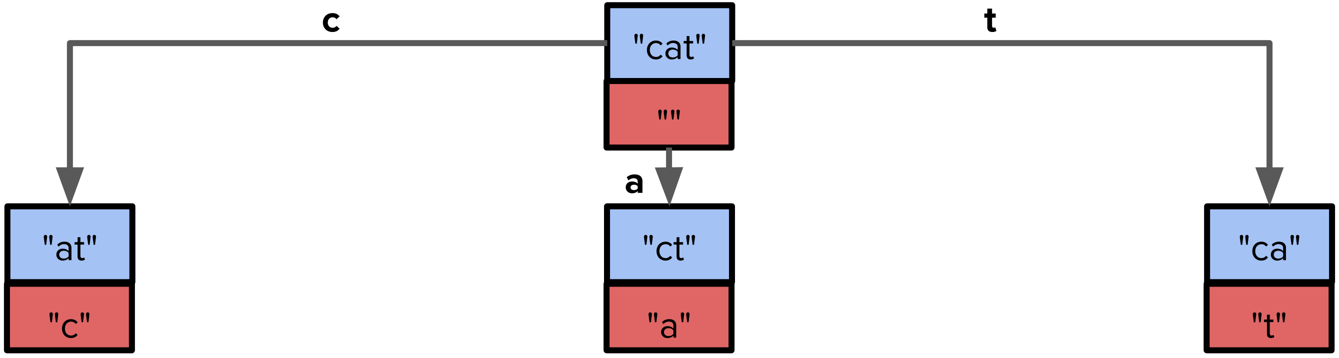
Decisions yet to be made
Decisions made so far

Decision tree: Find all permutations of "cat"



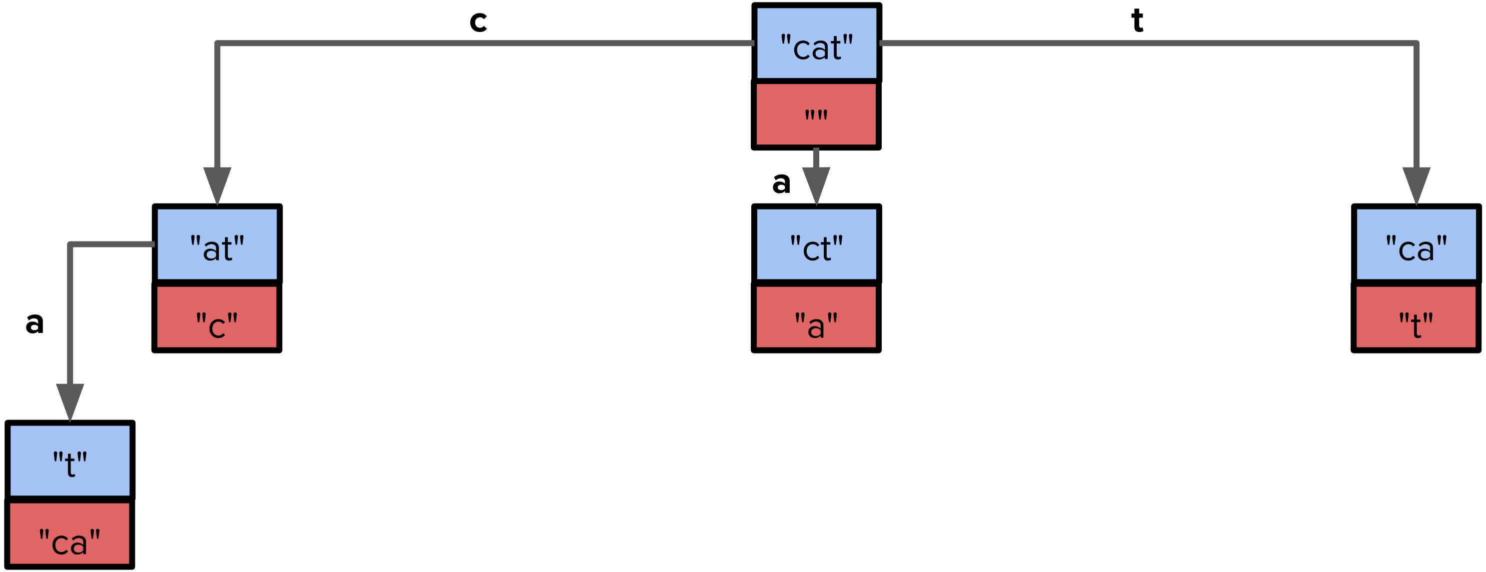
Decisions yet to be made
Decisions made so far

Decision tree: Find all permutations of "cat"



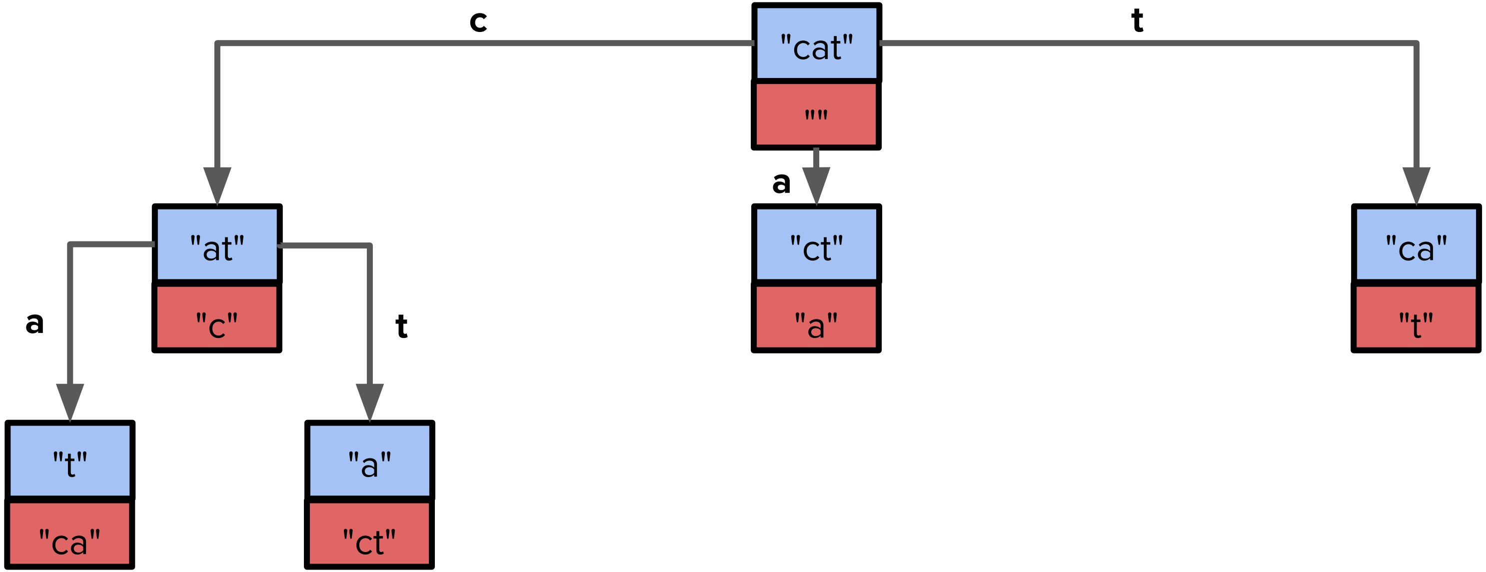
Decisions yet to be made
Decisions made so far

Decision tree: Find all permutations of "cat"



Decisions yet to be made
Decisions made so far

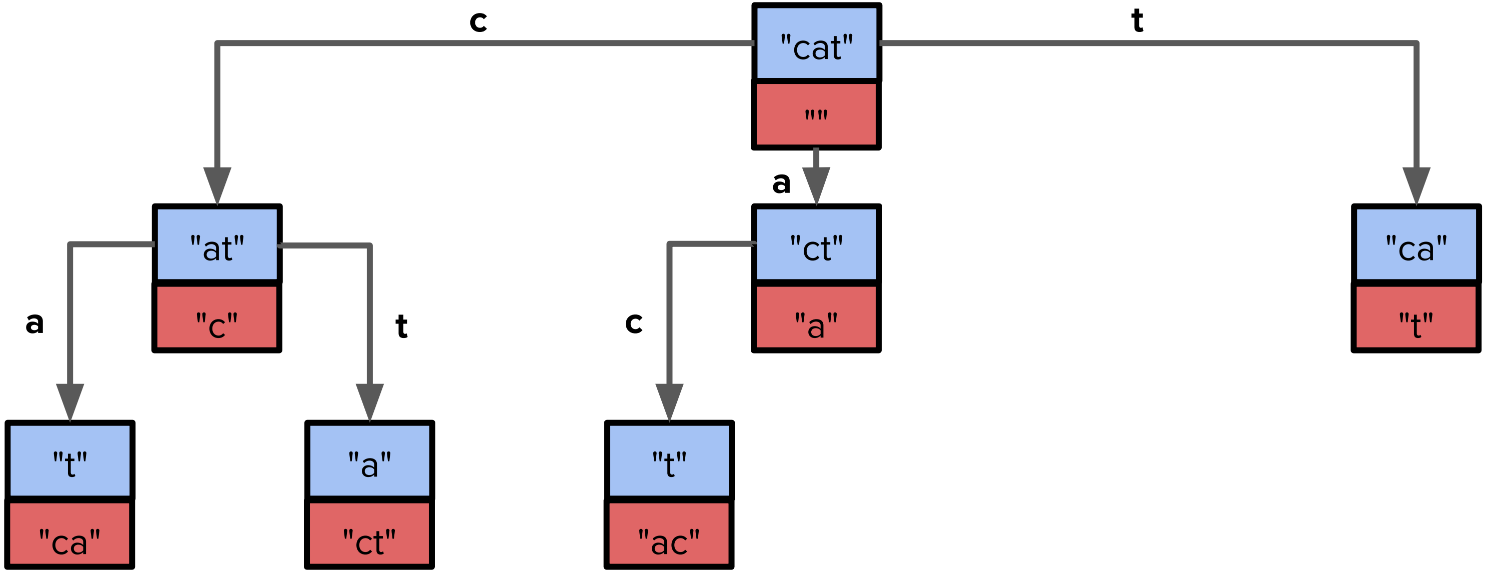
Decision tree: Find all permutations of "cat"



Decisions yet to be made

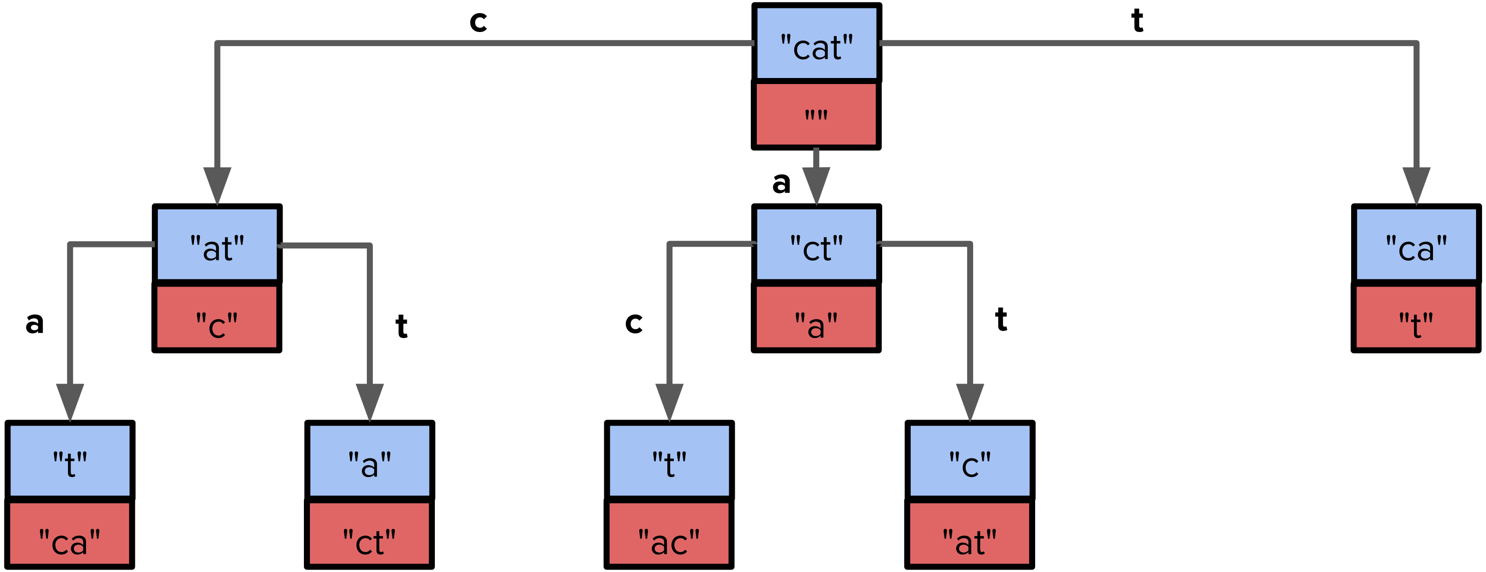
Decisions made so far

Decision tree: Find all permutations of "cat"



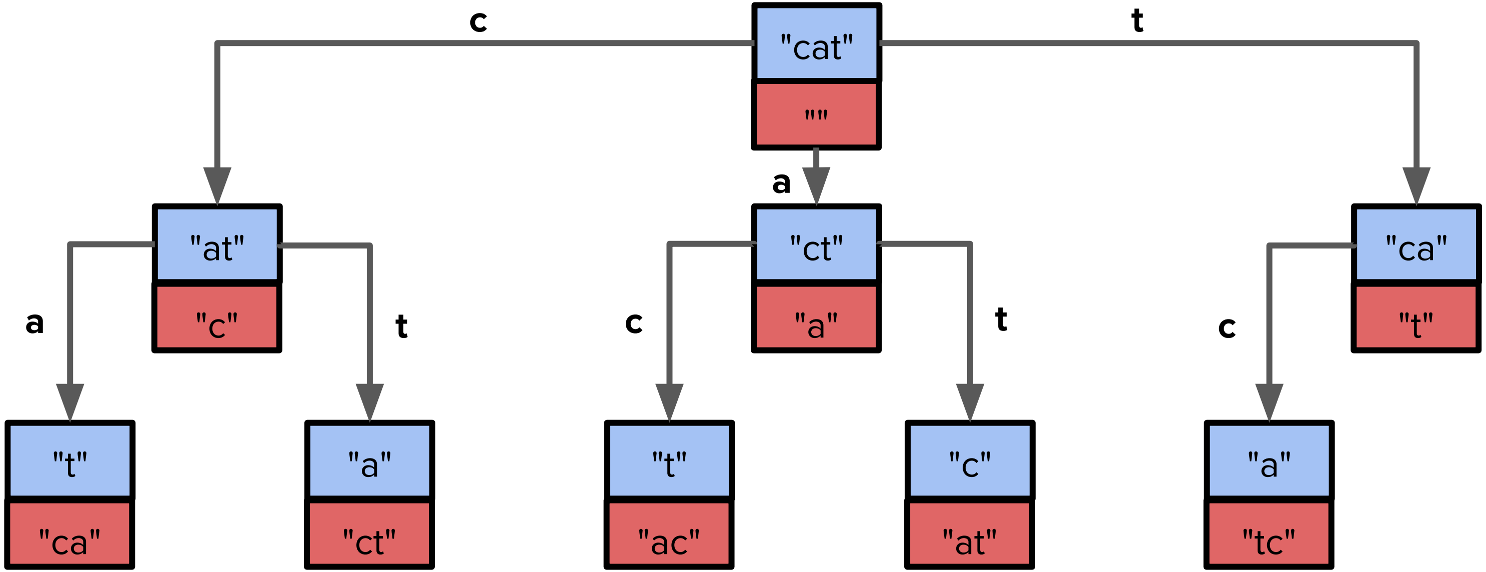
Decisions yet to be made
Decisions made so far

Decision tree: Find all permutations of "cat"



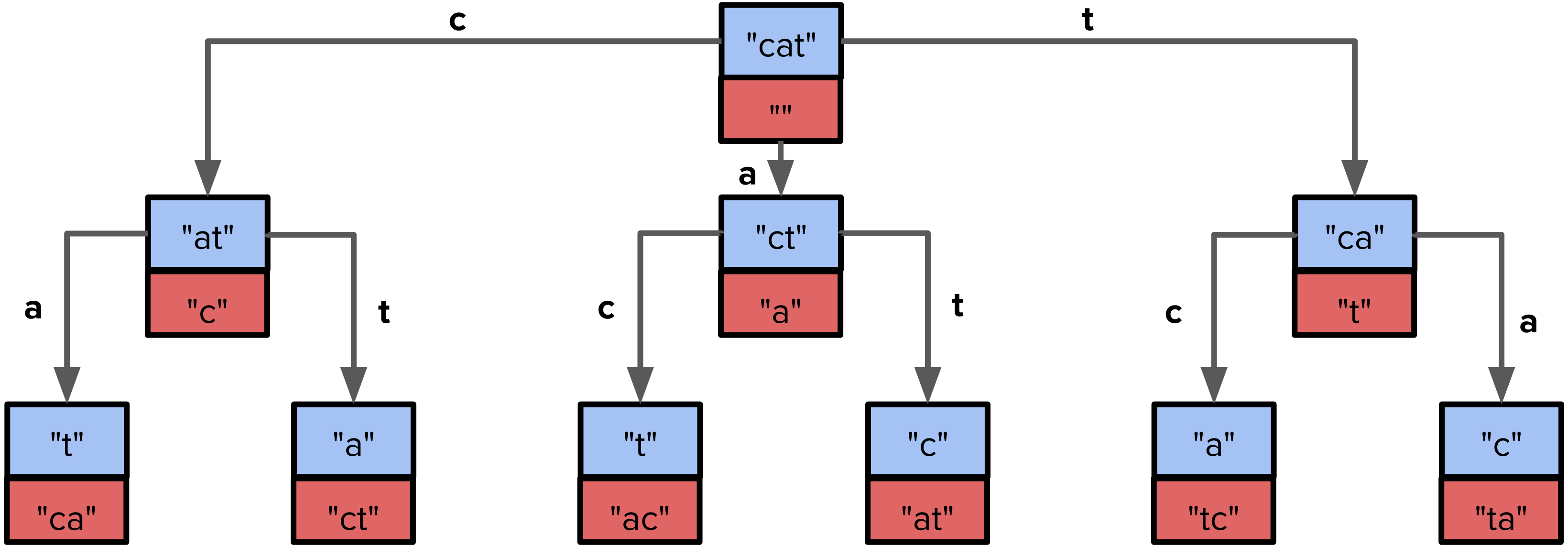
Decisions yet to be made
Decisions made so far

Decision tree: Find all permutations of "cat"



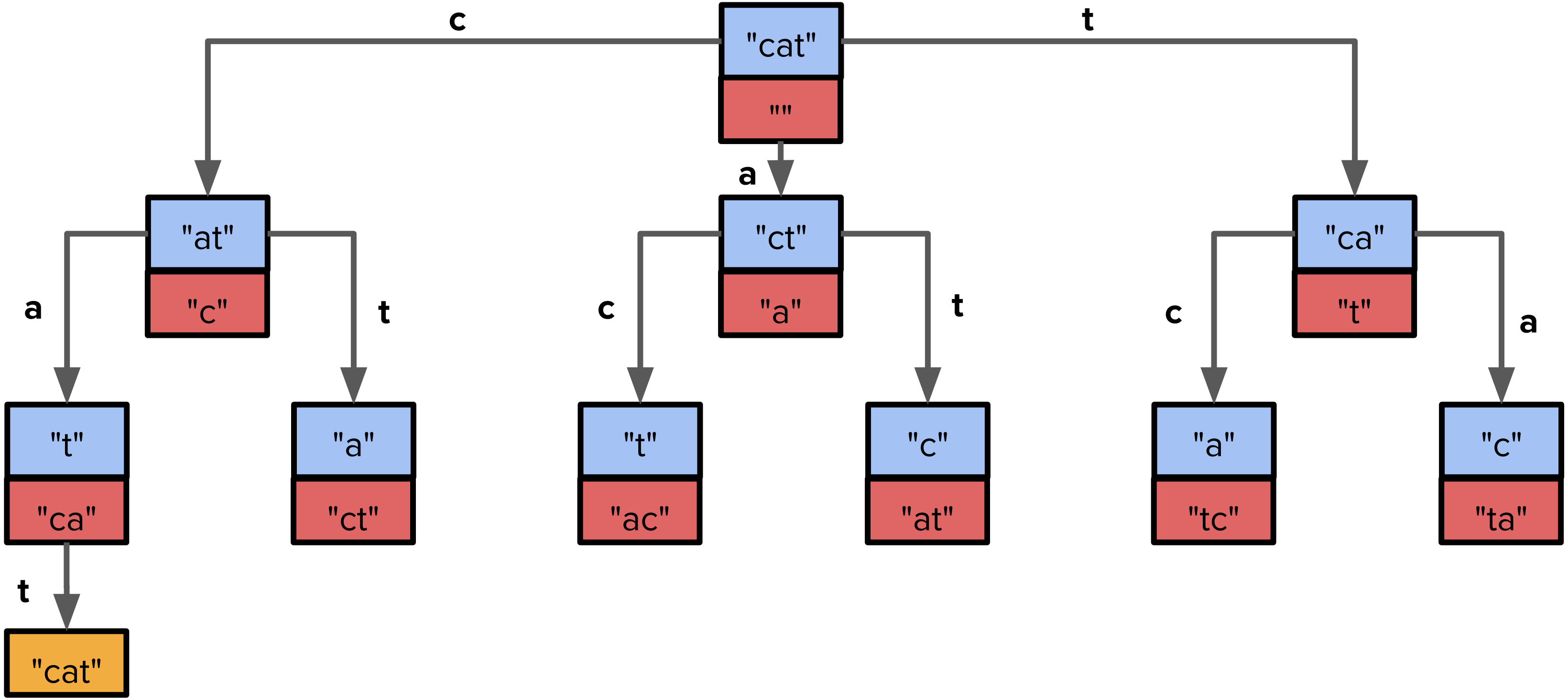
Decisions yet to be made
Decisions made so far

Decision tree: Find all permutations of "cat"



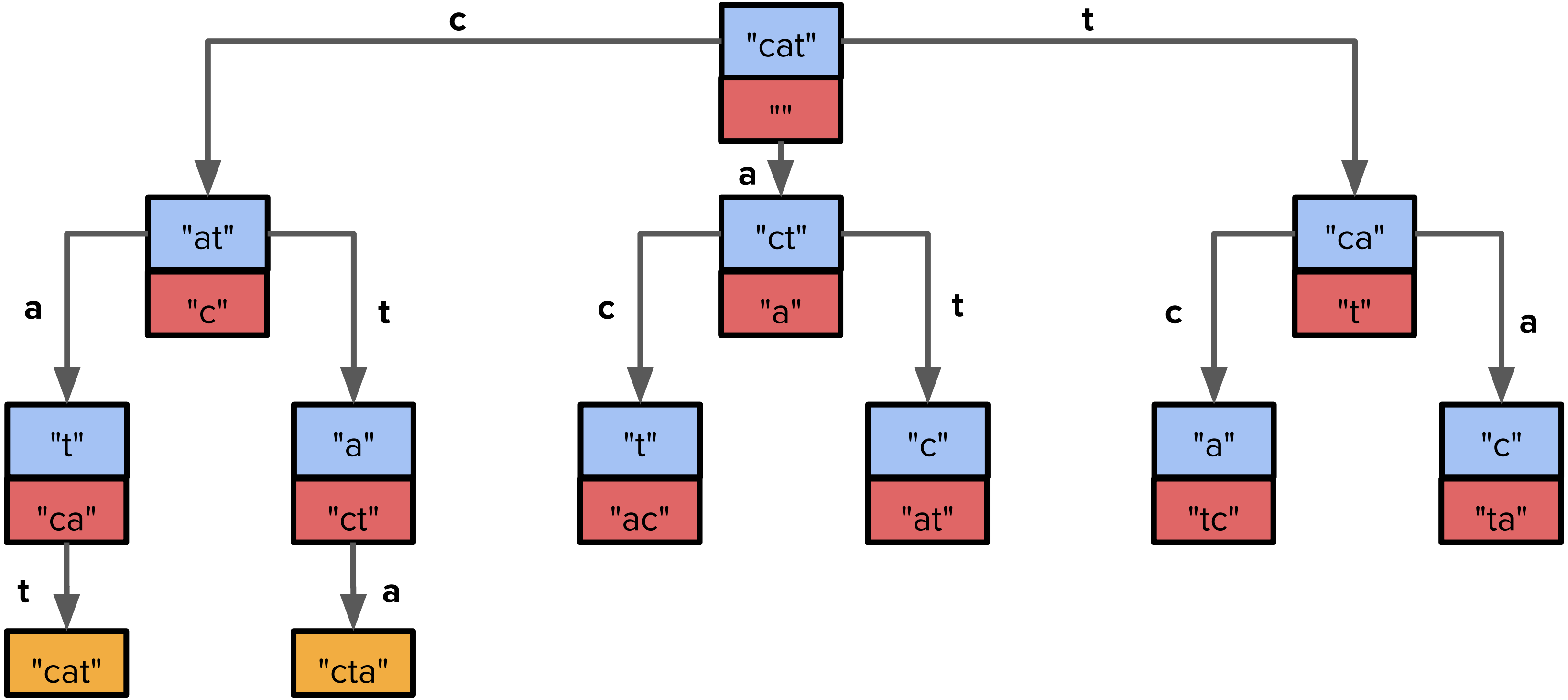
Decisions yet to be made
Decisions made so far

Decision tree: Find all permutations of "cat"



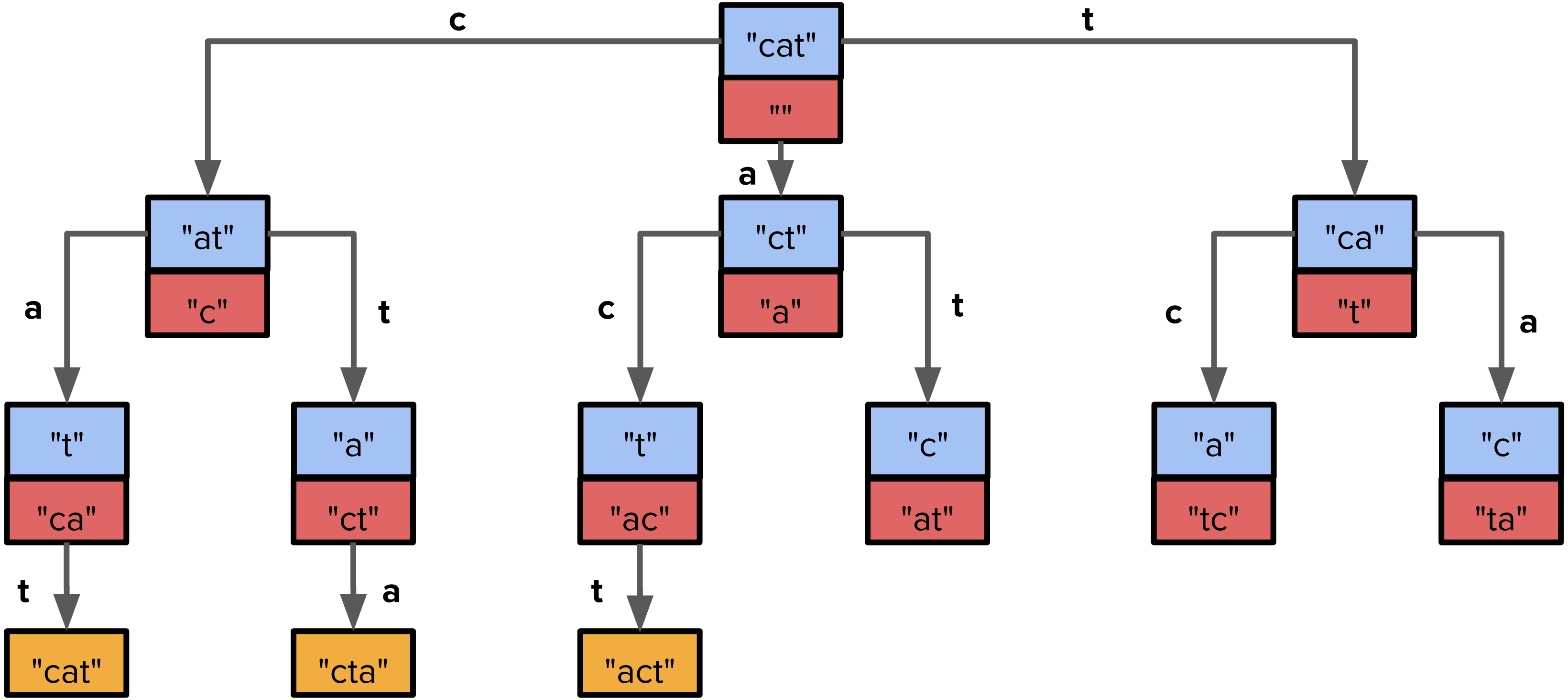
Decisions yet to be made
Decisions made so far

Decision tree: Find all permutations of "cat"



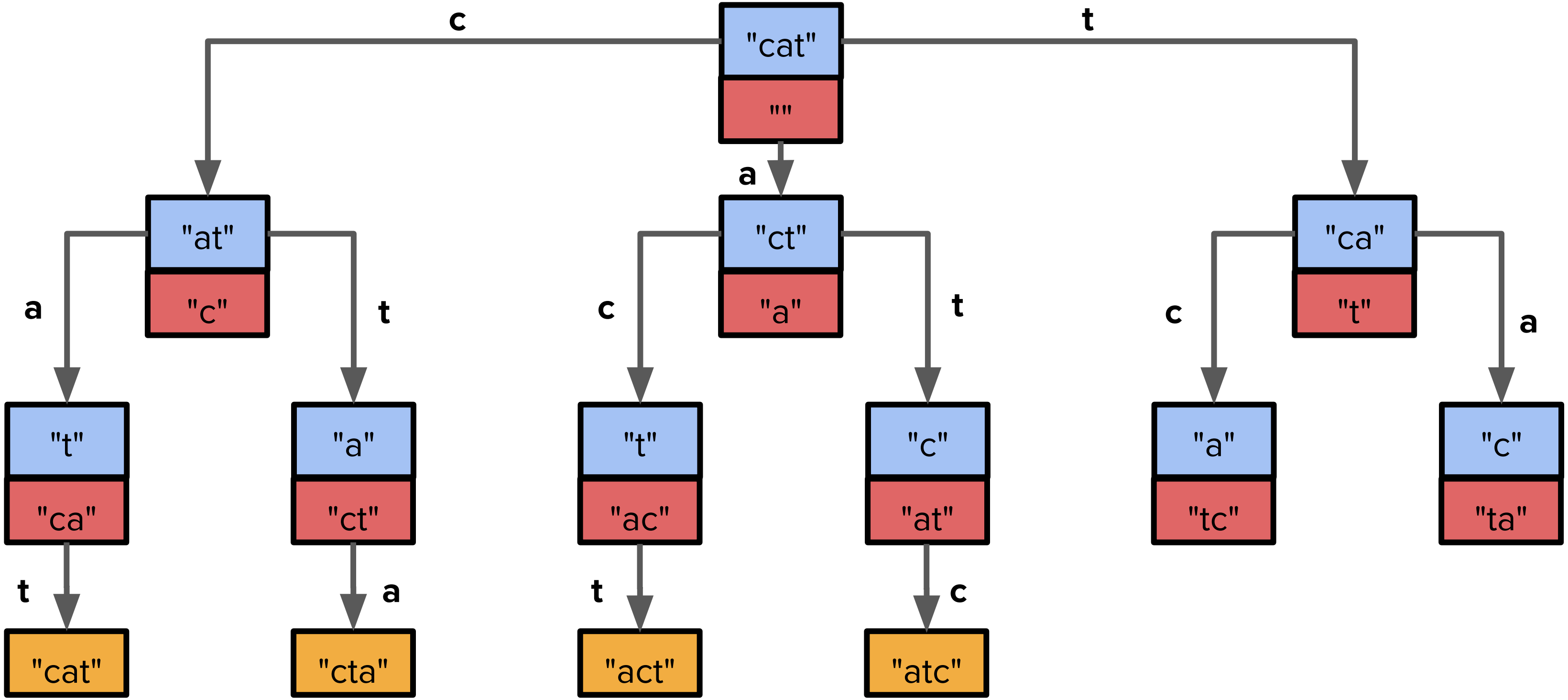
Decisions yet to be made
Decisions made so far

Decision tree: Find all permutations of "cat"



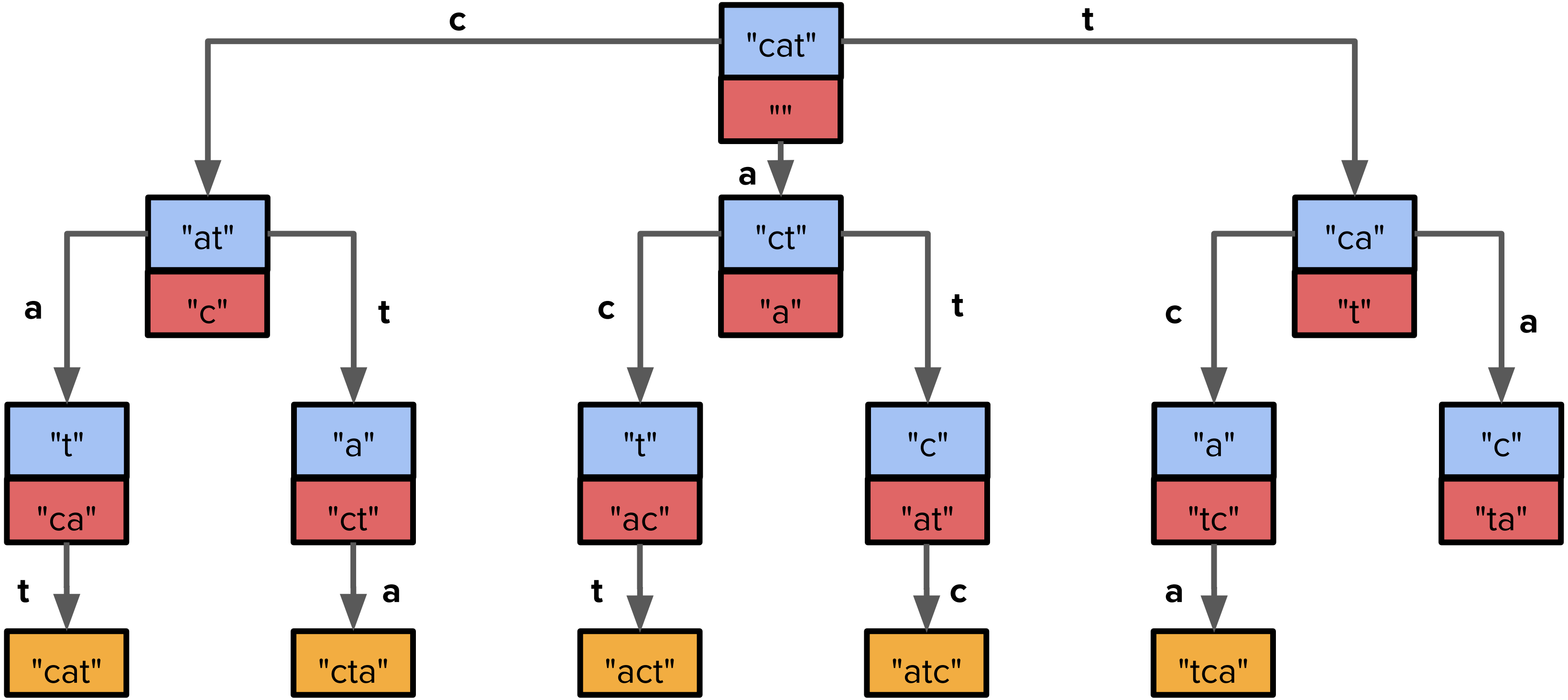
Decisions yet to be made
Decisions made so far

Decision tree: Find all permutations of "cat"



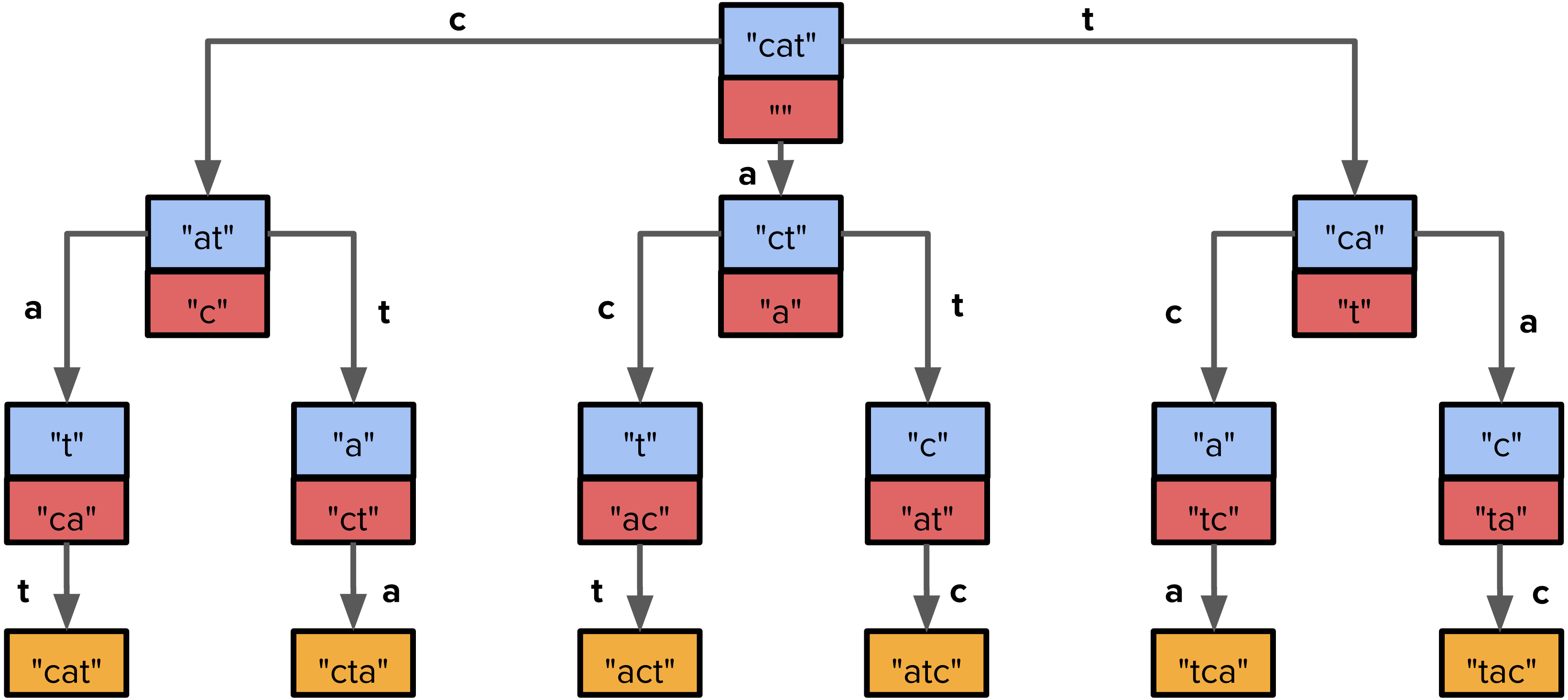
Decisions yet to be made
Decisions made so far

Decision tree: Find all permutations of "cat"



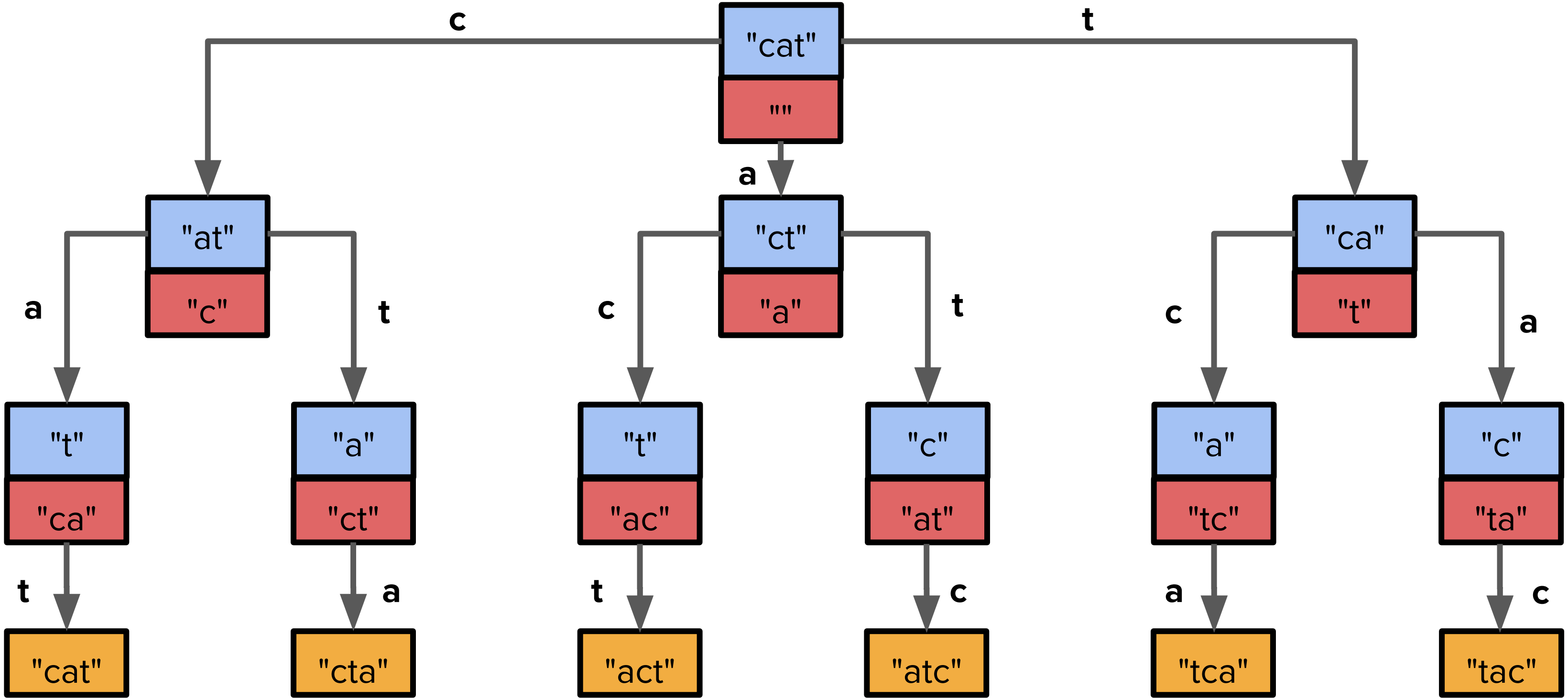
Decision tree: Find all permutations of "cat"

Decisions yet to be made
Decisions made so far



Decisions yet to be made
Decisions made so far

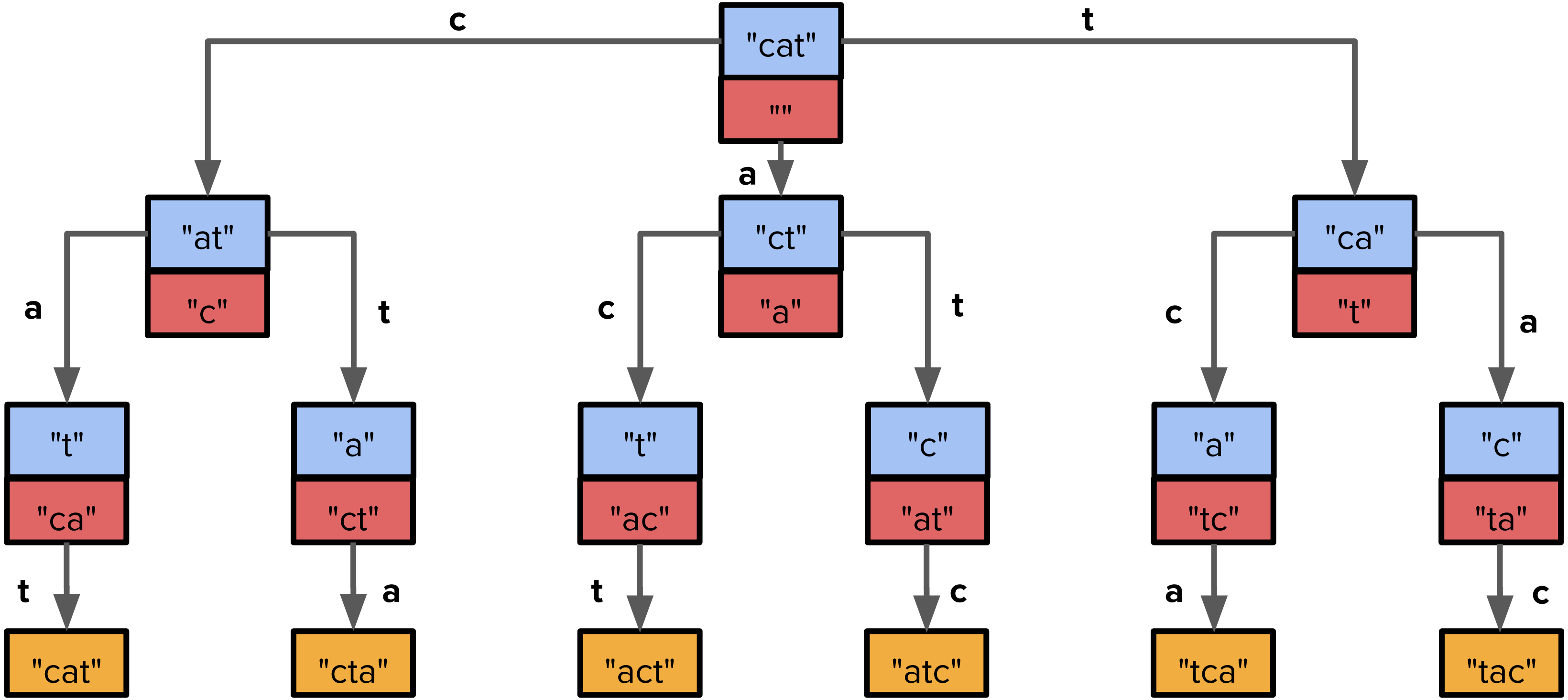
Decision tree: Find all permutations of "cat"



Base case: No letters remaining to choose!

Decisions yet to be made
Decisions made so far

Decision tree: Find all permutations of "cat"



Recursive case: For every letter remaining, add that letter to the current permutation and recurse!

Let’s code it!

Permutations Code

```
void listPermutations(string s){
    listPermutationsHelper(s, "");
}

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

Permutations Code

```
void listPermutations(string s){
   listPermutationsHelper(s, "");
}

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

Use of recursive helper function with empty string as starting point

Permutations Code

Decisions yet to be made

```
void listPermutations(string s){
    listPermutationsHelper(s, "");
}

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

Permutations Code

Decisions yet to be made

Decisions already made

```
void listPermutations(string s){
   listPermutationsHelper(s, "");
}

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

Permutations Code

Decisions yet to be made

Decisions already made

```
void listPermutations(string s){
   listPermutationsHelper(s, "");
}

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

Base case: No decisions remain

Permutations Code

```
void listPermutations(string s){
   listPermutationsHelper(s, "");
}

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

Decisions yet to be made

Decisions already made

Base case: No decisions remain

Recursive case: Try all options for next decision

Takeaways

- The specific model of the general **"choose / explore / unchoose"** pattern in backtracking recursion that we applied here can be thought of as **"copy, edit, recurse"**
    - Since we passed all our parameters by value, each recursive stack frame had its own independent copy of the string data that it could edit as appropriate
    - The "unchoose" step is **implicit** since there is no need to undo anything by virtue of the fact that editing a copy only has local consequences.

Takeaways

- The specific model of the general **"choose / explore / unchoose"** pattern in backtracking recursion that we applied here can be thought of as **"copy, edit, recurse"**

- At each step of the recursive backtracking process, it is important to keep track of **the decisions we've made so far** and **the decisions we have left to make**

Takeaways

- The specific model of the general **"choose / explore / unchoose"** pattern in backtracking recursion that we applied here can be thought of as **"copy, edit, recurse"**

- At each step of the recursive backtracking process, it is important to keep track of **the decisions we've made so far** and **the decisions we have left to make**

- Backtracking recursion can have **variable branching factors** at each level

Takeaways

- The specific model of the general **"choose / explore / unchoose"** pattern in backtracking recursion that we applied here can be thought of as **"copy, edit, recurse"**

- At each step of the recursive backtracking process, it is important to keep track of **the decisions we've made so far** and **the decisions we have left to make**

- Backtracking recursion can have **variable branching factors** at each level

- Use of helper functions and initial empty params that get built up is common

Shrinkable Words

“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?”

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

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

How can we determine if a word is shrinkable?

- 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.

- Idea: Let’s use a decision tree to remove letters and determine **shrinkability**!

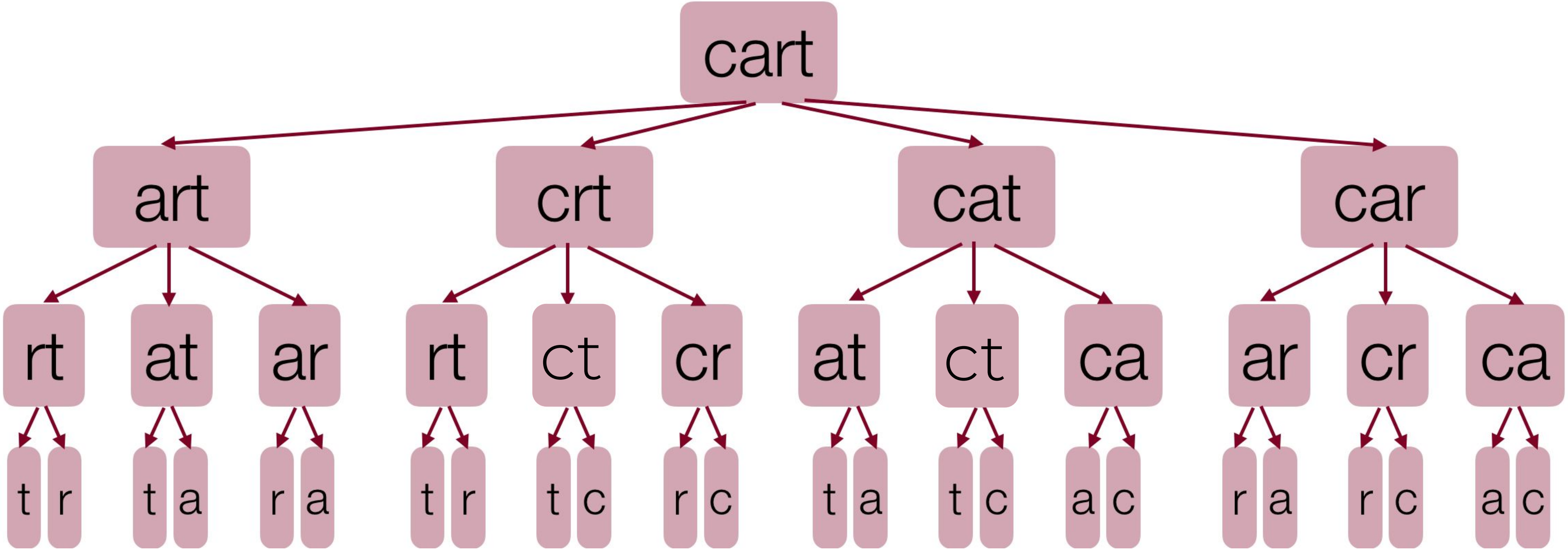
What defines our shrinkable decision tree?

- **Decision** at each step (each level of the tree):
  - What letter are going to remove?

- **Options** at each decision (branches from each node):
  - The remaining letters in the string

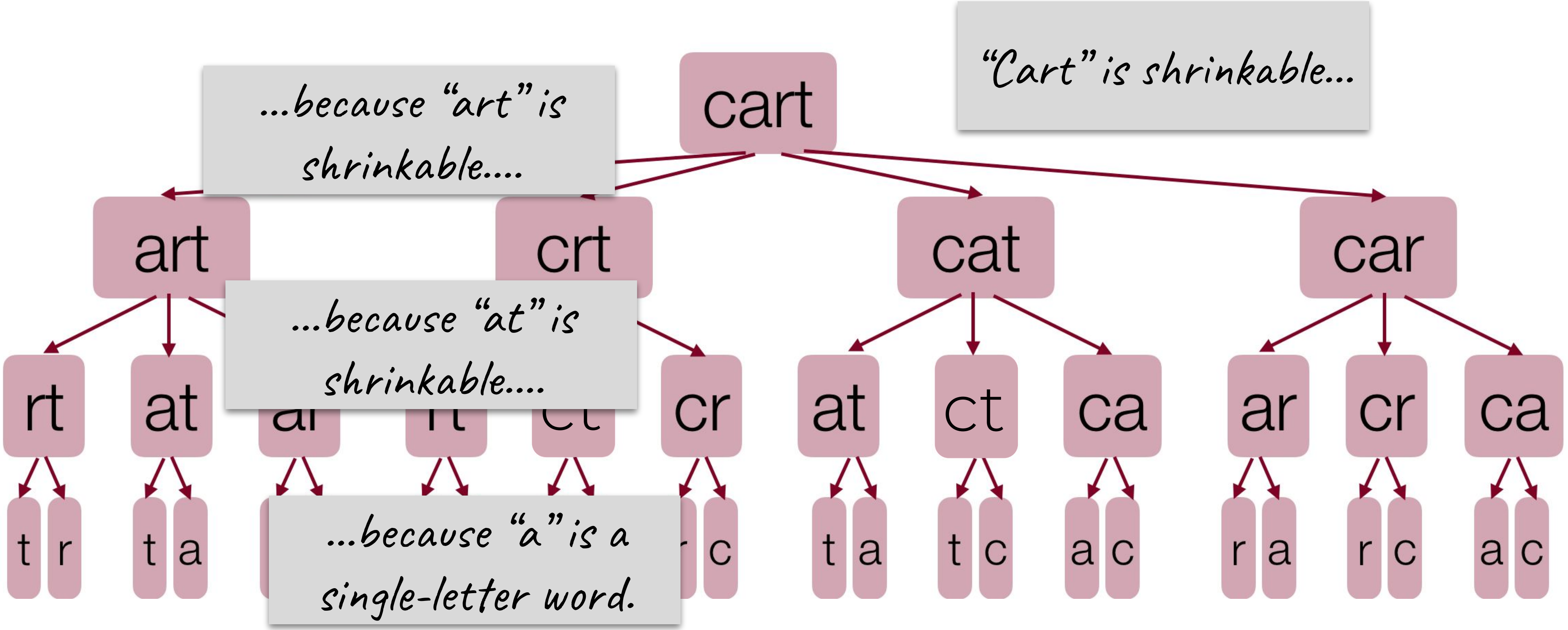
- Information we need to store along the way:
  - The shrinking string

What defines our shrinkable decision tree?



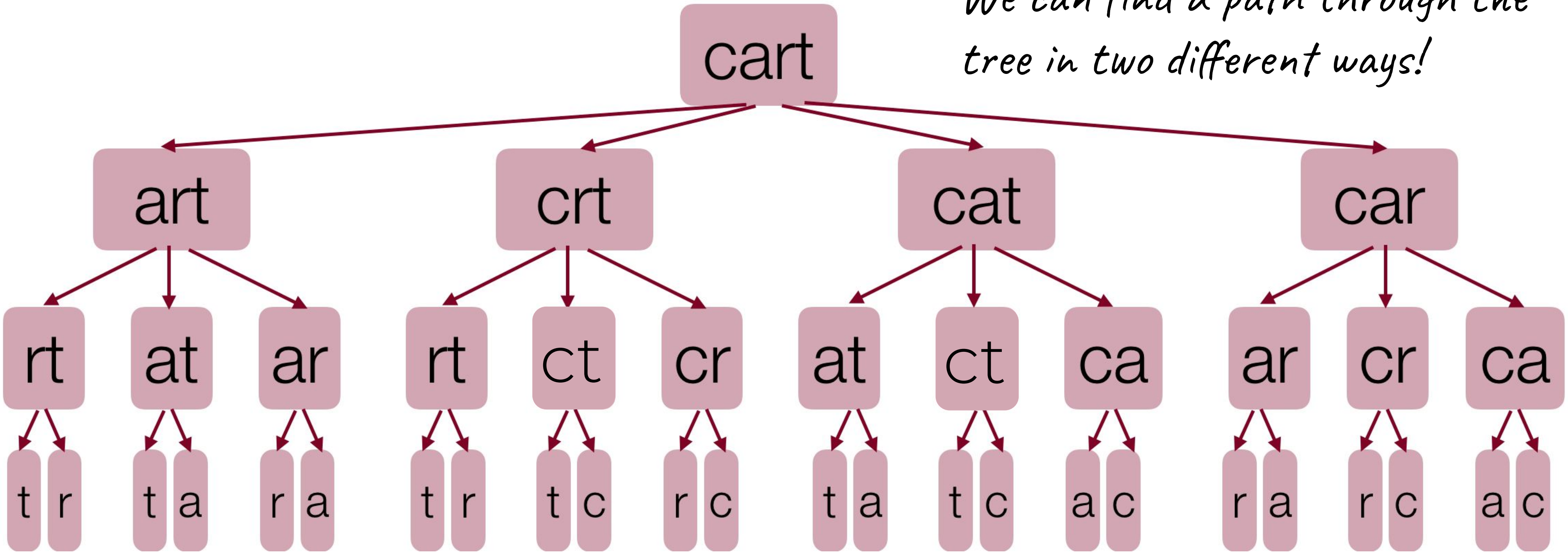
Examples from Chris Gregg and Keith Schwarz

What defines our shrinkable decision tree?



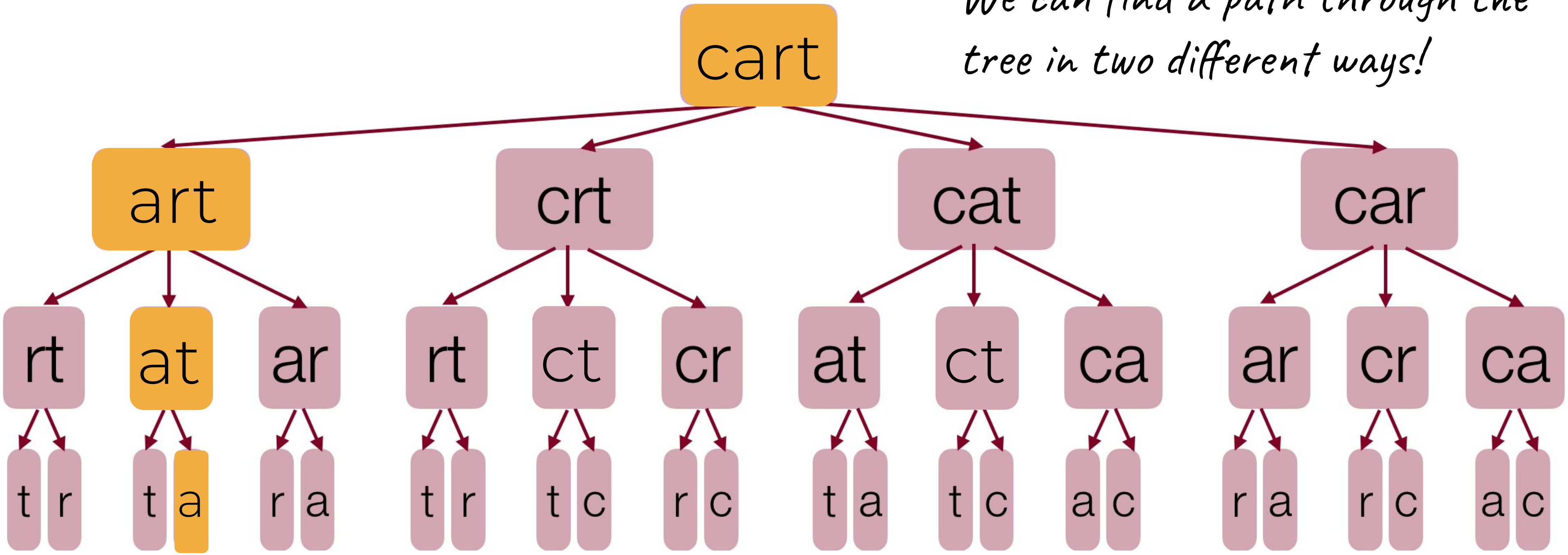
What defines our shrinkable decision tree?

We can find a path through the tree in two different ways!

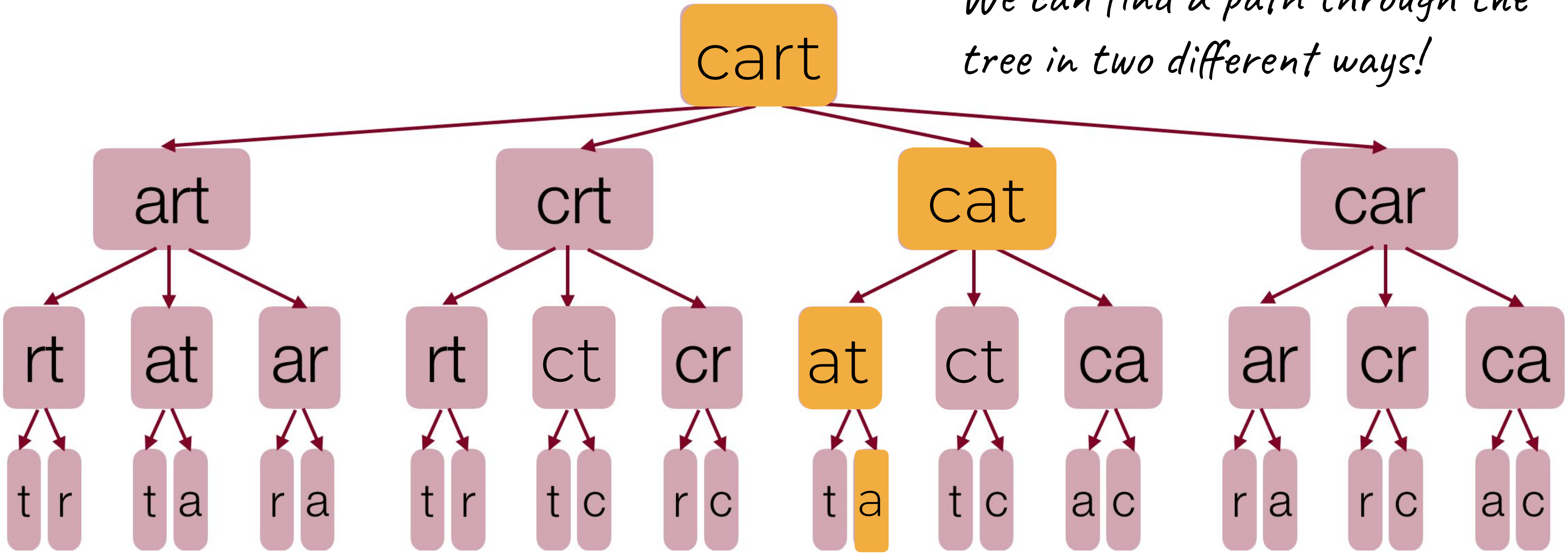


What defines our shrinkable decision tree?

We can find a path through the tree in two different ways!

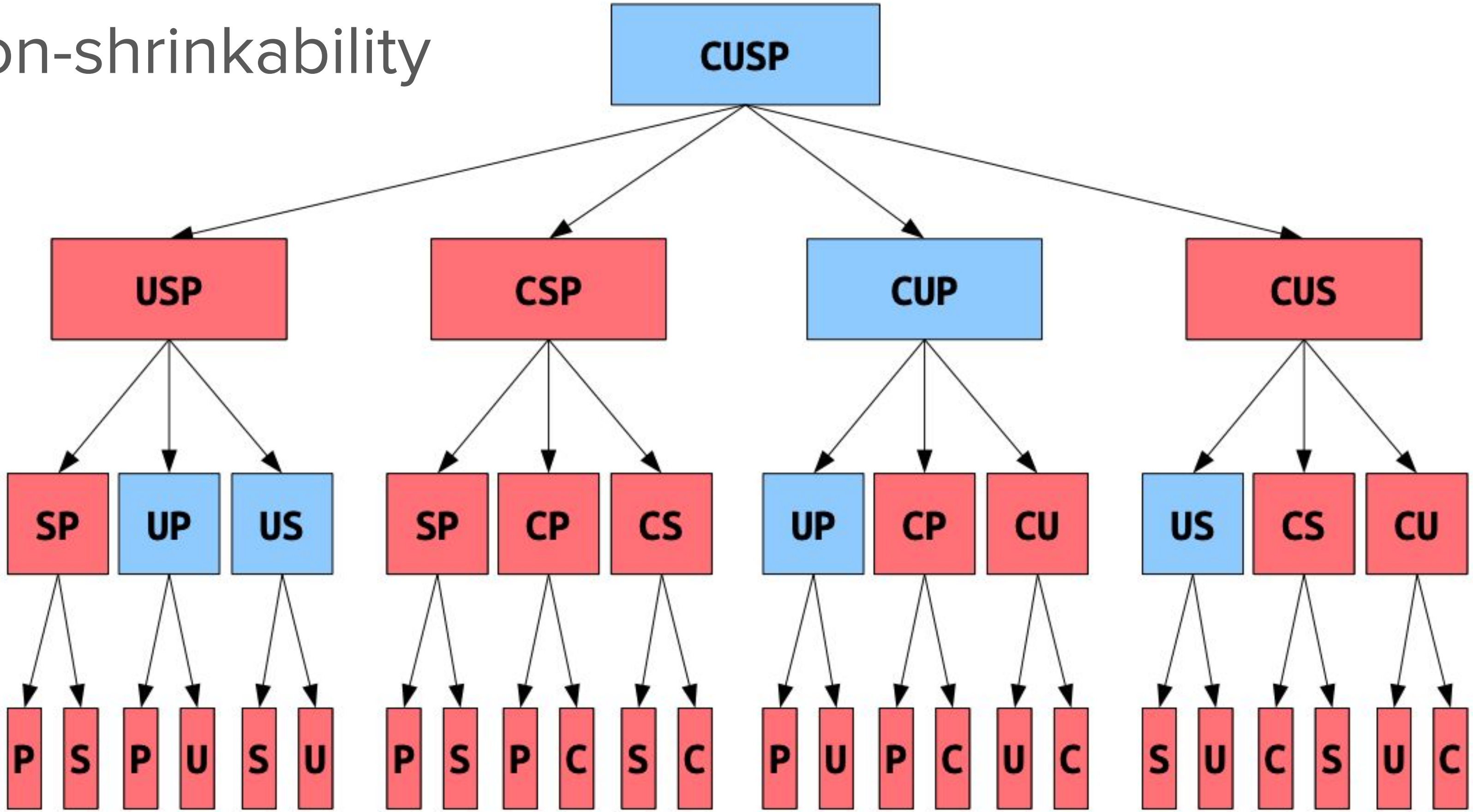


What defines our shrinkable decision tree?

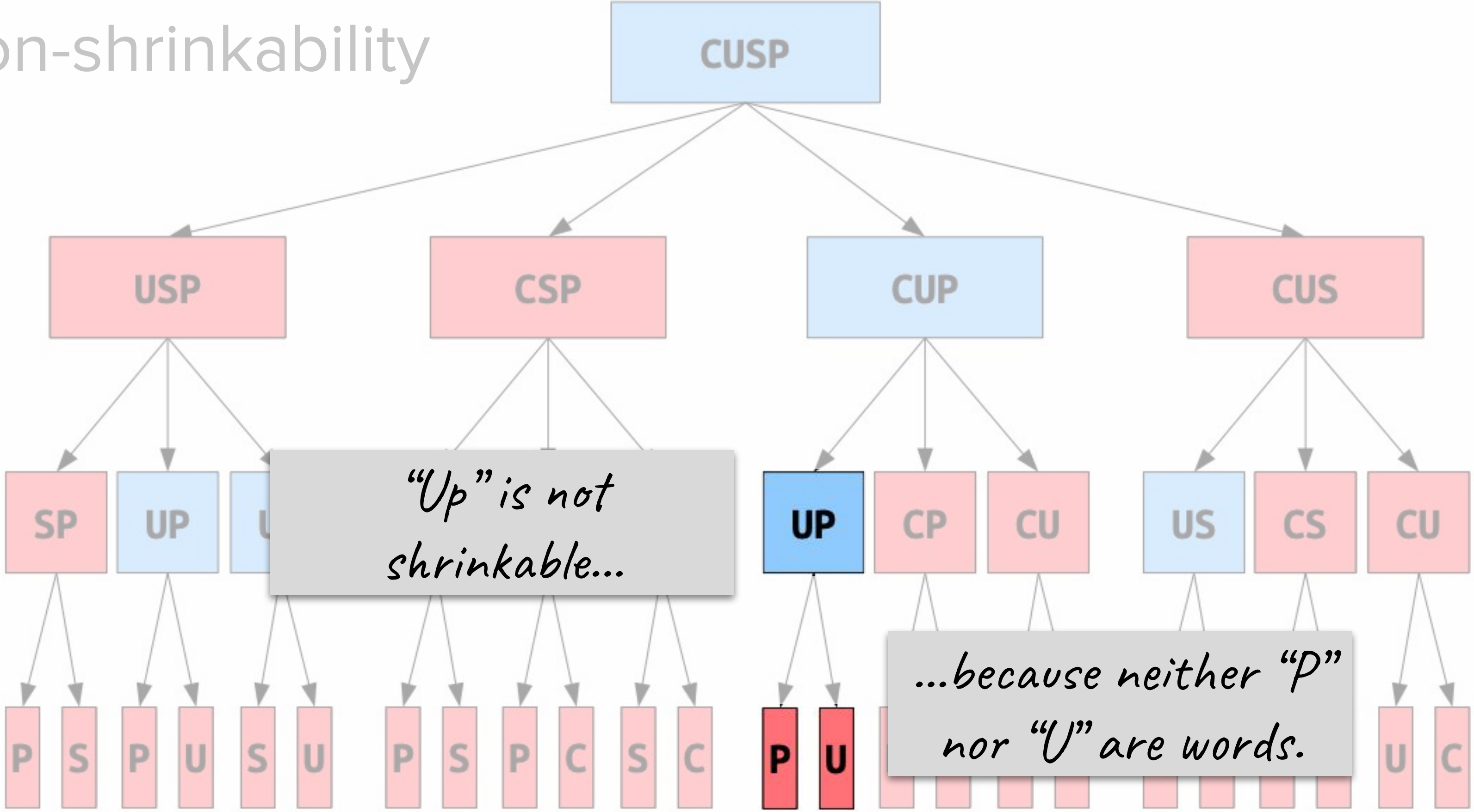


We can find a path through the tree in two different ways!

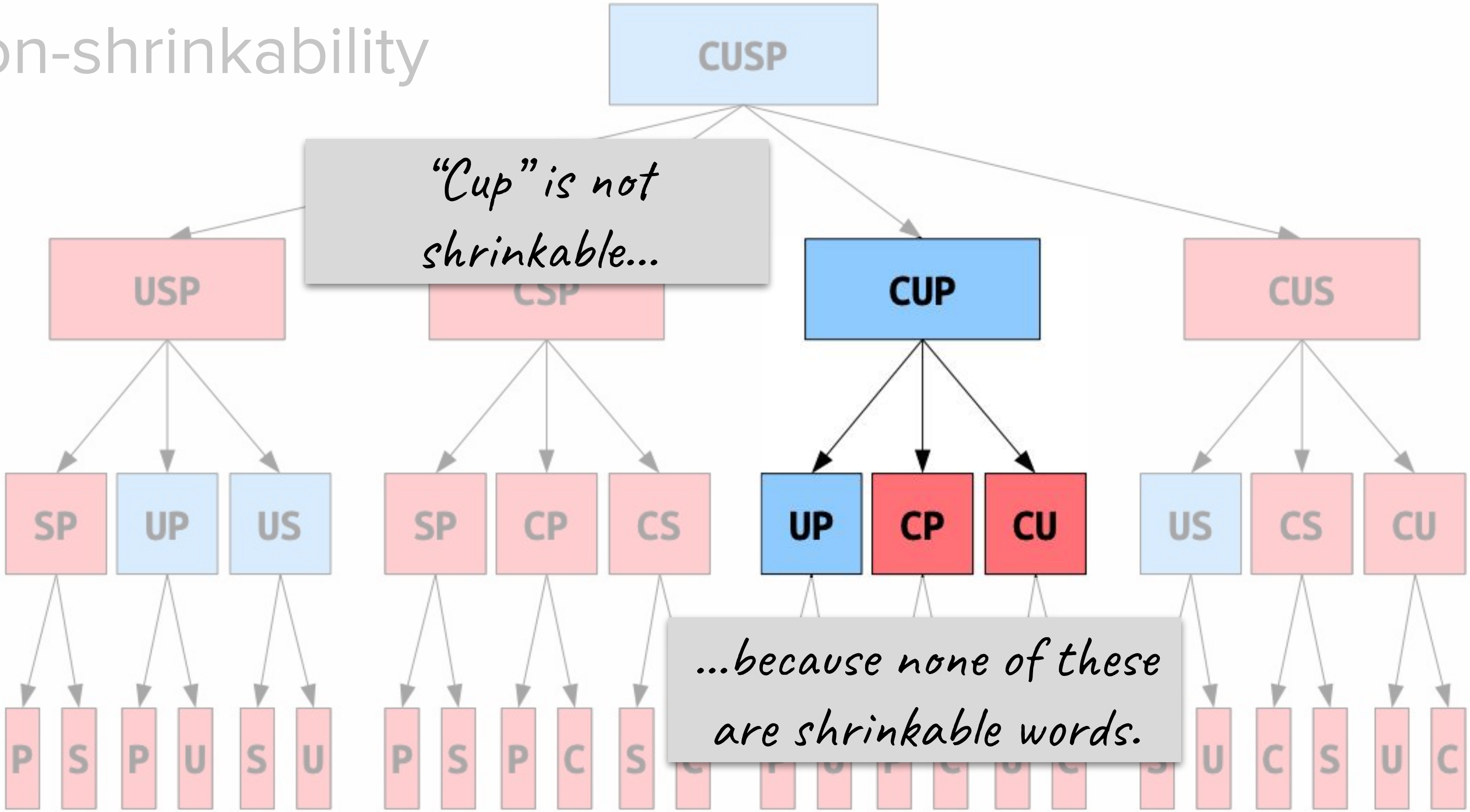
Non-shrinkability



Non-shrinkability

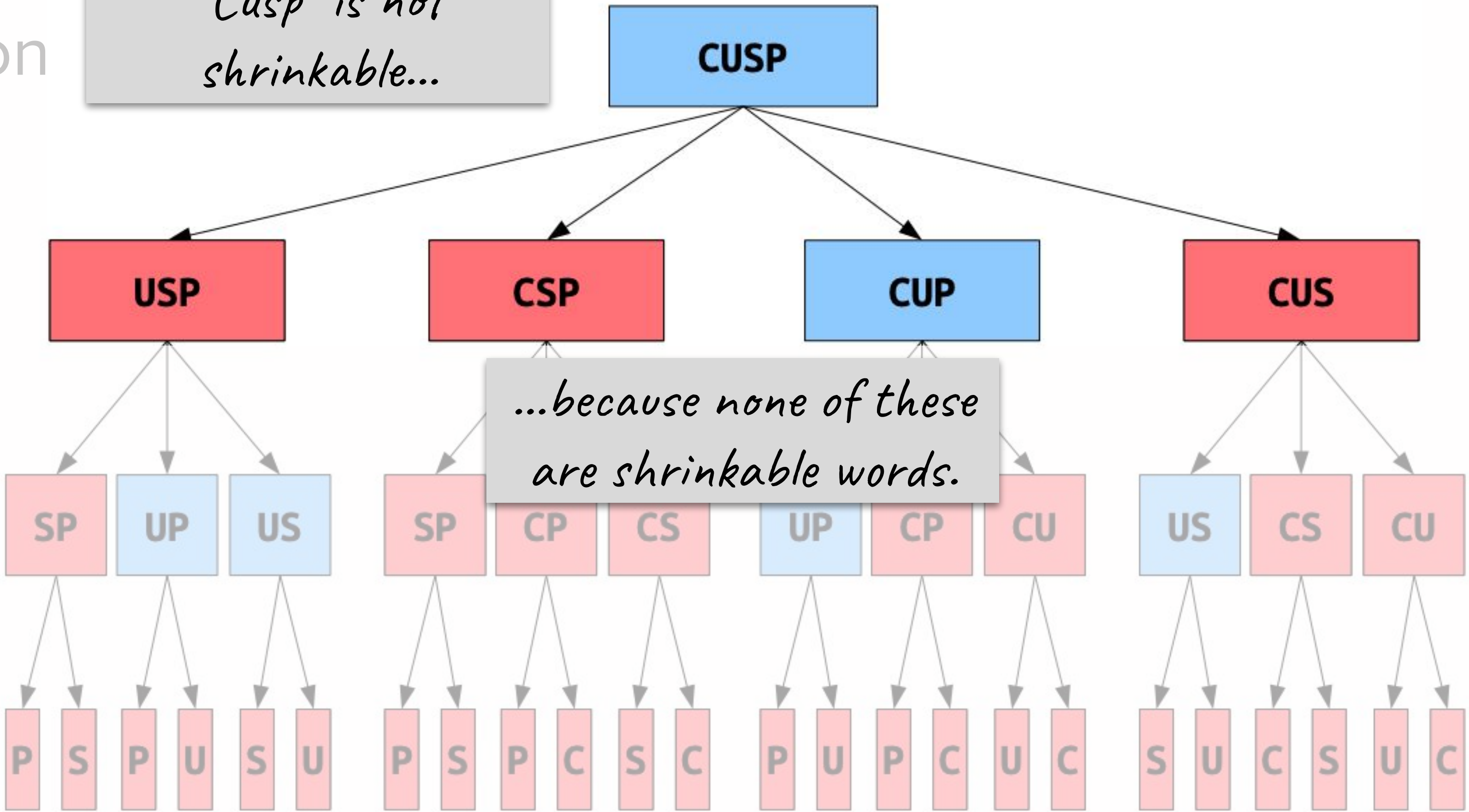


Non-shrinkability



Non

"Cusp" is not shrinkable...



...because none of these are shrinkable words.

Examples from Chris Gregg and Keith Schwarz

How can we determine if a word is shrinkable?

- **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 cases**:
  - 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.

Lexicon

- Lexicon is a helpful ADT provided by the Stanford C++ libraries (in **`lexicon.h`**) that is used specifically for storing many words that make up a dictionary

- Generally, Lexicons offer faster lookup than normal Sets, which is why we choose to use them when dealing with words and large dictionaries

- ```
  Lexicon lex("res/EnglishWords.txt"); // create from file
  lex.contains("koala"); // returns true
  lex.contains("zzzzz"); // returns false
  lex.containsPrefix("fi"); // returns true if there are
  any words starting with "fi" in the dictionary
  ```
  ```

Let’s code it!

Takeaways

- This is another example of **copy-edit-recurse** to choose, explore, and then implicitly unchoose!

- In this problem, we're using backtracking to **find if a solution exists**.
  - Notice the way the recursive case is structured:

```
for all options at each decision point:
    if recursive call returns true:
        return true;
return false if all options are exhausted;
```

Announcements

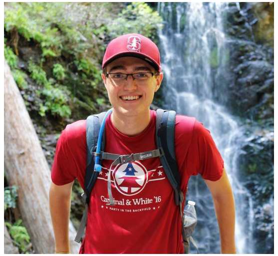
Announcements

- The grace period for Assignment 2 expires tonight at 11:59pm PDT.
- Assignment 3 will be released by the end of the day today.
- The Assignment 3 YEAH session will be hosted by Trip tomorrow evening at 6pm PDT. The slides and recording will be posted shortly after the session is over.
- We will be releasing practice problems and information about the diagnostic over the weekend.

Subsets

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:



Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

{}

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

{ }

Even though we may not care about this "team," the empty set is a subset of our original set!

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

{}

As humans, it might be easiest to think about all teams (subsets) of a particular size.

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

```
{}
{"Nick"}
{"Kylie"}
{"Trip"}
```

As humans, it might be easiest to think about all teams (subsets) of a particular size.

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

```
{}
{“Nick”}
{“Kylie”}
{“Trip”}
{“Nick”, “Kylie”}
{“Nick”, “Trip”}
{“Kylie”, “Trip”}
```

As humans, it might be easiest to think about all teams (subsets) of a particular size.

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

```
{}
{"Nick"}
{"Kylie"}
{"Trip"}
{"Nick", "Kylie"}
{"Nick", "Trip"}
{"Kylie", "Trip"}
{"Nick", "Kylie", "Trip"}
```

As humans, it might be easiest to think about all teams (subsets) of a particular size.

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

```
{}
{"Nick"}
{"Kylie"}
{"Trip"}
{"Nick", "Kylie"}
{"Nick", "Trip"}
{"Kylie", "Trip"}
{"Nick", "Kylie", "Trip"}
```

Another case of "generate/count all solutions" using recursive backtracking!

Discuss in breakouts:

What are the possible subsets of the choices {"c++", "python", "java", "javascript"}?

What potential recursive insights about generating subsets can you glean from this example?

[Time-permitting] Can you come up with a base case and recursive case for generating permutations?

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

```
{}
{"Nick"}
{"Kylie"}
{"Trip"}
{"Nick", "Kylie"}
{"Nick", "Trip"}
{"Kylie", "Trip"}
{"Nick", "Kylie", "Trip"}
```

For computers generating subsets (and thinking about decisions), there's another pattern we might notice...

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

```
{}
{"Nick"}
{"Kylie"}
{"Trip"}
{"Nick", "Kylie"}
{"Nick", "Trip"}
{"Kylie", "Trip"}
{"Nick", "Kylie", "Trip"}
```

Half the subsets contain "Nick"

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

```
{}
{"Nick"}
{"Kylie"}
{"Trip"}
{"Nick", "Kylie"}
{"Nick", "Trip"}
{"Kylie", "Trip"}
{"Nick", "Kylie", "Trip"}
```

Half the subsets contain "Kylie"

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

```
{}
{"Nick"}
{"Kylie"}
{"Trip"}
{"Nick", "Kylie"}
{"Nick", "Trip"}
{"Kylie", "Trip"}
{"Nick", "Kylie", "Trip"}
```

Half the subsets contain "Trip"

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

```
{}
{"Nick"}
{"Kylie"}
{"Trip"}
{"Nick", "Kylie"}
{"Nick", "Trip"}
{"Kylie", "Trip"}
{"Nick", "Kylie", "Trip"}
```

Half the subsets that contain "Trip" also contain "Nick"

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

```
{}
{"Nick"}
{"Kylie"}
{"Trip"}
{"Nick", "Kylie"}
{"Nick", "Trip"}
{"Kylie", "Trip"}
{"Nick", "Kylie", "Trip"}
```

Half the subsets that contain both "Trip" and "Nick" contain "Kylie"

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

```
{}
{"Nick"}
{"Kylie"}
{"Trip"}
{"Nick", "Kylie"}
{"Nick", "Trip"}
{"Kylie", "Trip"}
{"Nick", "Kylie", "Trip"}
```

What defines our subsets decision tree?

- **Decision** at each step (each level of the tree):
  - Are we going to include a given element in our subset?

What defines our subsets decision tree?

- **Decision** at each step (each level of the tree):
  - Are we going to include a given element in our subset?

- **Options** at each decision (branches from each node):
  - Include element
  - Don’t include element

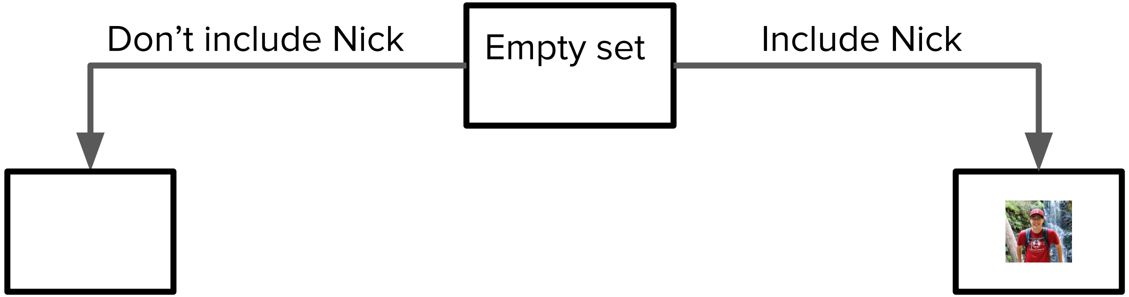
What defines our subsets decision tree?

- **Decision** at each step (each level of the tree):
  - Are we going to include a given element in our subset?

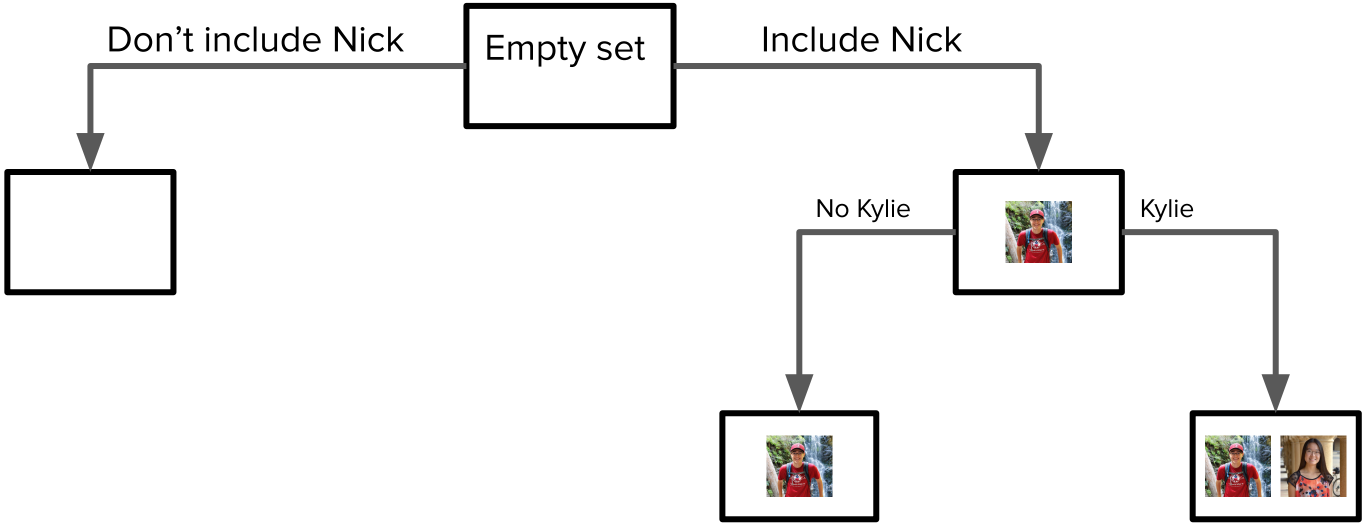
- **Options** at each decision (branches from each node):
  - Include element
  - Don’t include element

- Information we need to store along the way:
  - The set you’ve built so far
  - The remaining elements in the original set

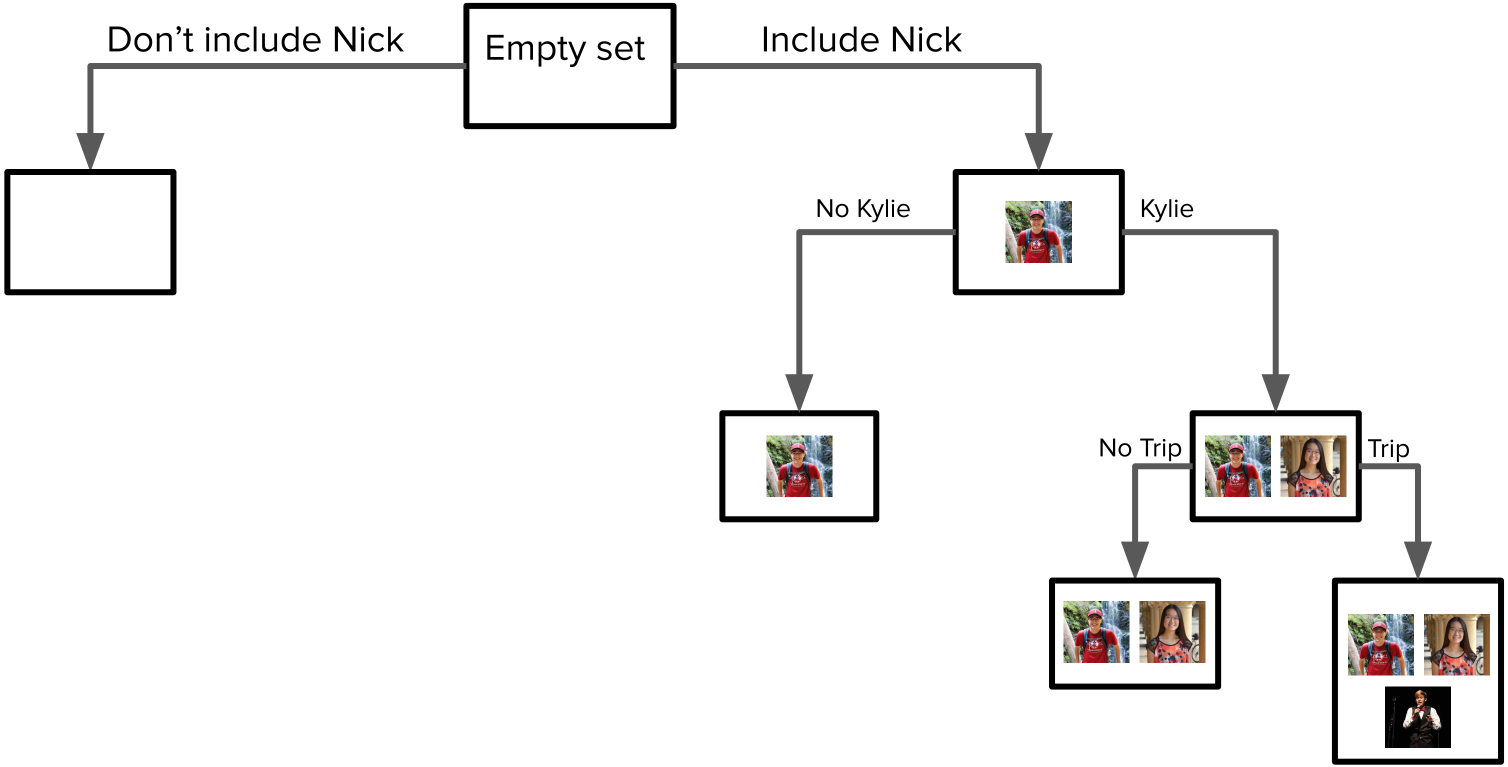
Decision tree



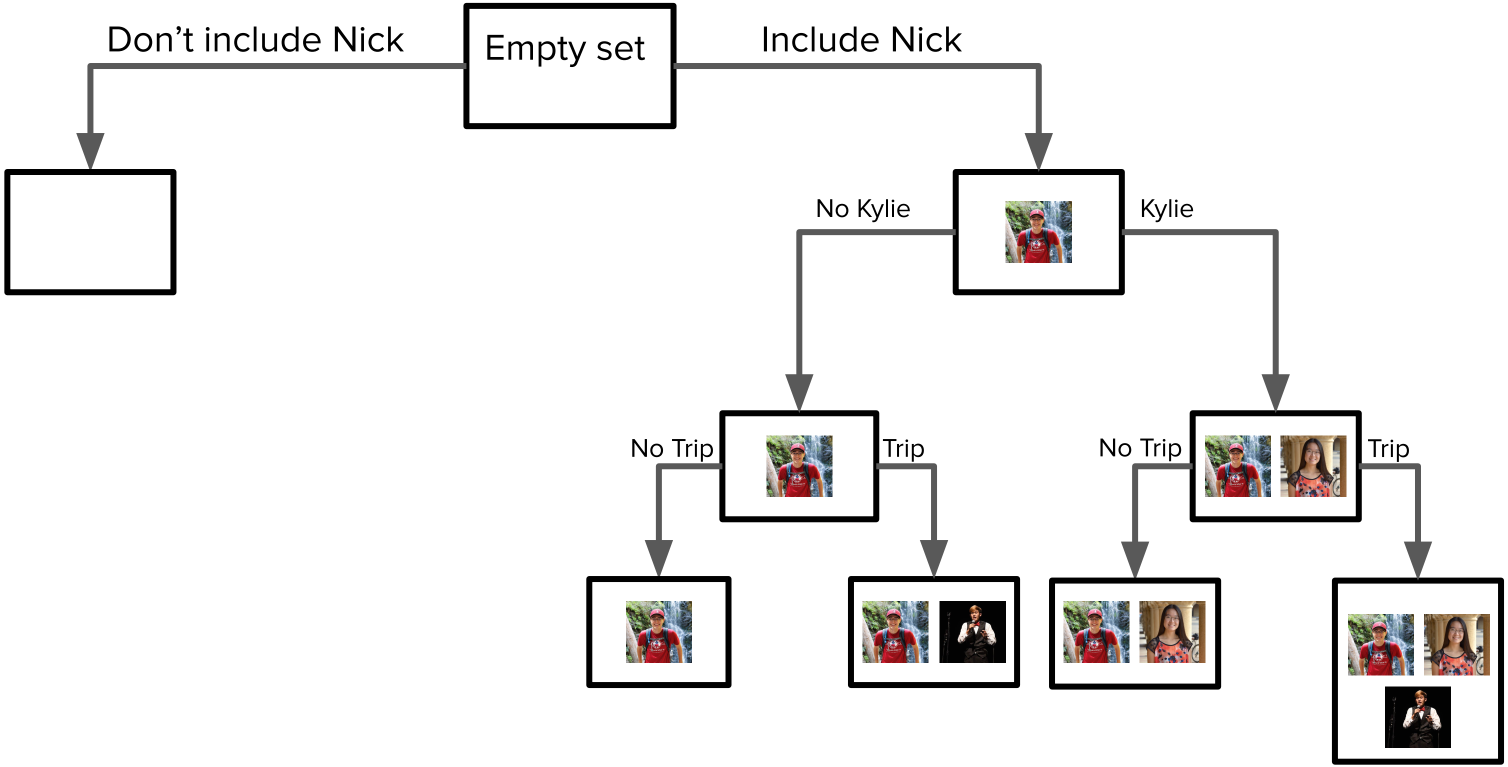
Decision tree



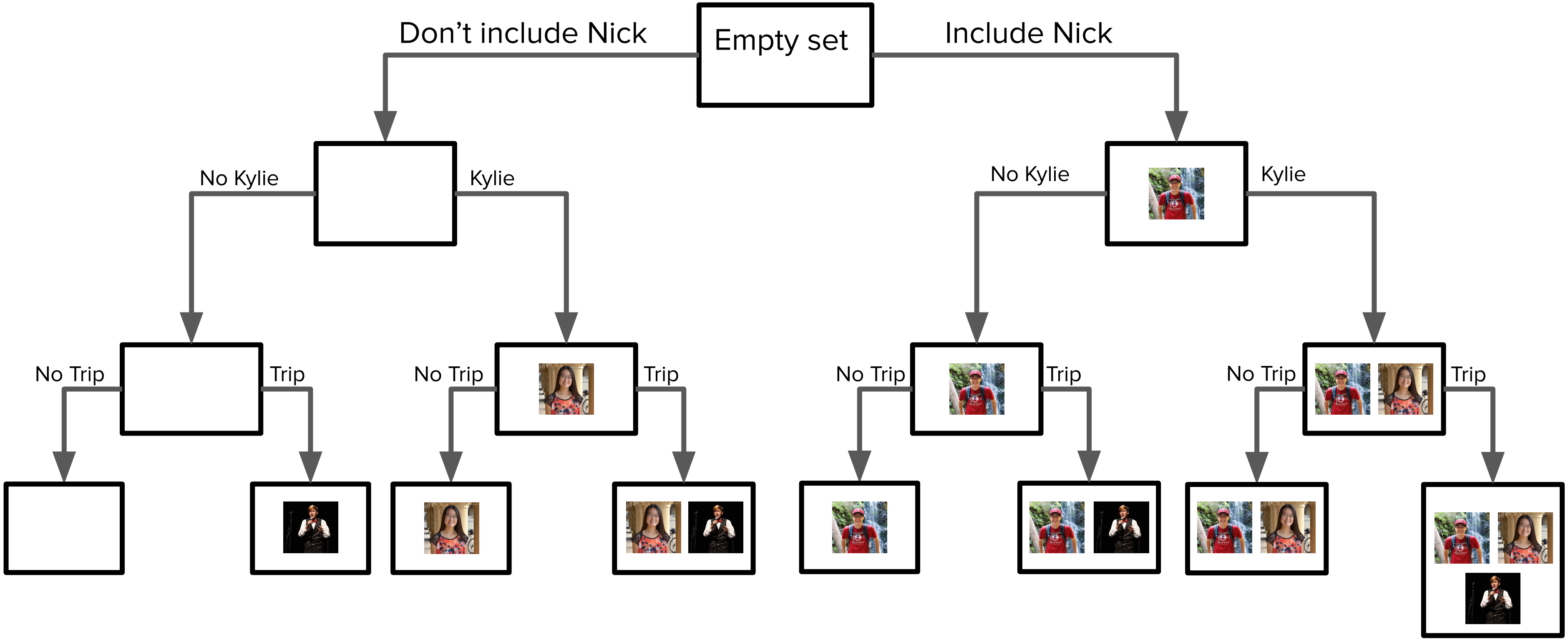
Decision tree



Decision tree



Decision tree



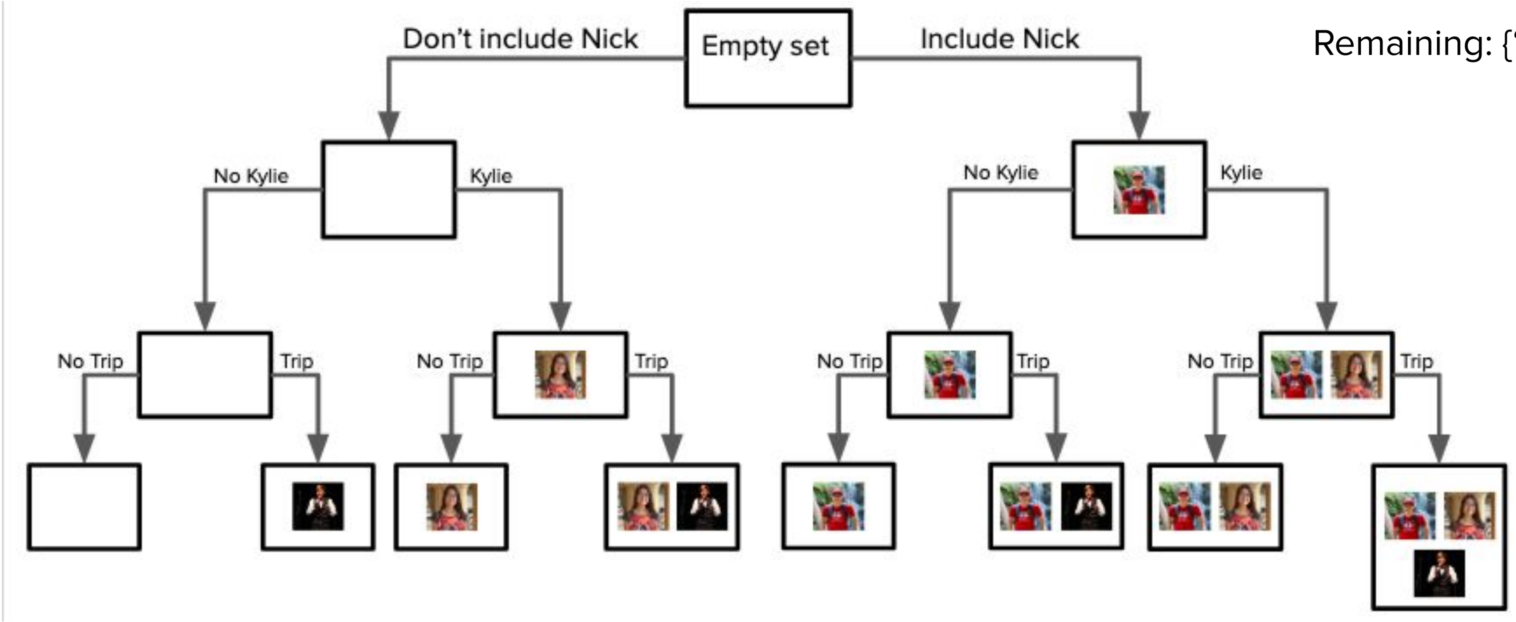
What defines our subsets decision tree?

- **Decision** at each step (each level of the tree):
  - Are we going to include a given element in our subset?

- **Options** at each decision (branches from each node):
  - Include element
  - Don’t include element

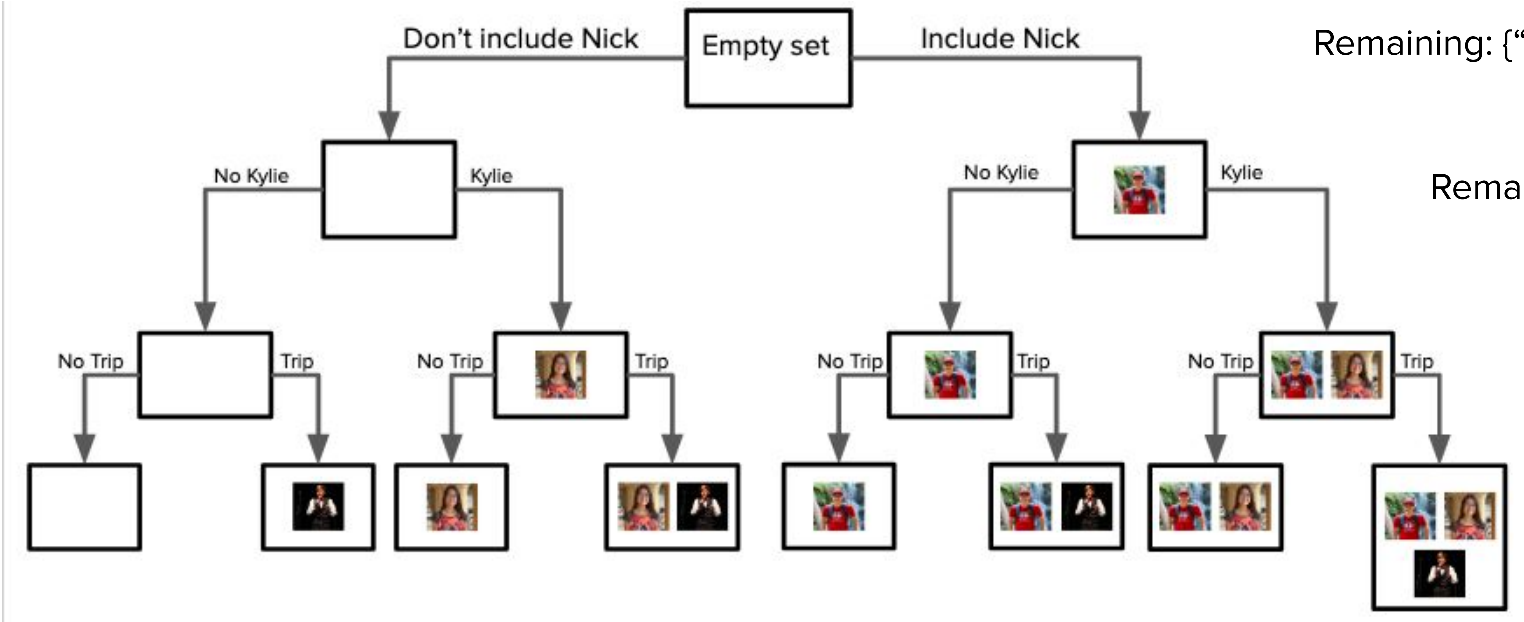
- Information we need to store along the way:
  - The set you’ve built so far
  - **The remaining elements in the original set**

Decision tree



Remaining: {"Nick", "Kylie", "Trip"}

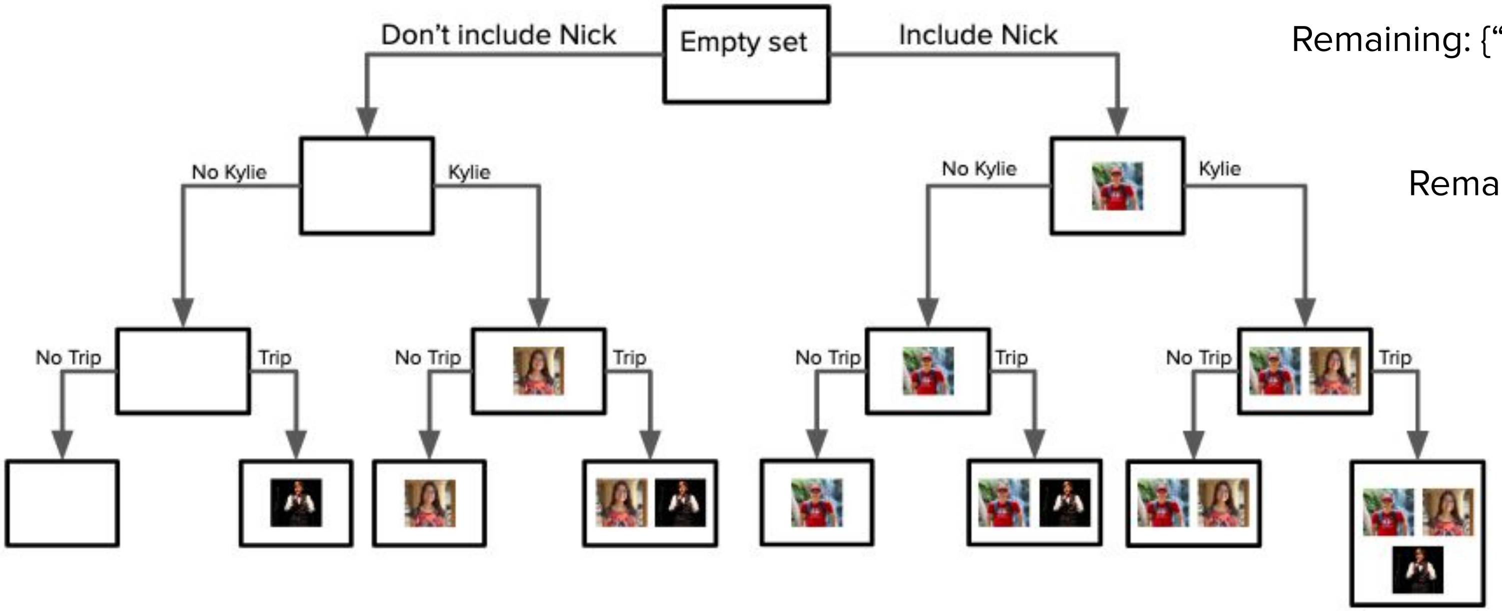
Decision tree



Remaining: {"Nick", "Kylie", "Trip"}

Remaining: {"Kylie", "Trip"}

Decision tree

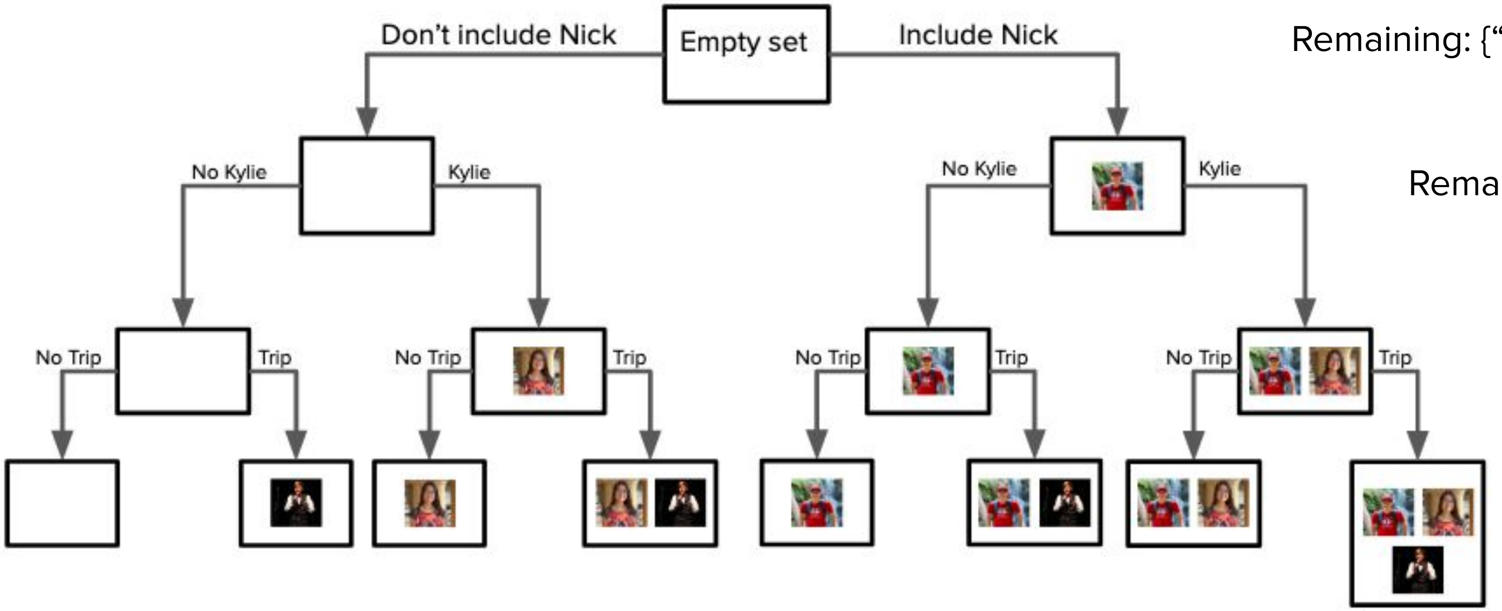


Remaining: {"Nick", "Kylie", "Trip"}

Remaining: {"Kylie", "Trip"}

Remaining: {"Trip"}

Decision tree



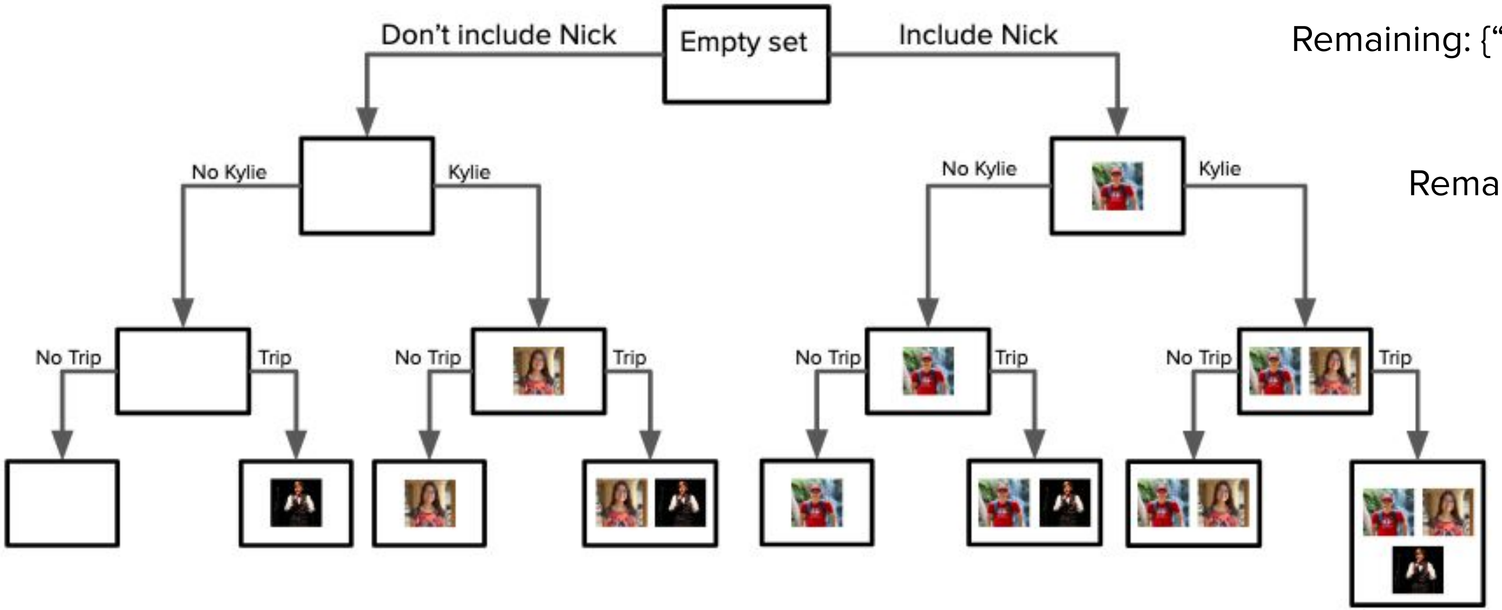
Remaining: {"Nick", "Kylie", "Trip"}

Remaining: {"Kylie", "Trip"}

Remaining: {"Trip"}

Remaining: {}

Decision tree



Remaining: {"Nick", "Kylie", "Trip"}

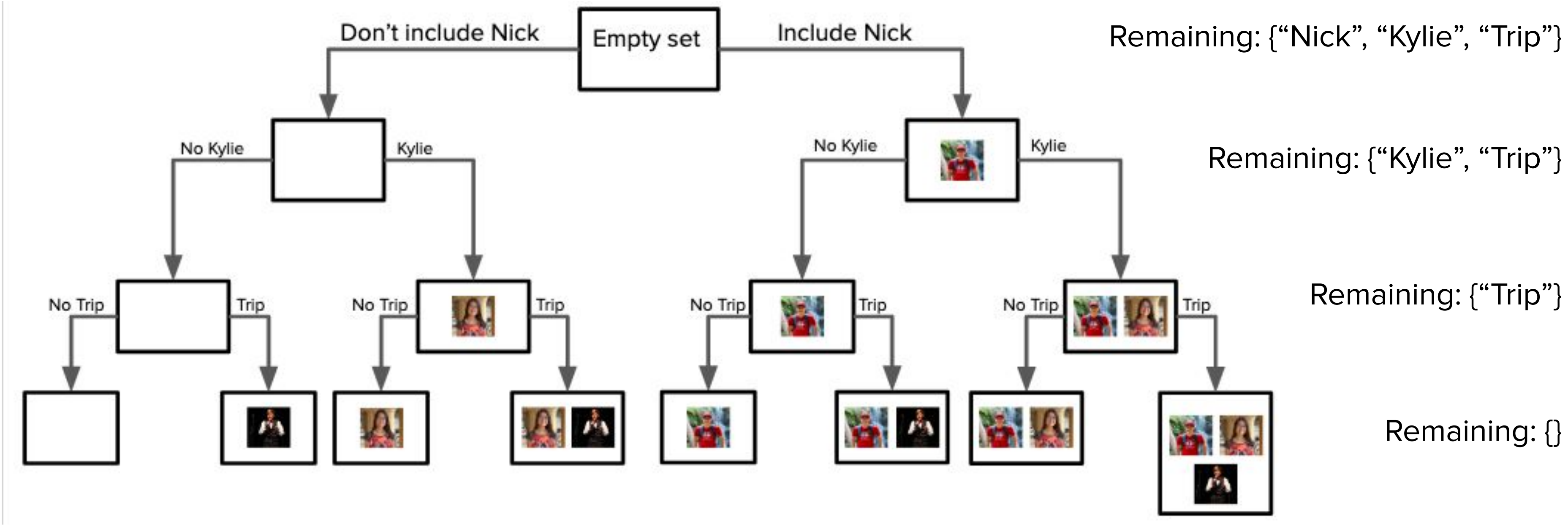
Remaining: {"Kylie", "Trip"}

Remaining: {"Trip"}

Remaining: {}

Base case: No people remaining to choose from!

Decision tree



Recursive case: Pick someone in the set. Choose to include or not include them.

Let’s code it!

Takeaways

- This is our first time seeing an explicit “unchoose” step
    - This is necessary because we’re passing sets by reference and editing them!

Takeaways

- This is our first time seeing an explicit “unchoose” step
  - This is necessary because we’re passing sets by reference and editing them!

```
string elem = remaining.first();
// remove this element from possible choices
remaining = remaining - elem;
listSubsetsHelper(remaining, chosen); // do not add elem to chosen
chosen = chosen + elem;
listSubsetsHelper(remaining, chosen); // add elem to chosen
chosen = chosen - elem;
// add this element back to possible choices
remaining = remaining + elem;
```

Takeaways

- This is our first time seeing an explicit "unchoose" step
  - This is necessary because we're passing sets by reference and editing them!

Choose

```
string elem = remaining.first();
// remove this element from possible choices
remaining = remaining - elem;
listSubsetsHelper(remaining, chosen); // do not add elem to chosen
chosen = chosen + elem;
listSubsetsHelper(remaining, chosen); // add elem to chosen
chosen = chosen - elem;
// add this element back to possible choices
remaining = remaining + elem;
```

Takeaways

- This is our first time seeing an explicit “unchoose” step
  - This is necessary because we’re passing sets by reference and editing them!

Explore (part 1)

```
string elem = remaining.first();
// remove this element from possible choices
remaining = remaining - elem;
listSubsetsHelper(remaining, chosen); // do not add elem to chosen
chosen = chosen + elem;
listSubsetsHelper(remaining, chosen); // add elem to chosen
chosen = chosen - elem;
// add this element back to possible choices
remaining = remaining + elem;
```

Takeaways

- This is our first time seeing an explicit “unchoose” step
  - This is necessary because we’re passing sets by reference and editing them!

Explore (part 2)

```
string elem = remaining.first();
// remove this element from possible choices
remaining = remaining - elem;
listSubsetsHelper(remaining, chosen); // do not add elem to chosen
chosen = chosen + elem;
listSubsetsHelper(remaining, chosen); // add elem to chosen
chosen = chosen - elem;
// add this element back to possible choices
remaining = remaining + elem;
```

Takeaways

- This is our first time seeing an explicit “unchoose” step
  - This is necessary because we’re passing sets by reference and editing them!

```
string elem = remaining.first();
// remove this element from possible choices
remaining = remaining - elem;
listSubsetsHelper(remaining, chosen); // do not add elem to chosen
chosen = chosen + elem;
listSubsetsHelper(remaining, chosen); // add elem to chosen
chosen = chosen - elem;
// add this element back to possible choices
remaining = remaining + elem;
```

Explicit Unchoose (i.e. undo)

Takeaways

- This is our first time seeing an explicit “unchoose” step
    - This is necessary because we’re passing sets by reference and editing them!

Without this step, we could not explore the other side of the tree

```
string elem = remaining.first();
// remove this element from possible choices
remaining = remaining - elem;
listSubsetsHelper(remaining, chosen); // do not add elem to chosen
chosen = chosen + elem;
listSubsetsHelper(remaining, chosen); // add elem to chosen
chosen = chosen - elem;
// add this element back to possible choices
remaining = remaining + elem;
```

Takeaways

- This is our first time seeing an explicit “unchoose” step
  - This is necessary because we’re passing sets by reference and editing them!

- It’s important to consider not only decisions and options at each decision, but also to keep in mind what information you have to keep track of with each recursive call. This might help you define your base case.

Takeaways

- This is our first time seeing an explicit “unchoose” step
    - This is necessary because we’re passing sets by reference and editing them!

- It’s important to consider not only decisions and options at each decision, but also to keep in mind what information you have to keep track of with each recursive call. This might help you define your base case.

- The subset problem contains themes we’ve seen in backtracking recursion:
    - Building up solutions as we go down the decision tree
    - Using a helper function to abstract away implementation details

Summary

Backtracking recursion: **Exploring many possible solutions**

Overall paradigm: choose/explore/unchoose

Two ways of doing it

- **Choose explore undo**
  - Uses pass by reference; usually with large data structures
  - Explicit unchoose step by "undoing" prior modifications to structure
  - E.g. Generating subsets (one set passed around by reference to track subsets)
- **Copy edit explore**
  - Pass by value; usually when memory constraints aren't an issue
  - Implicit unchoose step by virtue of making edits to copy
  - E.g. Building up a string over time

Three use cases for backtracking

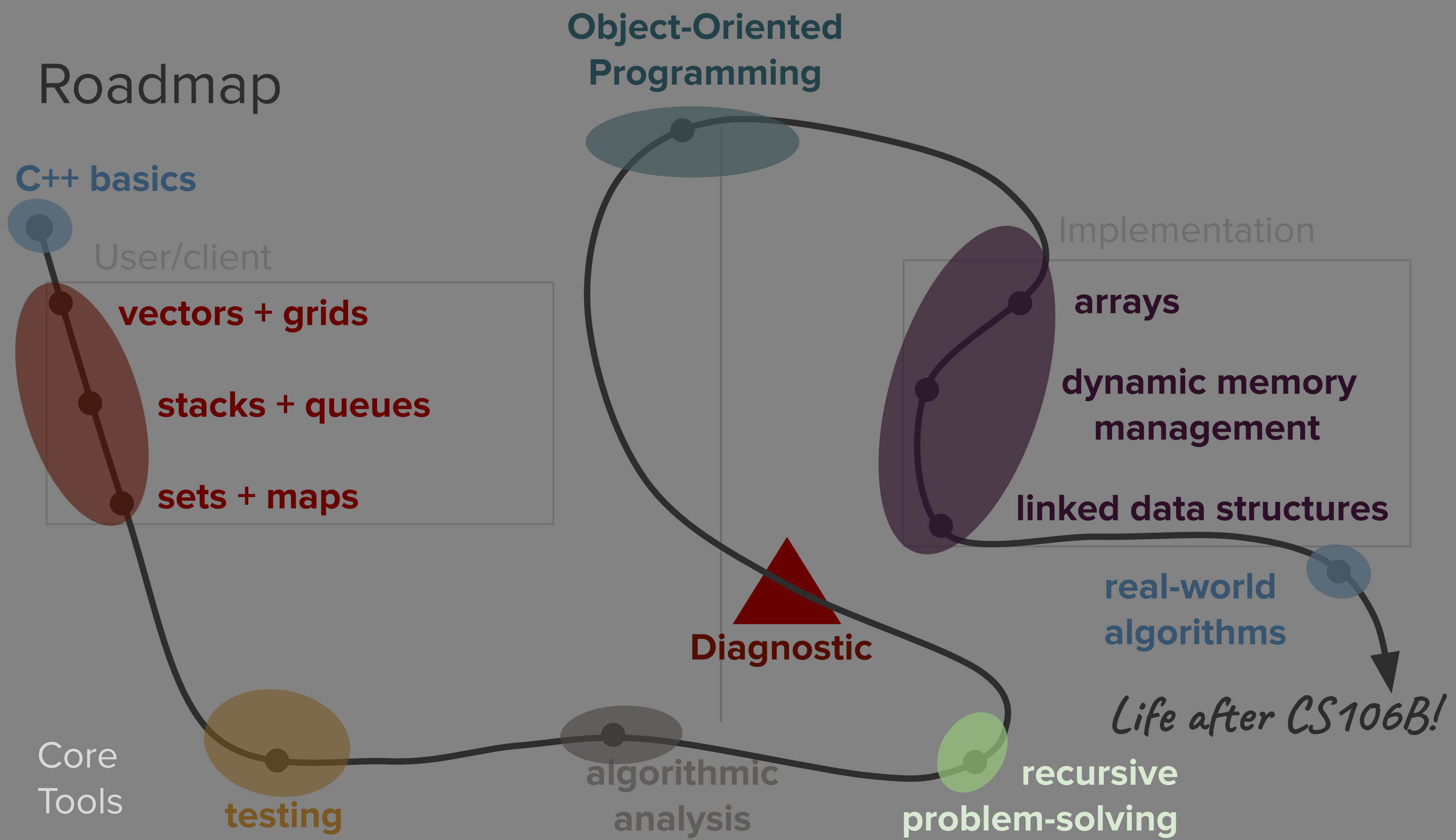
1. Generate/count all solutions (enumeration)
2. Find one solution (or prove existence)
3. Pick one best solution

General examples of things you can do:

- Permutations
- Subsets
- Combinations
- etc.

What’s next?

Roadmap


Object-Oriented
Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory
management
linked data structures
real-world
algorithms
Diagnostic
Life after CS106B!
Core
Tools
testing
algorithmic
analysis
recursive
problem-solving

More Recursive Backtracking