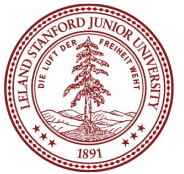
Recursive Backtracking Revisited

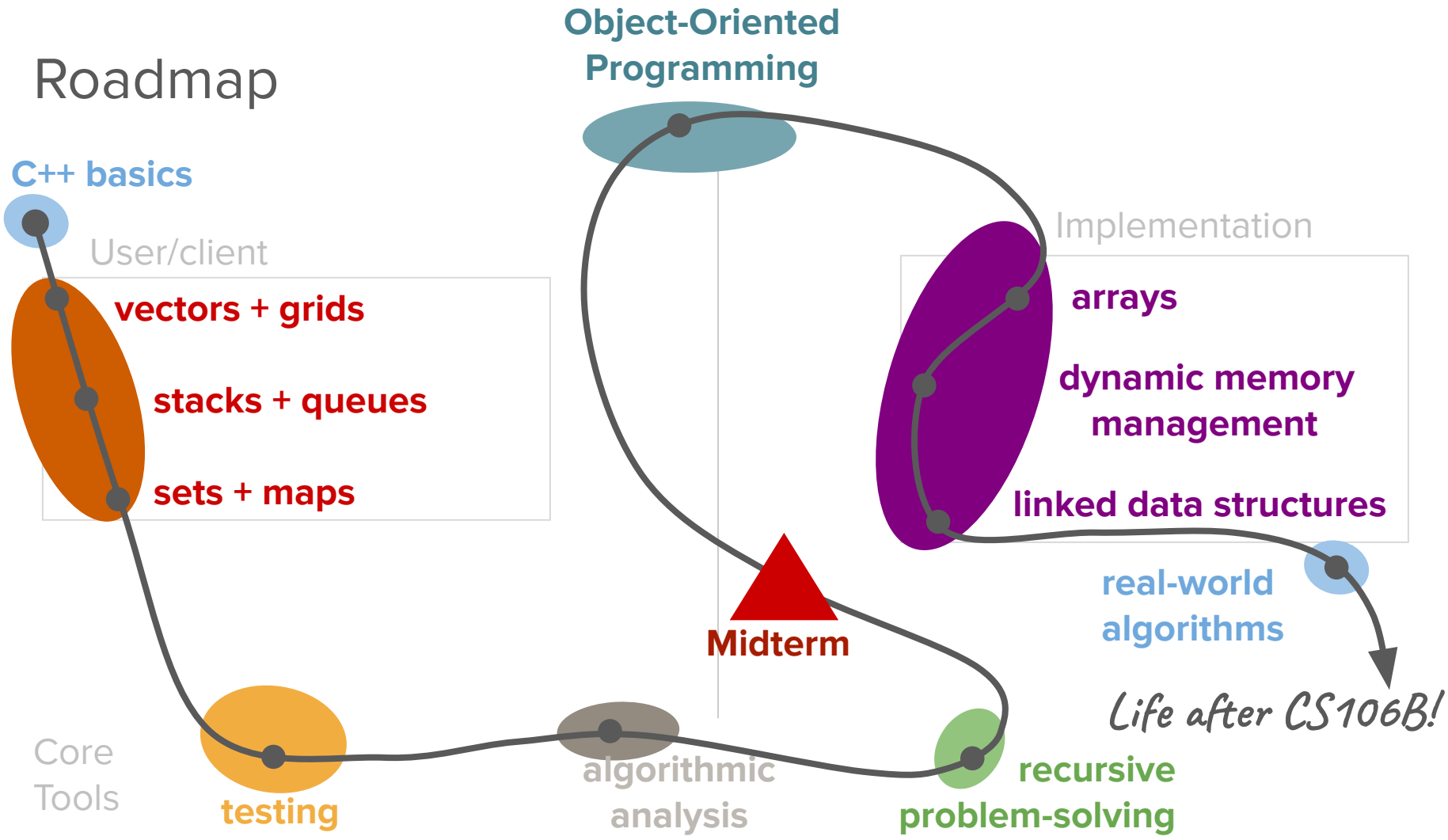
What has been your favorite part of the first 4 weeks of the course?

PollEv.com/cs106bpolls

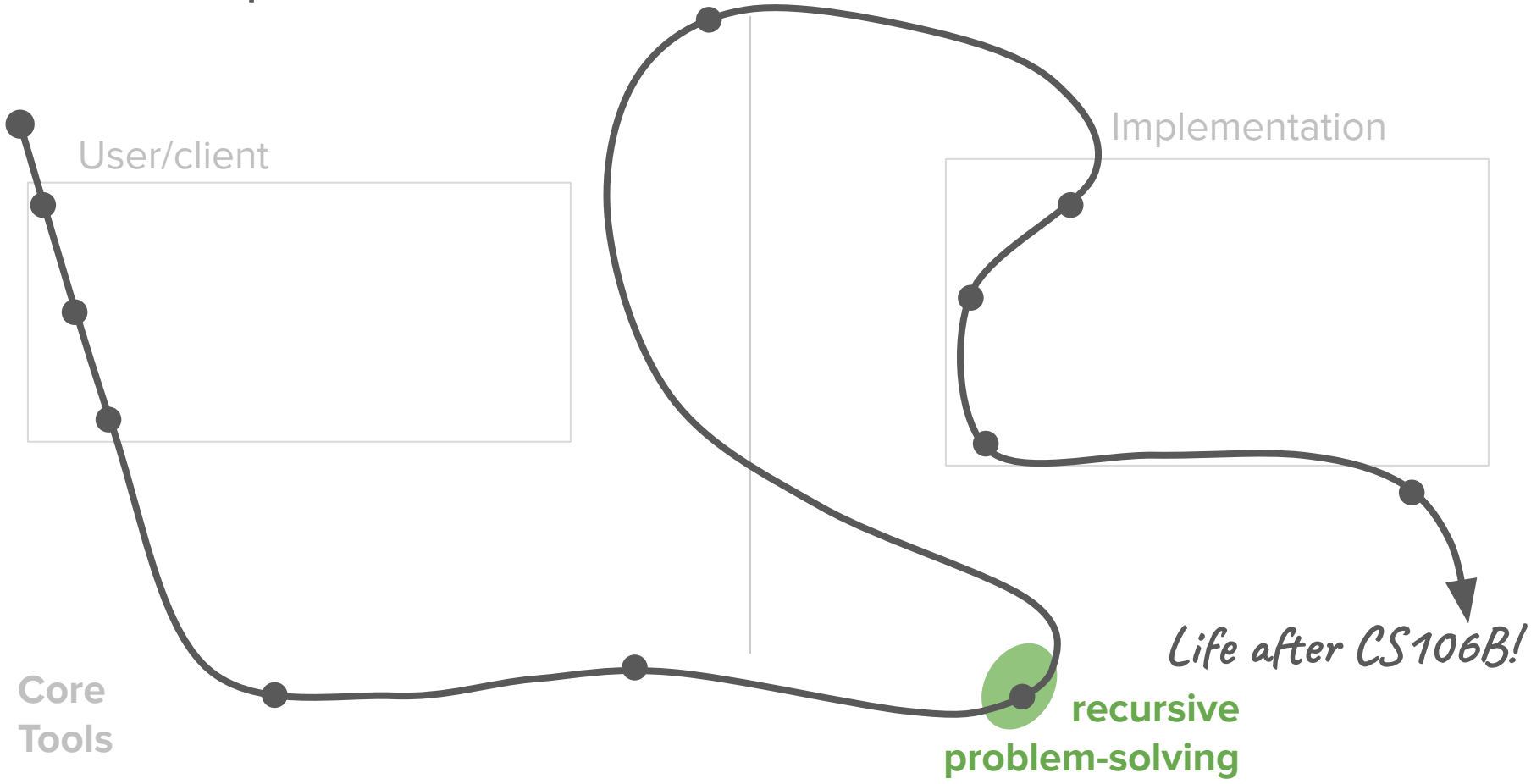


What has been your favorite part of the first half of the course?

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
Midterm
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Roadmap


User/client
Implementation
Life after CS106B!
Core
Tools
recursive
problem-solving

Today's question

What strategies should we use when solving recursive backtracking problems?

Today's topics

1. Review + Subsets

2. Recursive backtracking strategies

3. Practice applying strategies
   a. Making combinations
   b. Solving mazes with DFS

4. A brief intro to optimization (if time)

Review

(intro to recursive backtracking)

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
- Common pattern: choose/explore/unchoose

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

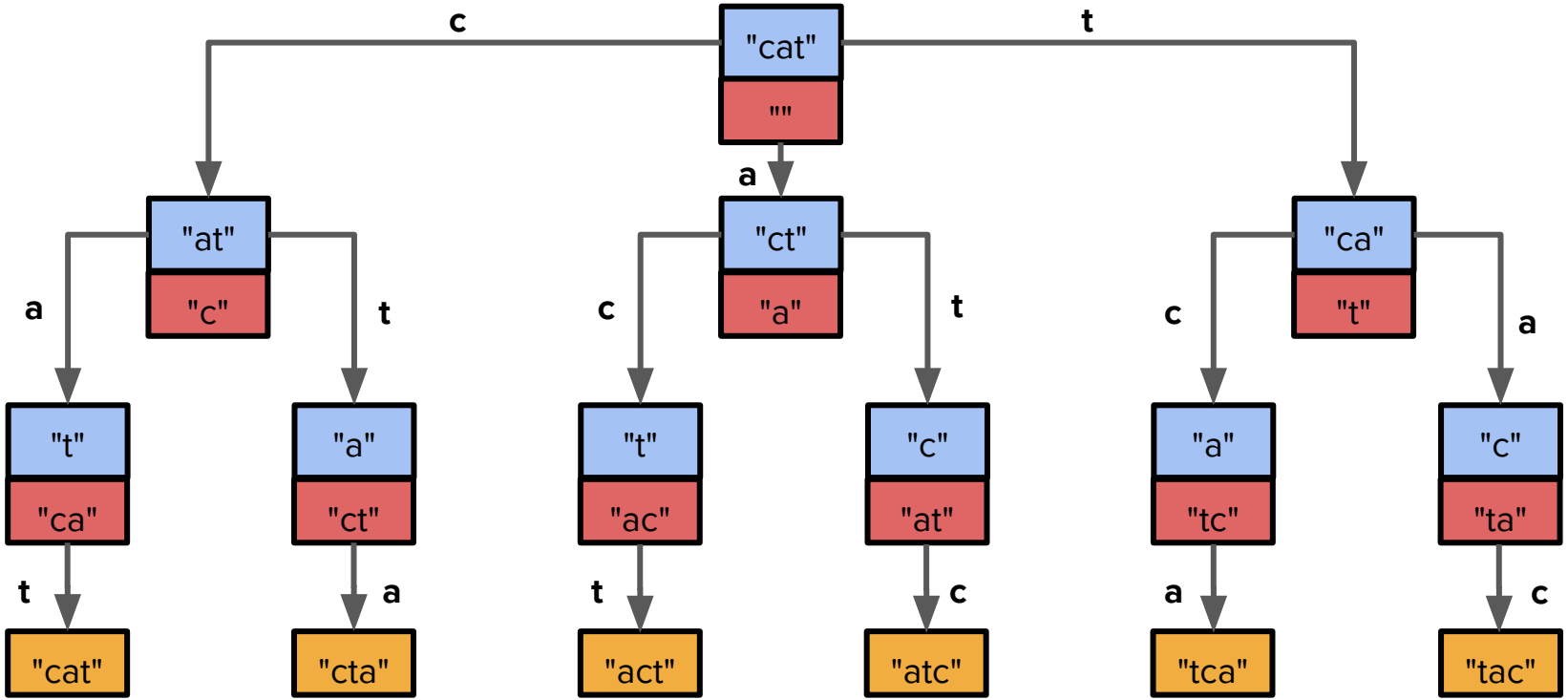
Word Scramble:
Finding all *permutations*

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 - we must use all the items and we care about their order**
  - Generating subsets
  - Generating combinations
  - And many, many more

Decisions yet to be made
Decisions made so far

Decision tree: Find all permutations of "cat"



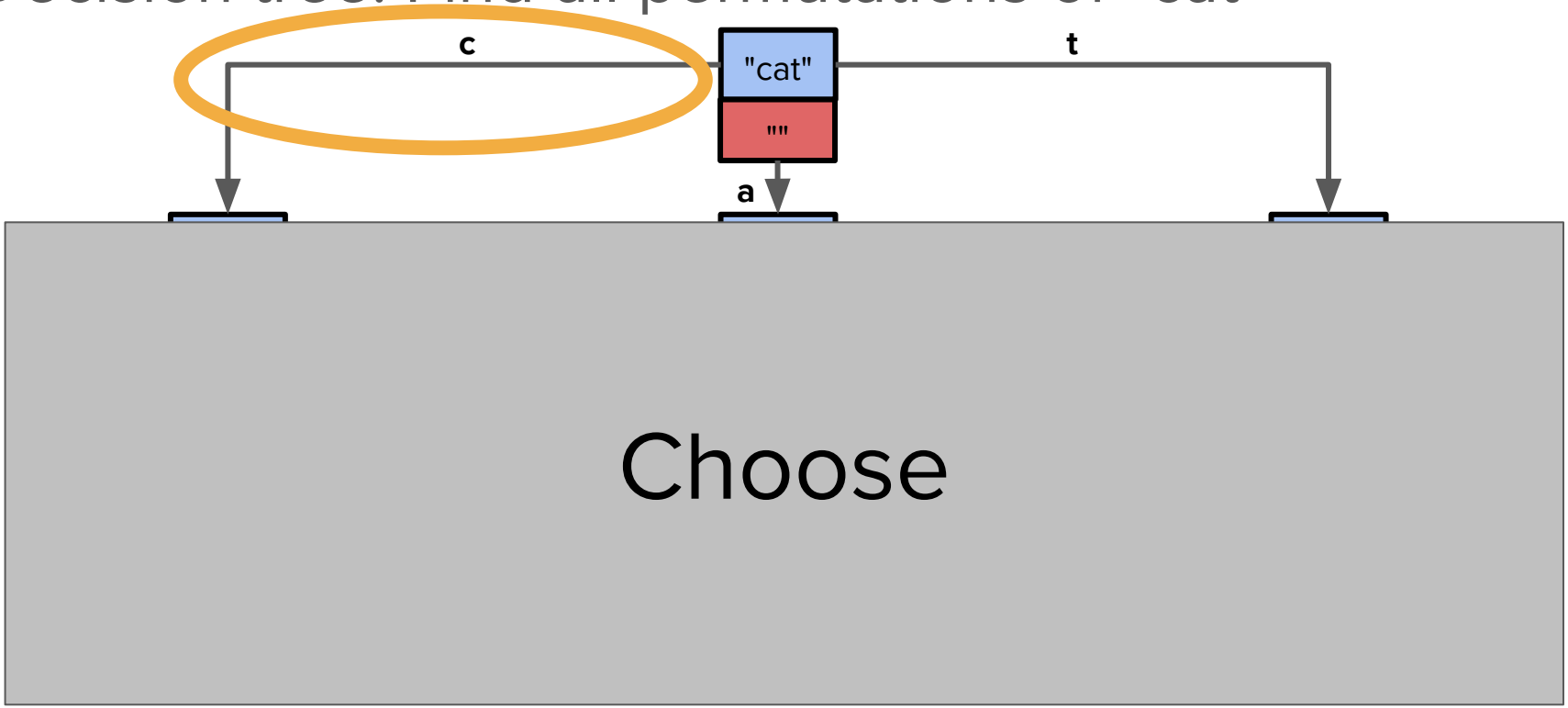
Decisions yet to be made
Decisions made so far

Decision tree: Find all permutations of "cat"

Remember:
choose/explore/unchoose!

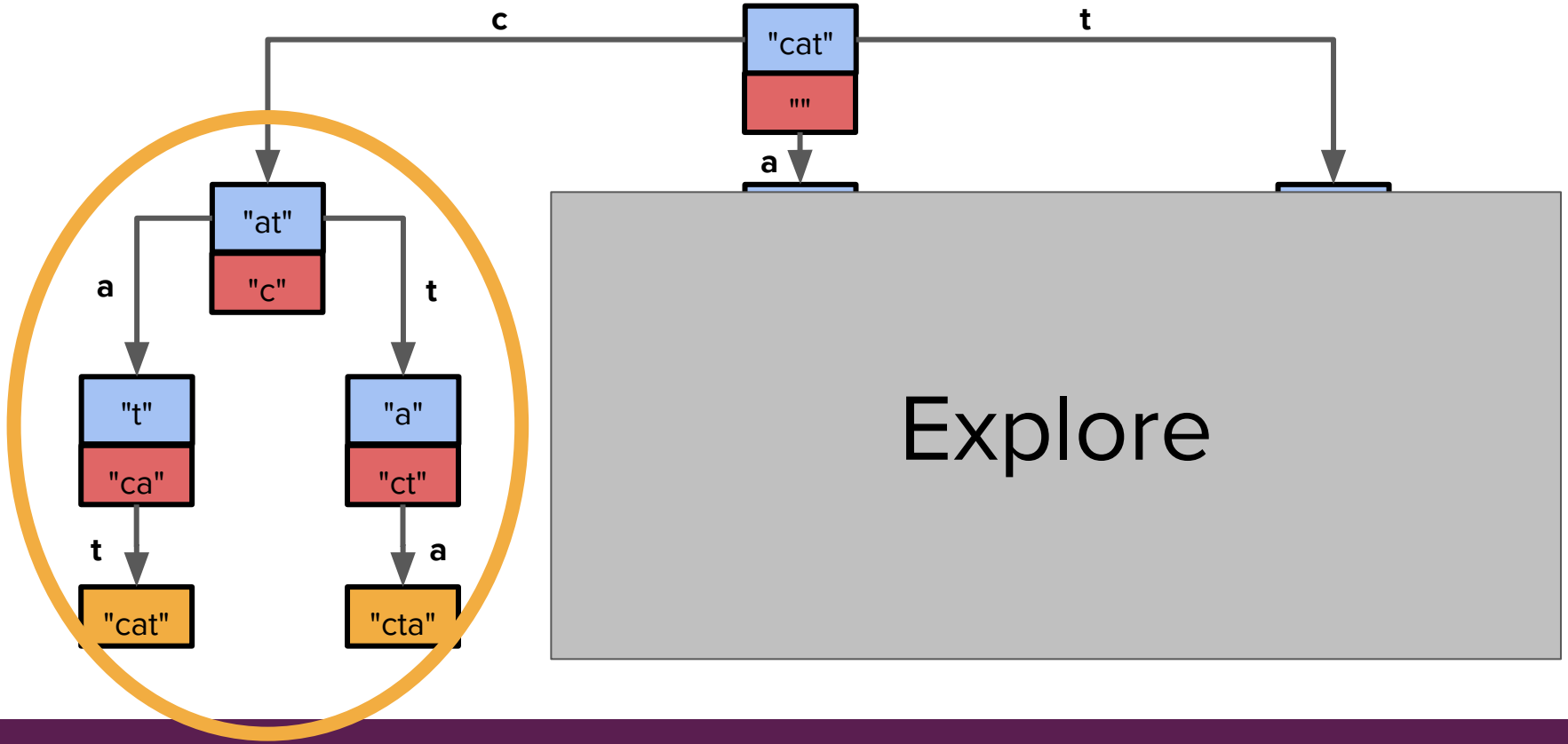
Decisions yet to be made
Decisions made so far

Decision tree: Find all permutations of "cat"



Decisions yet to be made
Decisions made so far

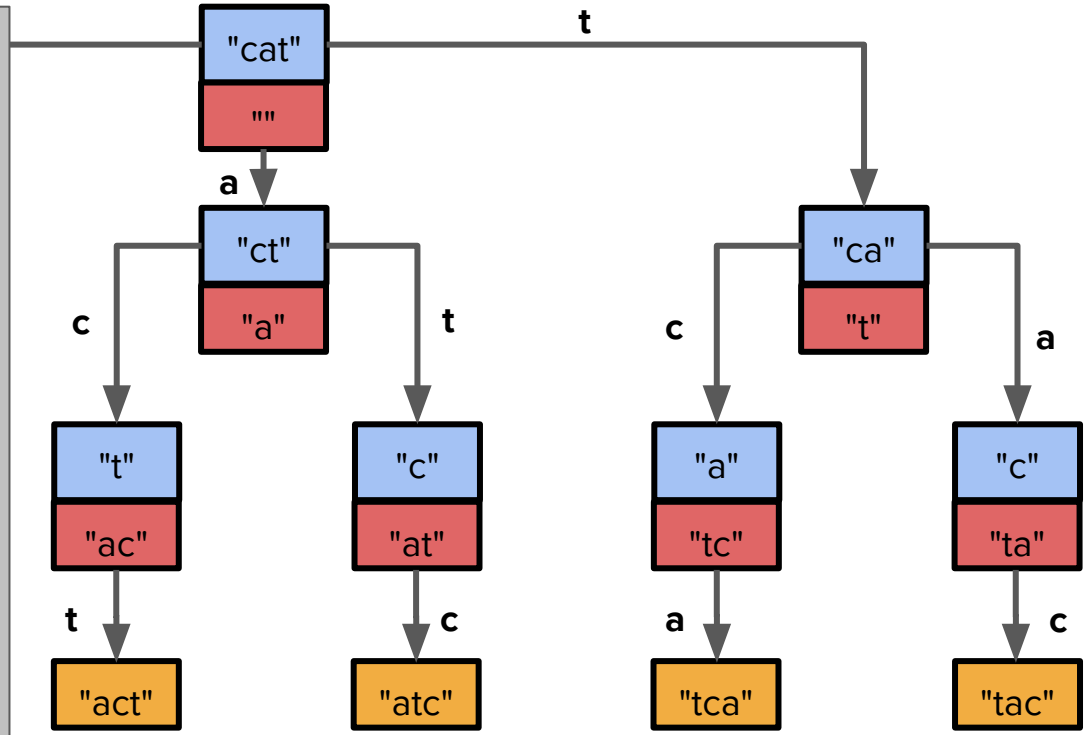
Decision tree: Find all permutations of "cat"



Decisions yet to be made
Decisions made so far

Decision tree: Find all permutations of "cat"

Unchoose



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

```
void listPermutationsHelper(string remaining, string soFar) {
    // base case: if no letters remaining, then we're done
    if (remaining == "") {
        cout << soFar << endl;
    }
    // recursive case: for every letter, append to soFar and then recurse
    else {
        for (int i = 0; i < remaining.length(); i++) {
            char currentLetter = remaining[i];
            string rest = remaining.substr(0, i) + remaining.substr(i+1);
            listPermutationsHelper(rest, soFar + currentLetter);
        }
    }
}

void listPermutations(string s){
    /* TODO: Fill in this function! */
    listPermutationsHelper(s, "");
}
```

Takeaways

- The specific model of the general **"choose / explore / unchoose"** pattern in backtracking recursion that we applied to generate permutation can be thought of as **"copy, edit, recurse"**
  - We make new copies of the **`remaining`** and **`soFar`** strings every time we recurse.
  - We're going to see a different method of doing choose/explore/unchoose when we work with ADTs and don't want to make lots of copies.

Takeaways

- The specific model of the general **"choose / explore / unchoose"** pattern in backtracking recursion that we applied to generate permutation 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**
    - We created a helper function that had parameters for the **`remaining`** and **`soFar`** strings.
    - Calling the helper function with an initial empty parameter that gets built up (**`soFar`**) will be a common technique for recursive backtracking.

Takeaways

- The specific model of the general **"choose / explore / unchoose"** pattern in backtracking recursion that we applied to generate permutation 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
  - The number of branches in our tree at each level depended on how many letters we had left.
  - We combined iteration and recursion to help us make the right number of recursive calls.

Takeaways

- The specific model of the general **"choose / explore / unchoose"** pattern in backtracking recursion that we applied to generate permutation 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**

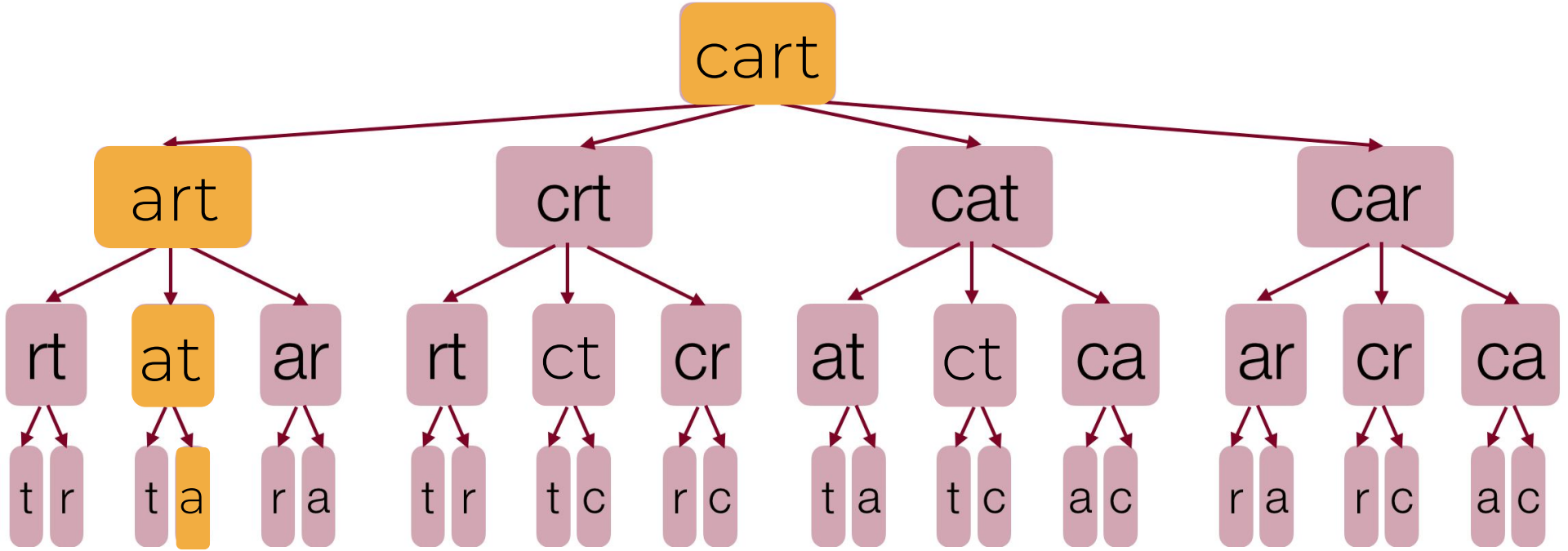
- Backtracking recursion can have **variable branching factors** at each level

Shrinkable Words:
Seeing if a solution exists

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 - all possible shrinkable words within a word**

What defines our shrinkable decision tree?



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

```
bool isShrinkable(Lexicon& lex, string word) {
    /* TODO: Fill in isShrinkable function! */
    // base case:
    // 1) if not valid english word, return false
    // 2) if single letter word, return true
    if (!lex.contains(word)) {
        return false;
    }
    if (word.length() == 1) {
        return true;
    }
    // recursive case:
    // try removing each letter one at a time
    // if we find a shrinkable word recursively, return true
    // if we find nothing at all, return false
    for (int i = 0; i < word.length(); i++) {
        string remainingWord = word.substr(0, i) + word.substr(i + 1);
        if (isShrinkable(lex, remainingWord)) {
            return true;
        }
    }
    return false;
}
```

Takeaways

- This is another example of **copy-edit-recurse** with **variable branching factors** at each level!

Takeaways

- This is another example of **copy-edit-recurse** with **variable branching factors** at each level!

- 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;
```

Takeaways

- This is another example of **copy-edit-recurse** with **variable branching factors** at each level!

- In this problem, we're using backtracking to **find if a solution exists**.

- We **don't need a helper function** because we don't need to keep track of the letters we removed, only the ones we have left.

Takeaways

- This is another example of **copy-edit-recurse** with **variable branching factors** at each level!
- In this problem, we're using backtracking to **find if a solution exists**.
- We **don't need a helper function** because we don't need to keep track of the letters we removed, only the ones we have left.
- Because we stop as soon as a word isn't valid or shrinkable, we have **multiple base cases** (in addition to multiple recursive calls).

Making teams:
Generating all possible
subsets

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

Subsets (for grading the midterm)

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!

Subsets (cont.)

Subsets

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

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

We noticed a pattern...

Subsets

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

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

Half the subsets contain “Jenny”

Subsets

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

```
{}
{“Jenny”}
{“Kylie”}
{“Trip”}
{“Jenny”, “Kylie”}
{“Jenny”, “Trip”}
{“Kylie”, “Trip”}
{“Jenny”, “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:

```
{}
{“Jenny”}
{“Kylie”}
{“Trip”}
{“Jenny”, “Kylie”}
{“Jenny”, “Trip”}
{“Kylie”, “Trip”}
{“Jenny”, “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:

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

Half the subsets that contain “Trip” also contain “Jenny”

Subsets

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

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

Half the subsets that contain both “Trip” and “Jenny” contain “Kylie”

Subsets

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

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

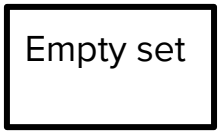
What defines our subsets decision tree?

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

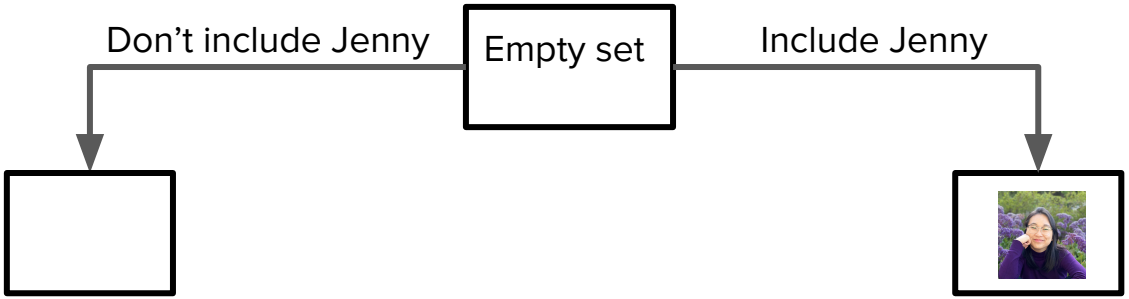
- **Options** at each decision (branches from each node):
  - Include element
  - Don’t include element

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

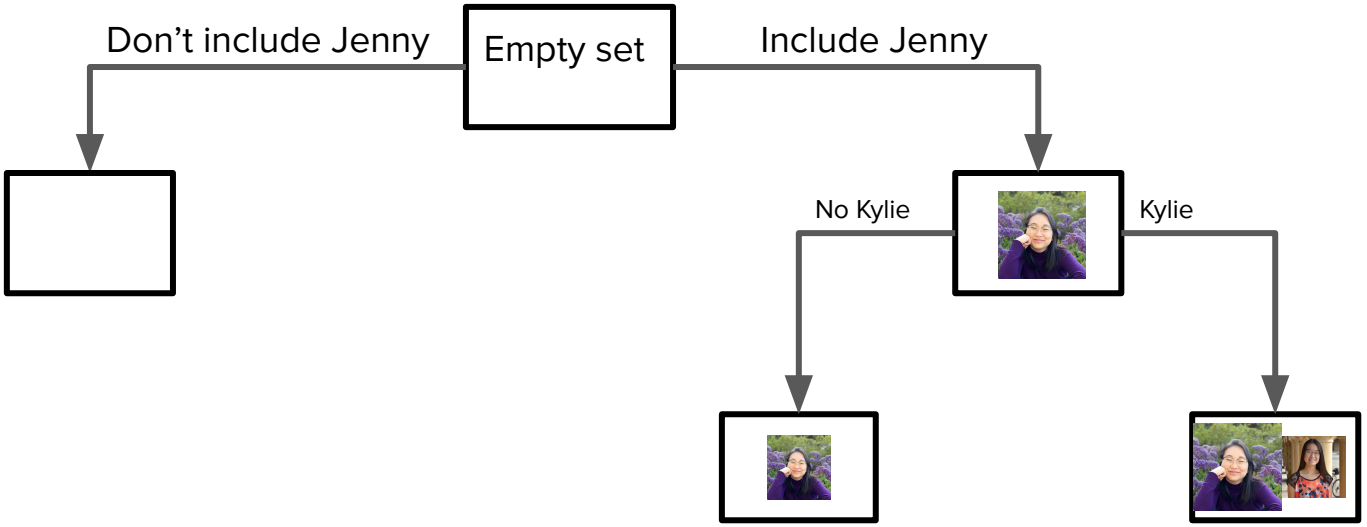
Decision tree



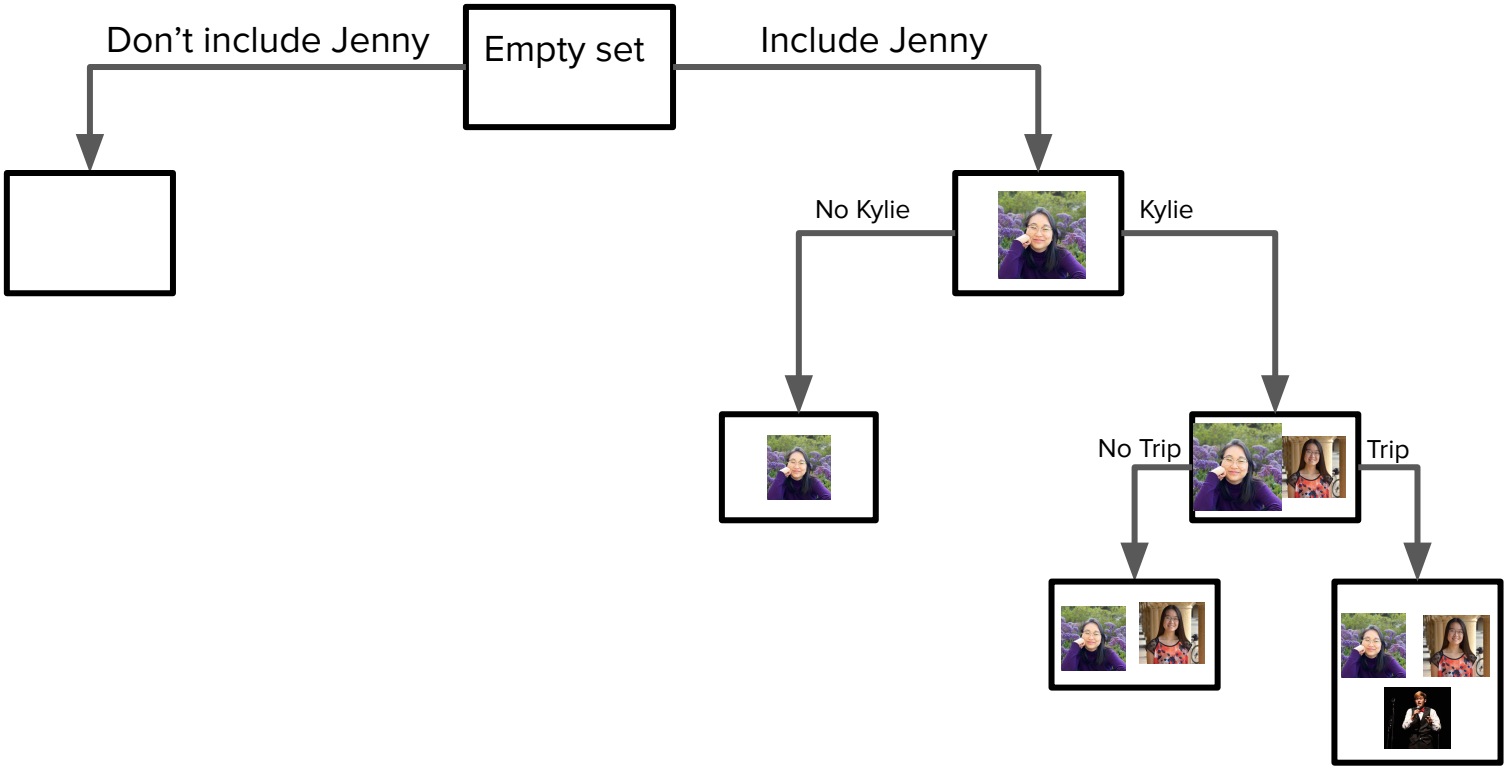
Decision tree



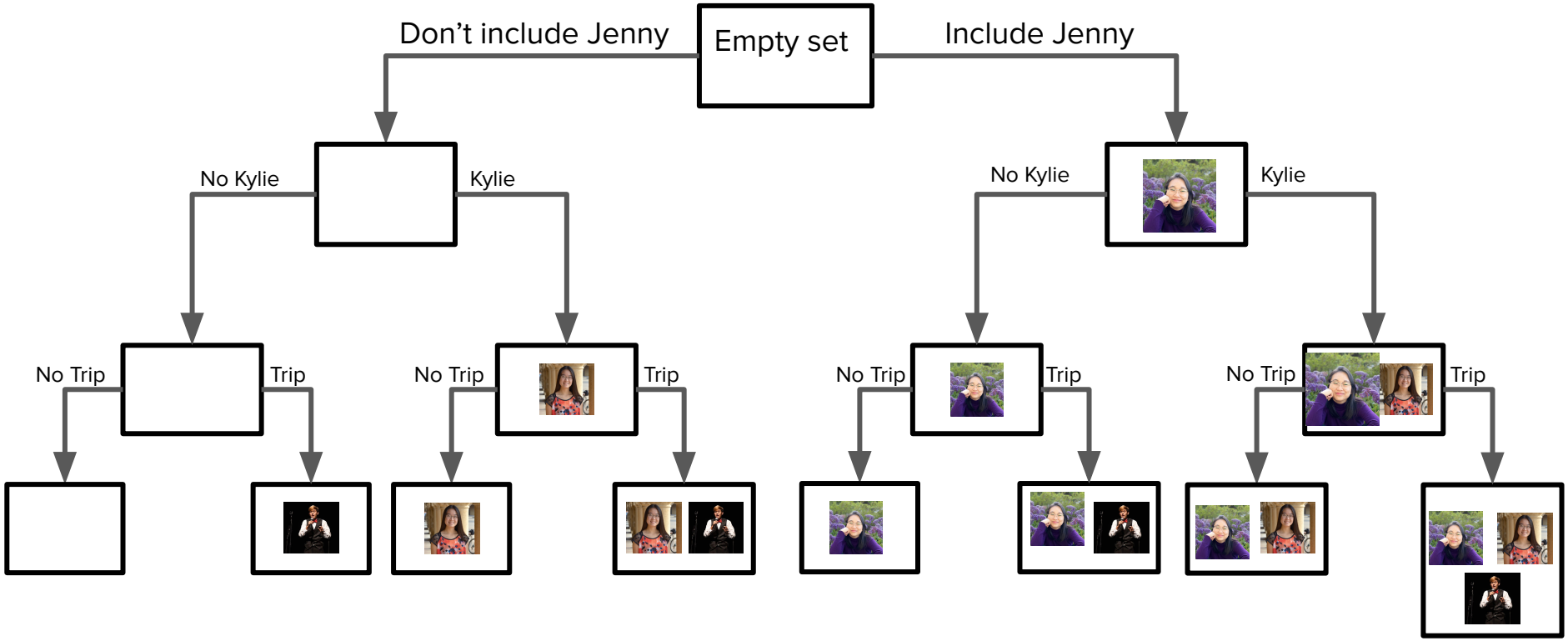
Decision tree



Decision tree



Decision tree



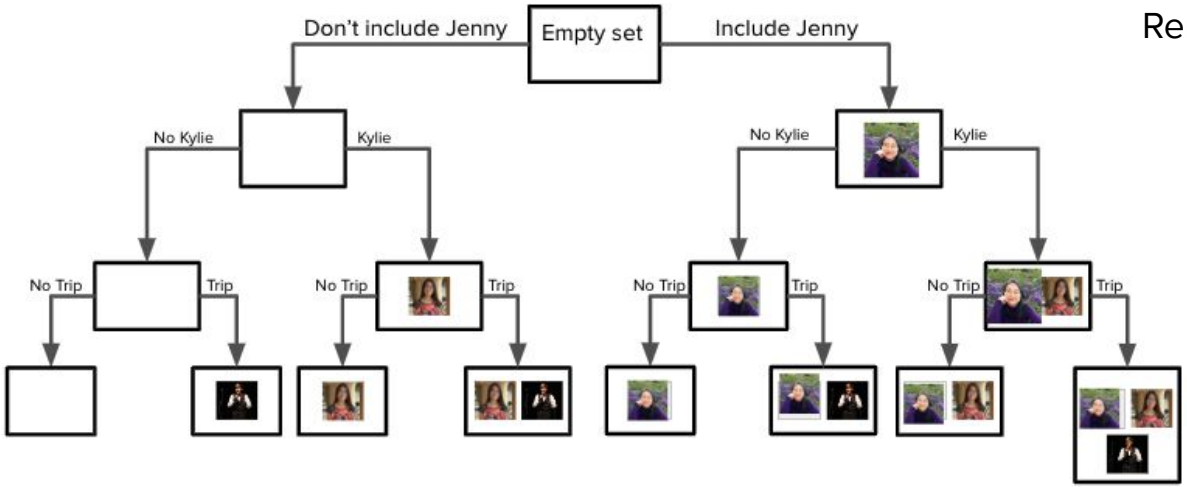
What defines our subsets decision tree?

- **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

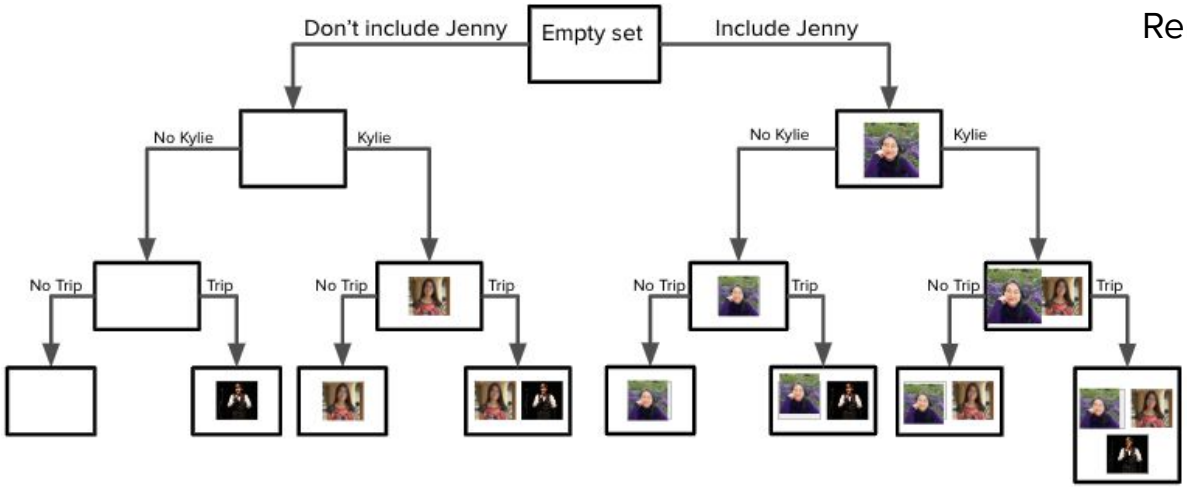
- Information we need to store along the way:
  - The set you’ve built so far
  - **The remaining elements in the original set**

Decision tree



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

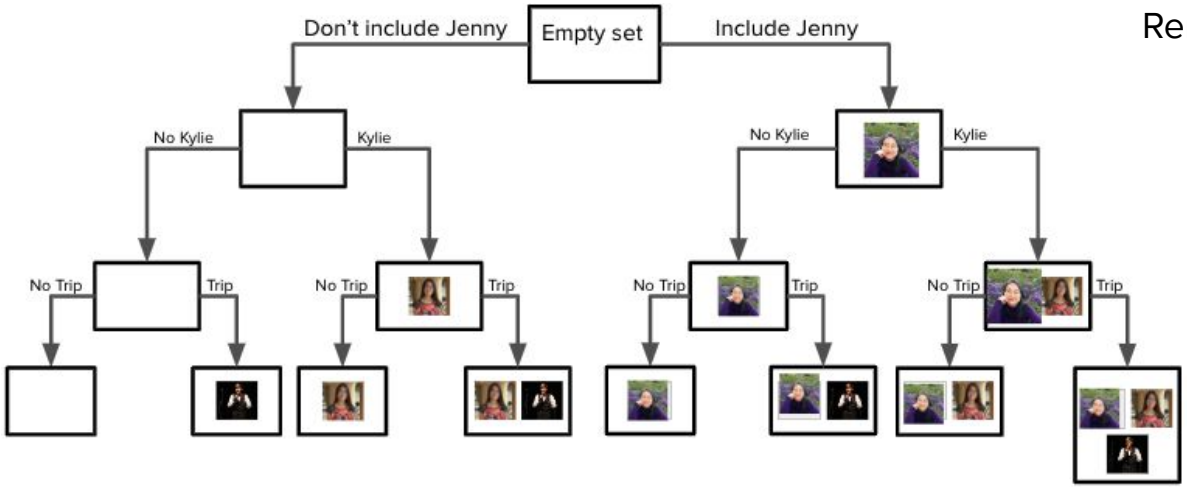
Decision tree



Remaining: {“Jenny”, “Kylie”, “Trip”}

Remaining: {“Kylie”, “Trip”}

Decision tree

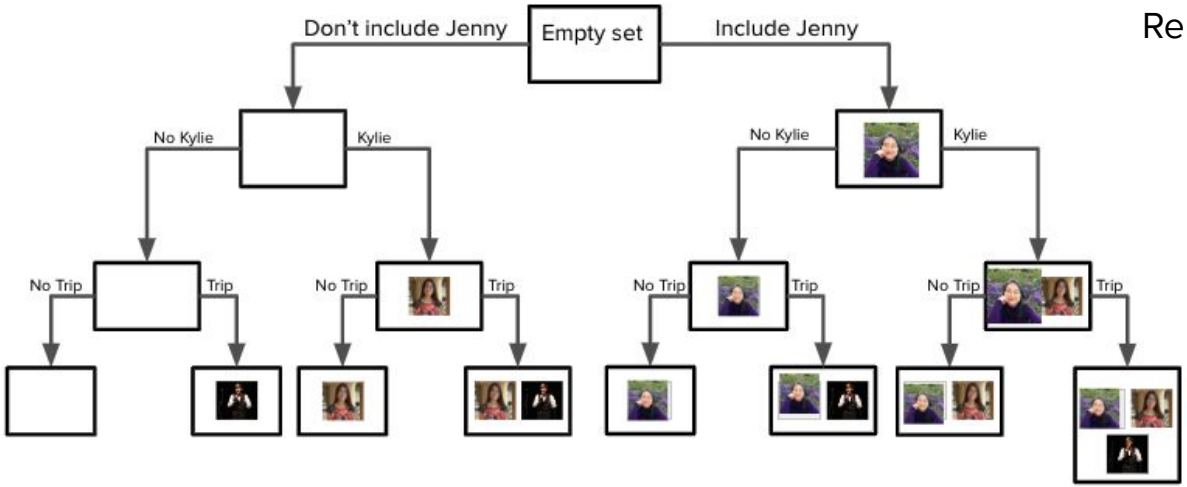


Remaining: {“Jenny”, “Kylie”, “Trip”}

Remaining: {“Kylie”, “Trip”}

Remaining: {“Trip”}

Decision tree



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

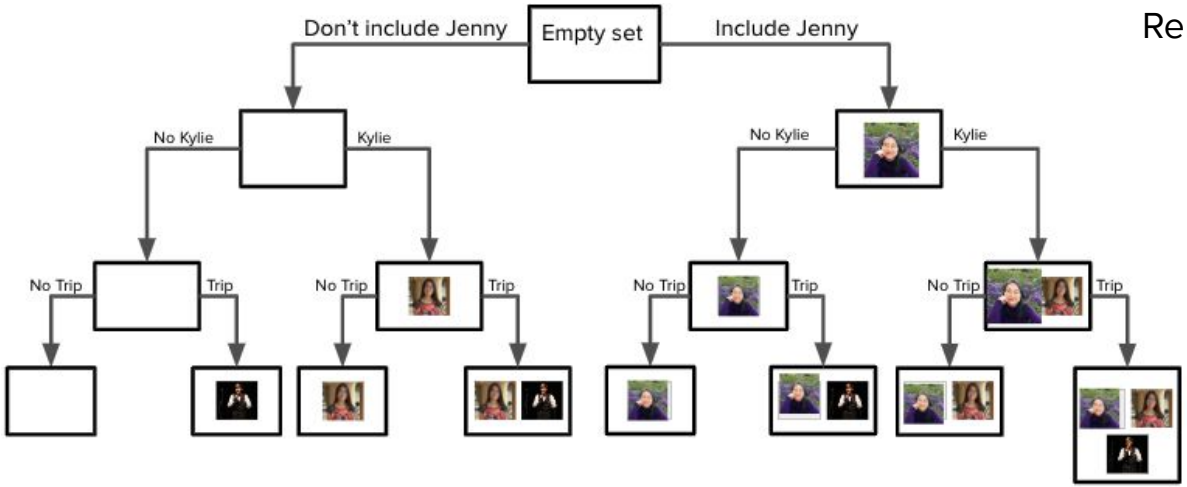
Remaining: {"Kylie", "Trip"}

Remaining: {"Trip"}

Remaining: {}

Base case: No people remaining to choose from!

Decision tree



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

Remaining: {"Kylie", "Trip"}

Remaining: {"Trip"}

Remaining: {}

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!

Explicit Unchoose (i.e. undo)

```
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!

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!

- Note the difference in the options at each step in this problem vs. the previous two.
  - It’s also important to keep in mind what information you have to keep track of with each recursive call. This might help you define your base case (e.g. having an empty remaining set).

Takeaways

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

- Note the difference in the options at each step in this problem vs. the previous two.
  - It’s also important to keep in mind what information you have to keep track of with each recursive call. This might help you define your base case (e.g. having an empty remaining set).

- This was our first example using ADTs with recursion, and we’ll see more today!

Backtracking recursion: **Exploring many possible solutions**

Two methods of choose/explore/unchoose

- **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

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 process should we use to solve recursive backtracking problems?

Solving backtracking recursion problems

- Which of our three use cases does our problem fall into? (generate/count all solutions, find one solution/prove its existence, pick one best solution)
- What are we building up as our “many possibilities” in order to find our solution?

- What’s the provided function prototype and requirements? Do we need a helper function?
  - What are we returning as our solution?
  - Do we care about returning or keeping track of the path we took to get to our solution? If yes, what parameters are we already given and what others might be useful?

- What are our base and recursive cases?
  - What does my decision tree look like? (decisions, options, what to keep track of)
  - In addition to what we’re building up, are there any additional constraints on our solutions?
  - Does it make sense to use choose/explore/undo OR copy/edit/recurse for the recursion?

Solving backtracking recursion problems

- Which of our three use cases does our problem fall into? (generate/count all solutions, find one solution/prove its existence, pick one best solution)
- What are we building up as our “many possibilities” in order to find our solution? **(subsets, permutations, or something else)**

- What’s the provided function prototype and requirements? Do we need a helper function?
  - What are we returning as our solution? **(a boolean, void but printing out a string or ADT)**
  - Do we care about returning or keeping track of the path we took to get to our solution? If yes, what parameters are we already given and what others might be useful? (**sets**, **strings**)

- What are our base and recursive cases?
  - What does my decision tree look like? (decisions, options, what to keep track of)
  - In addition to what we’re building up, are there any additional constraints on our solutions?
  - Does it make sense to use choose/explore/undo OR copy/edit/recurse for the recursion?

Combinations

Creating fixed-size teams: Generating all possible combinations

Subsets vs. Combinations

- Our goal: We want to pick a combination of 5 graders out of a group of 8.
  - More useful than our generating subsets solution!

Subsets vs. Combinations

- Our goal: We want to pick a combination of 5 graders out of a group of 8.

- This sounds very similar to the problem we solved when we generated subsets – these 5 graders would be a subset of the overall group of 8.

Subsets vs. Combinations

- Our goal: We want to pick a combination of 5 graders out of a group of 8.

- This sounds very similar to the problem we solved when we generated subsets – these 5 graders would be a subset of the overall group of 8.

- What distinguishes a combination from a subset?
  - Combinations always have a specified **size,** unlike subsets (which can be any size)
  - We can think of combinations as **"subsets with constraints"**

Subsets vs. Combinations

- Our goal: We want to pick a combination of 5 graders out of a group of 8.
- This sounds very similar to the problem we solved when we generated subsets – these 5 graders would be a subset of the overall group of 8.
- What distinguishes a combination from a subset?
  - Combinations always have a specified **size**, unlike subsets (which can be any size)
  - We can think of combinations as **"subsets with constraints"**
- Could we use the code from before, generate all subsets, and then filter out all those of size 5?
  - We could, but that would be inefficient. Let's develop a better approach for combinations!

How do we approach this problem?

Solving backtracking recursion problems

- Which of our three use cases does our problem fall into? (generate/count all solutions, find one solution/prove its existence, pick one best solution)
- What are we building up as our “many possibilities” in order to find our solution?

- What’s the provided function prototype and requirements? Do we need a helper function?
  - What are we returning as our solution?
  - Do we care about returning or keeping track of the path we took to get to our solution? If yes, what parameters are we already given and what others might be useful?

- What are our base and recursive cases?
  - What does my decision tree look like? (decisions, options, what to keep track of)
  - In addition to what we’re building up, are there any additional constraints on our solutions?
  - Does it make sense to use choose/explore/undo OR copy/edit/recurse for the recursion?

Solving backtracking recursion problems

- **Which of our three use cases does our problem fall into? (generate/count all solutions, find one solution/prove its existence, pick one best solution)**
- **What are we building up as our “many possibilities” in order to find our solution?**

- What’s the provided function prototype and requirements? Do we need a helper function?
  - What are we returning as our solution?
  - Do we care about returning or keeping track of the path we took to get to our solution? If yes, what parameters are we already given and what others might be useful?

- What are our base and recursive cases?
  - What does my decision tree look like? (decisions, options, what to keep track of)
  - In addition to what we’re building up, are there any additional constraints on our solutions?
  - Does it make sense to use choose/explore/undo OR copy/edit/recurse for the recursion?

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

Solving backtracking recursion problems

- Which of our three use cases does our problem fall into? (generate/count all solutions, find one solution/prove its existence, pick one best solution)
- What are we building up as our “many possibilities” in order to find our solution?

- **What’s the provided function prototype and requirements? Do we need a helper function?**
  - What are we returning as our solution?
  - Do we care about returning or keeping track of the path we took to get to our solution? If yes, what parameters are we already given and what others might be useful?

- What are our base and recursive cases?
  - What does my decision tree look like? (decisions, options, what to keep track of)
  - In addition to what we’re building up, are there any additional constraints on our solutions?
  - Does it make sense to use choose/explore/undo OR copy/edit/recurse for the recursion?

What are we returning as our solution?

- Each combination of **k** graders can be represented as a **`Set<string>`**.

- In our string examples, we were content with just printing out all solutions. But what if we wanted to store all of them to be able to do something with them later?

- We want to return a container holding all possible combinations:

`Set<Set<string>>`

It's not that unusual to see containers nested this way!

What are we returning as our solution?

- Each combination of **k** graders can be represented as a **Set<string>**.
- In our string examples, we were content with just printing out all solutions. But what if we wanted to store all of them to be able to do something with them later?

```
Set<Set<string>> combinationsOf(Set<string>& graders, int k)
```

Do we need a helper function?

```
Set<Set<string>> combinationsOf(Set<string>& graders, int k)
```

Attendance ticket:

https://tinyurl.com/combinationsHelper

Please don’t send this link to students who are not here. It’s on your honor!

Do we need a helper function?

```
Set<Set<string>> combinationsOf(Set<string>& graders, int k)
```

Do we need a helper function?

```
Set<Set<string>> combinationsOf(Set<string>& graders, int k)
```

We'll need to keep track of a current set of graders as we're building up each possible set of strings. (We need a helper!)

Do we need a helper function?

```
Set<Set<string>> combinationsOf(Set<string>& graders, int k)
```

We'll need to keep track of a current set of graders as we're building up each possible set of strings. (We need a helper!)

```
Set<Set<string>> combinationsHelper(Set<string>& remaining, int k, Set<string>&
chosen)
```

Solving backtracking recursion problems

- Which of our three use cases does our problem fall into? (generate/count all solutions, find one solution/prove its existence, pick one best solution)
- What are we building up as our “many possibilities” in order to find our solution?
- What’s the provided function prototype and requirements? Do we need a helper function?

An exercise for you to try at home!

(solution is in the lecture code)

- **What are our base and recursive cases?**
  - What does my decision tree look like? (decisions, options, what to keep track of)
  - In addition to what we’re building up, are there any additional constraints on our solutions?
  - Does it make sense to use choose/explore/undo OR copy/edit/recurse for the recursion?

Announcements

Announcements

- Assignment 3 is due next Tuesday at 11:59pm PDT. The grace period ends Wednesday at 11:59pm PDT.

- Assignment 2 revisions will be due Friday, July 22 at 11:59pm PDT.

- The final project criteria have been released!
  - Please read through the entire page on the course website.
  - The timeline is suggested but there are three required milestones:
    - Project proposal: Sunday, July 24
    - Project write-up: Sunday, August 7
    - Project presentations: August 11-14

Revisiting mazes

Solving mazes with breadth-first search (BFS)

Solving mazes with breadth-first search (BFS)

Can we do it recursively?

How do we approach this problem?

Solving backtracking recursion problems

- **Which of our three use cases does our problem fall into? (generate/count all solutions, find one solution/prove its existence, pick one best solution)**
- **What are we building up as our “many possibilities” in order to find our solution?**

- What’s the provided function prototype and requirements? Do we need a helper function?
  - What are we returning as our solution?
  - Do we care about returning or keeping track of the path we took to get to our solution? If yes, what parameters are we already given and what others might be useful?

- What are our base and recursive cases?
  - What does my decision tree look like? (decisions, options, what to keep track of)
  - In addition to what we’re building up, are there any additional constraints on our solutions?
  - Does it make sense to use choose/explore/undo OR copy/edit/recurse for the recursion?

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 - all possible routes through a maze**

Solving backtracking recursion problems

- Which of our three use cases does our problem fall into? (generate/count all solutions, find one solution/prove its existence, pick one best solution)
- What are we building up as our “many possibilities” in order to find our solution?

- **What’s the provided function prototype and requirements? Do we need a helper function?**
  - What are we returning as our solution?
  - Do we care about returning or keeping track of the path we took to get to our solution? If yes, what parameters are we already given and what others might be useful?

- What are our base and recursive cases?
  - What does my decision tree look like? (decisions, options, what to keep track of)
  - In addition to what we’re building up, are there any additional constraints on our solutions?
  - Does it make sense to use choose/explore/undo OR copy/edit/recurse for the recursion?

Do we need a helper function?

- Recall our solveMaze prototype:

```
Stack<GridLocation> solveMaze(Grid<bool>& maze)
```

Do we need a helper function?

- Recall our solveMaze prototype:

```
Stack<GridLocation> solveMaze(Grid<bool>& maze)
```

- We need a helper function to keep track of our path through the maze!
  - Our helper function will have as **parameters**: the maze itself and the path we're building up.
  - We also want the helper to be able to tell us whether or not the maze is solvable – let's have it return a boolean.

Do we need a helper function?

- Recall our solveMaze prototype:

```
Stack<GridLocation> solveMaze(Grid<bool>& maze)
```

- We need a helper function to keep track of our path through the maze!

```
bool solveMazeHelper(Grid<bool>& maze,
                     Stack<GridLocation>& path)
```

Solving backtracking recursion problems

- Which of our three use cases does our problem fall into? (generate/count all solutions, find one solution/prove its existence, pick one best solution)
- What are we building up as our “many possibilities” in order to find our solution?

- What’s the provided function prototype and requirements? Do we need a helper function?
  - What are we returning as our solution?
  - Do we care about returning or keeping track of the path we took to get to our solution? If yes, what parameters are we already given and what others might be useful?

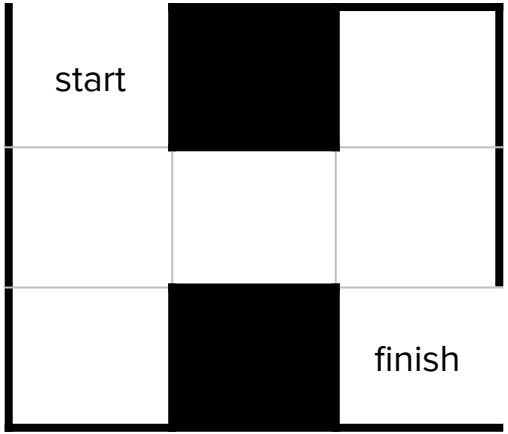
- **What are our base and recursive cases?**
  - What does my decision tree look like? (decisions, options, what to keep track of)
  - In addition to what we’re building up, are there any additional constraints on our solutions?
  - Does it make sense to use choose/explore/undo OR copy/edit/recurse for the recursion?

A **recursive** algorithm for solving mazes

- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we’re at the end of the maze

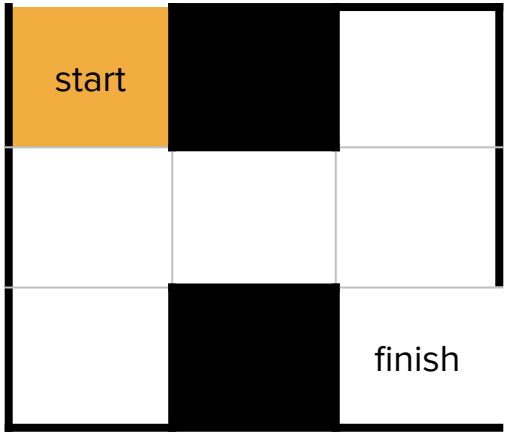
A **recursive** algorithm for solving mazes

- Start at the entrance



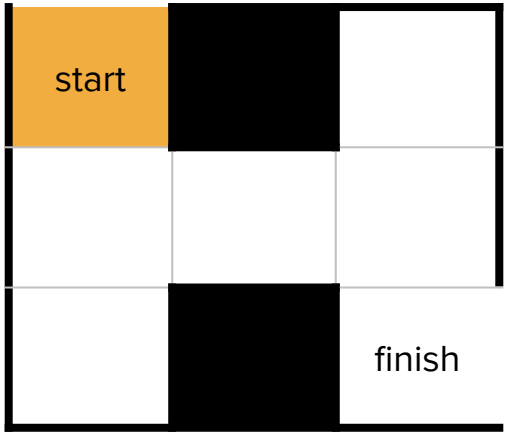
A **recursive** algorithm for solving mazes

- Start at the entrance
- Take one step North, South, East, or West



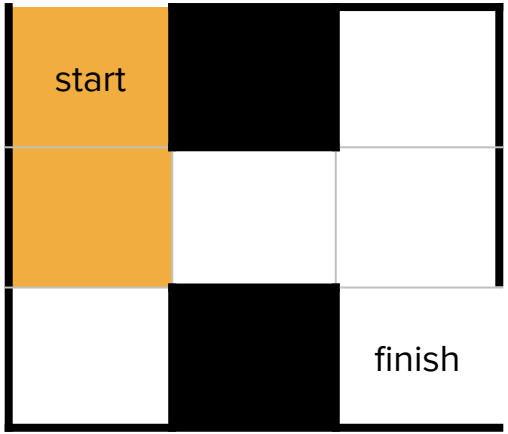
A **recursive** algorithm for solving mazes

- Start at the entrance
- Take one step ~~North~~, South, East, or West



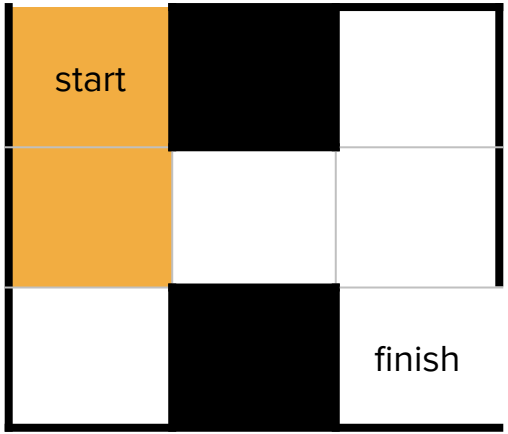
A **recursive** algorithm for solving mazes

- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze



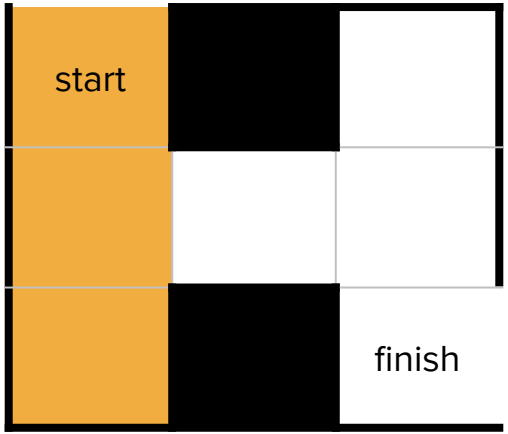
A **recursive** algorithm for solving mazes

- Start at the entrance
- Take one step ~~North~~, South, East, or West



A **recursive** algorithm for solving mazes

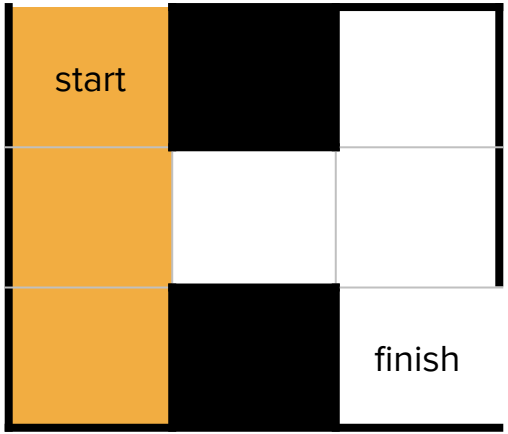
- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze



A **recursive** algorithm for solving mazes

- Start at the entrance
- Take one step ~~North, South, East, or West~~

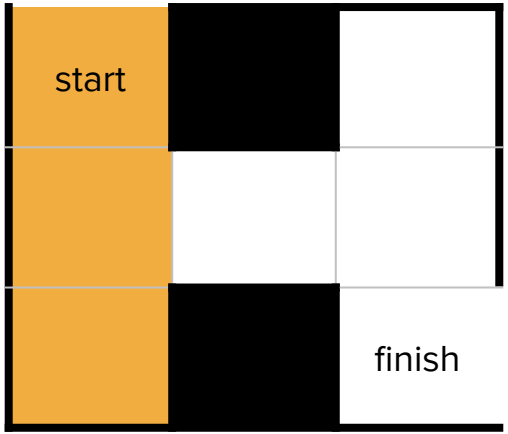
Dead end!
(cannot go North,
South, East, or West)



A **recursive** algorithm for solving mazes

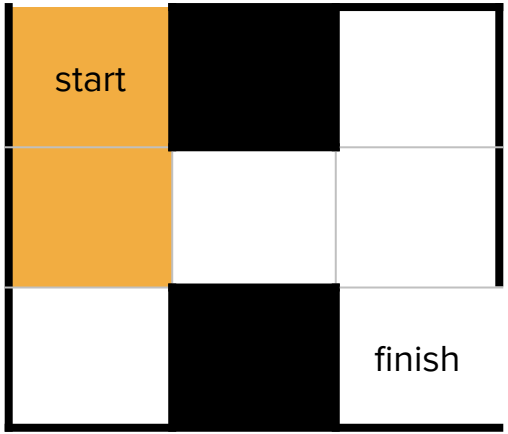
- Start at the entrance
- Take one step ~~North, South, East, or West~~

We must go back one step.



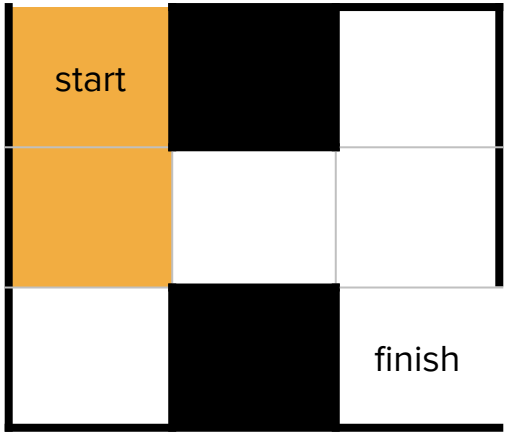
A **recursive** algorithm for solving mazes

- Start at the entrance
- Take one step ~~North~~, South, East, or West



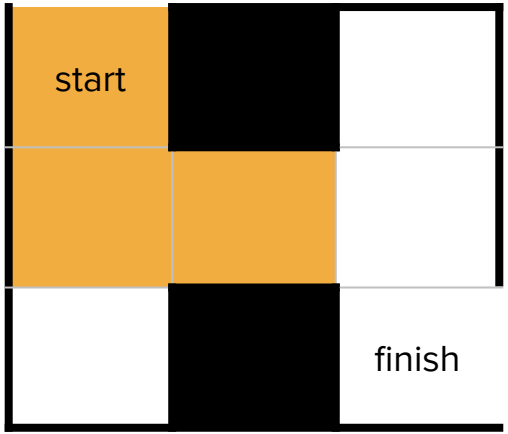
A **recursive** algorithm for solving mazes

- Start at the entrance
- Take one step ~~North~~, ~~South~~, East, or West



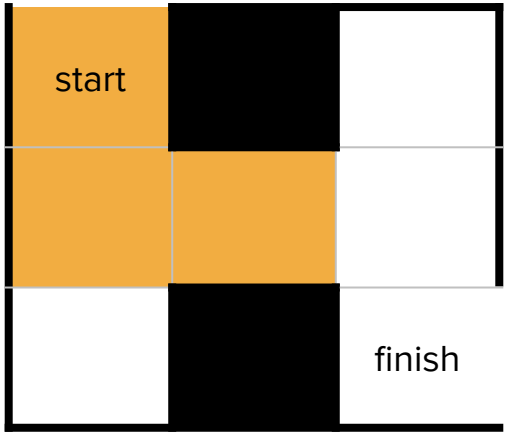
A **recursive** algorithm for solving mazes

- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze



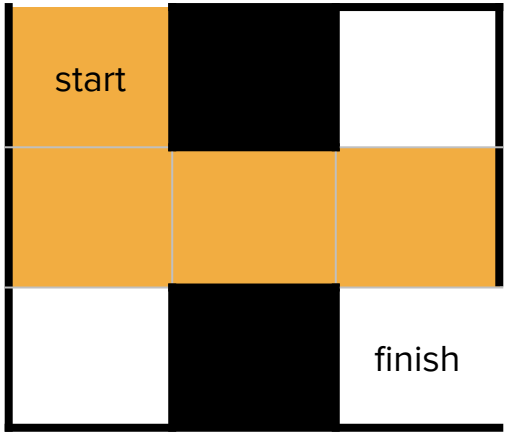
A **recursive** algorithm for solving mazes

- Start at the entrance
- Take one step ~~North~~, ~~South~~, East, or West



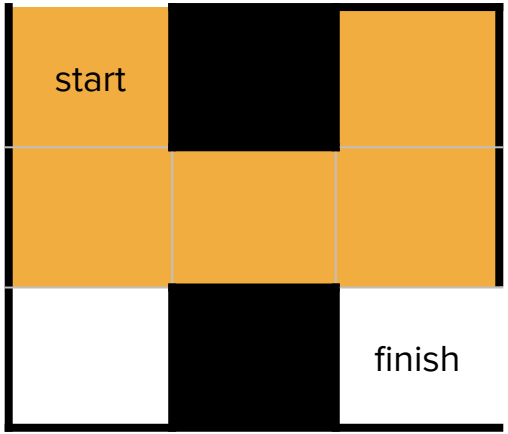
A **recursive** algorithm for solving mazes

- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze



A **recursive** algorithm for solving mazes

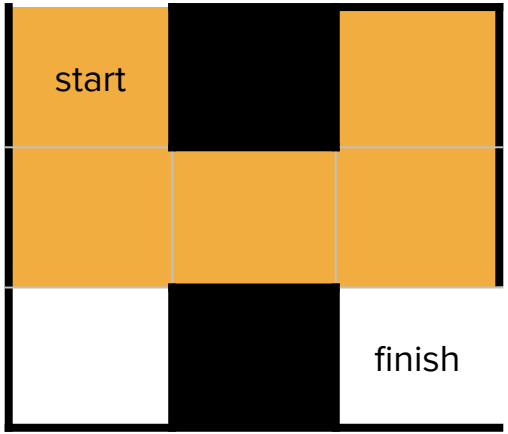
- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze



A **recursive** algorithm for solving mazes

- Start at the entrance
- Take one step ~~North, South, East, or West~~

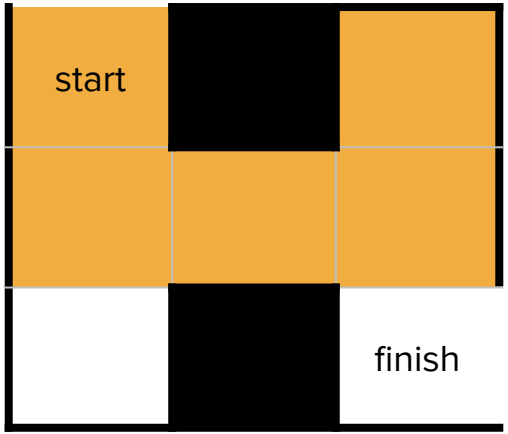
Dead end!
(cannot go North, South, East, or West)



A **recursive** algorithm for solving mazes

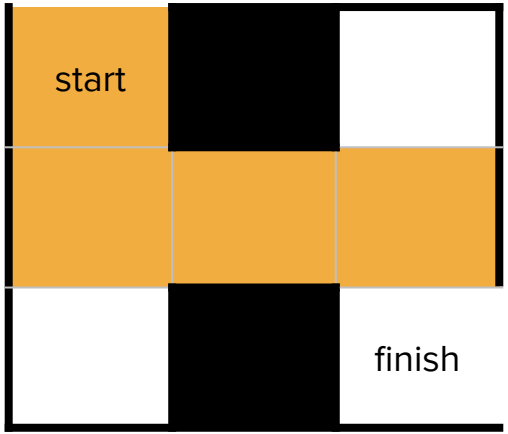
- Start at the entrance
- Take one step ~~North, South, East, or West~~

We must go back one step.



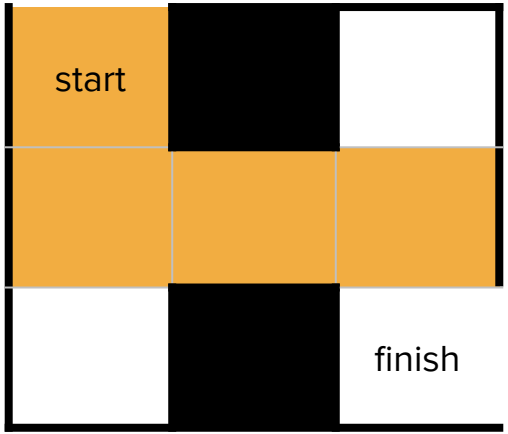
A **recursive** algorithm for solving mazes

- Start at the entrance
- Take one step North, South, East, or West



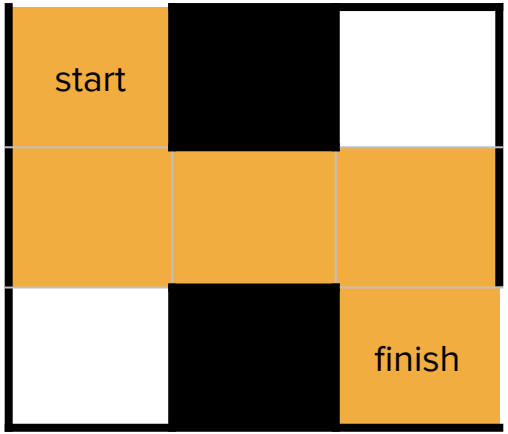
A **recursive** algorithm for solving mazes

- Start at the entrance
- Take one step ~~North~~, South, East, or West



A **recursive** algorithm for solving mazes

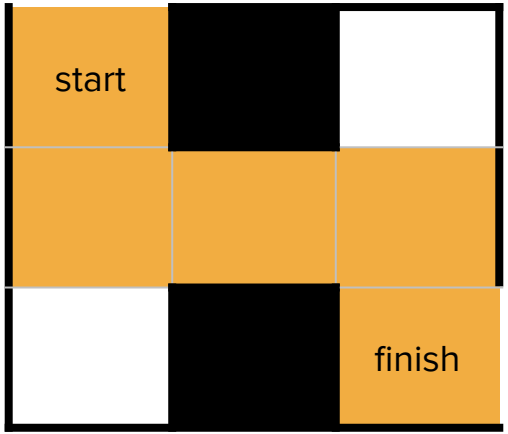
- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze



A **recursive** algorithm for solving mazes

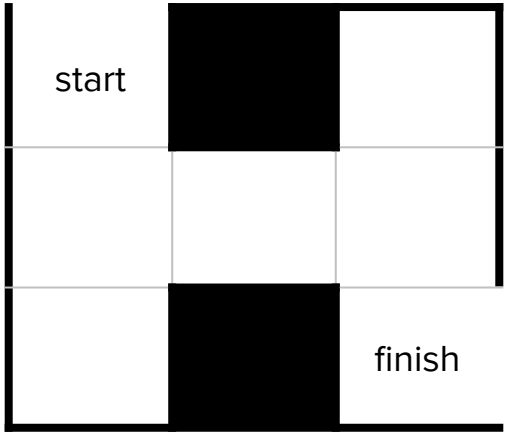
- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze

End of the maze!



A **recursive** algorithm for solving mazes

- **Base case**: If we're at the end of the maze, stop
- **Recursive case**: Explore North, South, East, then West



What defines our maze decision tree?

- **Decision** at each step (each level of the tree):
  - Which valid move will we take?

- **Options** at each decision (branches from each node):
  - All valid moves (in bounds, not a wall, not previously visited) that are either North, South, East, or West of the current location

- Information we need to store along the way:
  - The path we've taken so far (a Stack we're building up)
  - Where we've already visited
  - Our current location

What defines our maze decision tree?

- **Decision** at each step (each level of the tree):
  - Which valid move will we take?

- **Options** at each decision (branches from each node):
  - All valid moves (in bounds, not a wall, not previously visited) that are either North, South, East, or West of the current location

- Information we need to store along the way:
  - The path we've taken so far (a Stack we're building up)
  - Where we've already visited
  - Our current location

Exercise for home:
Draw the decision tree.

What defines our maze decision tree?

- **Decision** at each step (each level of the tree):
  - Which valid move will we take?

- **Options** at each decision (branches from each node):
  - All valid moves (in bounds, not a wall, not previously visited) that are either North, South, East, or West of the current location

- Information we need to store along the way:
  - The path we've taken so far (a Stack we're building up)
  - Where we've already visited
  - **Our current location**

We need to make an adjustment!

- Recall our solveMaze prototype:

```
Stack<GridLocation> solveMaze(Grid<bool>& maze)
```

- We need a helper function to keep track of our path through the maze!

```
bool solveMazeHelper(Grid<bool>& maze,
                     Stack<GridLocation>& path,
                     GridLocation cur)
```

Pseudocode

- Our helper function will have as **parameters**: the maze itself, the path we're building up, and the current location.
  - **Idea**: Use the boolean Grid (the maze itself) to store information about whether or not a location has been visited by flipping the cell to false once it's in the path (to avoid loops) ➔ This works with our existing **`generateValidMoves()`** function

Pseudocode

- Our helper function will have as **parameters**: the maze itself, the path we're building up, and the current location.
  - **Idea**: Use the boolean Grid (the maze itself) to store information about whether or not a location has been visited by flipping the cell to false once it's in the path (to avoid loops) ➔ This works with our existing **`generateValidMoves()`** function

- **Recursive case**: Iterate over valid moves from **`generateValidMoves()`** and try adding them to our path
  - If any recursive call returns true, we have a solution
  - If all fail, return false

Pseudocode

- Our helper function will have as **parameters**: the maze itself, the path we're building up, and the current location.
  - **Idea**: Use the boolean Grid (the maze itself) to store information about whether or not a location has been visited by flipping the cell to false once it's in the path (to avoid loops) ➔ This works with our existing **`generateValidMoves()`** function

- **Recursive case**: Iterate over valid moves from **`generateValidMoves()`** and try adding them to our path
  - If any recursive call returns true, we have a solution
  - If all fail, return false

- **Base case**: We can stop exploring when we've reached the exit ➔ return true if the current location is the exit

Let’s code it!

Takeaways

- Recursive maze-solving uses **choose/explore/undo** because we have to explicitly “unchoose” by setting cells back to true after trying them.

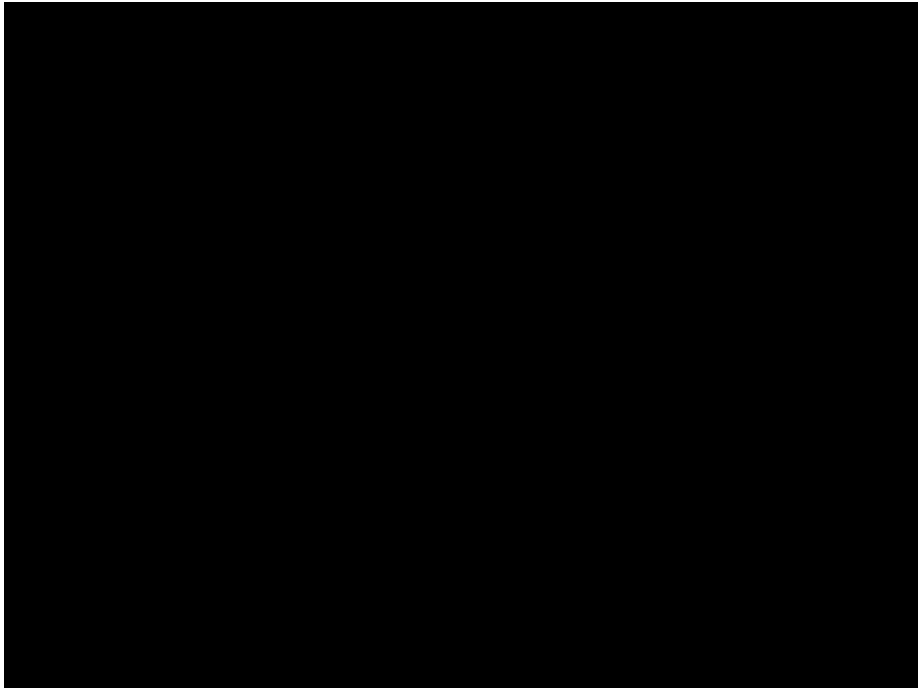
- Our helper function may have a different return type from our initial function prototype, and our wrapper function (not the helper) may be more complex than just a call to our helper function.

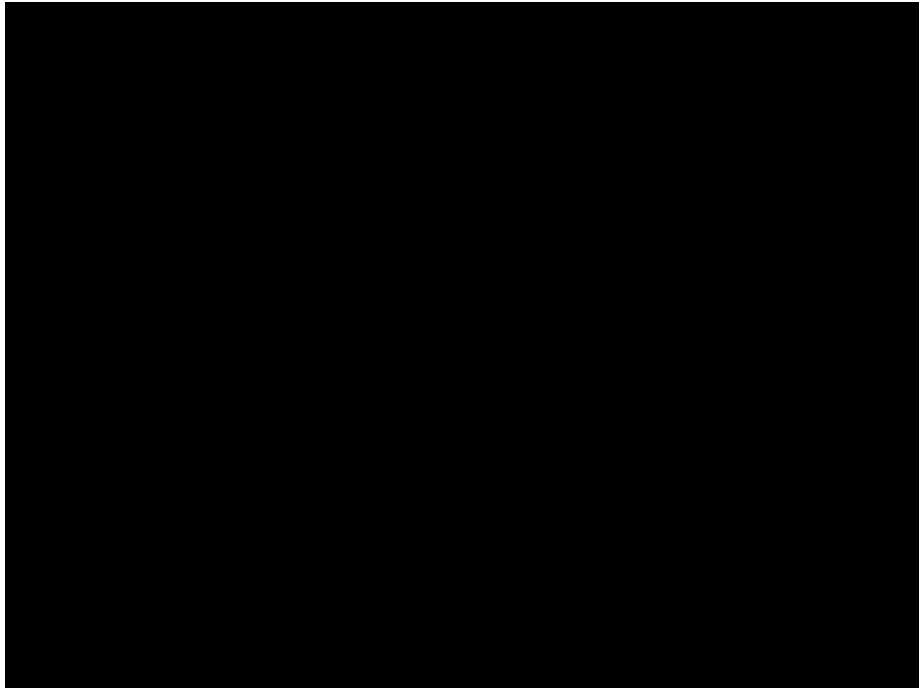
- It may be helpful to revisit and adjust our initial answers to our planning questions as we determine more about the algorithm we want to use (e.g. adding a parameter to our helper function).

Recursion is depth-first search (DFS)!

BFS vs. DFS comparison

Which do you think will be faster?





BFS vs. DFS comparison

- BFS is typically iterative while DFS is naturally expressed recursively.
- Although DFS is faster in this particular case, which search strategy to use depends on the problem you're solving.
- BFS looks at all paths of a particular length before moving on to longer paths, so it's guaranteed to find the shortest path (e.g. word ladder)!
- DFS doesn't need to store all partial paths along the way, so it has a smaller memory footprint than BFS does.

Recursive Optimization

(a brief intro)

"Hard" Problems

"Hard" Problems

- There are many different categories of problems in computer science that are considered to be "hard" to solve.
  - Formally, these are known as "NP-hard" problems. Take CS103 to learn more!

"Hard" Problems

- There are many different categories of problems in computer science that are considered to be "hard" to solve.
  - Formally, these are known as "NP-hard" problems. Take CS103 to learn more!

- For these categories of problems, there exist no known "good" or "efficient" ways to generate the best solution to the problem. The only known way to generate an exact answer is to **try all possible solutions** and select the best one.
  - Often times these problems involve finding permutations (`O(n!)` possible solutions) or combinations (`O(2^n)` possible solutions)

"Hard" Problems

- There are many different categories of problems in computer science that are considered to be "hard" to solve.
  - Formally, these are known as "NP-hard" problems. Take CS103 to learn more!

- For these categories of problems, there exist no known "good" or "efficient" ways to generate the best solution to the problem. The only known way to generate an exact answer is to **try all possible solutions** and select the best one.
  - Often times these problems involve finding permutations (`O(n!)` possible solutions) or combinations (`O(2^n)` possible solutions)

- **Backtracking recursion is an elegant way to solve these kinds of problems!**

Recursive optimization: “find the best solution”

- We won’t cover these in lecture because you won’t be writing any recursive optimization code on Assignment 3 or any of the assignments moving forward.

- But these types of problems are one of the most common applications of recursion in real-world engineering systems.

- If you love recursion after the last two weeks and want to dive deeper into topics we didn’t have time to explore, you can try out a recursive optimization problem for your final project!

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 → optimization!**
- There are many, many examples of specific problems that we can solve, including
  - Generating permutations
  - **Generating subsets (with additional constraints!)**
  - Generating combinations
  - And many, many more

Check out the Knapsack Problem in this week's section handout!

Limitations of recursive optimization

With great power comes great responsibility...

- Ask: What are you optimizing for?

- Keith Schwarz has a great recursive backtracking problem about optimizing shift scheduling for a company to maximize profit. It shows how optimizing for profit without considering how the schedule might severely affect workers' quality of life. And this has happened in real life!

- Computers can help you find strategies that maximize lots of different quantities. Make sure you pick a quantity that takes in the perspectives of all the stakeholders in a situation.

Recursion is memory intensive

- Because a stack frame gets created for every recursive call, recursion can be very memory intensive. You'll get to experience this in A3!

- Recursion is a powerful tool for understanding data structures and algorithms, especially in fields like artificial intelligence and systems design and programming languages.

- But it often can't be used in scenarios that require you to handle large amounts of data (without some sort of added optimizations).

Recursion Wrap-up

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
- Common pattern: choose/explore/unchoose

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.

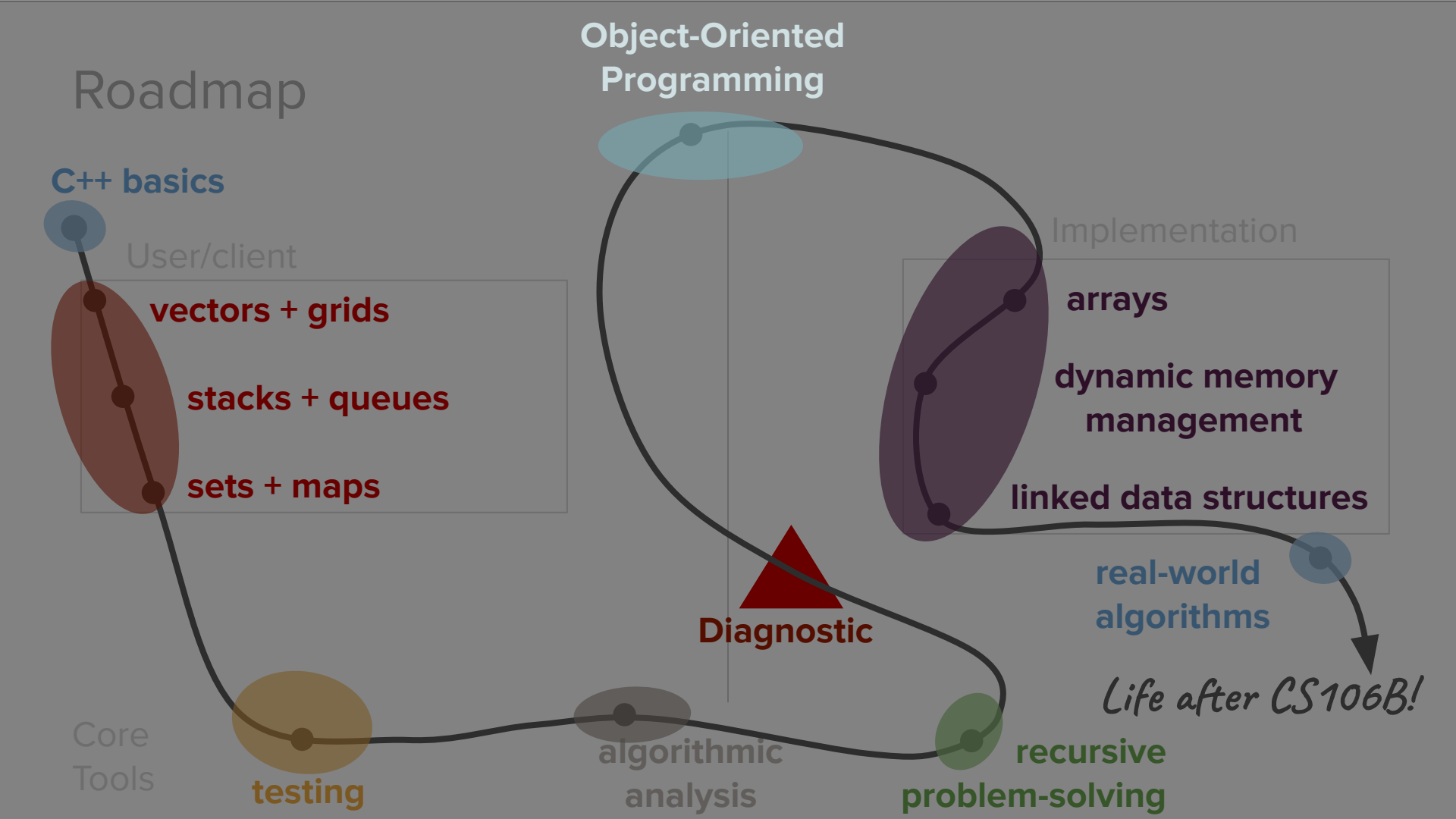
Solving backtracking recursion problems

- Which of our three use cases does our problem fall into? (generate/count all solutions, find one solution/prove its existence, pick one best solution)
- What are we building up as our “many possibilities” in order to find our solution? (subsets, permutations, combinations, or something else)

- What’s the provided function prototype and requirements? Do we need a helper function?
    - What are we returning as our solution? (a boolean, a final value, a set of results, etc.)
    - Do we care about returning or keeping track of the path we took to get to our solution? If yes, what parameters are we already given and what others might be useful?

- What are our base and recursive cases?
    - What does my decision tree look like? (decisions, options, what to keep track of)
    - In addition to what we’re building up, are there any additional constraints on our solutions?
    - Does it make sense to use choose/explore/undo OR copy/edit/recurse for the recursion? (Note: In some very complex problems, it might be some combination of the two.)

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
Life after CS106B!
Diagnostic
Core Tools
testing
algorithmic analysis
recursive problem-solving

Classes and Object-Oriented Programming