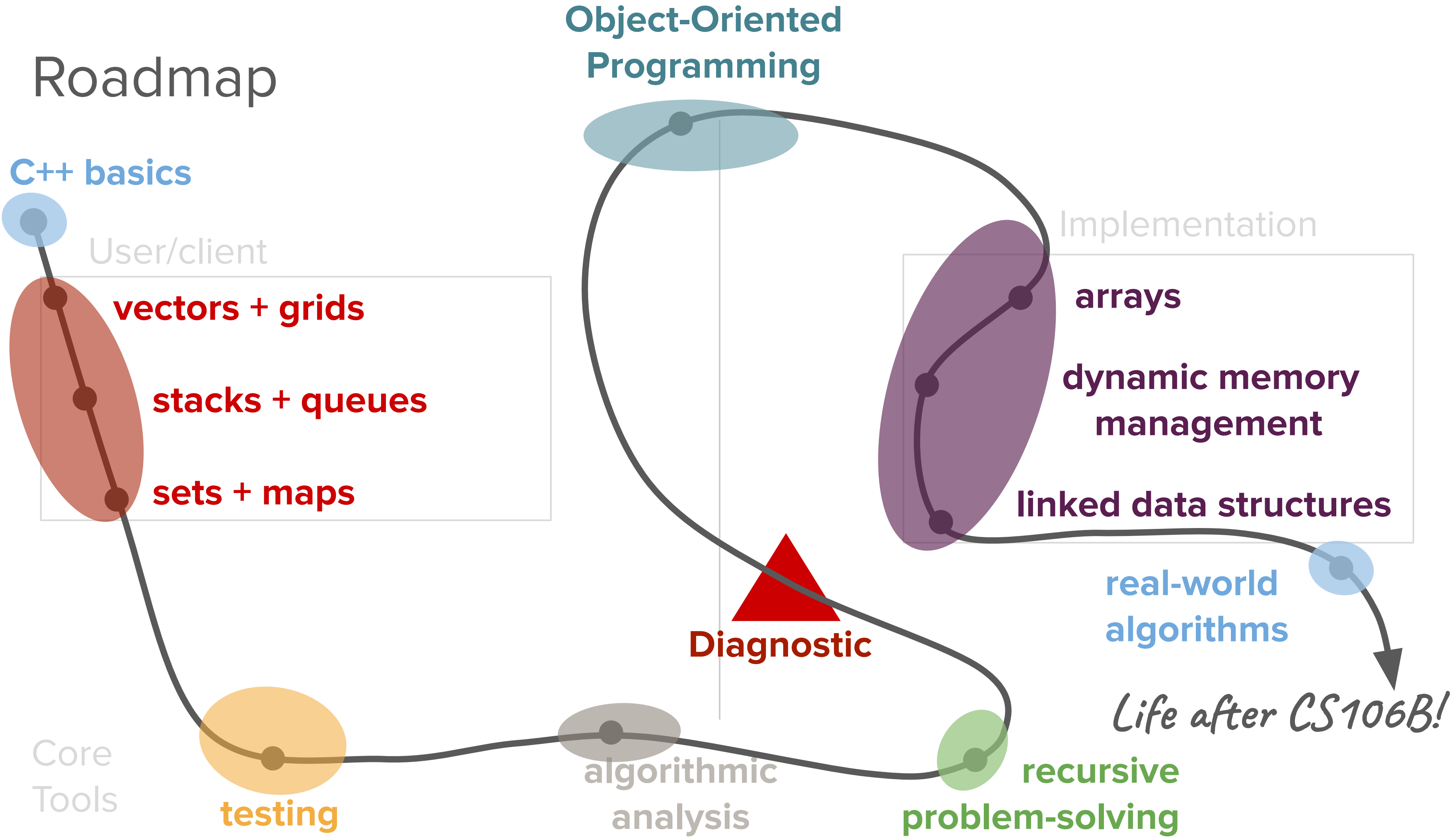
Recursive Backtracking Revisited

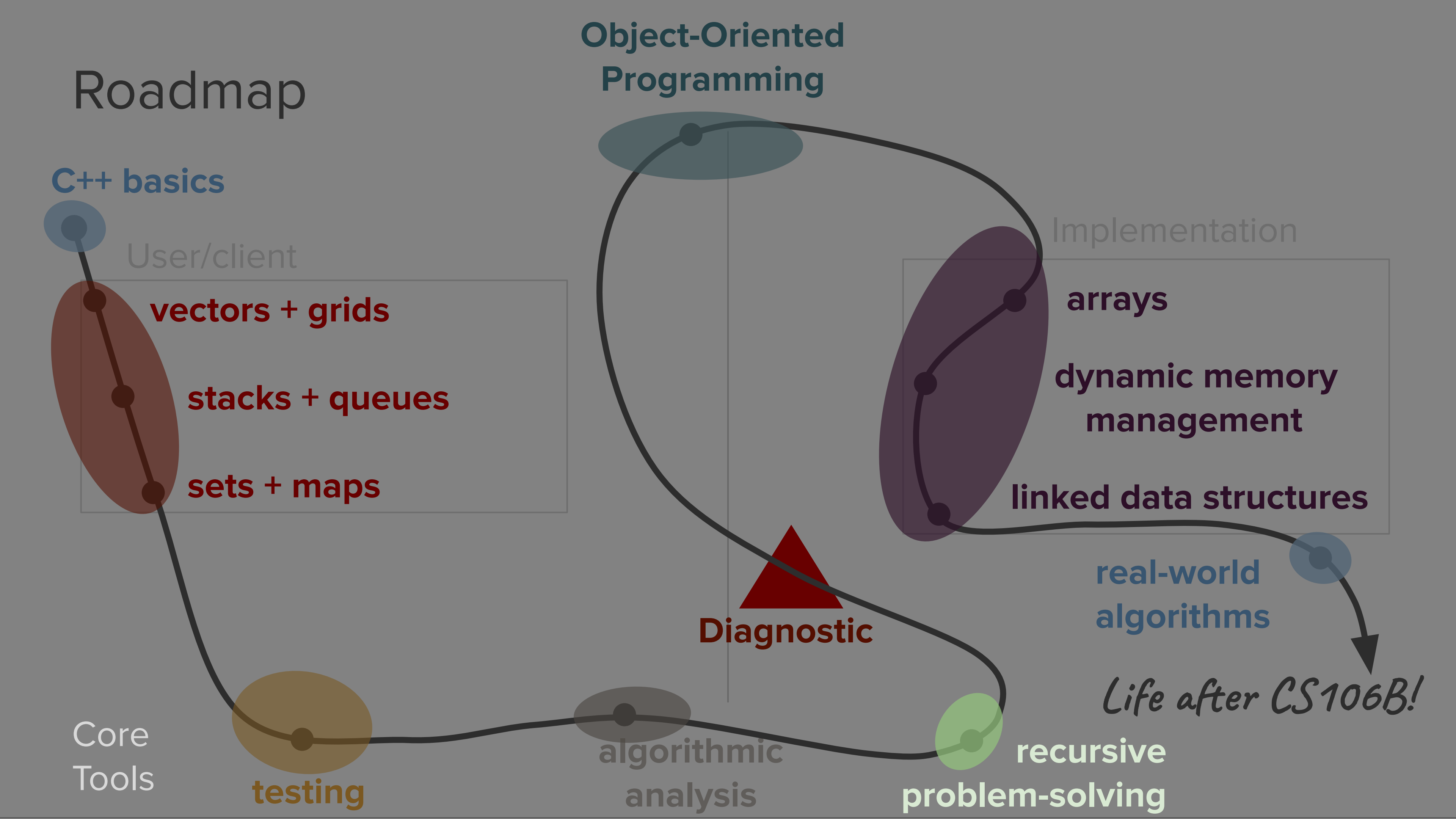
What has been your favorite part of the first 4 weeks of the course?

(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

What strategies should we use when solving recursive backtracking problems?

Today's topics

1. Review
2. Recursive backtracking strategies
3. Practice applying strategies
   a. Selecting fixed-size groups
   b. Solving mazes with DFS
   c. Knapsack problem

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

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

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**
    - Generating subsets
    - Generating combinations
    - And many, many more

What defines our permutations decision tree?

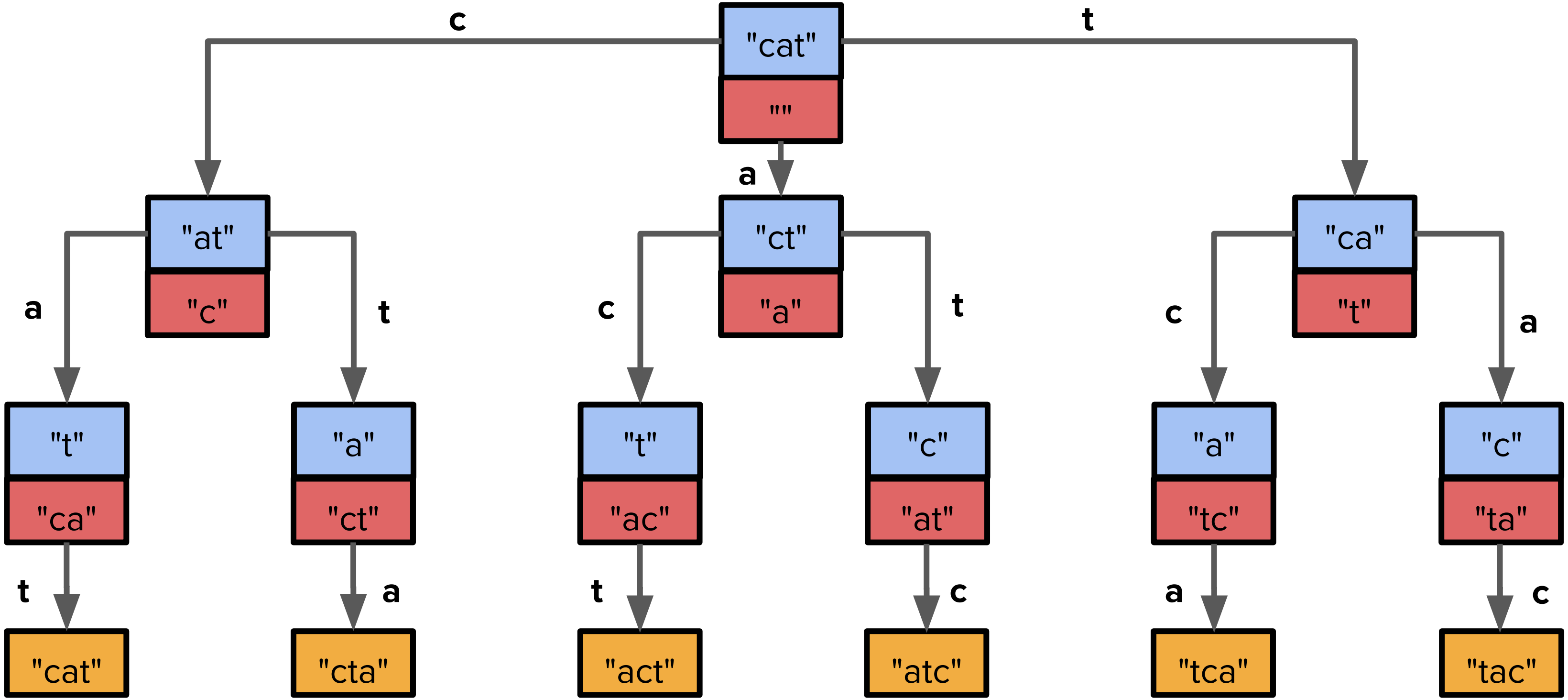
- **Decision** at each step (each level of the tree):
  - What is the next letter that is going to get added to the permutation?

- **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"



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

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

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**

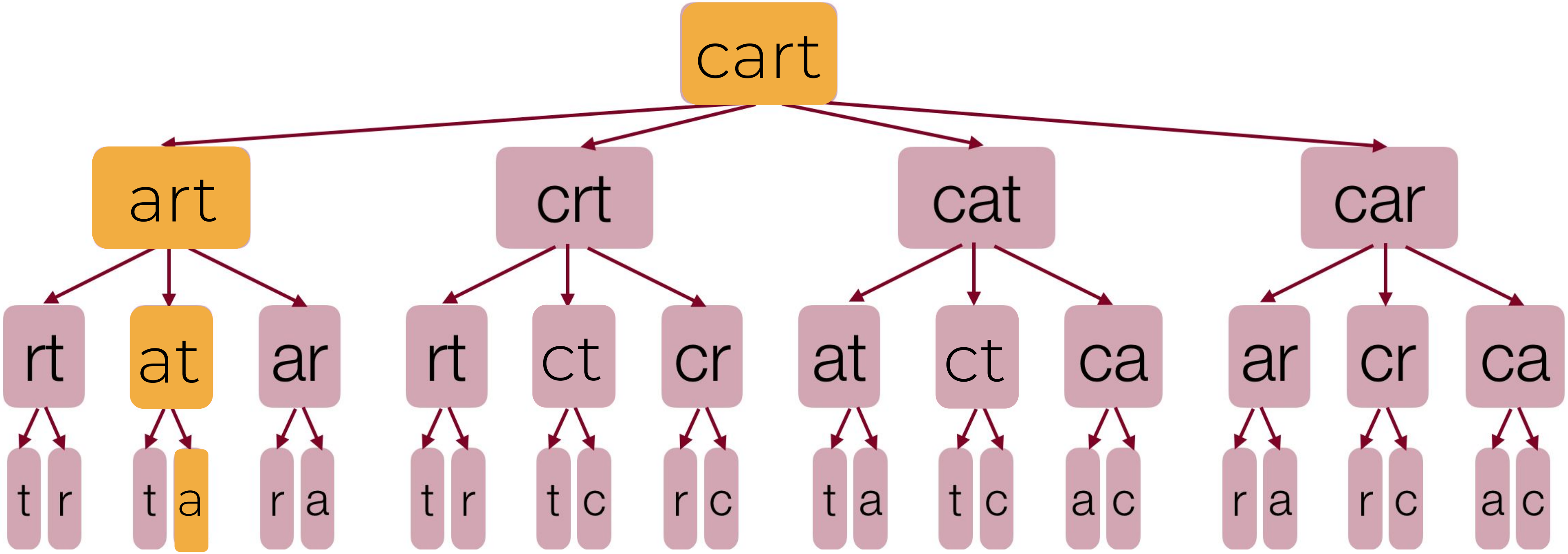
What defines our shrinkable decision tree?

- **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?



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;
```

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

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!

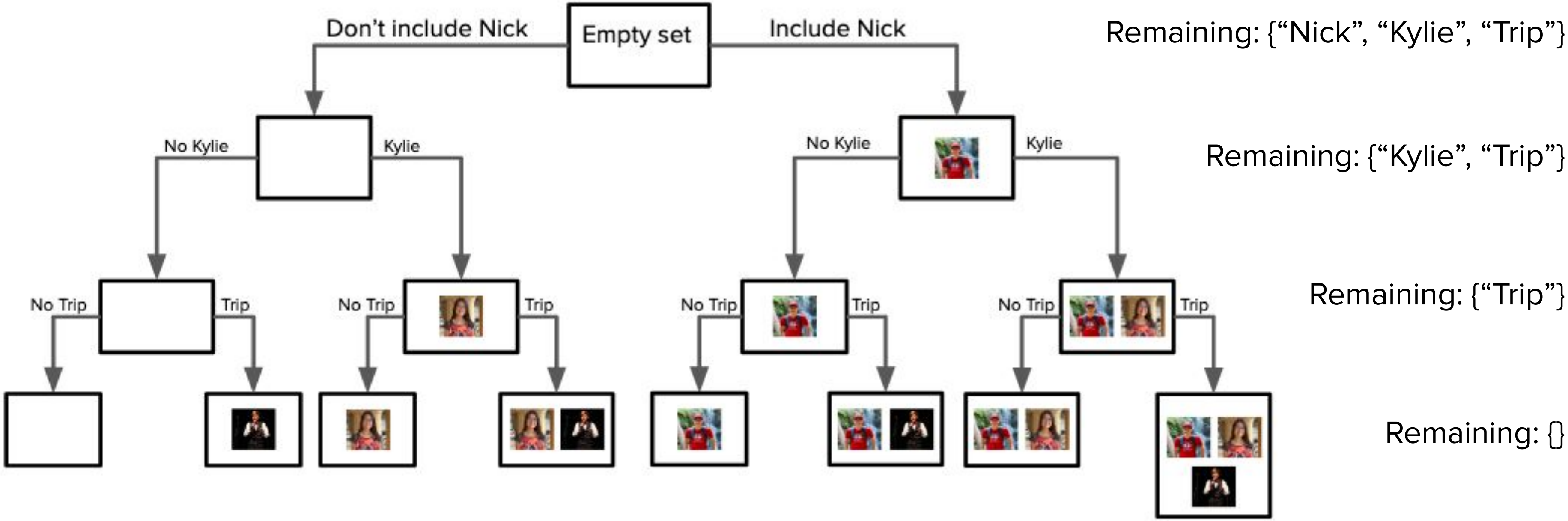
What defines our subsets decision tree?

- **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



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.

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

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

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

What are all the ways you can pick five justices off the U.S. Supreme Court?

Subsets vs. Combinations

- Our goal: We want to pick a combination of 5 justices out of a group of 9.

Subsets vs. Combinations

- Our goal: We want to pick a combination of 5 justices out of a group of 9.
- This sounds very similar to the problem we solved when we generated subsets – these 5 justices would be a subset of the overall group of 9.

Subsets vs. Combinations

- Our goal: We want to pick a combination of 5 justices out of a group of 9.

- This sounds very similar to the problem we solved when we generated subsets – these 5 justices would be a subset of the overall group of 9.

- 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 justices out of a group of 9.
- This sounds very similar to the problem we solved when we generated subsets – these 5 justices would be a subset of the overall group of 9.
- 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 last lecture, 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`** judges can be represented as a **`Set<string>`**.

- In Friday's 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** judges can be represented as a **Set<string>**.

- In Friday's 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>& judges, int k)
```

Do we need a helper function?

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

We'll need to keep track of a current set of judges 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>& judges, int k)
```

We'll need to keep track of a current set of judges 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)
```

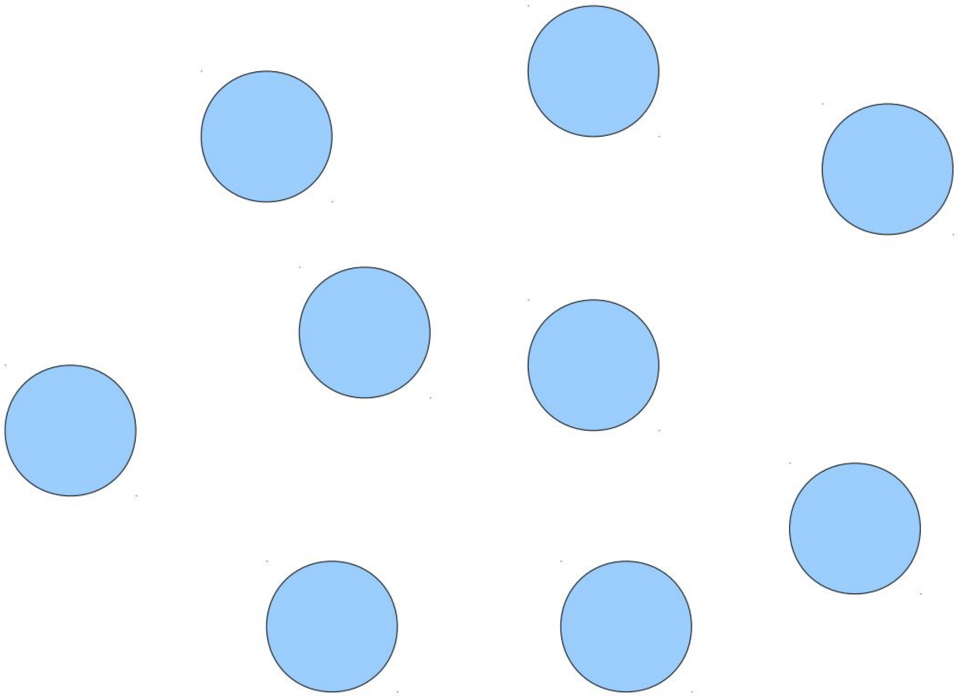
Solving backtracking recursion problems

- 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?

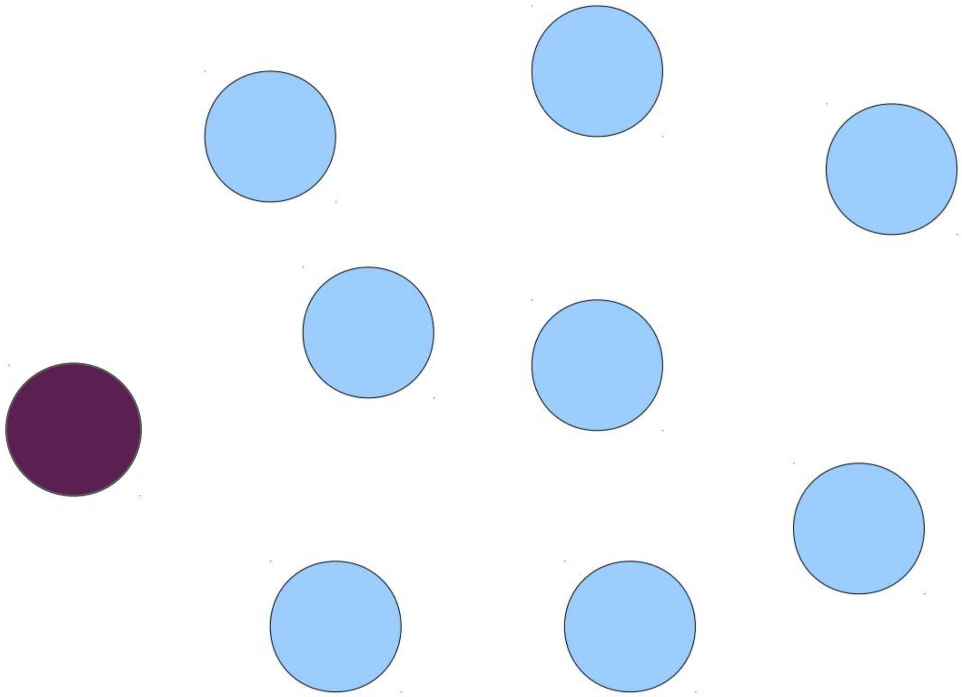
- What's the provided function prototype and requirements? Do we need a helper function?
  - What are we returning as our solution?
  - Do we care about returning or keeping track of the path we took to get to our solution? If yes, what parameters are we already given and what others might be useful?

- **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?

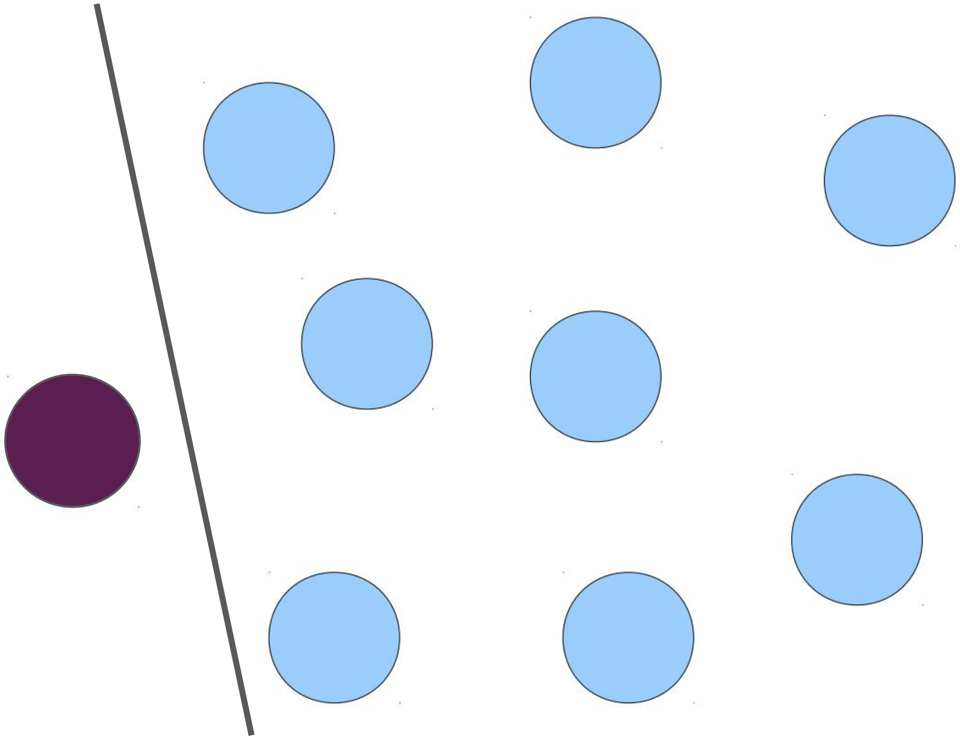
Generating Combinations



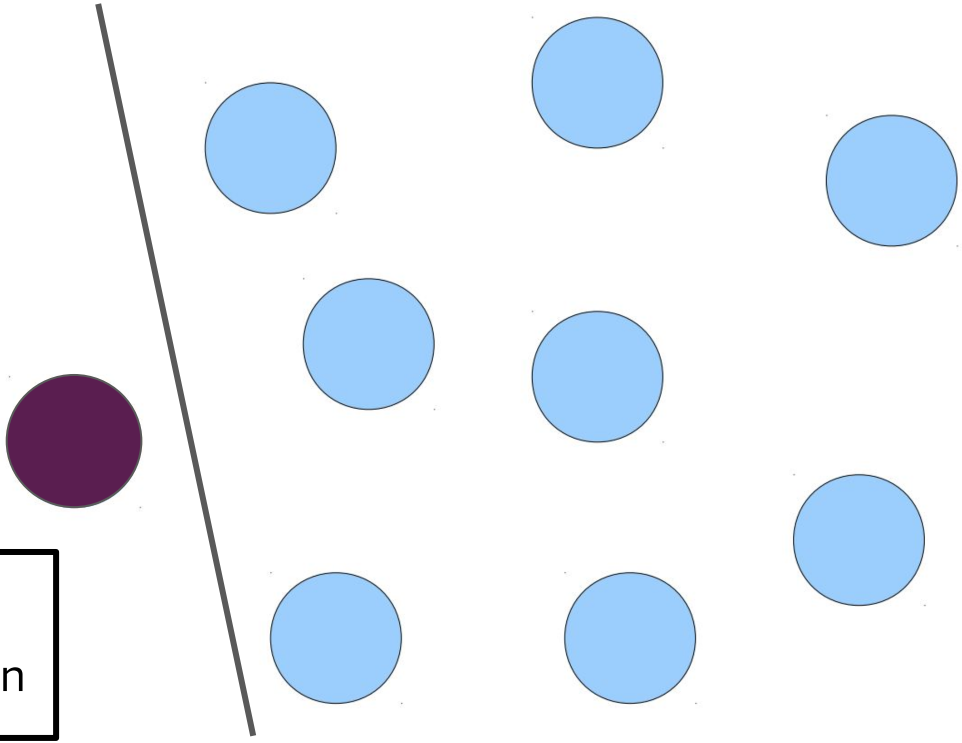
Generating Combinations



Generating Combinations

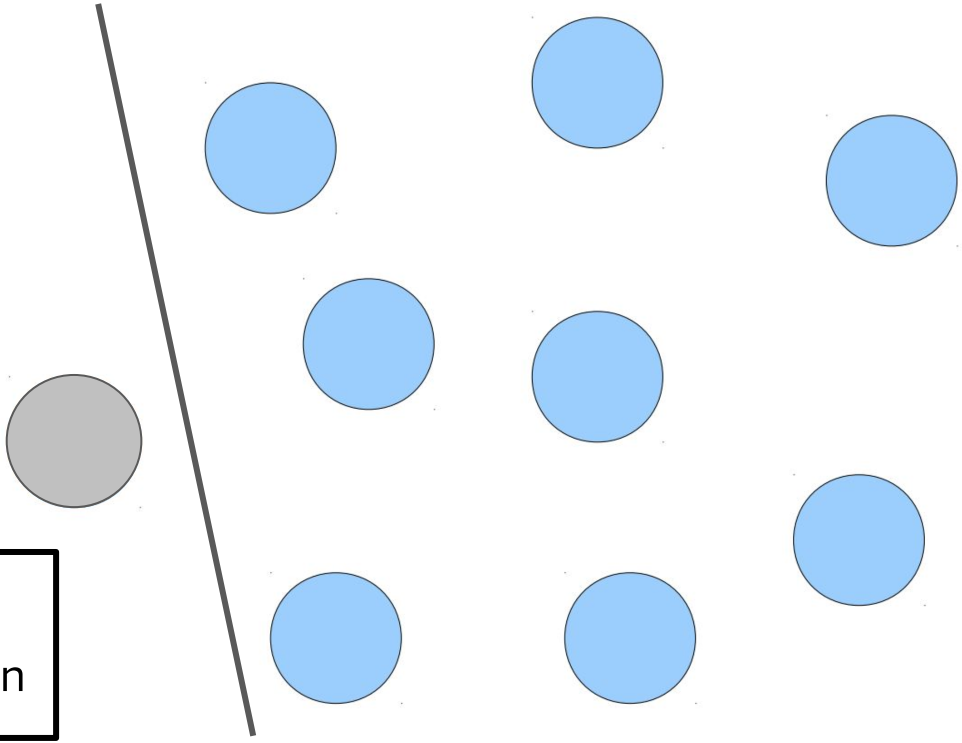


Generating Combinations



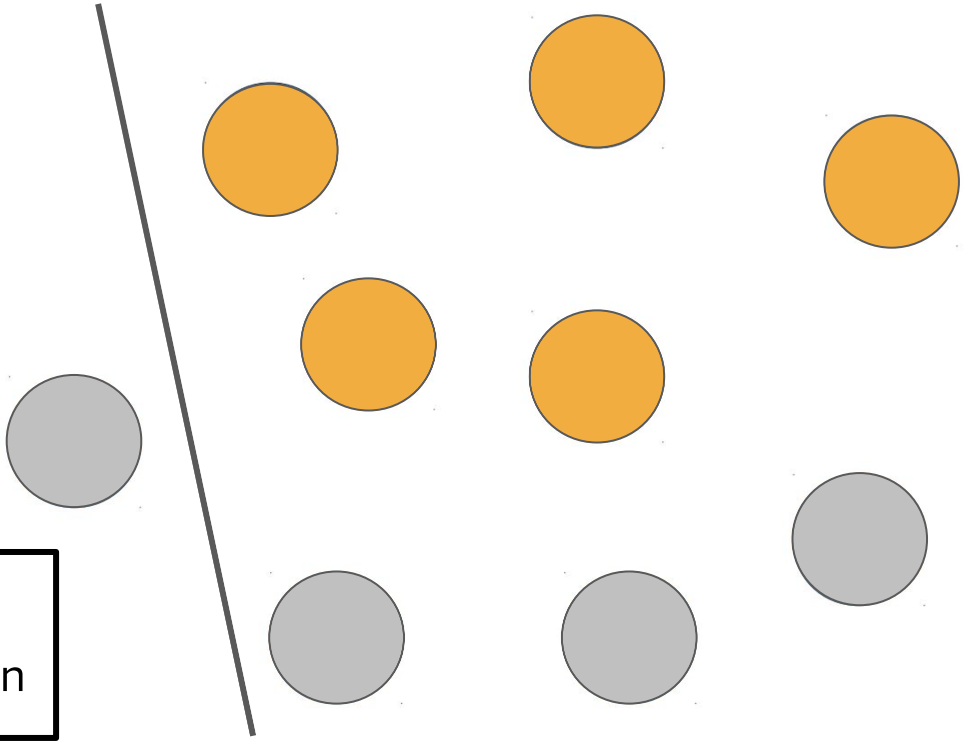
Option 1:
Exclude this person

Generating Combinations



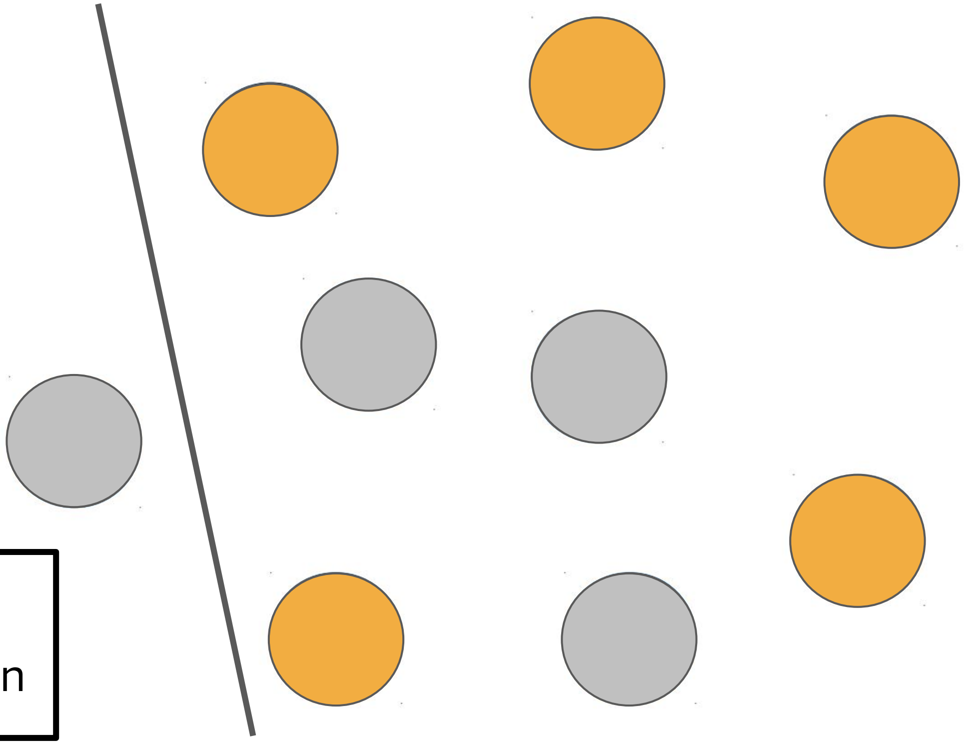
Option 1:
Exclude this person

Generating Combinations



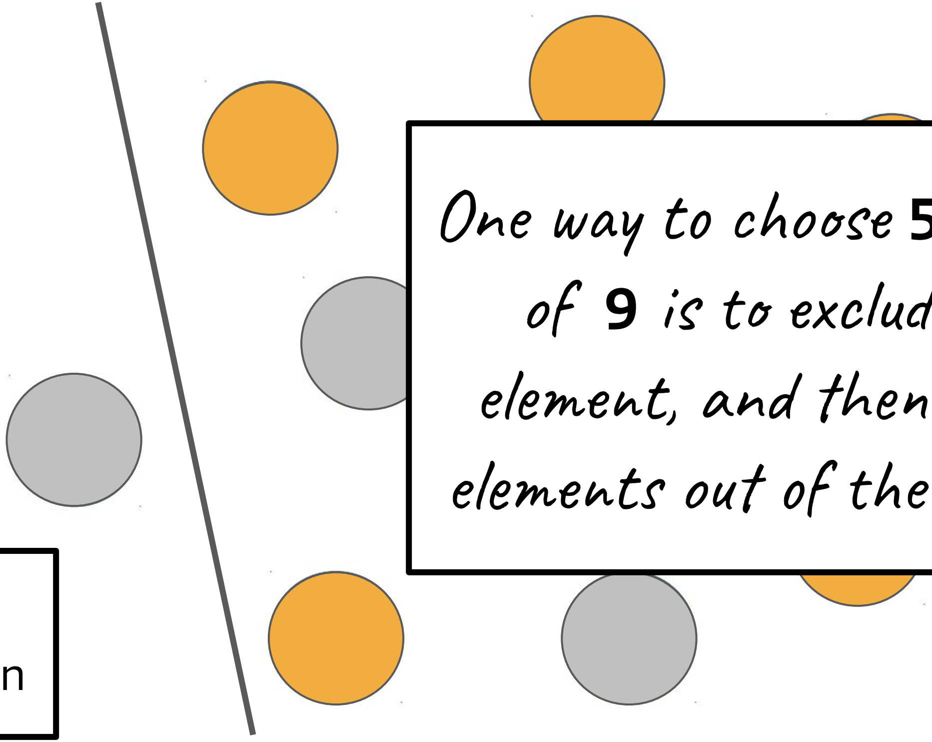
Option 1:
Exclude this person

Generating Combinations



Option 1:
Exclude this person

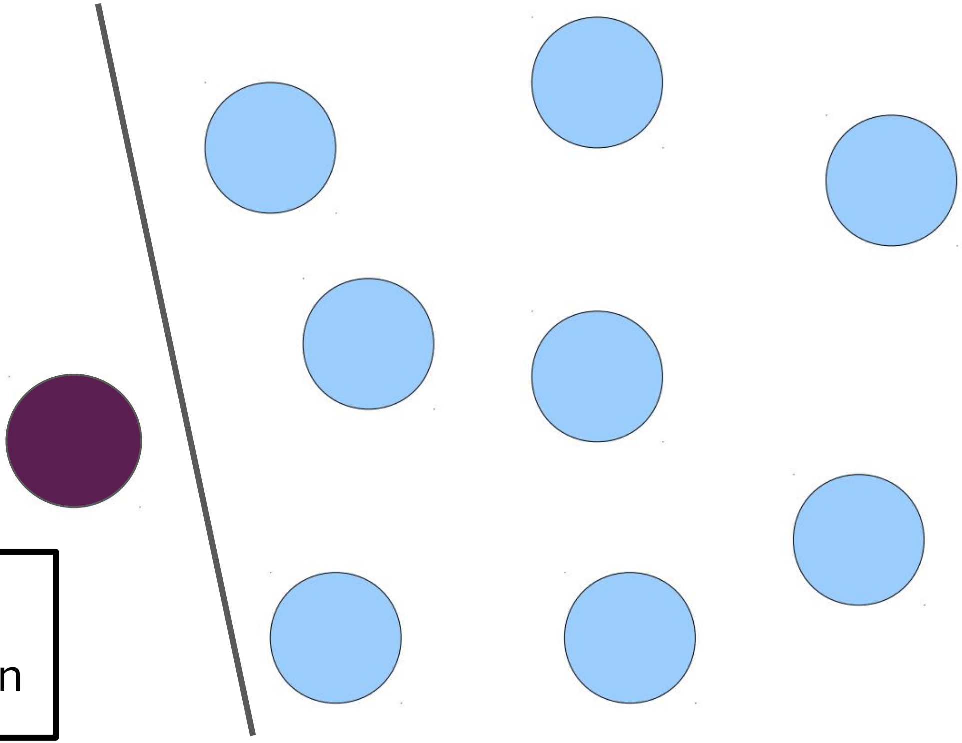
Generating Combinations



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

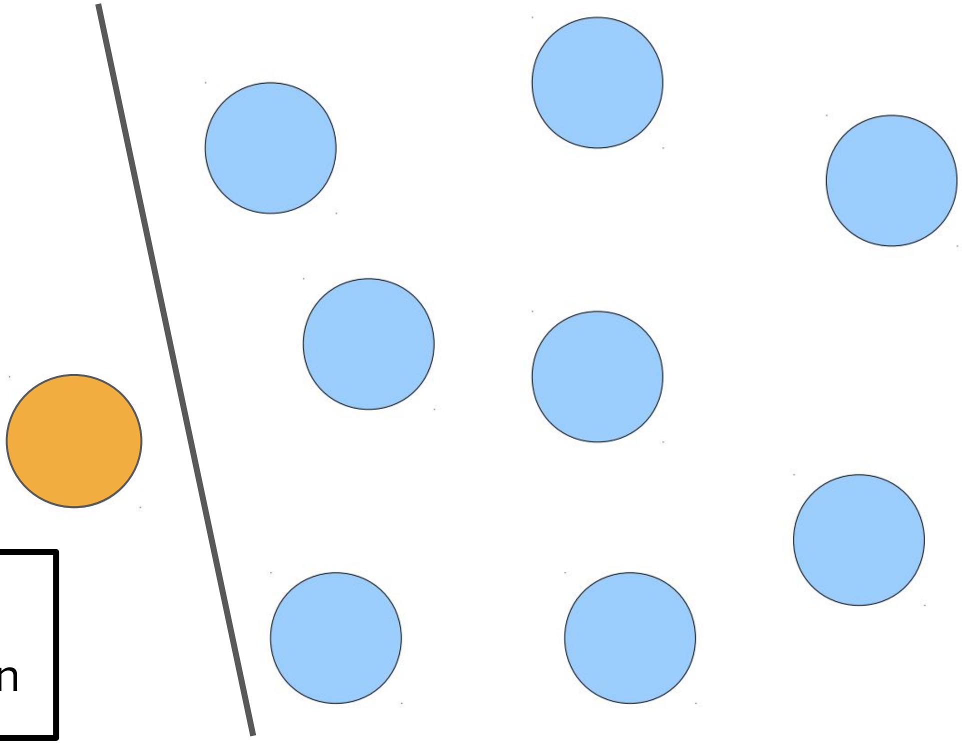
Option 1:
Exclude this person

Generating Combinations



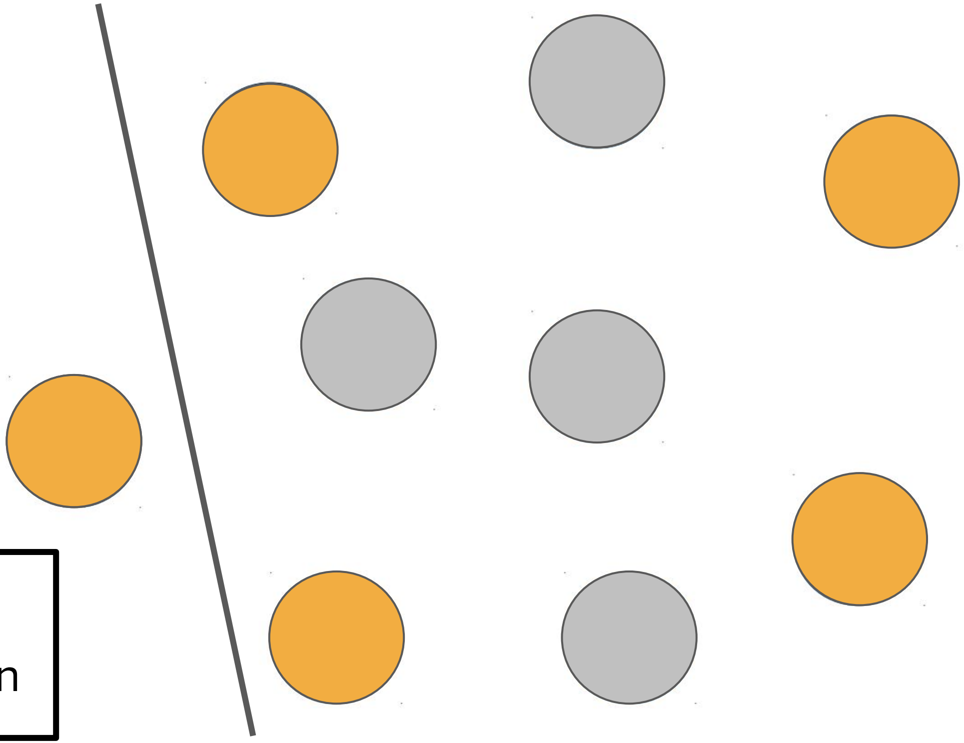
Option 2:
Include this person

Generating Combinations



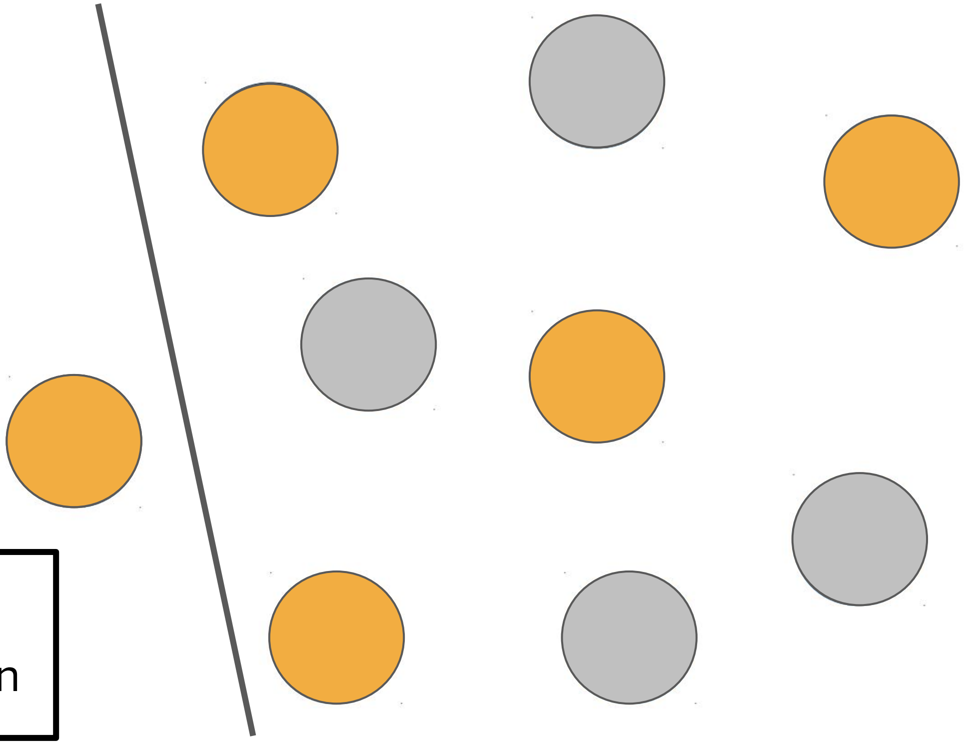
Option 2:
Include this person

Generating Combinations



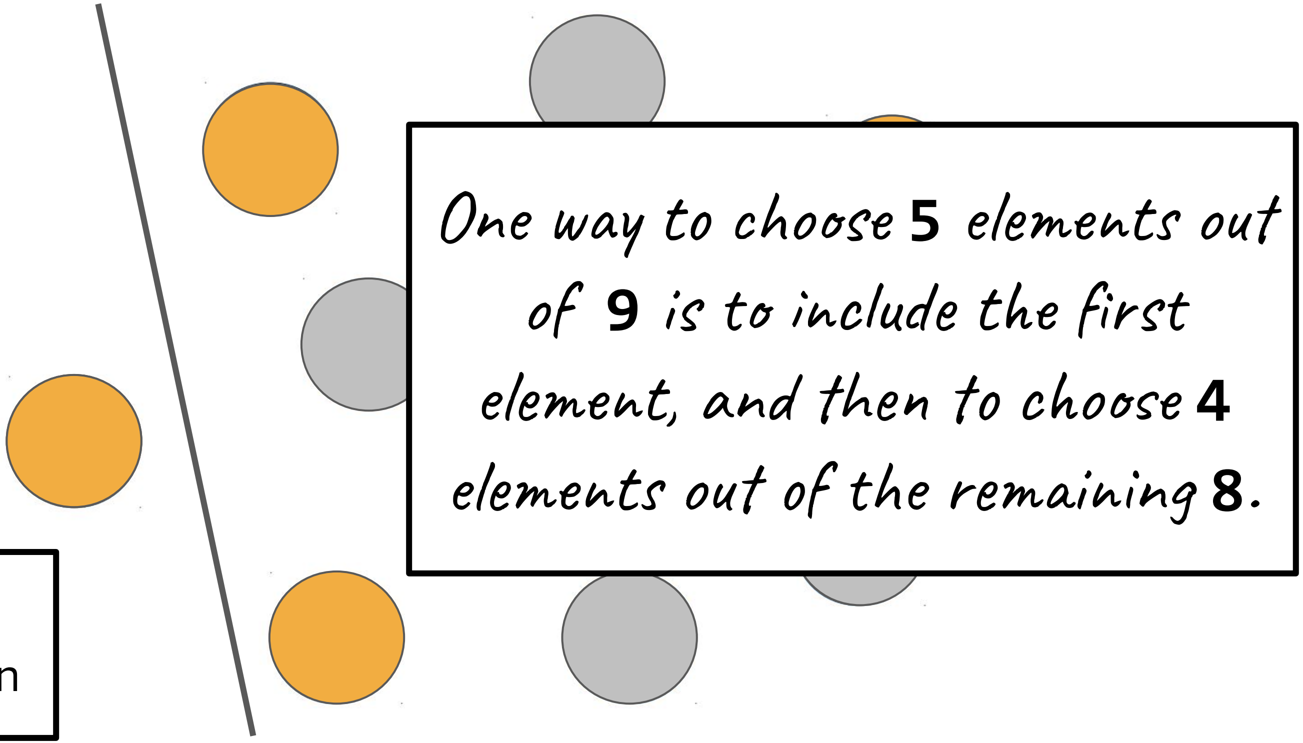
Option 2:
Include this person

Generating Combinations



Option 2:
Include this person

Generating Combinations



One way to choose **5** elements out of **9** is to include the first element, and then to choose **4** elements out of the remaining **8**.

Option 2:
Include this person

Writing functions that build combinations

- Suppose we get to the following scenario:

Pick 0 more Justices out of:

`{Kagan, Breyer}`

Chosen so far:

`{Ginsburg, Roberts, Gorsuch, Thomas, Sotomayor}`

- There's no need to keep looking! **What do we return in this case?**

Writing functions that build combinations

- Suppose we get to the following scenario:

Pick 0 more Justices out of:

`{Kagan, Breyer}`

Chosen so far:

`{Ginsburg, Roberts, Gorsuch, Thomas, Sotomayor}`

- There's no need to keep looking! **We can return a set containing the set of who we've chosen so far.**

Writing functions that build combinations

- Suppose we get to the following scenario:

Pick 0 more Justices out of:

`{Kagan, Breyer}`

Chosen so far:

`{Ginsburg, Roberts, Gorsuch, Thomas, Sotomayor}`

- There's no need to keep looking! **We can return a set containing the set of who we've chosen so far.**

This is our base case! (part 1)

Writing functions that build combinations

- Suppose we get to the following scenario:

Pick 5 more Justices out of:

`{Sotomayor, Thomas, Roberts, Gorsuch}`

Chosen so far:

`{}`

- There’s no need to keep looking! **What do we return in this case?**

Writing functions that build combinations

- Suppose we get to the following scenario:

Pick 5 more Justices out of:

`{Sotomayor, Thomas, Roberts, Gorsuch}`

Chosen so far:

`{}`

- There’s no need to keep looking! **We can return an empty set.**

Writing functions that build combinations

- Suppose we get to the following scenario:

Pick 5 more Justices out of:

`{Sotomayor, Thomas, Roberts, Gorsuch}`

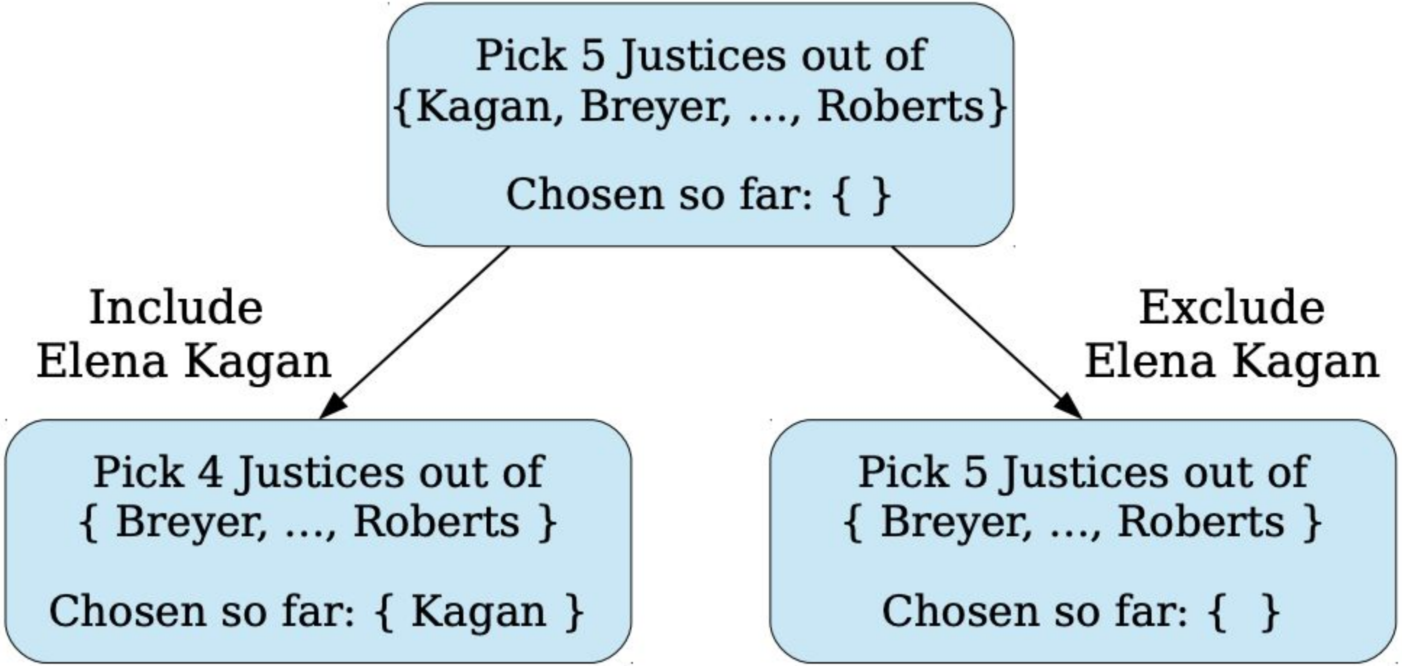
Chosen so far:

`{}`

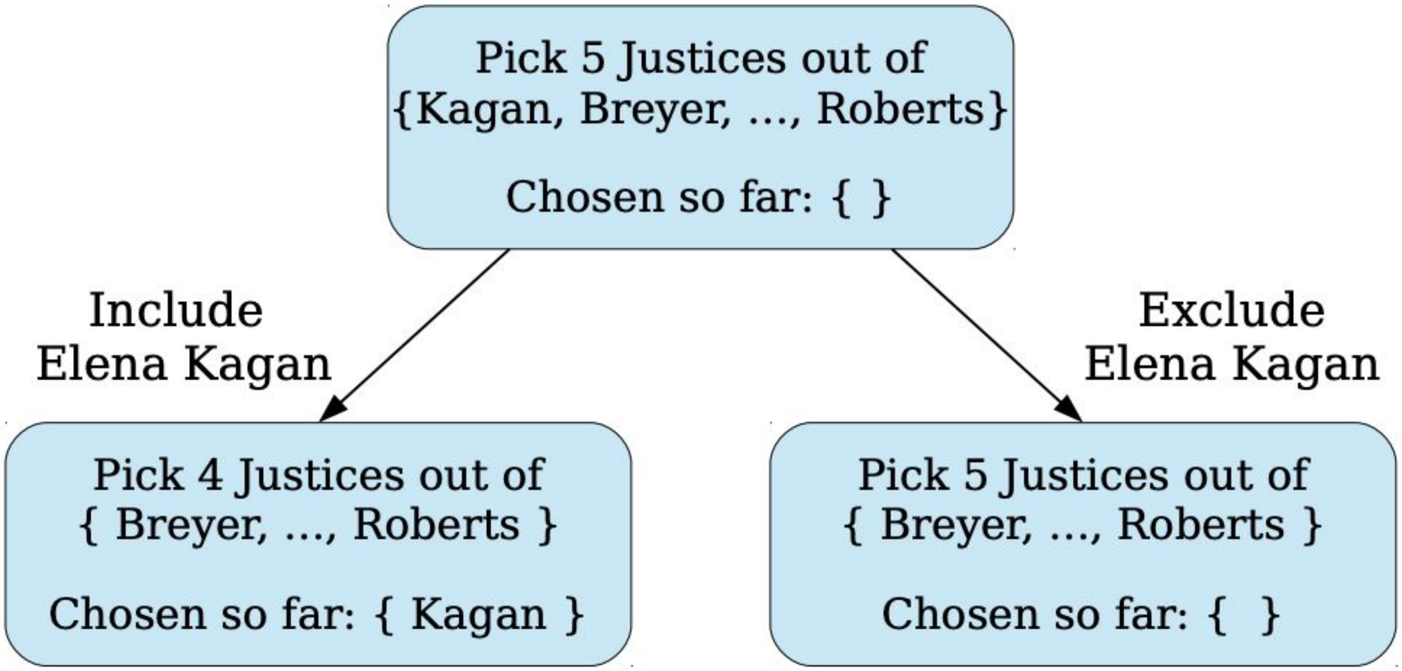
- There's no need to keep looking! **We can return an empty set.**

This is our base case! (part 2)

What about our combinations decision tree?



What about our combinations decision tree?



This is just the beginning of the tree, but helps us understand our recursive case.

What defines our combinations decision tree?

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

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

- Information we need to store along the way:
  - The combination you’ve built so far
  - The remaining elements to choose from
  - The remaining number of spots left to fill

What defines our combinations decision tree?

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

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

- Information we need to store along the way:
    - The combination you've built so far
    - The remaining elements to choose from
    - **The remaining number of spots left to fill**

Pseudocode

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

Pseudocode

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

- **Recursive case:**
  - Choose: Pick an element in remaining.
  - Explore: Try including and excluding the element and store resulting sets.
  - Unchoose: Restore our remaining and chosen sets.
  - Return the the combined returned sets from both inclusion and exclusion.

Pseudocode

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

- **Recursive case:**
  - Choose: Pick an element in remaining.
  - Explore: Try including and excluding the element and store resulting sets.
  - Unchoose: Restore our remaining and chosen sets.
  - **Return the the combined returned sets from both inclusion and exclusion.**

This is different from our usual recursion pattern!

Pseudocode

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

- **Recursive case:**
  - Choose: Pick an element in remaining.
  - Explore: Try including and excluding the element and store resulting sets.
  - Unchoose: Restore our remaining and chosen sets.
  - Return the the combined returned sets from both inclusion and exclusion.

- **Base cases:**
  - Not enough remaining elements to choose from ➔ return empty set
  - No more space in chosen (k is maxed out) ➔ return set with chosen

Let's code it!

Takeaways

- Making combinations is very similar to our recursive process for generating subsets!

- The differences:
  - We’re constraining the subsets’ size.
  - We’re building up a set of all valid subsets of that particular size (i.e. combinations).

- Instead of printing out subsets in our base case, we have to return individual sets in our base case and then build up and return our resulting set of sets in our recursive case

Announcements

Announcements

- Assignment 3 is due tonight at 11:59pm PDT. The grace period ends tomorrow at 11:59pm PDT.

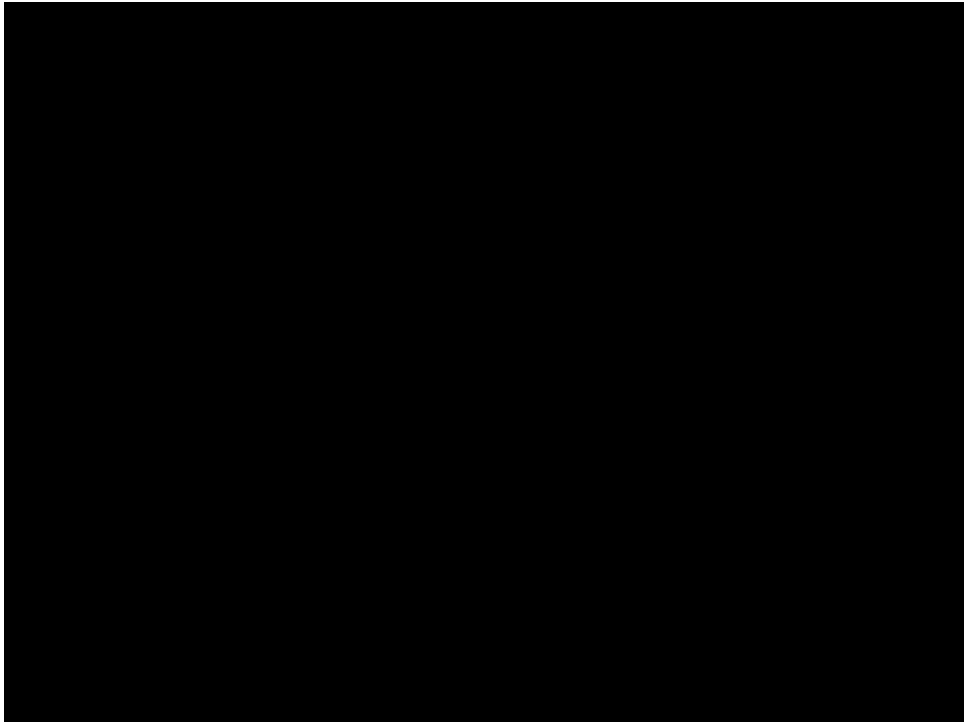
- Assignment 4 (backtracking recursion!) will be released by the end of the day.

- Assignment 2 revisions are due Thursday at 11:59pm PDT.

- The mid-quarter diagnostic is coming up at the end of this week.
  - Please check out the website and review last Wednesday's lecture for all the logistics!
  - Today is the last day of content that will be on the assessment.
  - Today's and Friday's lecture will only show up as extra credit.

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**

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?

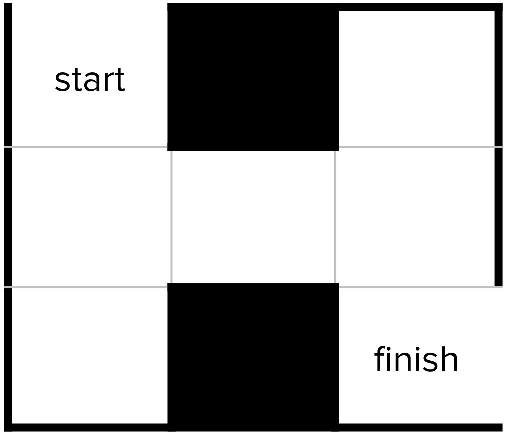
- **What are our base and recursive cases?**
  - What does my decision tree look like? (decisions, options, what to keep track of)
  - In addition to what we’re building up, are there any additional constraints on our solutions?
  - Does it make sense to use choose/explore/undo OR copy/edit/recurse for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

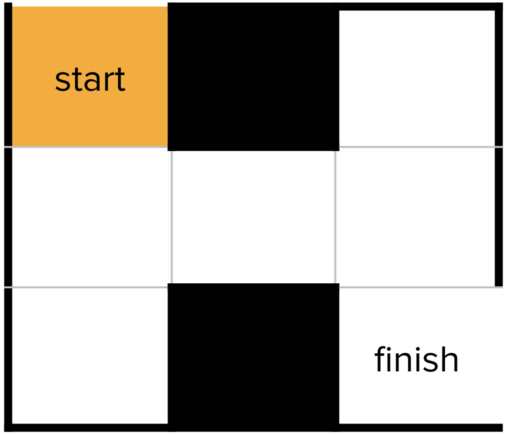
A **recursive** algorithm for solving mazes

- Start at the entrance



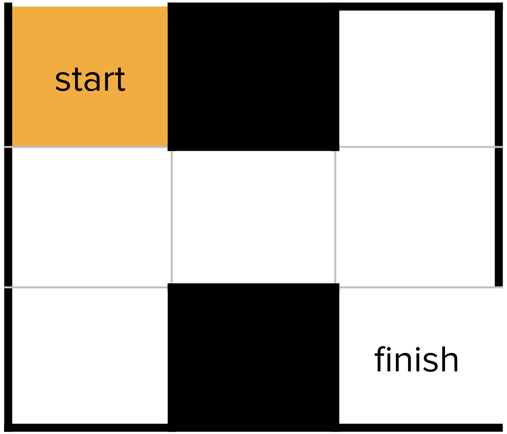
A **recursive** algorithm for solving mazes

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



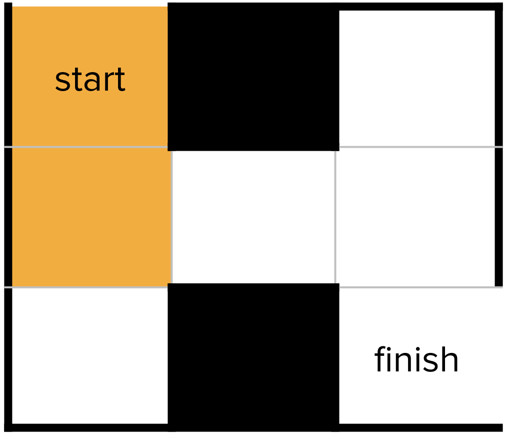
A **recursive** algorithm for solving mazes

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



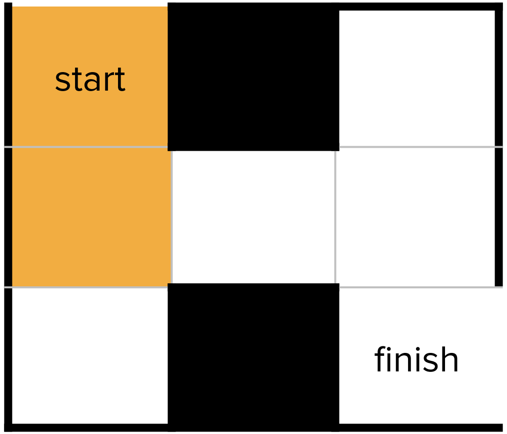
A **recursive** algorithm for solving mazes

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



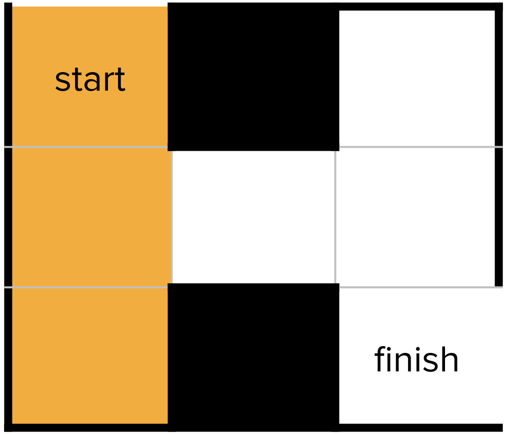
A **recursive** algorithm for solving mazes

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



A **recursive** algorithm for solving mazes

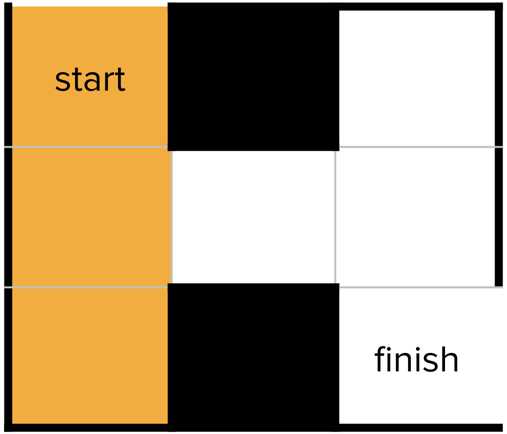
- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze



A **recursive** algorithm for solving mazes

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

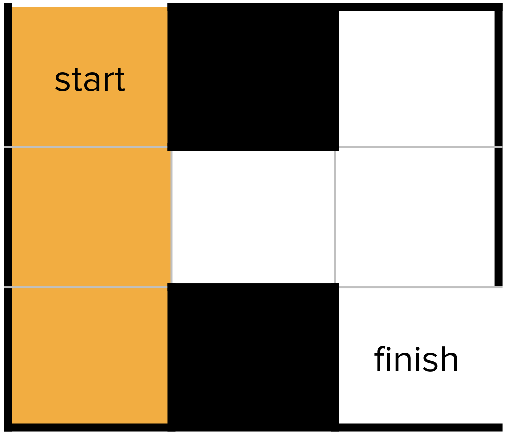
Dead end!
(cannot go North, South, East, or West)



A **recursive** algorithm for solving mazes

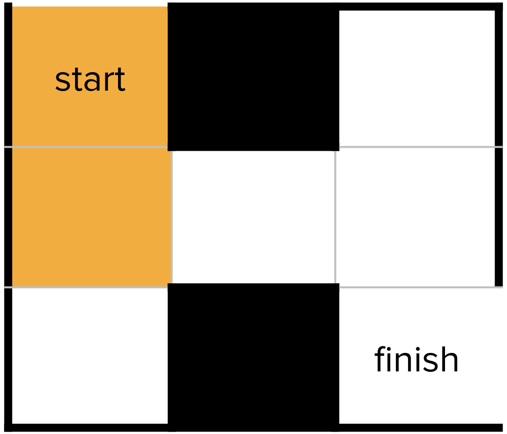
- Start at the entrance
- Take one step ~~North, South, East, or West~~

We must go back one step.



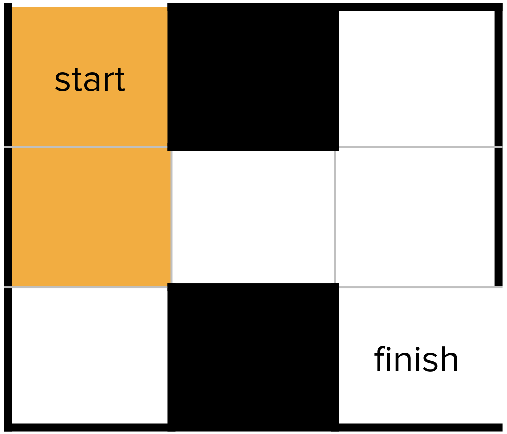
A **recursive** algorithm for solving mazes

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



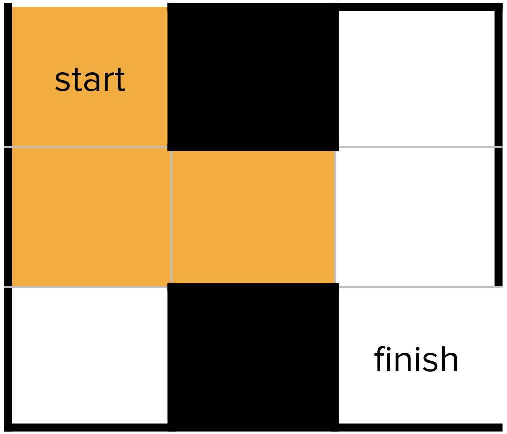
A **recursive** algorithm for solving mazes

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



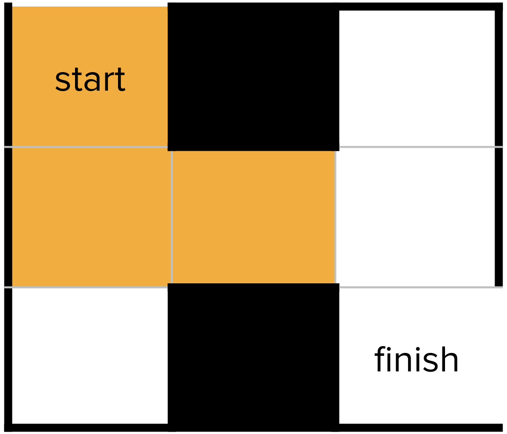
A **recursive** algorithm for solving mazes

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



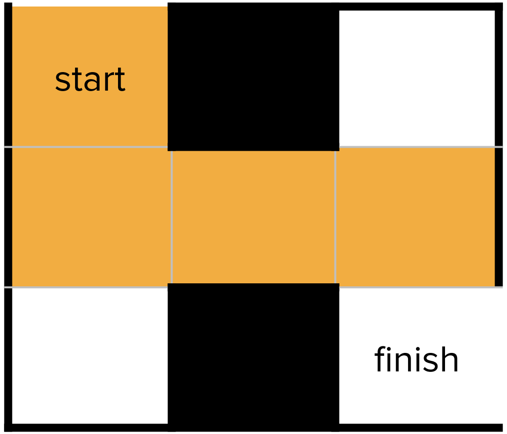
A **recursive** algorithm for solving mazes

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



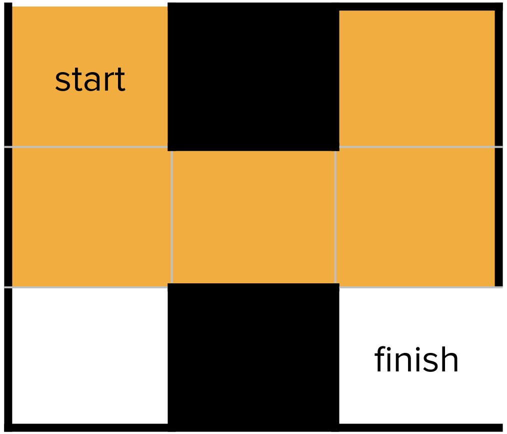
A **recursive** algorithm for solving mazes

- 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

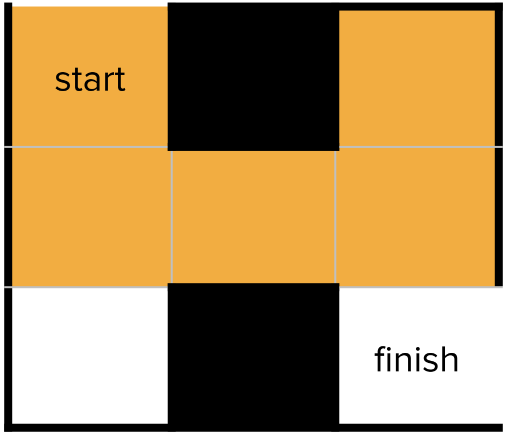
- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze



A **recursive** algorithm for solving mazes

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

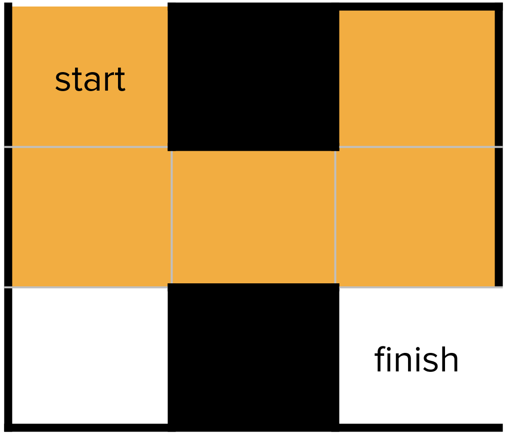
*Dead end!
(cannot go North,
South, East, or West)*



A **recursive** algorithm for solving mazes

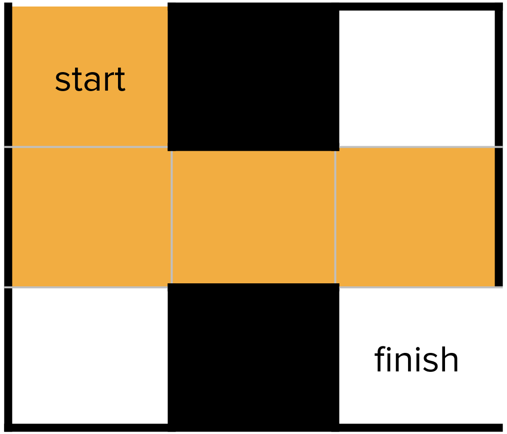
- Start at the entrance
- Take one step ~~North, South, East, or West~~

We must go back one step.



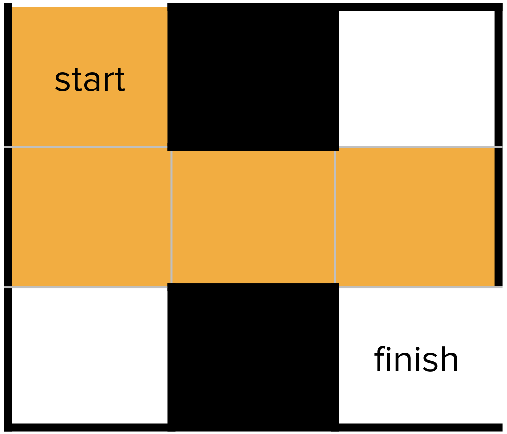
A **recursive** algorithm for solving mazes

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



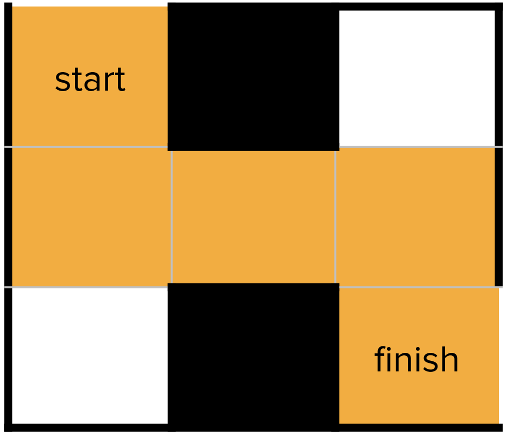
A **recursive** algorithm for solving mazes

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



A **recursive** algorithm for solving mazes

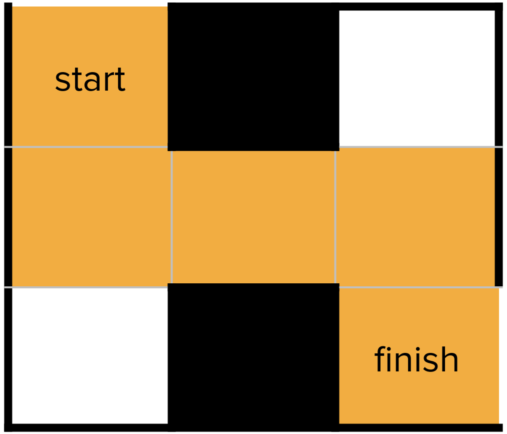
- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze



A **recursive** algorithm for solving mazes

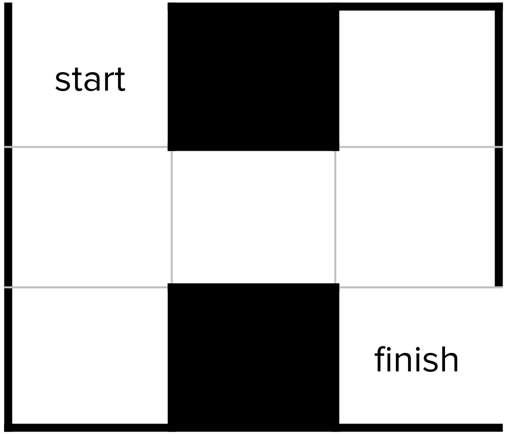
- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze

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?

Exercise for home: Draw the 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**

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.

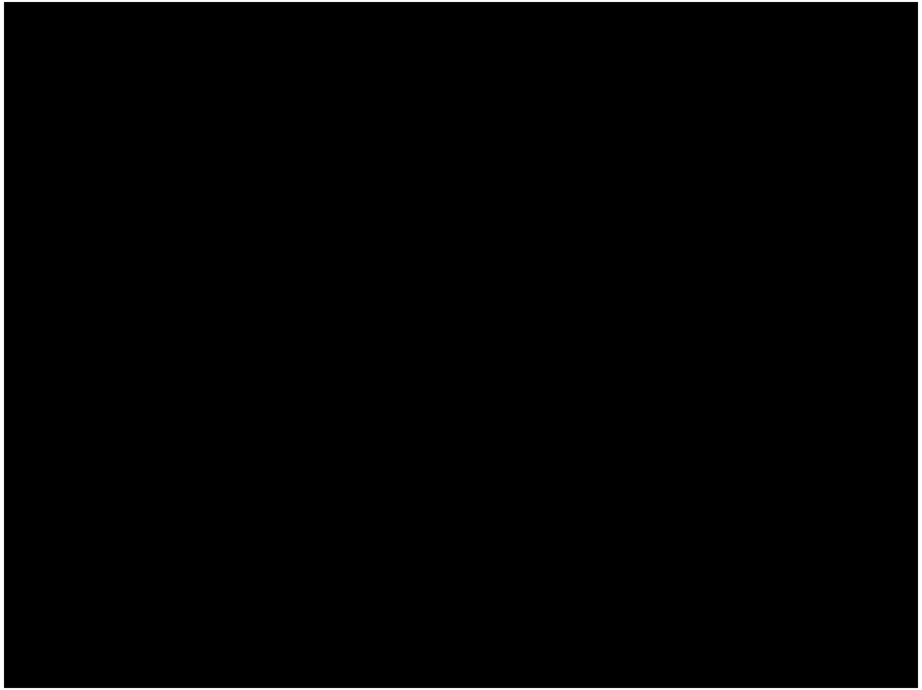
- Our helper function may have a different return type from our initial function prototype, and our wrapper function (not the helper) may be more complex than just a call to our helper function.

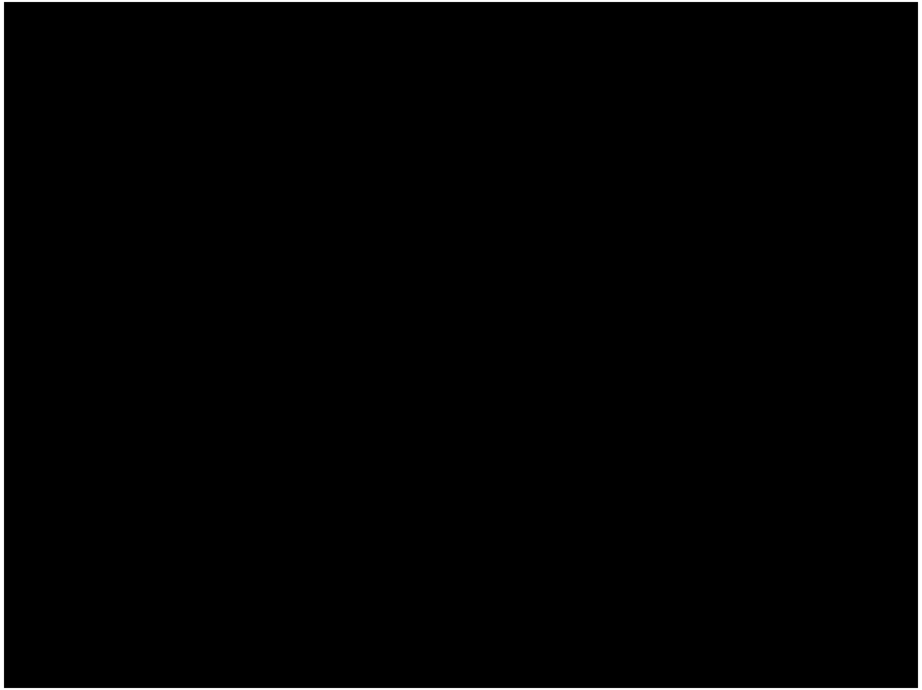
- It may be helpful to revisit and adjust our initial answers to our planning questions as we determine more about the algorithm we want to use (e.g. adding a parameter to our helper function).

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

"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!**

The Knapsack Problem

The Knapsack Problem

- Imagine yourself in a new lifestyle as a professional wilderness survival expert

The Knapsack Problem

- Imagine yourself in a new lifestyle as a professional wilderness survival expert

The Knapsack Problem

- Imagine yourself in a new lifestyle as a professional wilderness survival expert
- You are about to set off on a challenging expedition, and you need to pack your knapsack (or backpack) full of supplies.

The Knapsack Problem

- Imagine yourself in a new lifestyle as a professional wilderness survival expert
- You are about to set off on a challenging expedition, and you need to pack your knapsack (or backpack) full of supplies.
- You have a list full of supplies (each of which has a survival value and a weight associated with it) to choose from.

The Knapsack Problem

- Imagine yourself in a new lifestyle as a professional wilderness survival expert
- You are about to set off on a challenging expedition, and you need to pack your knapsack (or backpack) full of supplies.
- You have a list full of supplies (each of which has a survival value and a weight associated with it) to choose from.
- Your backpack is only sturdy enough to hold a certain amount of weight.

The Knapsack Problem

- Imagine yourself in a new lifestyle as a professional wilderness survival expert
- You are about to set off on a challenging expedition, and you need to pack your knapsack (or backpack) full of supplies.
- You have a list full of supplies (each of which has a survival value and a weight associated with it) to choose from.
- Your backpack is only sturdy enough to hold a certain amount of weight.
- Question: How can you **maximize the survival value** of your backpack?

Breakout Rooms: Solve a small knapsack example

The "Greedy" Approach

What happens if you always choose to include the item with the highest value that will still fit in your backpack?

Rope

- Value: 3
- Weight: 2

Axe

- Value: 4
- Weight: 3

Tent

- Value: 5
- Weight: 4

Canned food

- Value: 6
- Weight: 5

The "Greedy" Approach

What happens if you always choose to in hest value that will still fit in your backpack?

Bag is full!

Rope
- Value: 3
- Weight: 2

Axe
- Value: 4
- Weight: 3

Tent
- Value: 5
- Weight: 4

Canned food
- Value: 6
- **Weight: 5**

The "Greedy" Approach

What happens if you always choose to in

Why doesn't this work?

hest value that will still fit in your backpack?

Rope
- Value: 3
- Weight: 2

Axe
- Value: 4
- Weight: 3

Tent
- Value: 5
- Weight: 4

Canned food
- Value: 6
- **Weight: 5**

The "Greedy" Approach

What happens if you always choose hest value that will still fit in your backpack?

Items with lower individual values may sum to a higher total value!

Rope
- **Value: 3**
- Weight: 2

Axe
- **Value: 4**
- Weight: 3

Tent
- Value: 5
- Weight: 4

Canned food
- Value: 6
- Weight: 5

The Recursive Approach

Idea: Enumerate all subsets of weight <= 5 and pick the one with best total value.

The Recursive Approach

Idea: **Enumerate all subsets of weight <= 5** and pick the one with best total value.

This is generating 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?

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

The Recursive Approach

Idea: **Enumerate all combinations** and pick the one with best total value.

The Recursive Approach

Idea: Enumerate all combinations and **pick the one with best total value**.

Our final backtracking use case: "Pick one best solution"!

(i.e. optimization)

The Recursive Approach

Idea: Enumerate all combinations and **pick the one with best total value**.

We'll need to keep track of the total value we're building up, but for this version of the problem, we won't worry about finding the actual best subset of items itself.

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?

Problem Setup

```
int fillBackpack(Vector<BackpackItem>& items, int targetWeight);
```

- Assume that we have defined a custom **`BackpackItem`** struct, which packages up an item's **`survivalValue`** (int) and **`weight`** (int).
- We need to return the max value we can get from a combination of **`items`** under **`targetWeight`**.

Problem Setup

```
int fillBackpack(Vector<BackpackItem>& items, int targetWeight);
```

- Assume that we have defined a custom **`BackpackItem`** struct, which packages up an item's **`survivalValue`** (int) and **`weight`** (int).
- We need to return the max value we can get from a combination of **`items`** under **`targetWeight`**.

We need a helper function!

Pseudocode

- We need a helper function!

```
int fillBackpackHelper(Vector<BackpackItem>& items,
                                   int capacityRemaining, int curValue);
```

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 defines our knapsack decision tree?

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

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

- Information we need to store along the way:
  - The total value so far
  - The remaining elements to choose from
  - The remaining capacity (weight) in the backpack

What defines our knapsack decision tree?

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

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

- Information we need to store along the way:
  - The total value so far
  - The remaining elements to choose from
  - The remaining capacity (weight) in the backpack

This should look very similar to our previous combinations problem!

Pseudocode

- **Recursive case**:
  - Select an unconsidered item.
  - Recursively calculate the values both with and without the item.
  - Return the higher value.

- **Base cases**:
  - No remaining capacity in the knapsack ➜ return 0
    (not a valid combination with weight <= 5)
  - No more items to choose from ➜ return current value

Let's code it!

(if time allows)

Challenge extensions on knapsack

(for you to try at home)

Challenge #1: Improving our efficiency

- For efficiency, we'll use an **index** to keep track of which items we've already looked at inside **items**:

```
int fillBackpackHelper(Vector<BackpackItem>& items,
                                int capacityRemaining, int curValue,
                                int index);
```

Our adjusted pseudocode

- **Recursive case**:
  - Select an unconsidered item **based on the index**.
  - Recursively calculate the values both with and without the item.
  - Return the higher value.

- **Base cases**:
  - No remaining capacity in the knapsack ➜ return 0
    (not a valid combination with weight <= 5)
  - No more items to choose from ➜ return current value

Challenge #2: Tracking our items

- What if we wanted to know what combination of items resulted in the best value?

- Think about which answers to which questions in our recursive backtracking strategy would change.

Takeaways

- Finding the best solution to a problem (optimization) can often be thought of as an additional layer of complexity/decision making on top of the recursive enumeration we've seen before

- For "hard" problems, the best solution can only be found by enumerating all possible options and selecting the best one.

- Creative use of the return value of recursive functions can make applying optimization to an existing function straightforward.

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

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

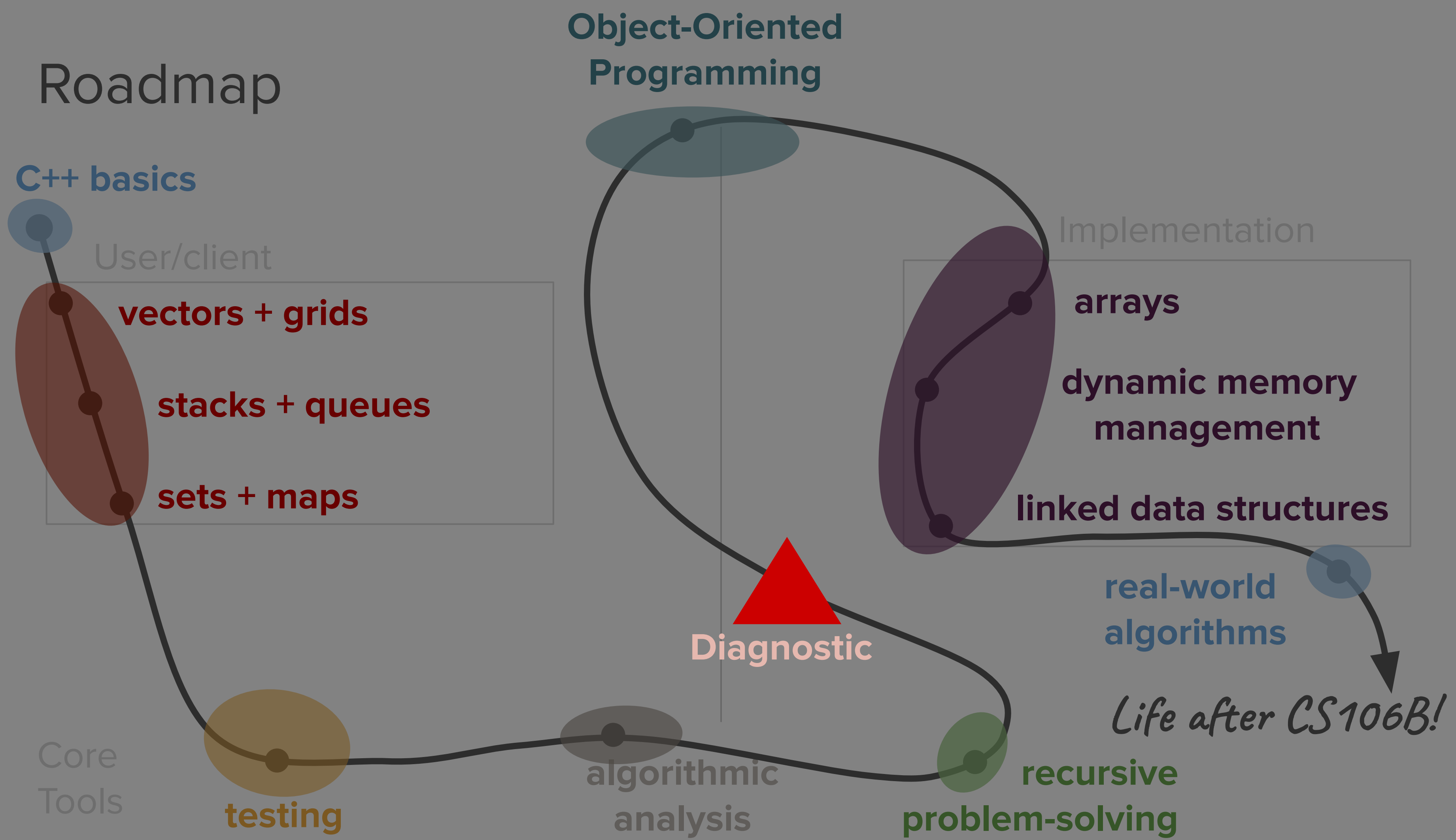
- Which of our three use cases does our problem fall into? (generate/count all solutions, find one solution/prove its existence, pick one best solution)
- What are we building up as our “many possibilities” in order to find our solution? (subsets, permutations, combinations, or something else)

- 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?

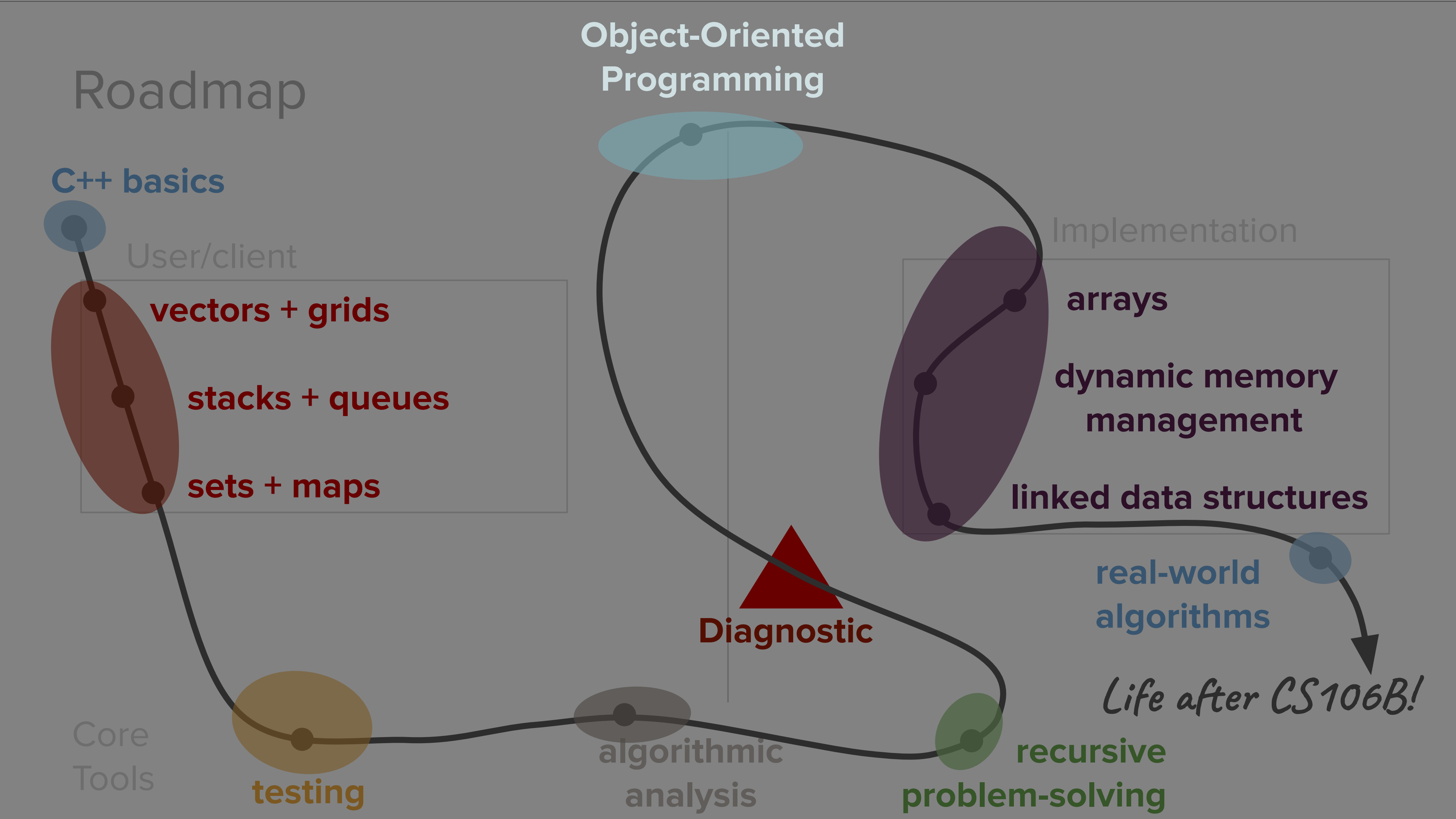
- What are our base and recursive cases?
  - What does my decision tree look like? (decisions, options, what to keep track of)
  - In addition to what we’re building up, are there any additional constraints on our solutions?
  - Does it make sense to use choose/explore/undo OR copy/edit/recurse for the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

Classes and Object-Oriented Programming