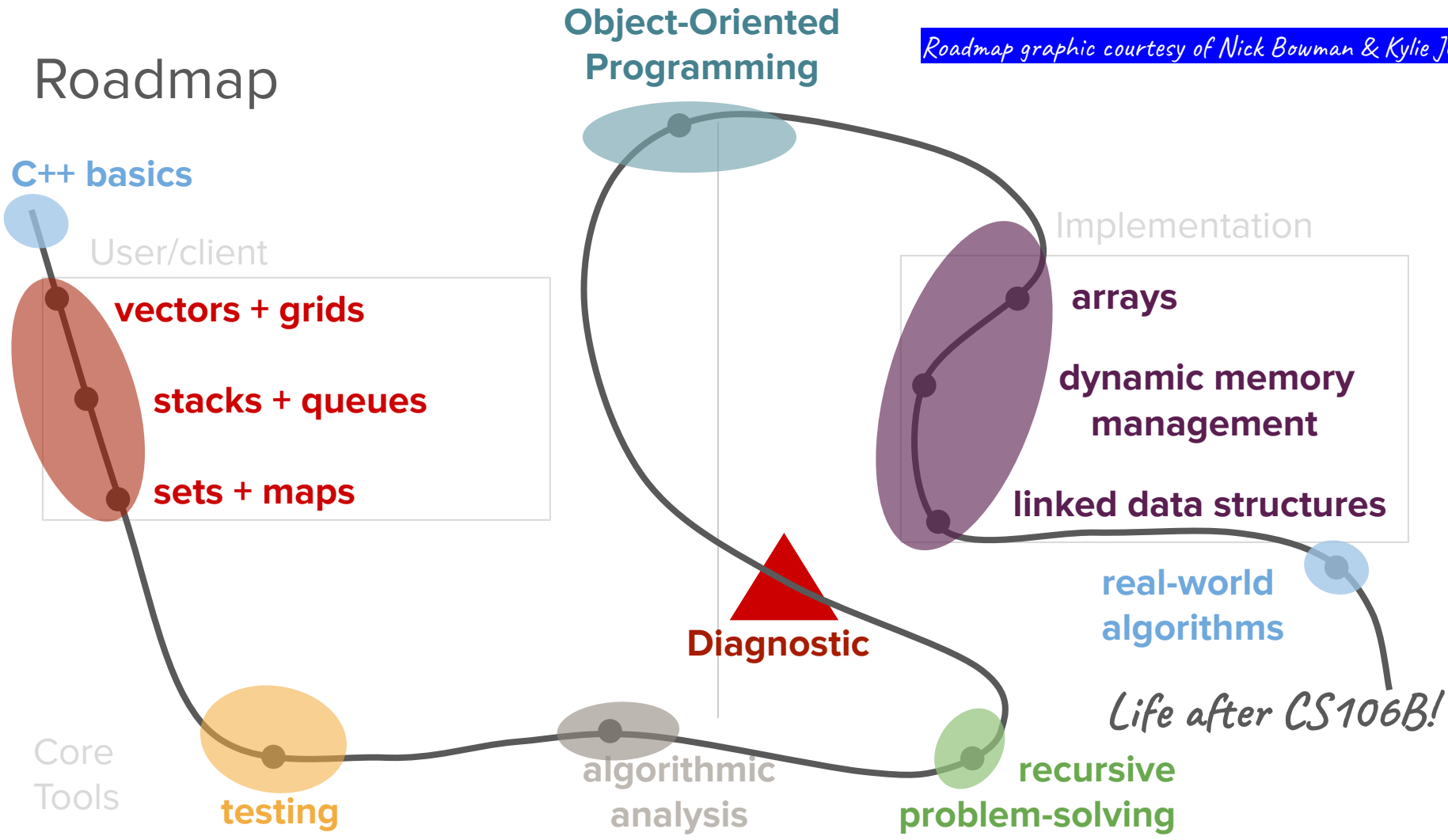
Recursive Backtracking and Optimization

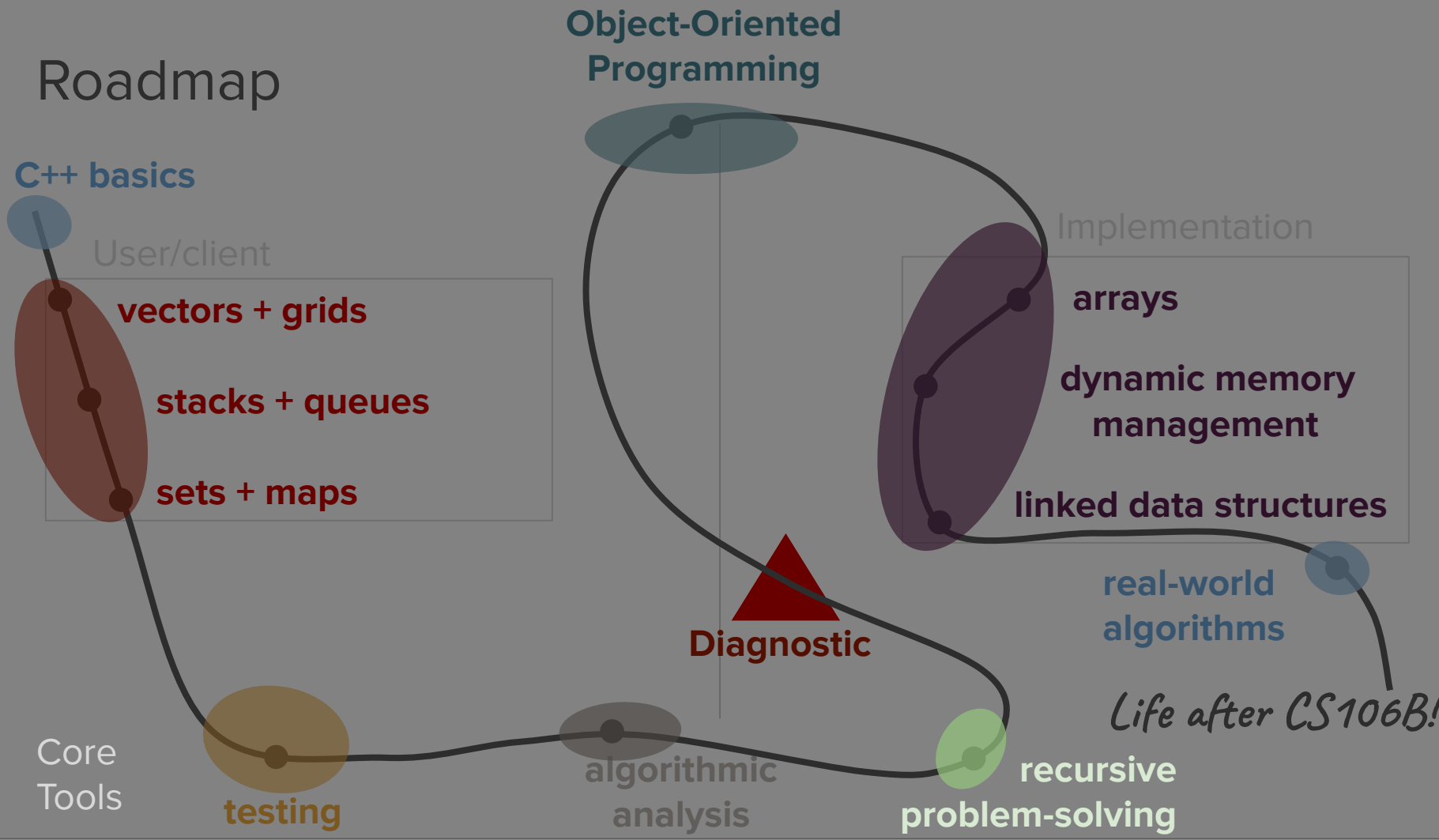
What has been your favorite part of the first 3 weeks of the course?

(please put your answers in the chat)

Roadmap


Object-Oriented Programming
Roadmap graphic courtesy of Nick Bowman & Kylie Jue
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

Roadma

C++ basics

User/clie

vectors

stack

sets

ntation

c memory

gement

a structures

world

ithms

after CS106B!

Core

Tools

Today's question

How can we use recursive backtracking to find the **best** solution to very challenging problems?

Today's topics

1. Review
2. Solving Mazes with DFS
3. Combinations
4. The Knapsack Problem

Review

(recursive backtracking with data structures)

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

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

Permutations

A ***permutation*** is a
rearrangement
of the elements of a sequence.

Lassen Volcanic National Park

Yosemite National Park

Coconino National Forest

Lava Beds National Monument

What defines our permutations decision tree?

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

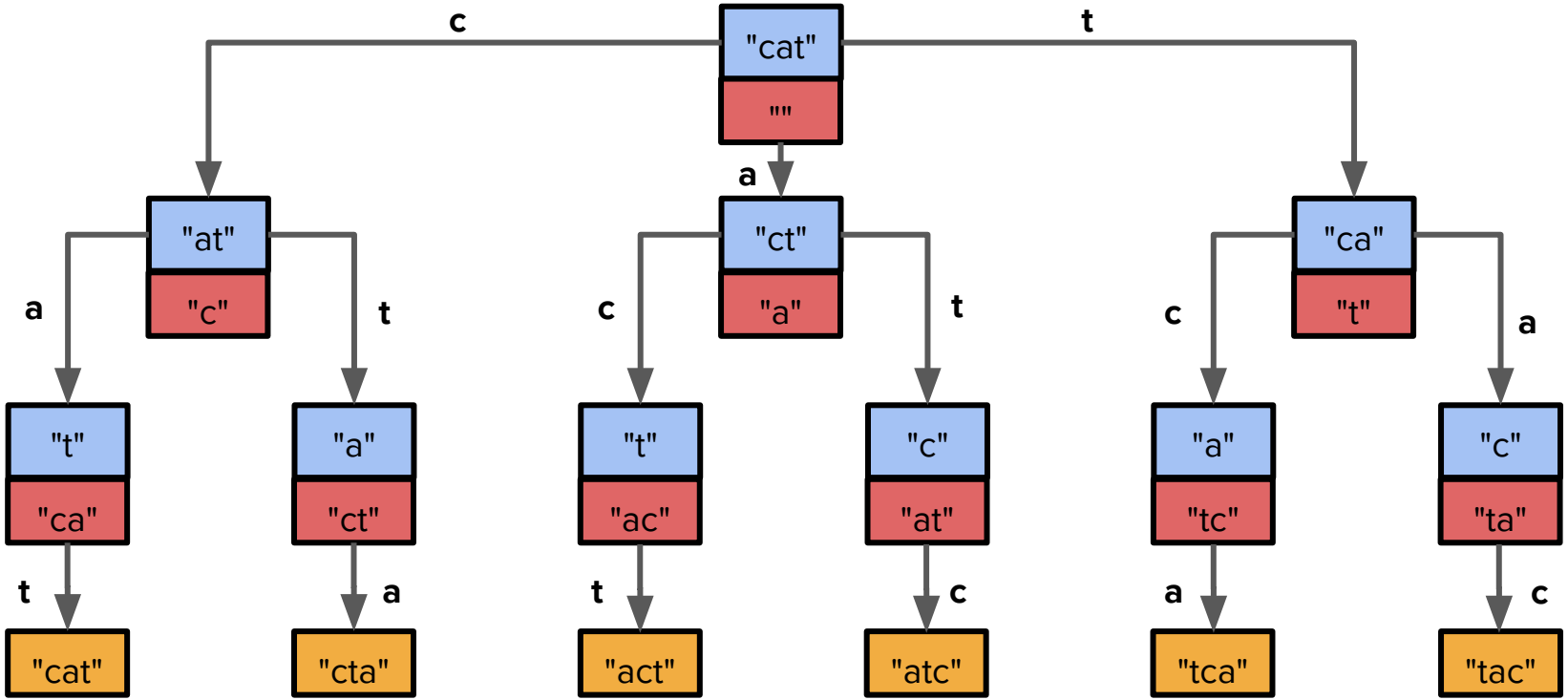
- **Options** at each decision (branches from each node):
  - One option for every remaining element that hasn't been selected yet
  - **Note: The number of options will be different at each level of the tree!**

- 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

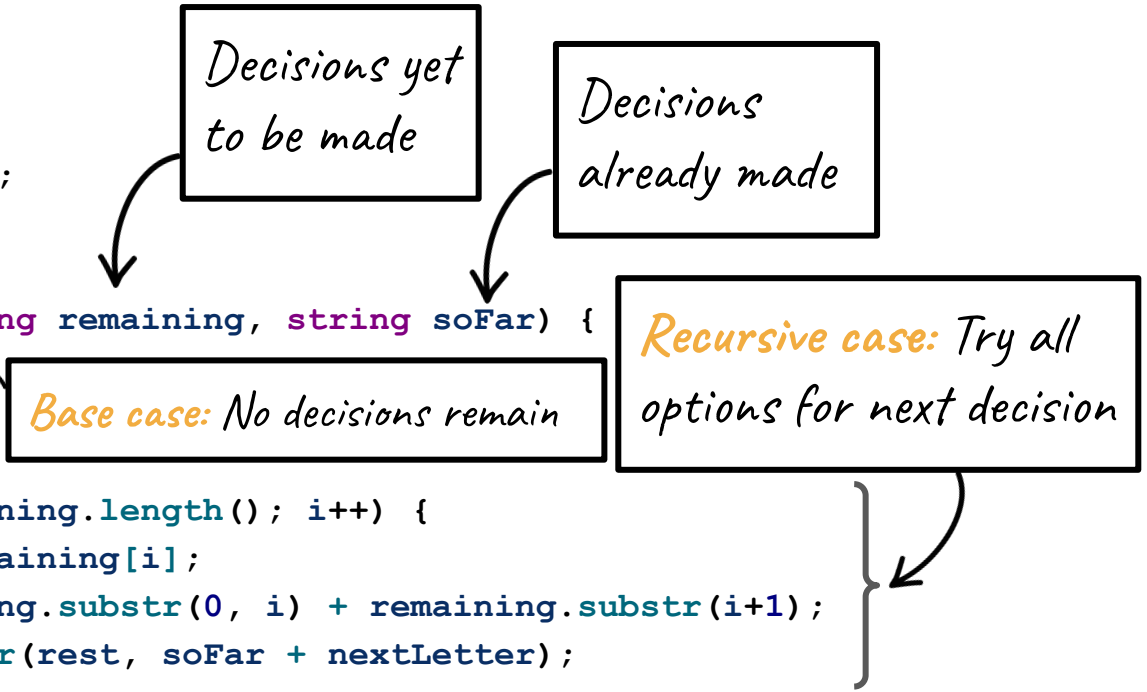
Decision tree: Find all permutations of "cat"



Permutations Code

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

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



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

Application: Shrinkable Words

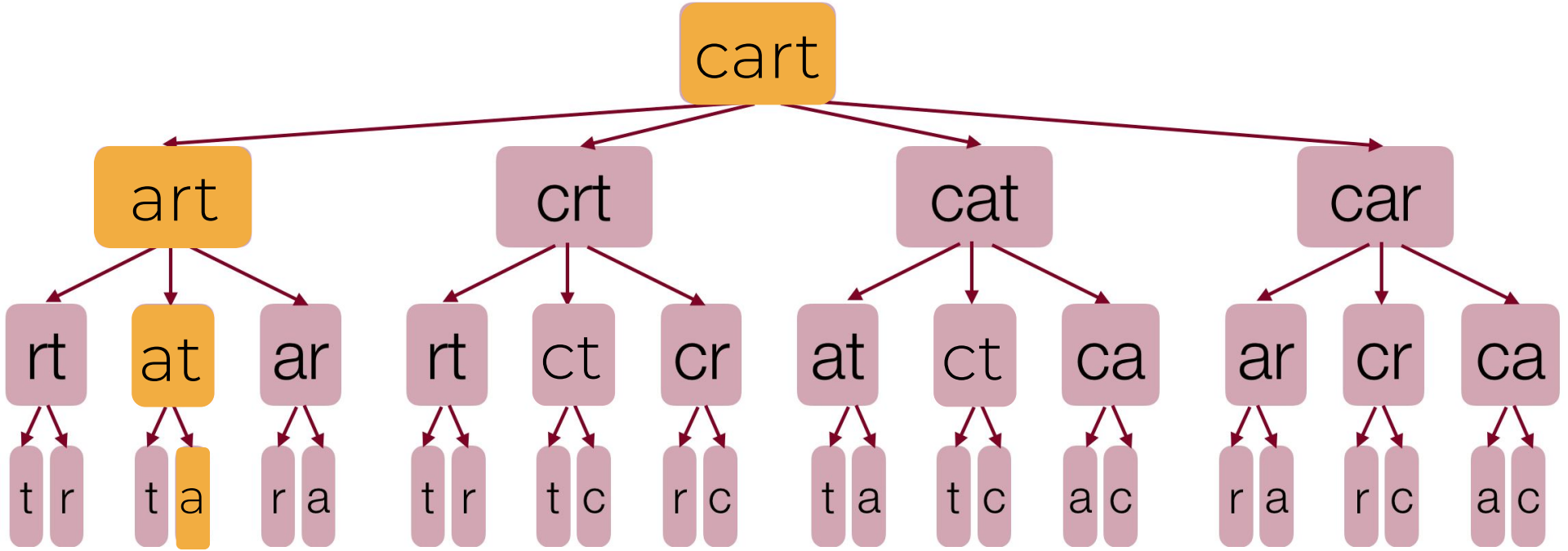
What defines our shrinkable decision tree?

- **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?



How do recursive backtracking solutions look different when data structures are involved?

Subsets

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!

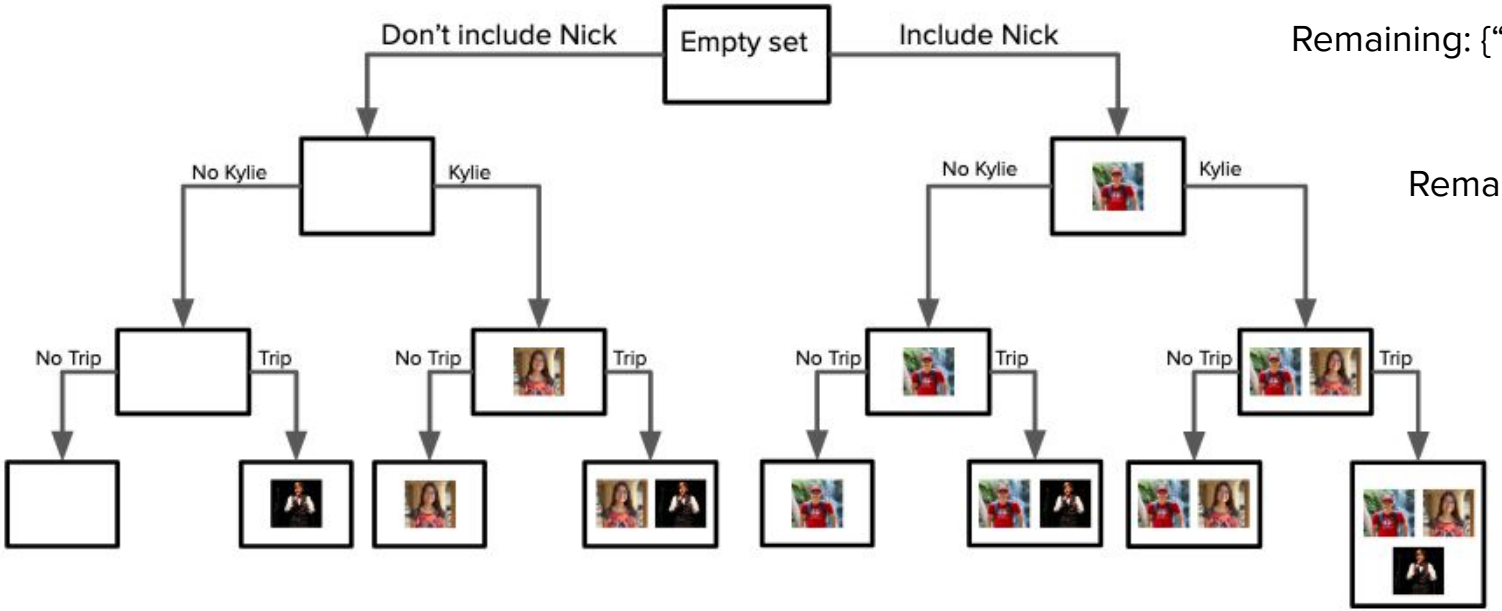
What defines our subsets decision tree?

- **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



Remaining: {“Nick”, “Kylie”, “Trip”}

Remaining: {“Kylie”, “Trip”}

Remaining: {“Trip”}

Remaining: {}

Subsets Summary

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

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

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

Subsets with a Property

Choosing an Unbiased Jury

What defines our jury selection decision tree?

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

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

- Information we need to store along the way:
  - The collection of candidates making up our jury so far
  - The remaining candidates to consider
  - The sum total bias of the current jury so far

Jury Selection Pseudocode

- Problem Setup
  - Assume that we have defined a custom **`juror`** struct, which packages up important information about a juror (their **`name`** and their **`bias`**, represented as an **`int`**)
  - Given a **`Vector<juror>`** (their may be duplicate name/bias pairs among candidates), we want to print out all possible unbiased juries that can be formed

- Recursive Case
  - Select a candidate that hasn't been considered yet.
  - Try not including them in the jury, and recursively find all possible unbiased juries.
  - Try including them in the jury, and recursively find all possible unbiased juries.

- Base Case
  - Once we're out of candidates to consider, check the bias of the current jury. If 0, display them!

Jury Selection Code v2.0

```
void findAllUnbiasedJuriesHelper(Vector<juror>& allCandidates, Vector<juror>& currentJury, int
currentBias, int index){
    if (index == allCandidates.size()){
        if (currentBias == 0){
            displayJury(currentJury);
        }
    } else {
        juror currentCandidate = allCandidates[index];

        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias, index + 1);
        currentJury.add(currentCandidate);
        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias + currentCandidate.bias,
index + 1);
        currentJury.remove(currentJury.size() - 1);
    }
}

void findAllUnbiasedJuries(Vector<juror>& allCandidates){
    Vector<juror> jury;
    findAllUnbiasedJuriesHelper(allCandidates, jury, 0, 0);
}
```

No more expensive addition/removal of possible candidates!

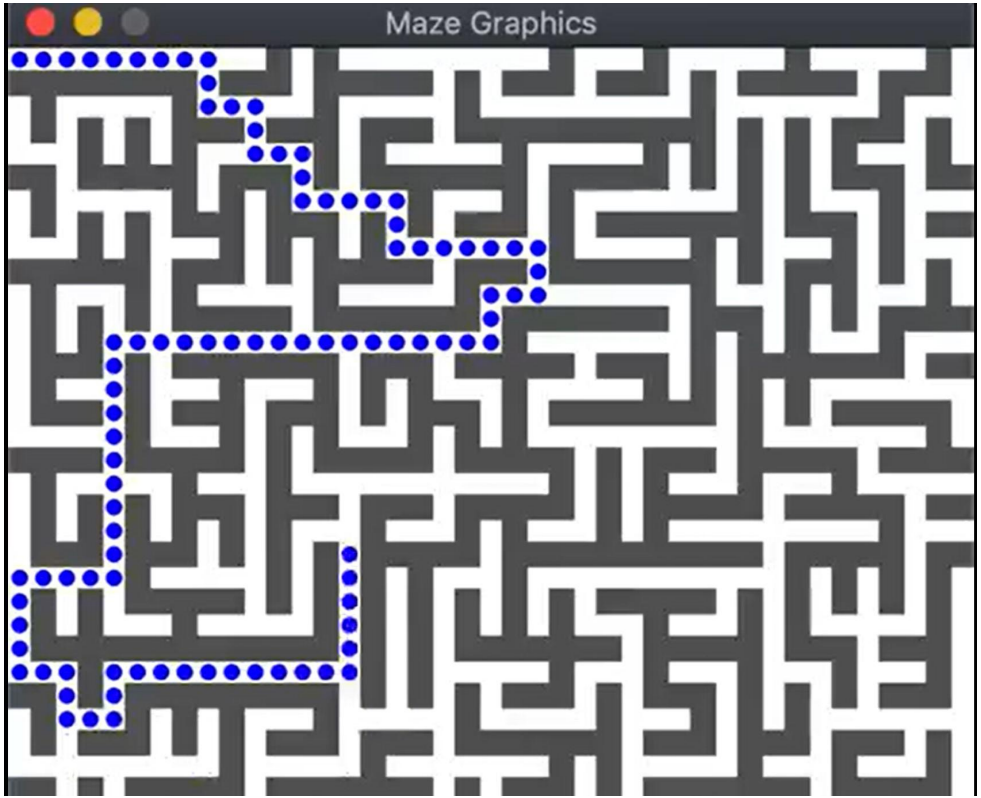
Jury Selection Summary

- Being able to enumerate all possible subsets and inspect subsets with certain constraints can be a powerful problem-solving tool.

- Maintaining an index of the current element under consideration for inclusion/exclusion in a collection is the most efficient way to keep track of the decision making process for subset generation

Revisiting mazes

Solving mazes with breadth-first search (BFS)

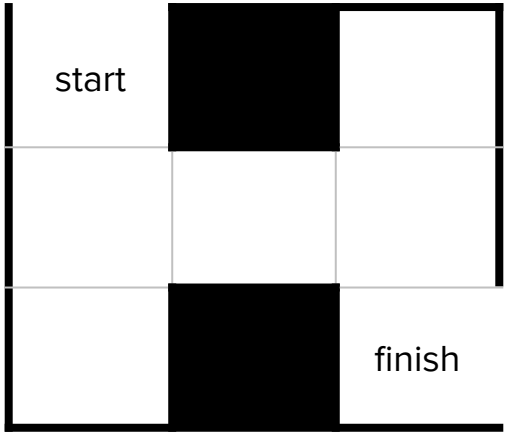


Solving mazes **recursively**

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

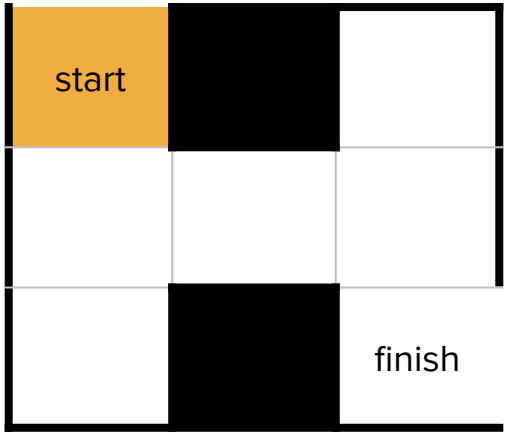
Solving mazes recursively

- Start at the entrance



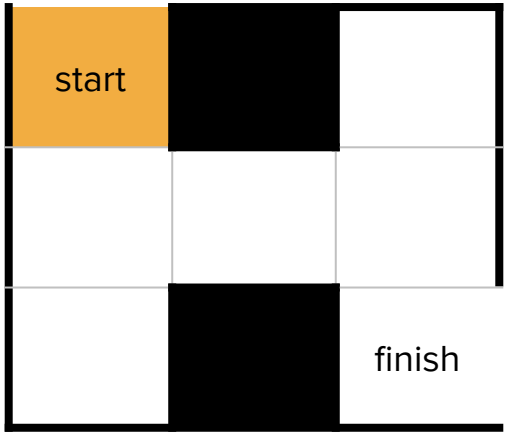
Solving mazes recursively

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



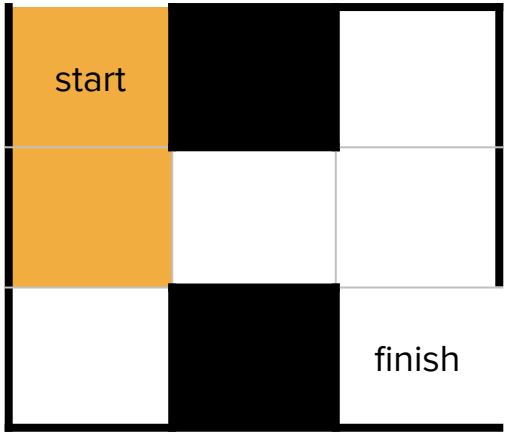
Solving mazes recursively

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



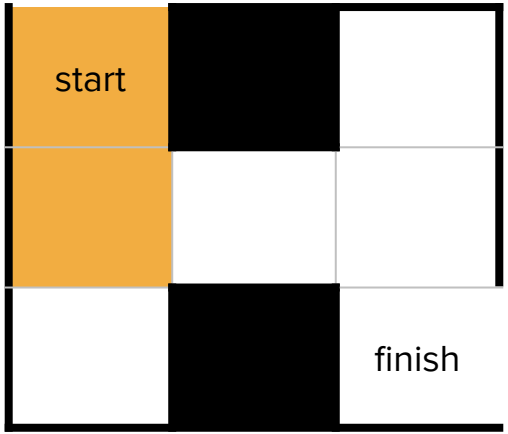
Solving mazes recursively

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



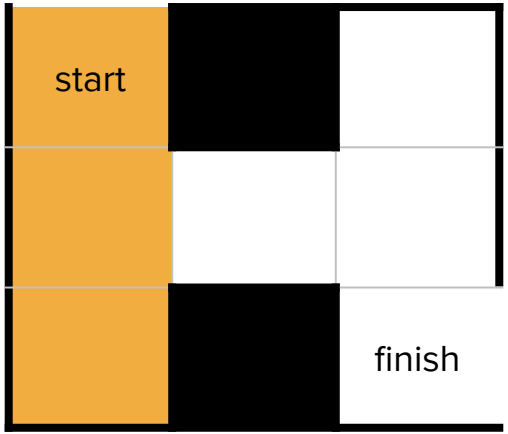
Solving mazes recursively

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



Solving mazes recursively

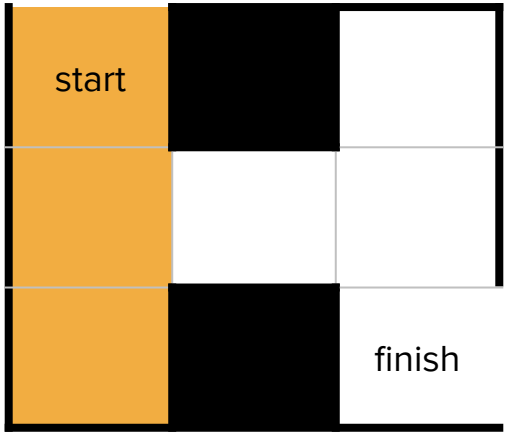
- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze



Solving mazes recursively

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

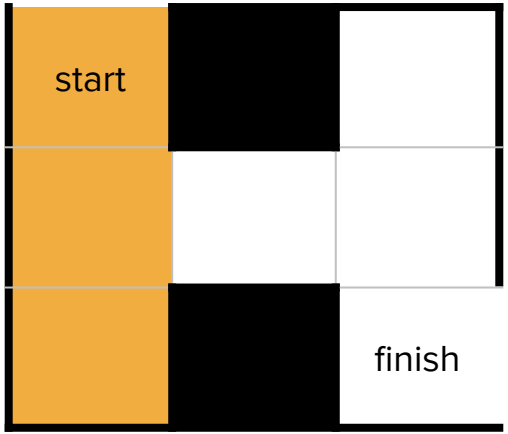
Dead end!
(cannot go North, South, East, or West)



Solving mazes recursively

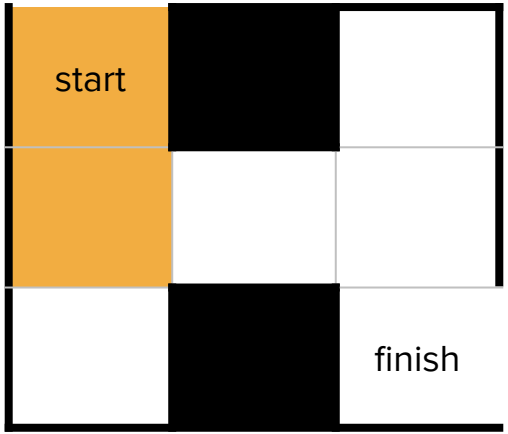
- Start at the entrance
- Take one step ~~North, South, East, or West~~

We must go back one step.



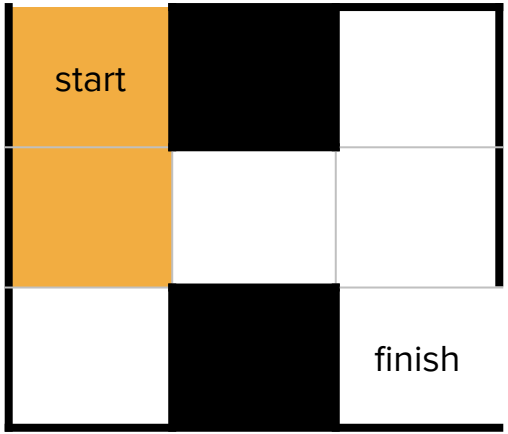
Solving mazes recursively

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



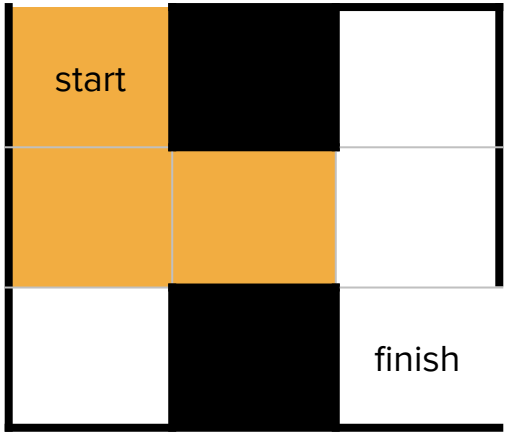
Solving mazes recursively

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



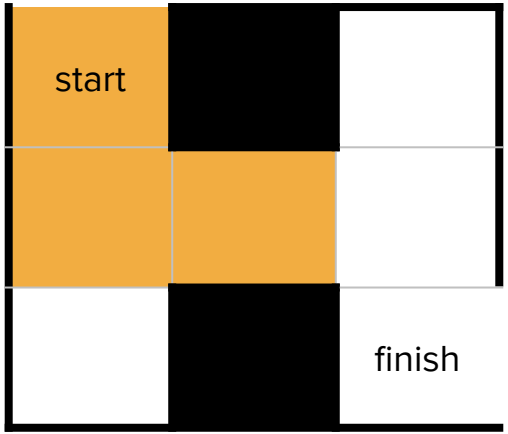
Solving mazes recursively

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



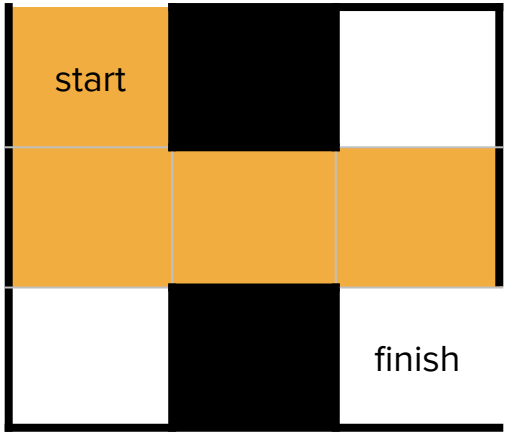
Solving mazes recursively

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



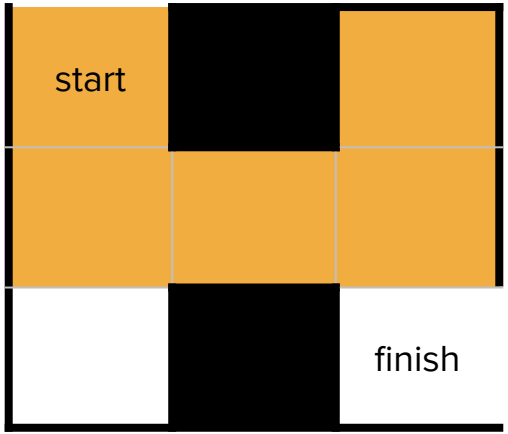
Solving mazes recursively

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



Solving mazes recursively

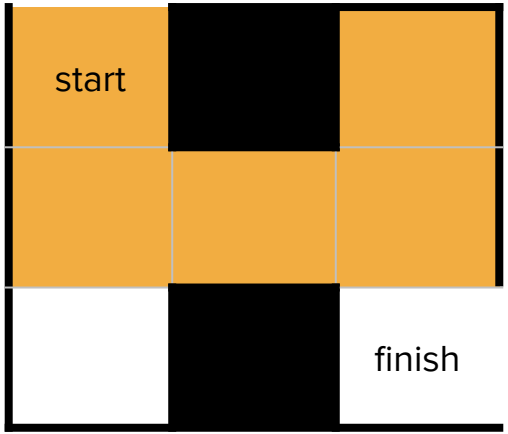
- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze



Solving mazes recursively

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

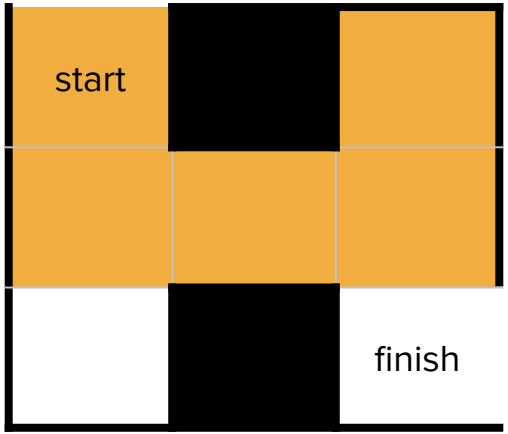
Dead end!
(cannot go North,
South, East, or West)



Solving mazes recursively

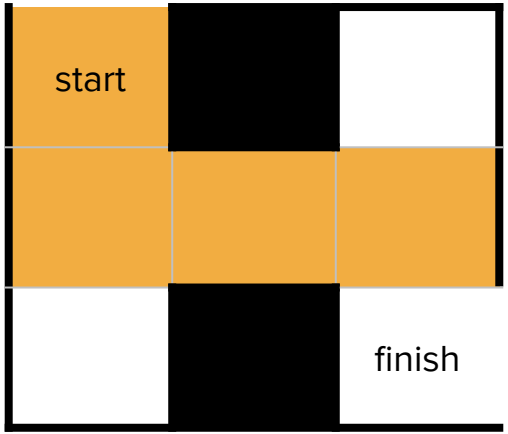
- Start at the entrance
- Take one step ~~North, South, East, or West~~

We must go back one step.



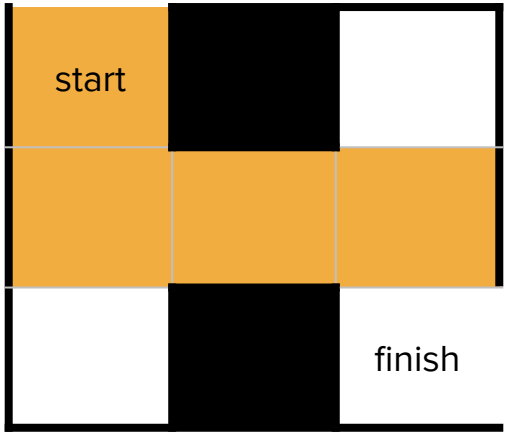
Solving mazes recursively

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



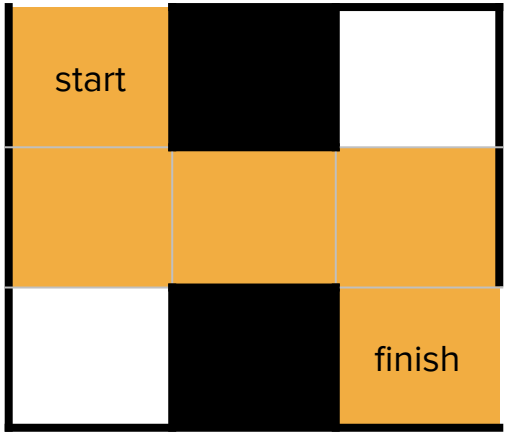
Solving mazes recursively

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



Solving mazes recursively

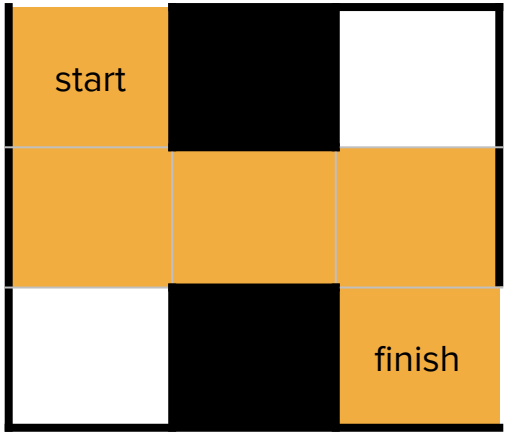
- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze



Solving mazes recursively

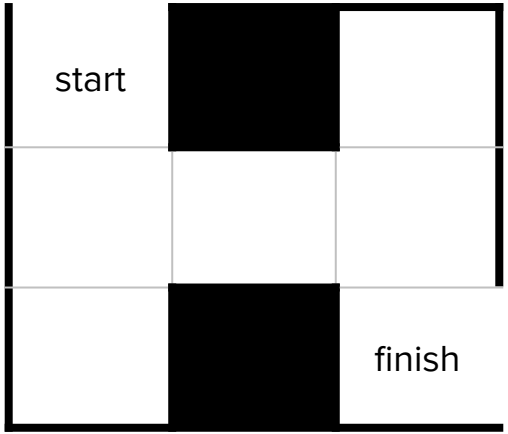
- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze

End of the maze!



Solving mazes recursively

- **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?

Exercise for home:
Draw the decision tree.

- **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

Pseudocode

- Recall our solveMaze prototype:

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

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

Pseudocode

- Recall our solveMaze prototype:

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

  We need a helper function!

Pseudocode

- Our helper function will have as **parameters**: the maze itself, the path we're building up, and the current location.

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

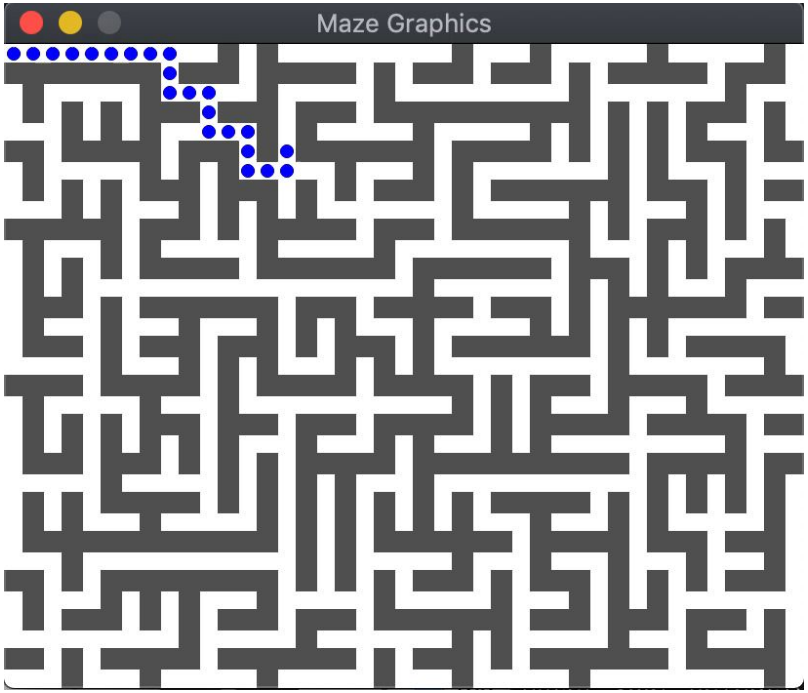
- **Base case**: We can stop exploring when we've reached the exit ➔ return true if the current location is the exit

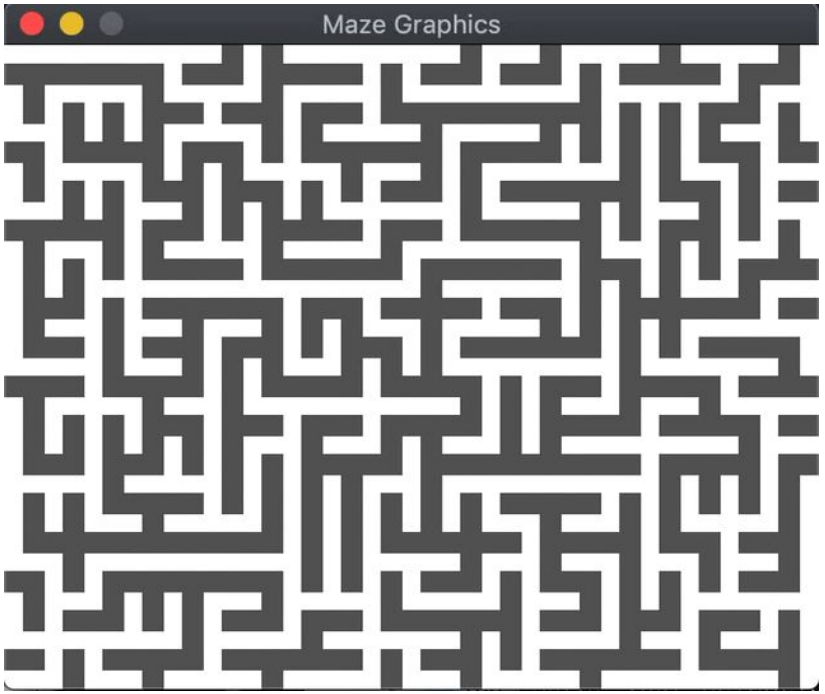
Let’s see the code!

Recursion is Depth-First Search (DFS)!

Breadth-First Search vs. Depth-First Search

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.

How can we use recursive backtracking to find the best solution to very challenging problems?

Using backtracking recursion

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

Combinations

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

What are **all the ways you can pick five justices** of the US 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.

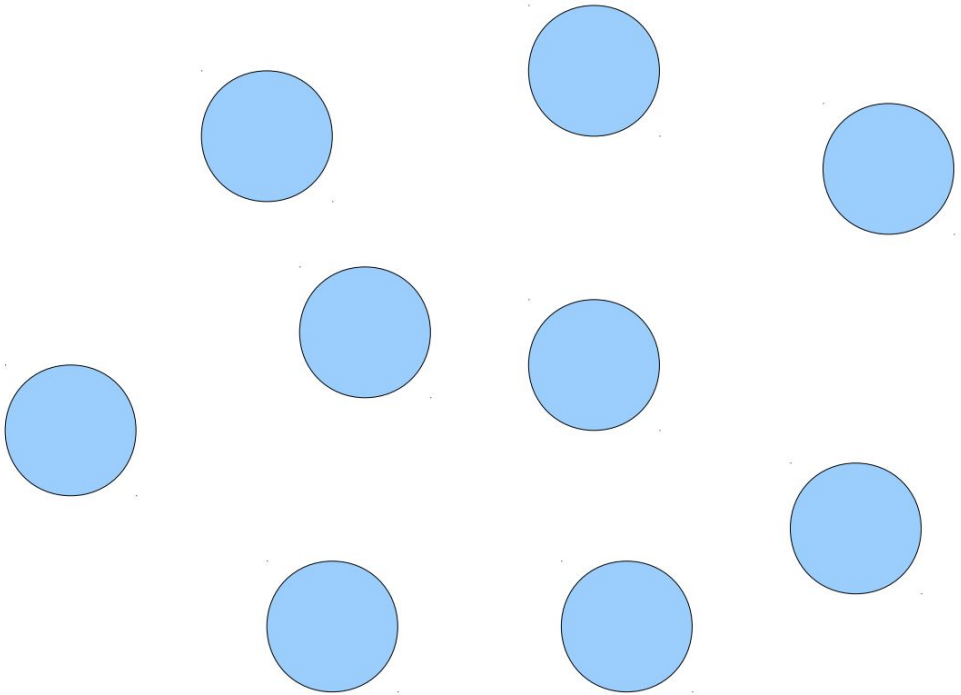
- This sounds very similar to the problem we solved when we generated subsets – these 5 justices would be a subset of the overall group of 9.

- 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"**

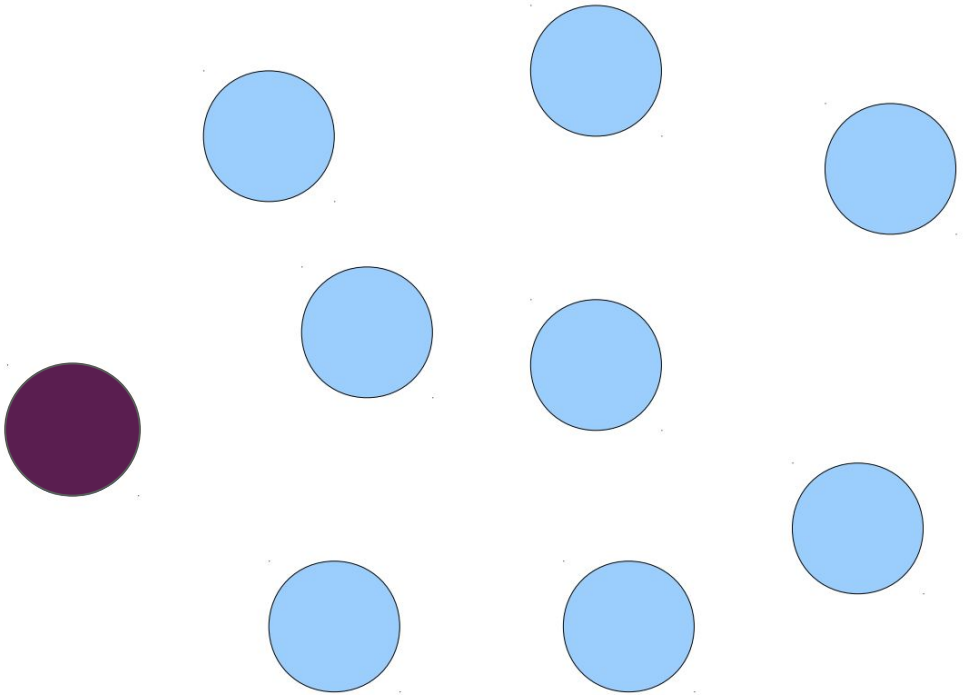
Subsets vs. Combinations

- 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 yesterday, 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!

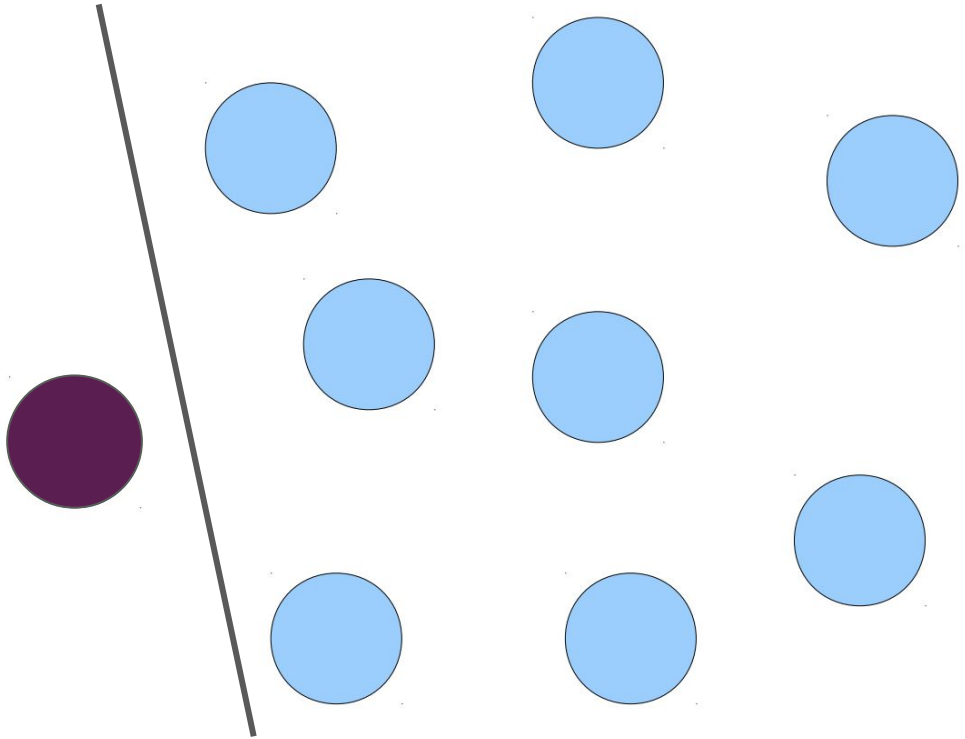
Generating Combinations



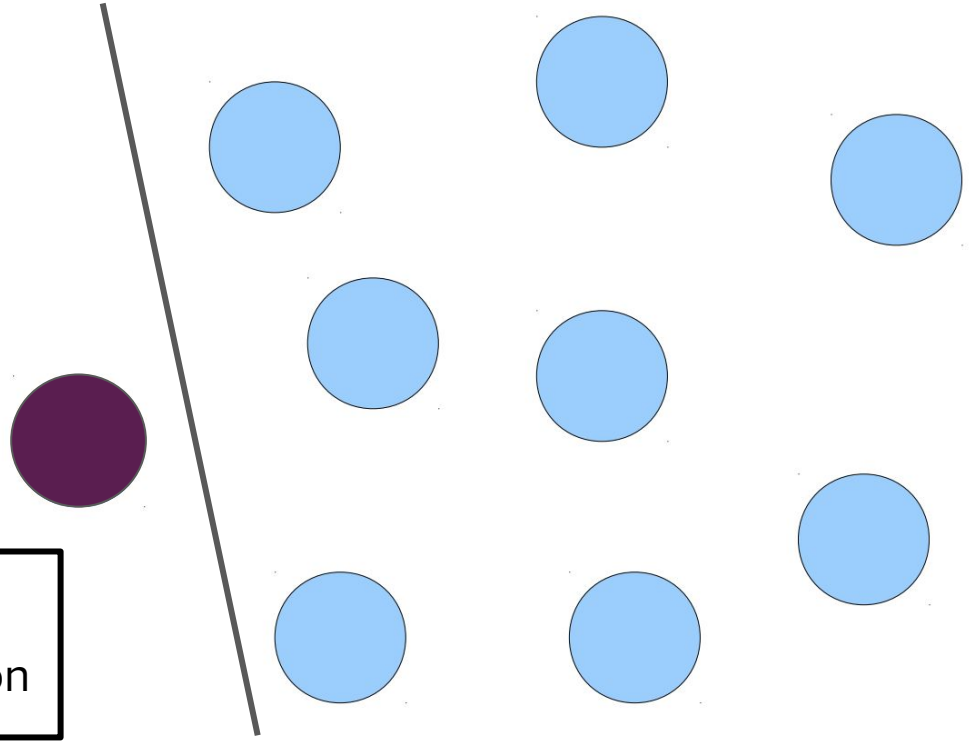
Generating Combinations



Generating Combinations

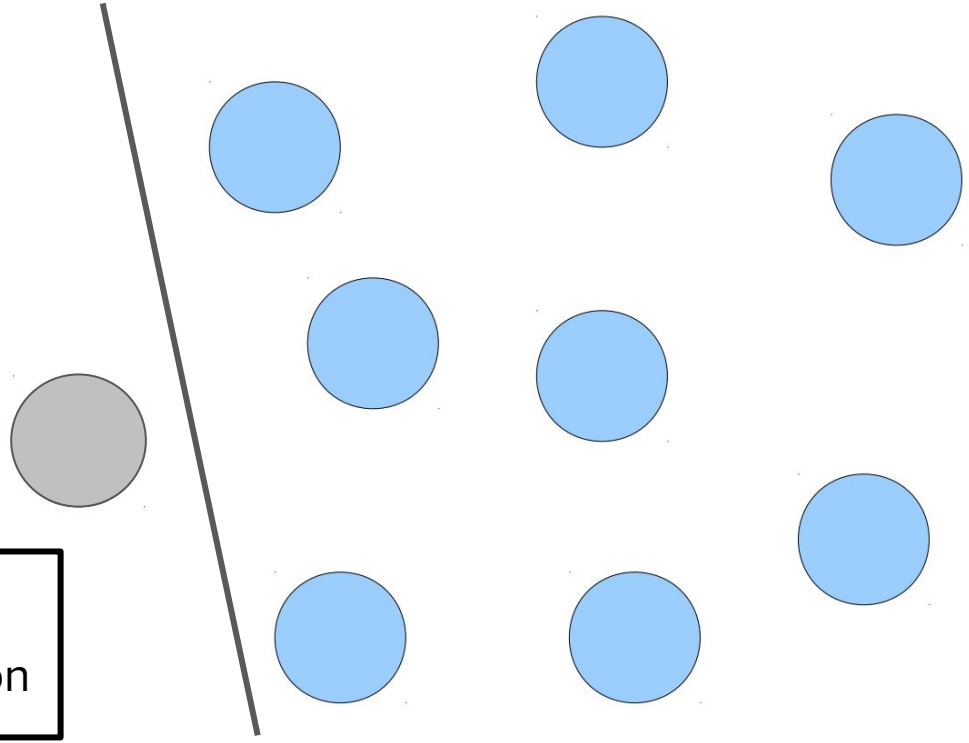


Generating Combinations



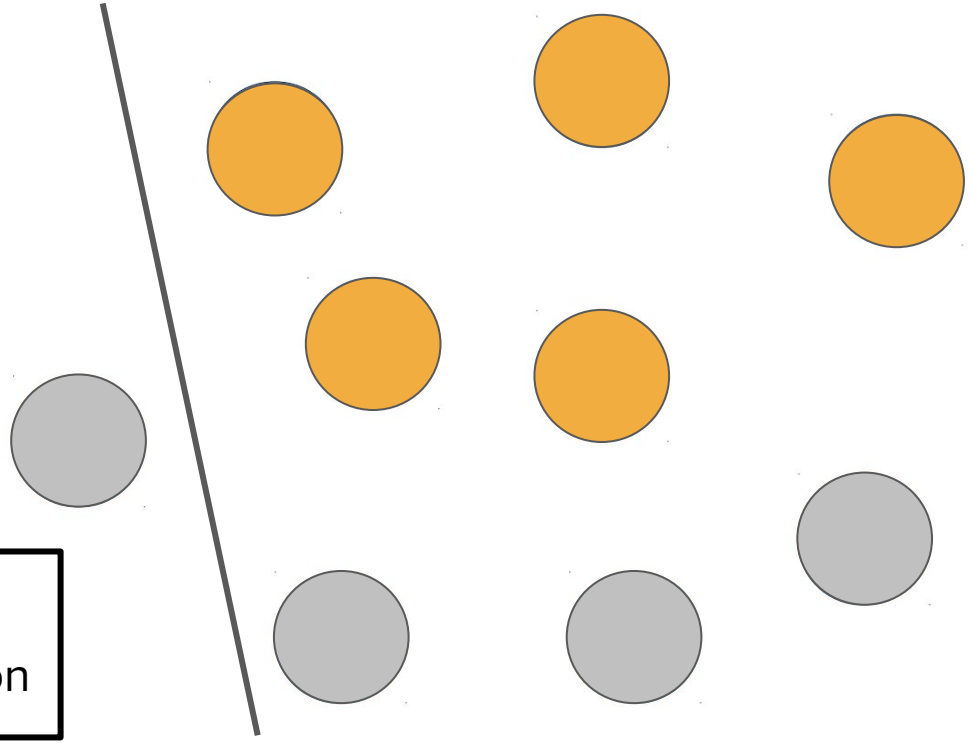
Option 1:
Exclude this person

Generating Combinations



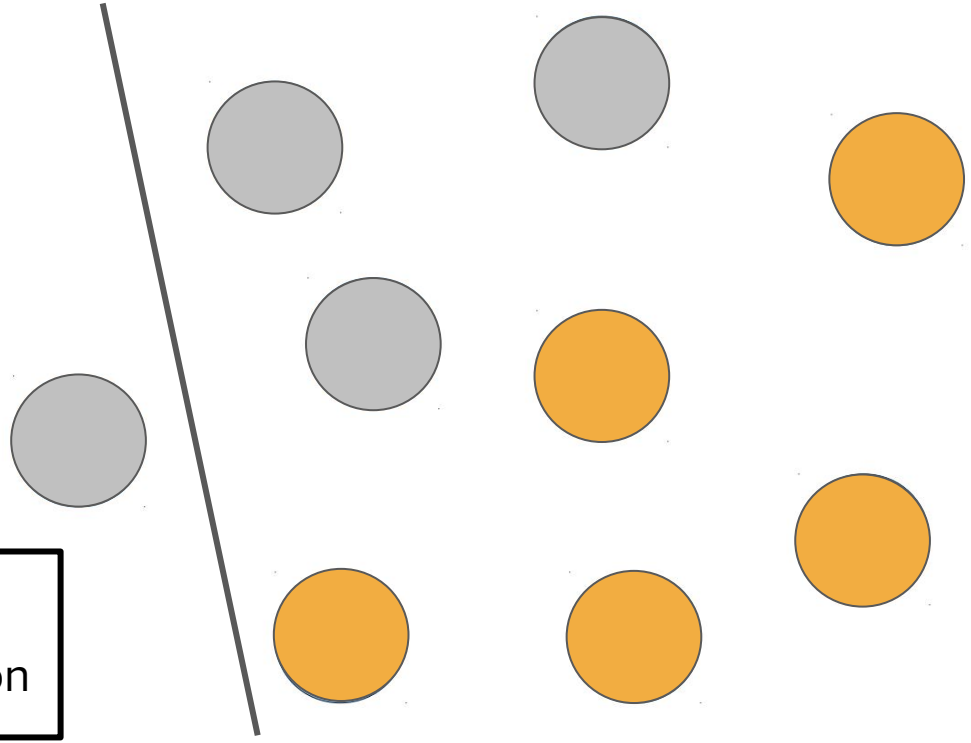
Option 1:
Exclude this person

Generating Combinations



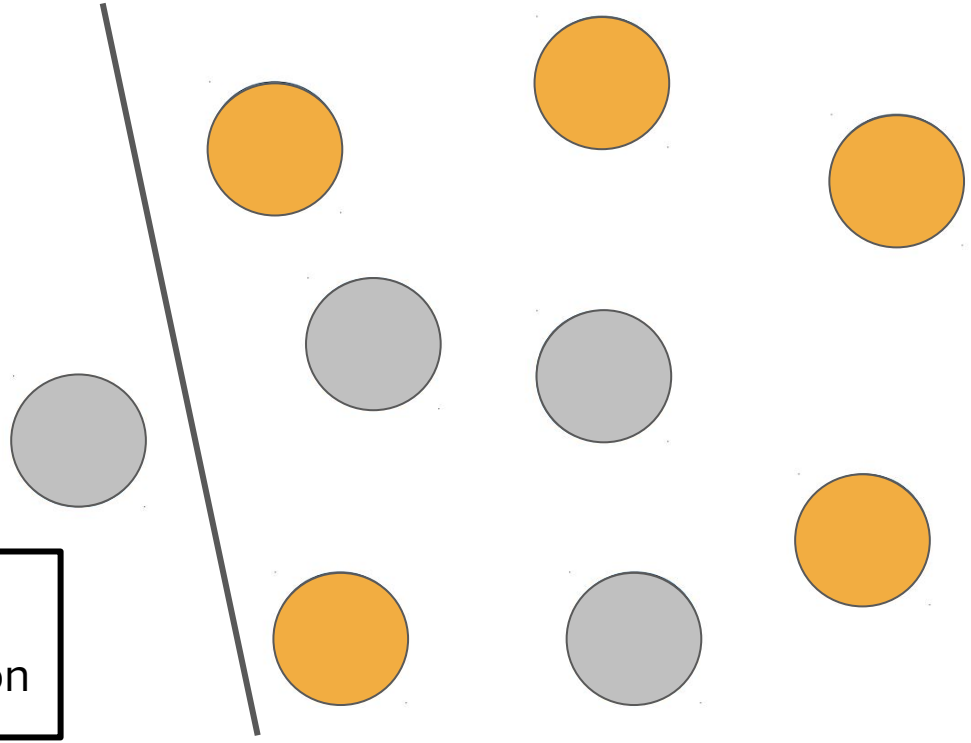
Option 1:
Exclude this person

Generating Combinations



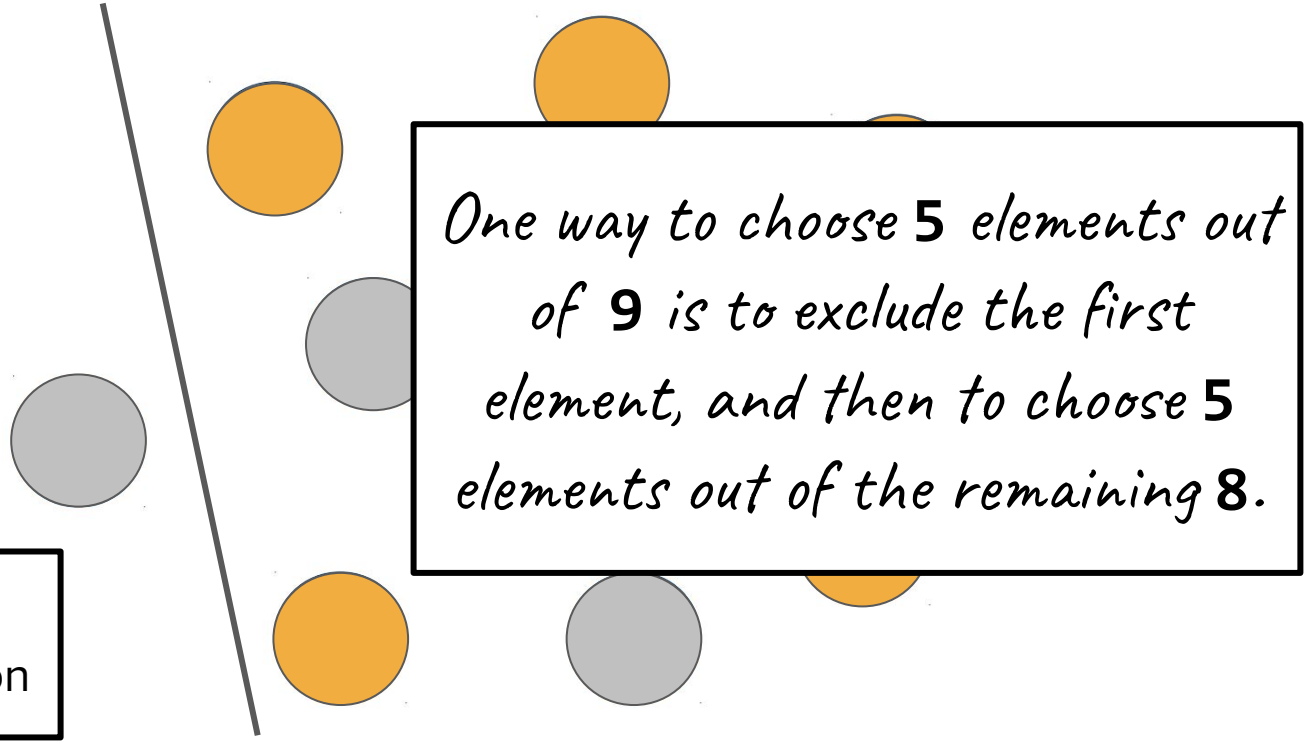
Option 1:
Exclude this person

Generating Combinations



Option 1:
Exclude this person

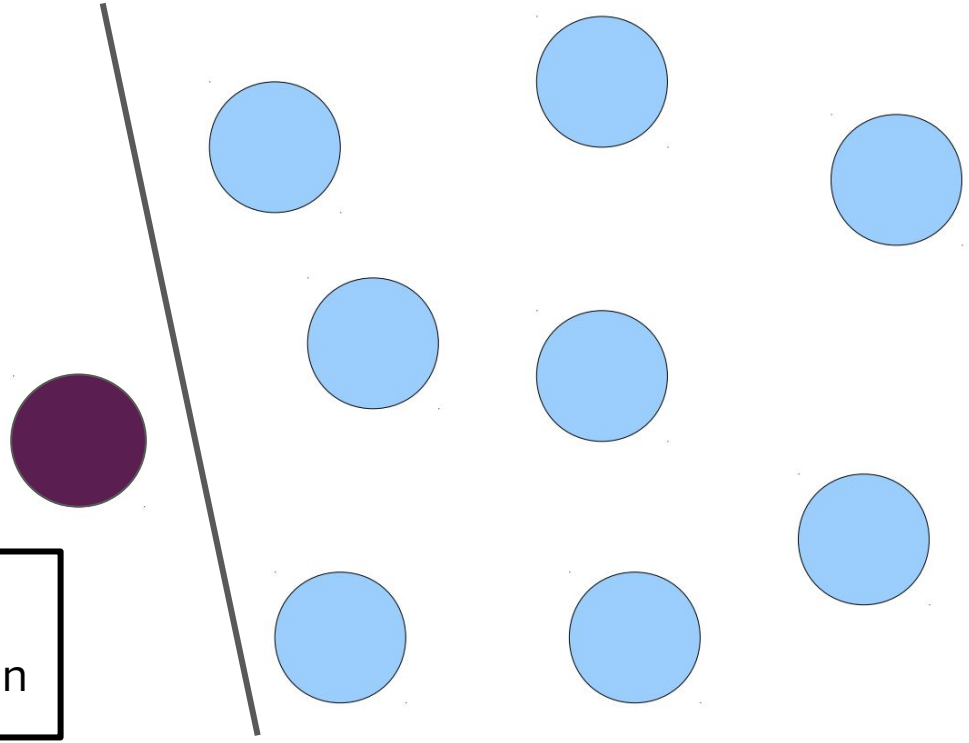
Generating Combinations



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**.

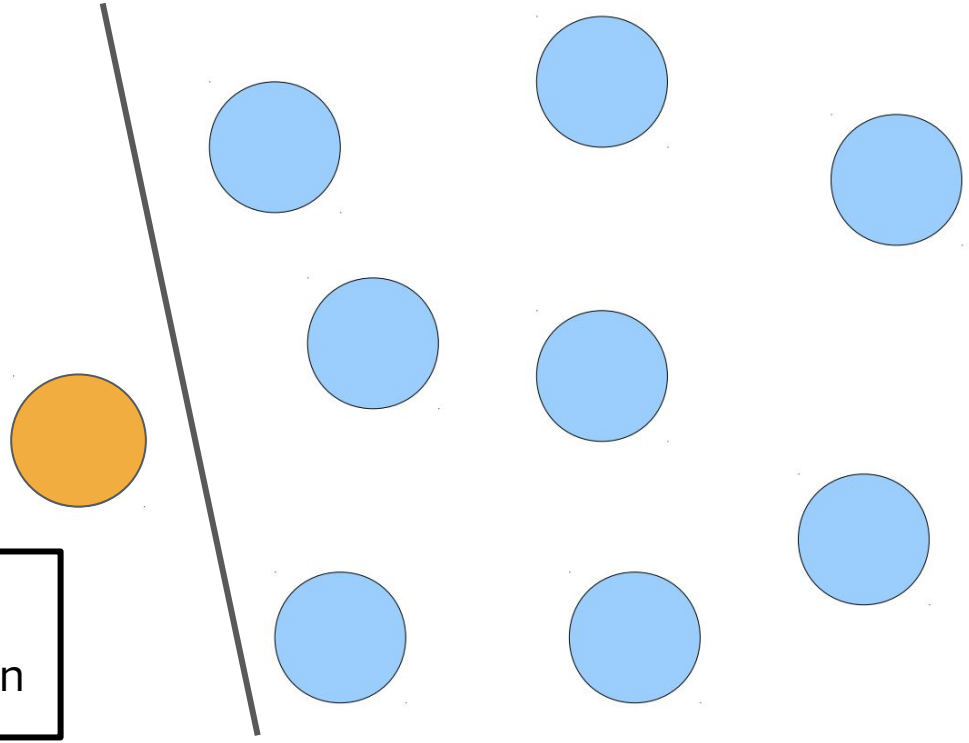
Option 1:
Exclude this person

Generating Combinations



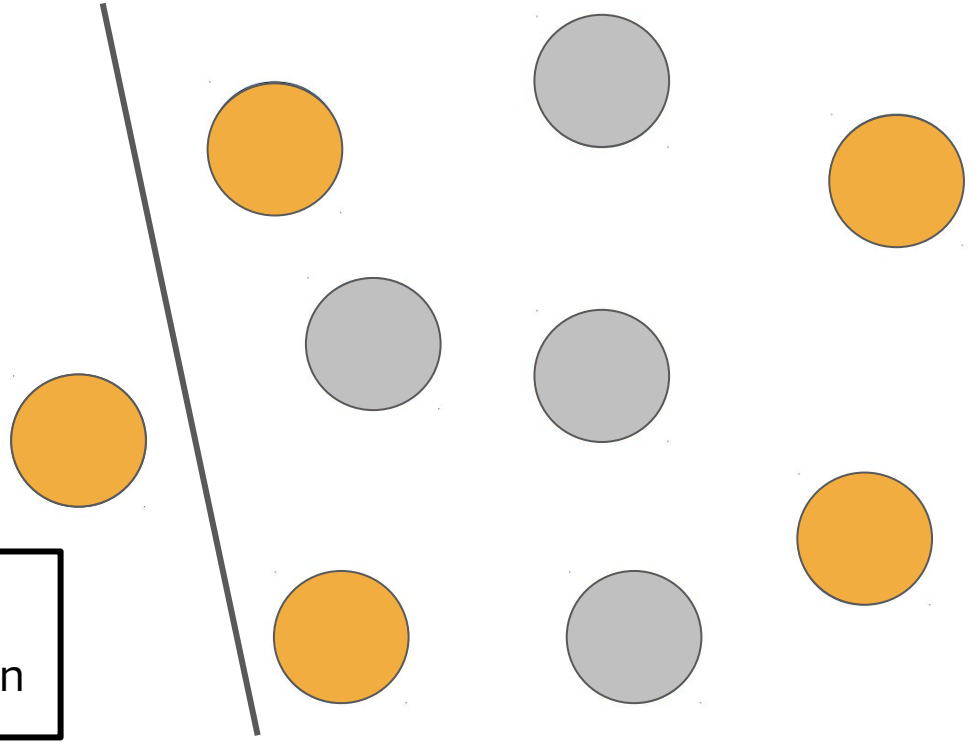
Option 2:
Include this person

Generating Combinations



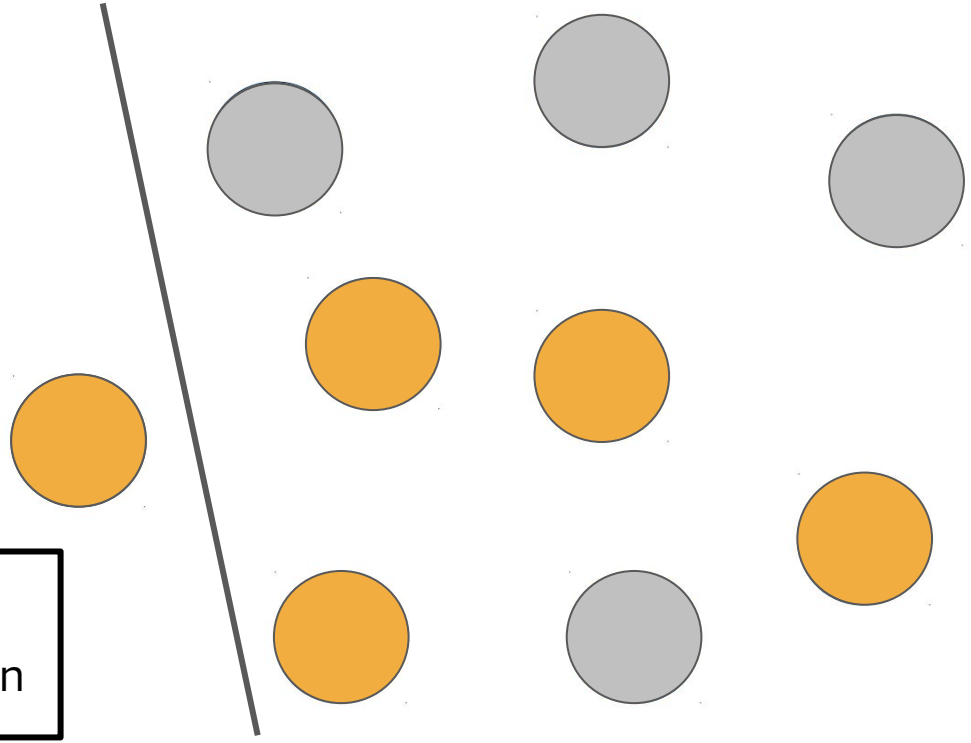
Option 2:
Include this person

Generating Combinations



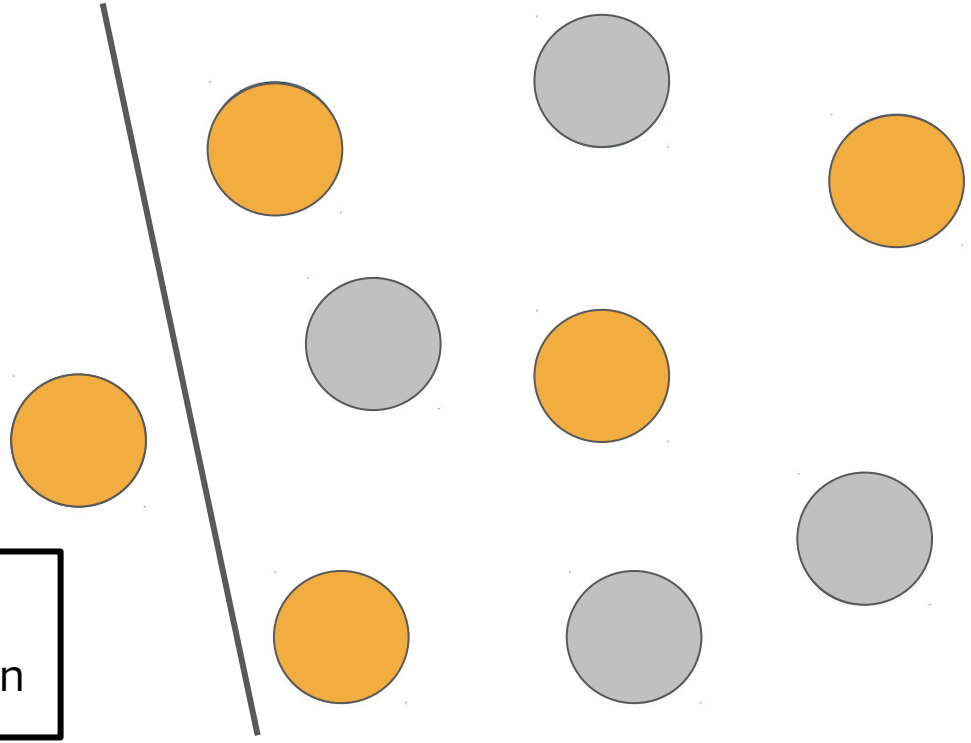
Option 2:
Include this person

Generating Combinations



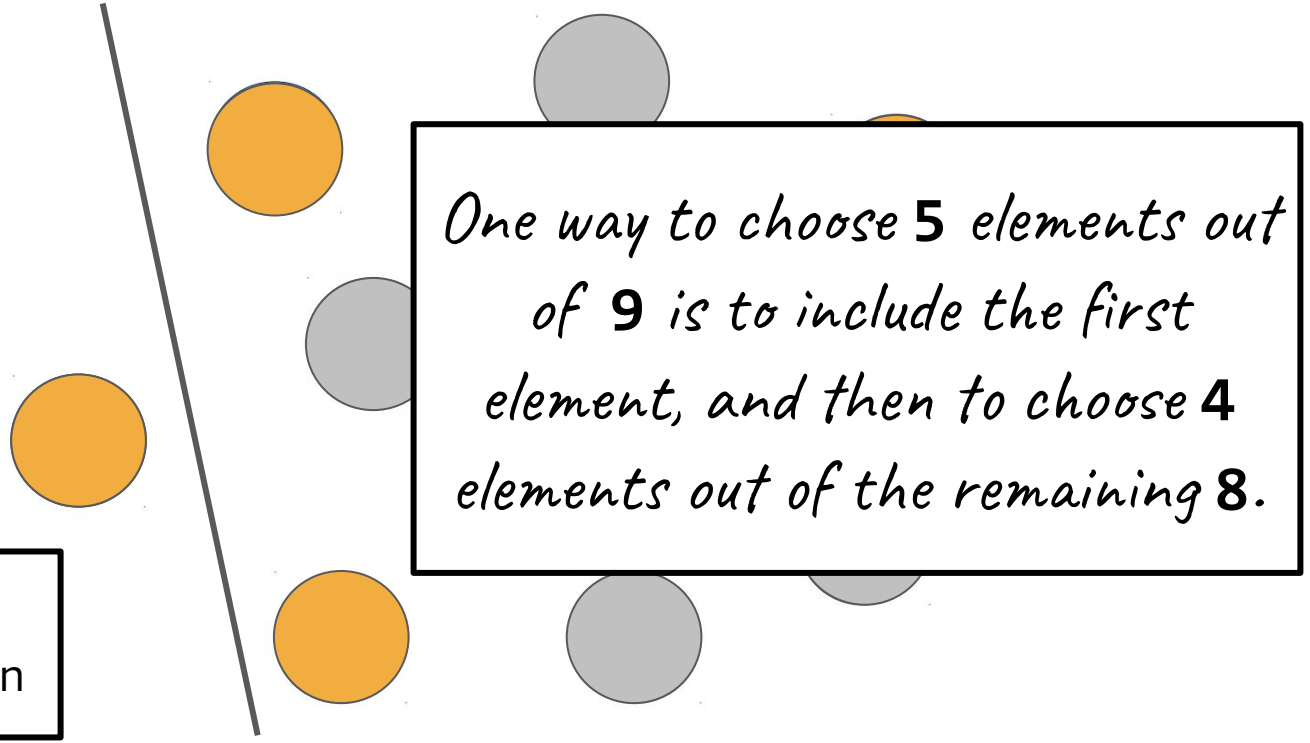
Option 2:
Include this person

Generating Combinations



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

- Each combination of **k** strings can be represented as a **`Set<string>`**.
- Before, 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!

Writing functions that build combinations

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

This is our function return type!

Writing functions that build combinations

- Suppose we get to the following scenario:

Pick 0 more Justices out of:

`{Kagan, Breyer}`

Chosen so far:

`{Barrett, 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:

`{Barrett, 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:

`{Barrett, 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 0 more Justices out of:
`{Sotomayor, Thomas}`
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 0 more Justices out of:
`{Sotomayor, Thomas}`
Chosen so far:
`{}`

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

Writing functions that build combinations

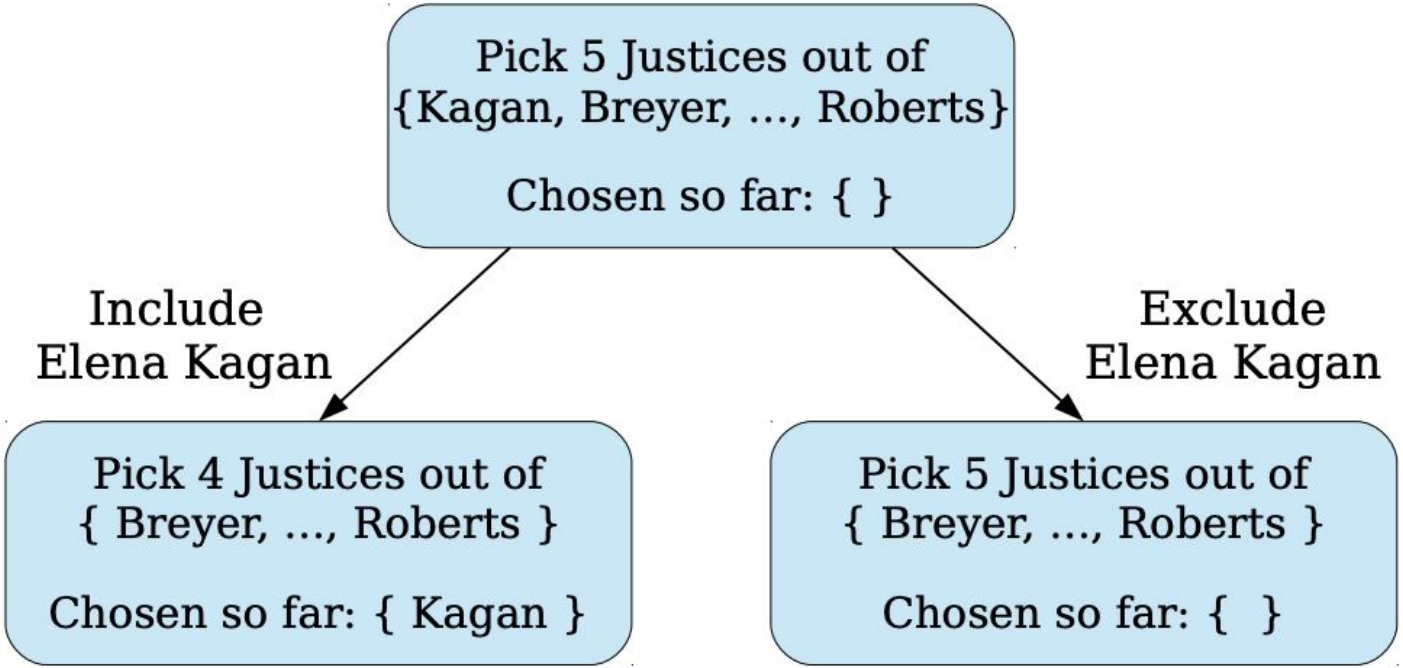
- Suppose we get to the following scenario:

Pick 0 more Justices out of:
`{Sotomayor, Thomas}`
Chosen so far:
`{}`

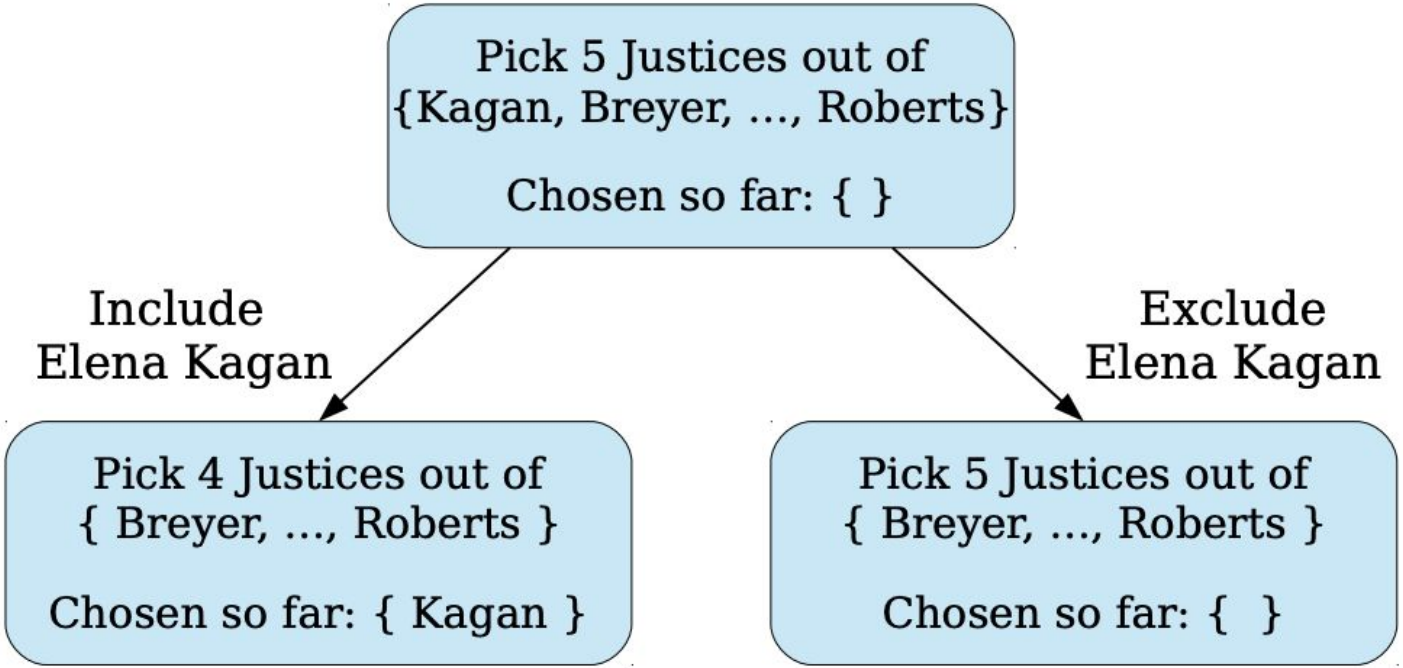
- There's no need to keep looking! **We can return an empty set.**

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

Pseudocode

```
Set<Set<string>> combinationsRec(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.
  - Return the the combined returned sets from both inclusion and exclusion.

Pseudocode

```
Set<Set<string>> combinationsRec(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.
  - **Return the the combined returned sets from both inclusion and exclusion.**

This is different from our usual recursion pattern!

Pseudocode

```
Set<Set<string>> combinationsRec(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.
  - Return the the combined returned sets from both inclusion and exclusion.

- **Base cases:**
  - No more remaining elements to choose from ➜ return empty set
  - No more space in chosen (k is maxed out) ➜ return set with chosen

Let’s see the code!

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

- A3 was released last Thursday and is due on Thursday, July 15 at 11:59pm.
  - Please note that using the grace period for A3 will push you into the mid-quarter diagnostic.
- Section leaders are currently working on grading and providing feedback on A2 submissions – feedback will be released by Wednesday night.
- The mid-quarter diagnostic is coming up at the end of this week.
  - You will have a 72-hour period of time from Friday to Sunday to complete the diagnostic.
  - The diagnostic is designed to take about an hour and a half to complete, but you can have up to 3 hours to work on it if you so choose.
  - The diagnostic will be administered via Gradescope.
  - A practice diagnostic and review materials have been posted on the diagnostic page.

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 a CS theory course 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 a CS theory course 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 a CS theory course 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

- 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

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

Your backpack holds up to 5 lbs max

What do you bring on your wilderness survival journey?

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 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…nest 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…nest value that will still fit in your backpack?

Why doesn't this work?

Rope
- Value: 3
- Weight: 2

Axe
- Value: 4
- Weight: 3

Tent
- Value: 5
- Weight: 4

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

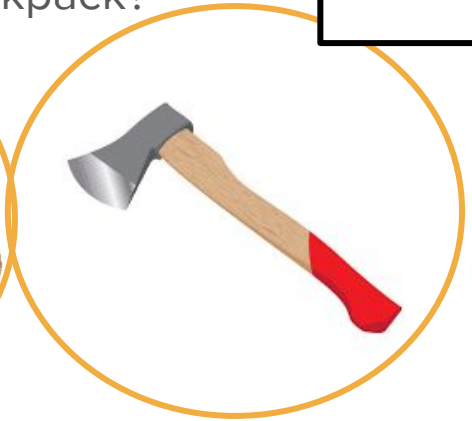
The "Greedy" Approac

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!

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.

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!

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, int index);
```

For efficiency, we'll use an **`index`** to keep track of which elements we've already looked at inside of **`items`** (like the unbiased jury problem).

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

Let’s see the code!

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

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.

Summary

Backtracking recursion:
Exploring many possible solutions

Overall paradigm: choose/explore/unchoose

Two ways of doing it

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

Three use cases for backtracking

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

General examples of things you can do:

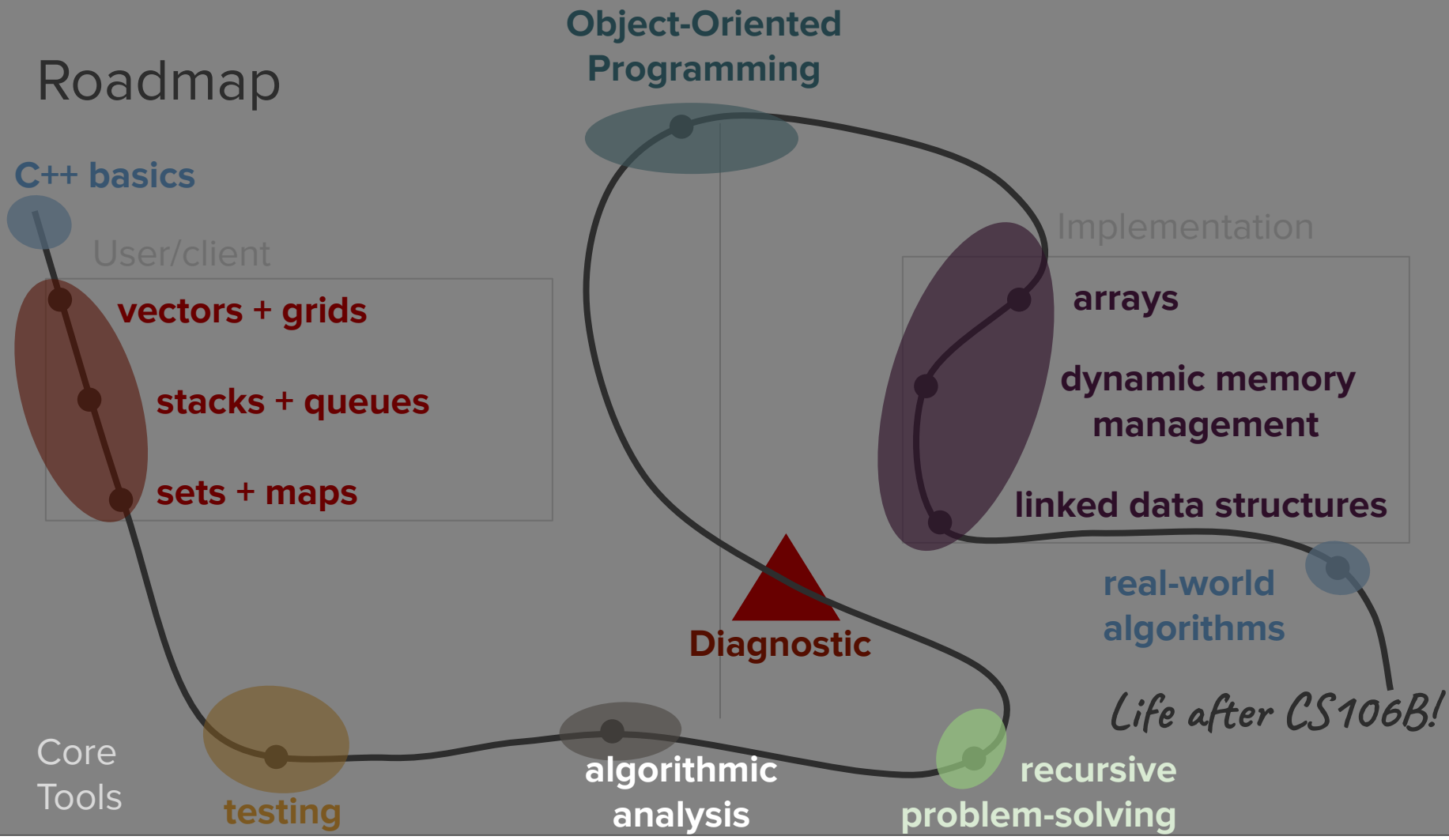
- Permutations
- Subsets
- Combinations
- etc.

Questions to ask
when planning your backtracking strategy

- What does the decision tree look like? (decisions, options, what to keep track of)
- What are the base and recursive cases?
- What's the provided function prototype and requirements? Is a helper function needed?
- Do you care about returning or keeping track of the path you took to get your solution?
- Which of the 3 use cases does the problem fall into? (generate/count all solutions, find one solution/prove its existence, or pick one best solution)
- What are you returning as your solution? (a boolean, a final value, a set of results, etc.)
- What are you building up as your "many possibilities" in order to find your solution? (subsets, permutations, combinations, or something else)

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

Beyond Efficiency:

Algorithmic Analysis and Ethics