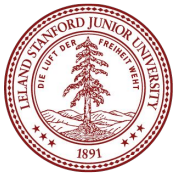
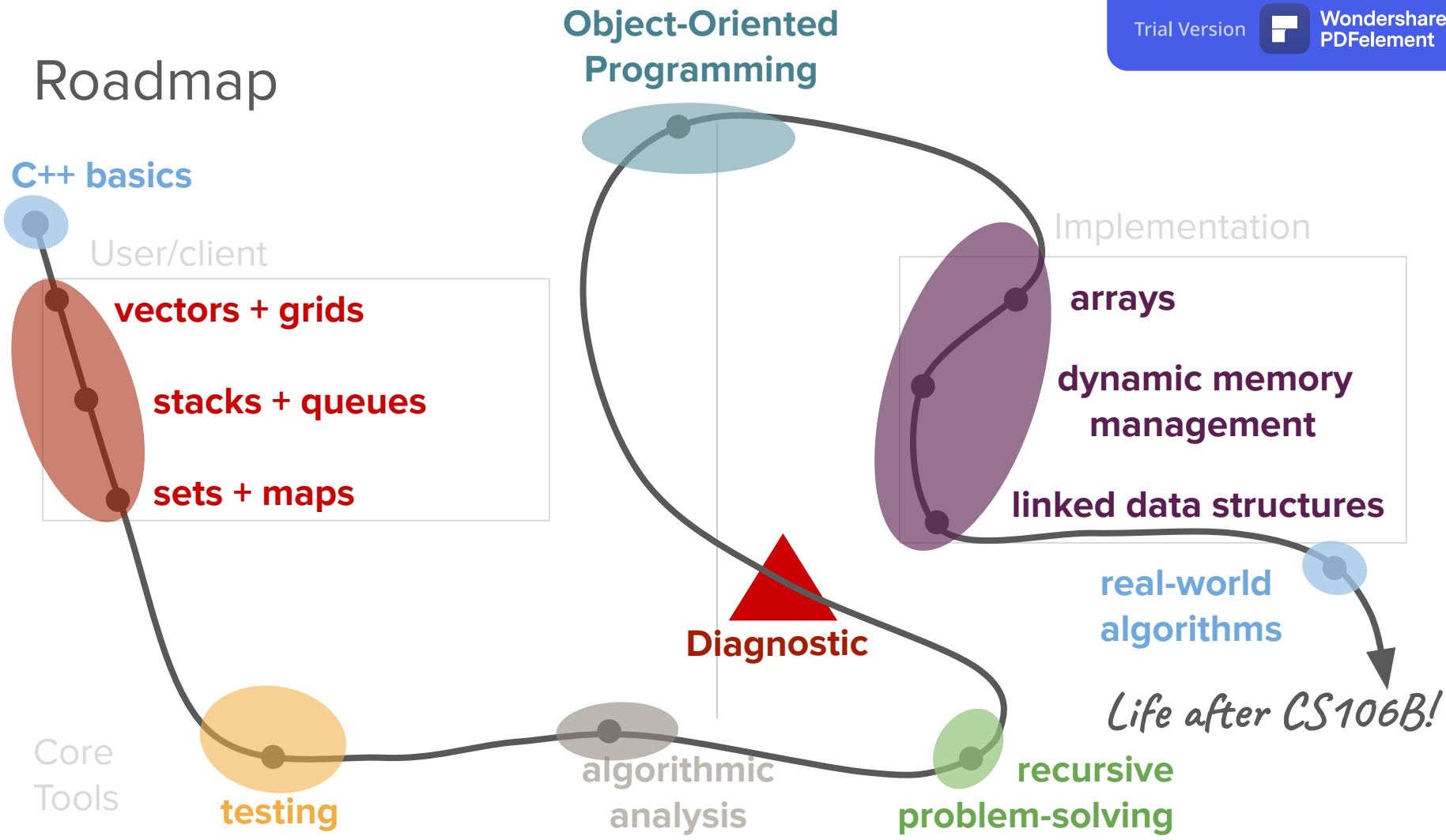
Recursive Backtracking Revisited

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

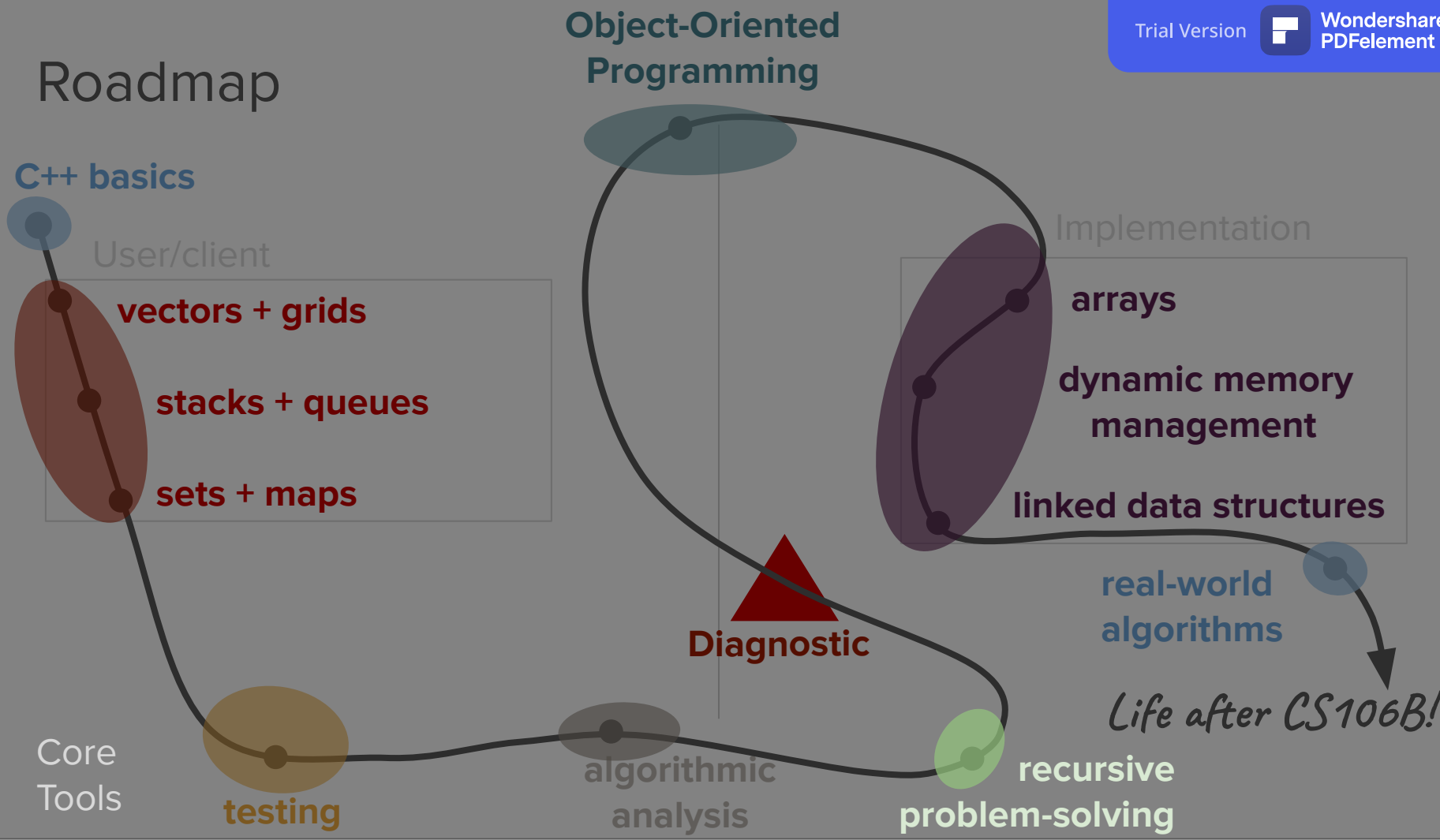
(put your answers the chat)



Roadmap



Roadmap



Today's question

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

Today's topics

1. Review
2. Implementing Subsets
3. Selecting Unbiased Juries
4. Solving Mazes with DFS

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

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

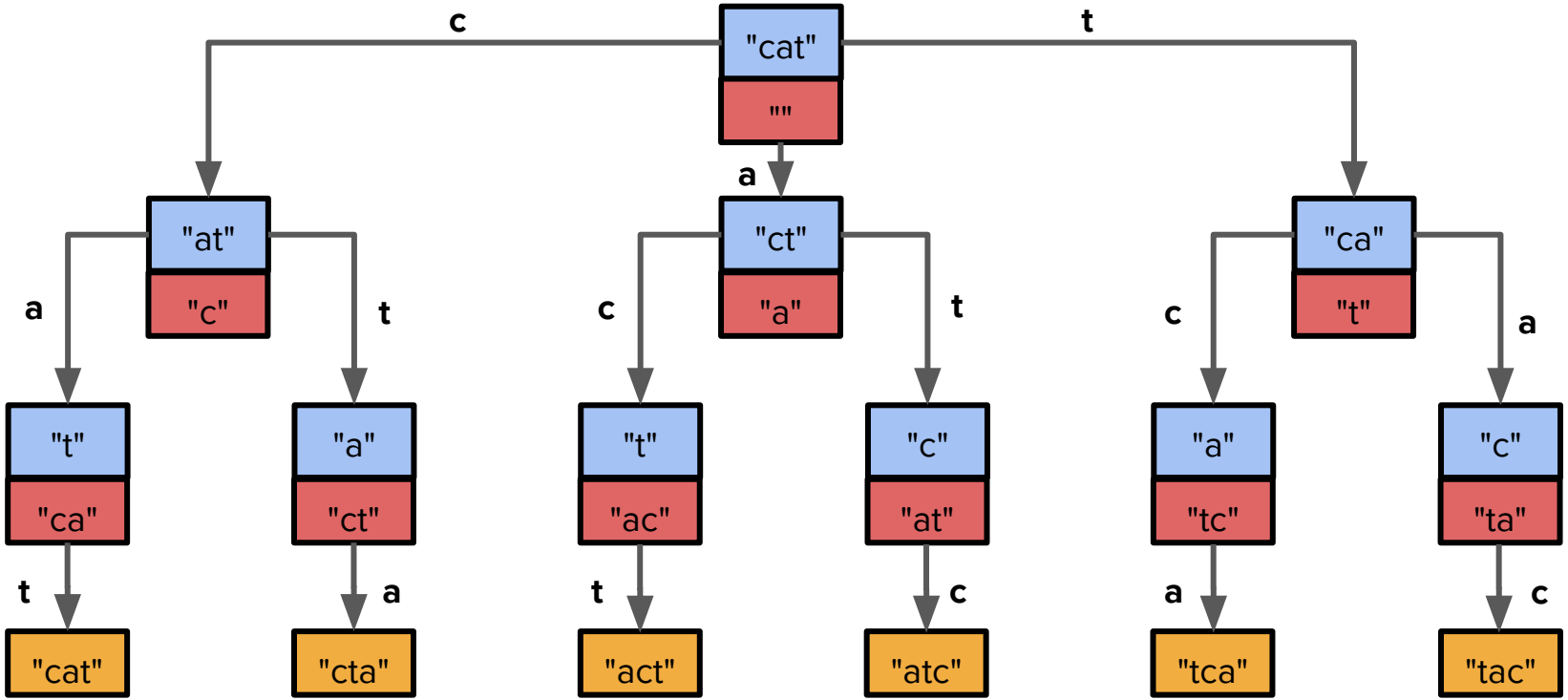
What defines our permutations decision tree?

- **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

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);
        }
    }
}
```

Decisions yet to be made

Decisions already made

Base case: No decisions remain

Recursive case: Try all options for next decision

Takeaways

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

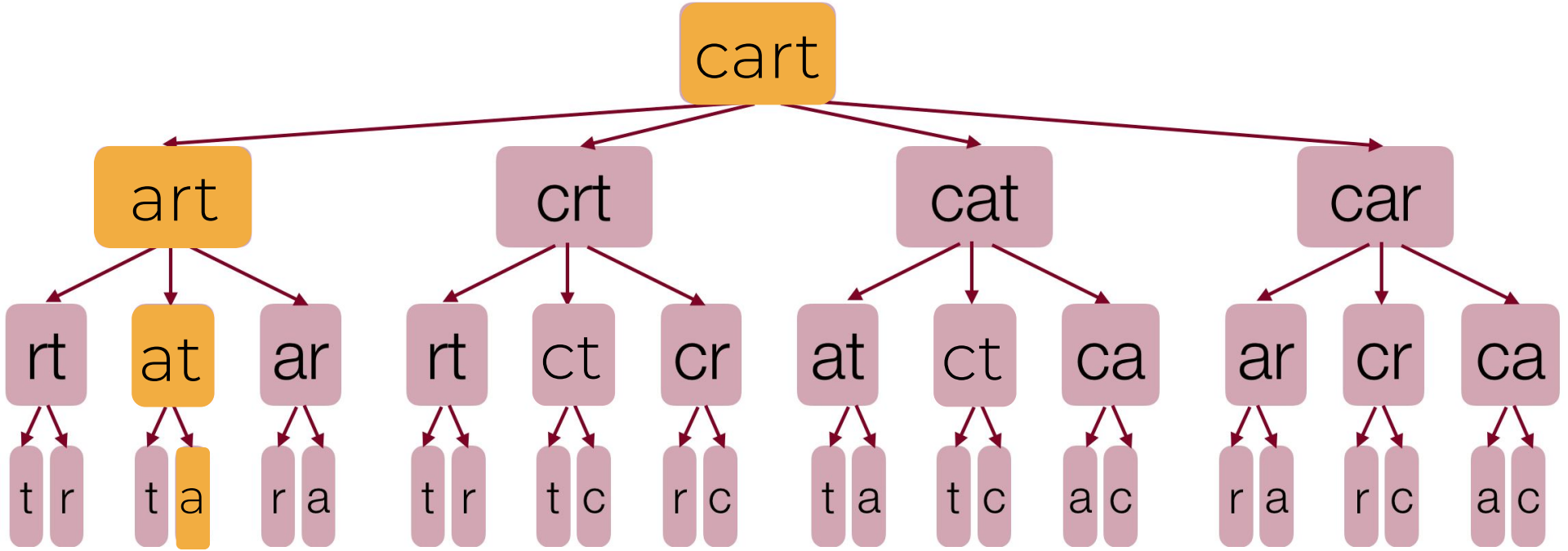
- Use of helper functions and initial empty params that get built up is common

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?



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;
```

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:

In general, a "subset" is any subcollection of elements from an initial collection of options.

Subsets

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

{}

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

Subsets

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

{}

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

Subsets

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

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

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

Subsets

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

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

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

Discuss in breakouts:

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

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

[Time-permitting] Can you come up with a base case and recursive case for generating 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”}
```

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

Subsets

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

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

Half the subsets contain “Nick”

Subsets

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

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

Half the subsets contain “Kylie”

Subsets

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

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

Half the subsets contain “Trip”

Subsets

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

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

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

Subsets

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

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

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

Subsets

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

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

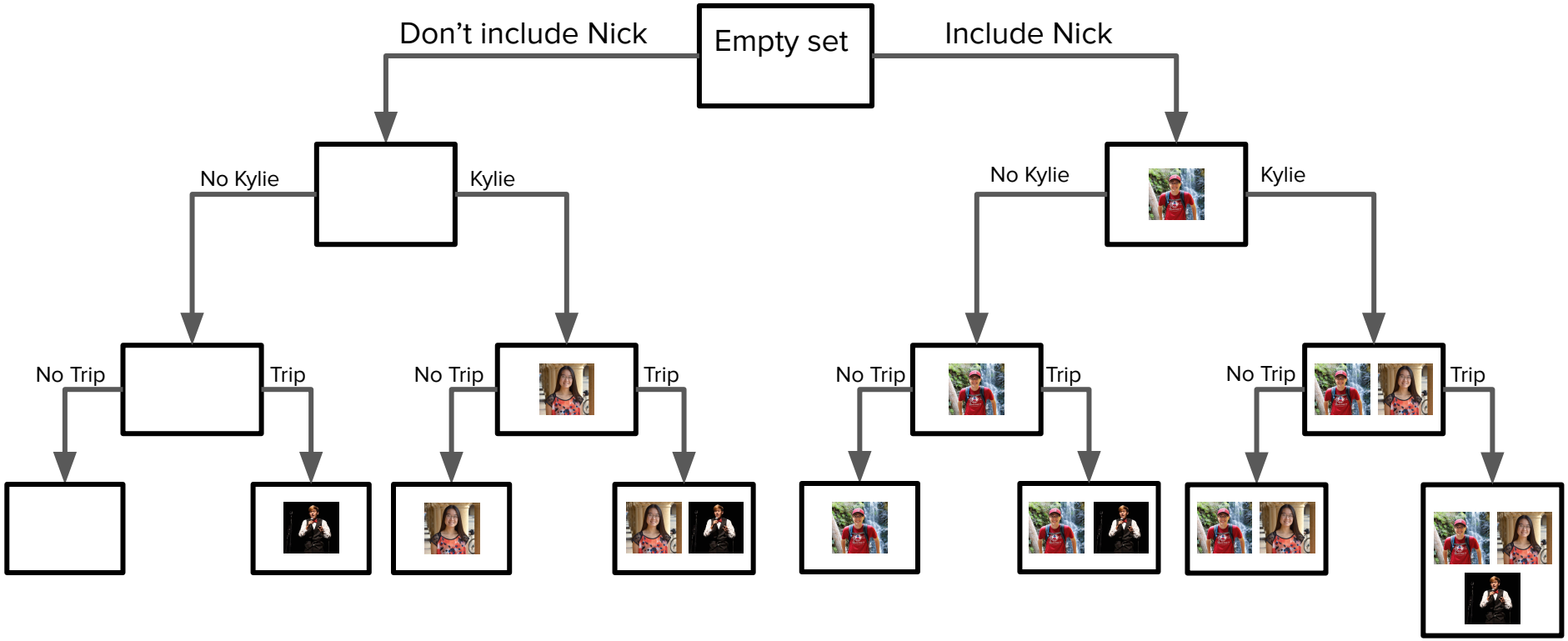
What defines our subsets decision tree?

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

- **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



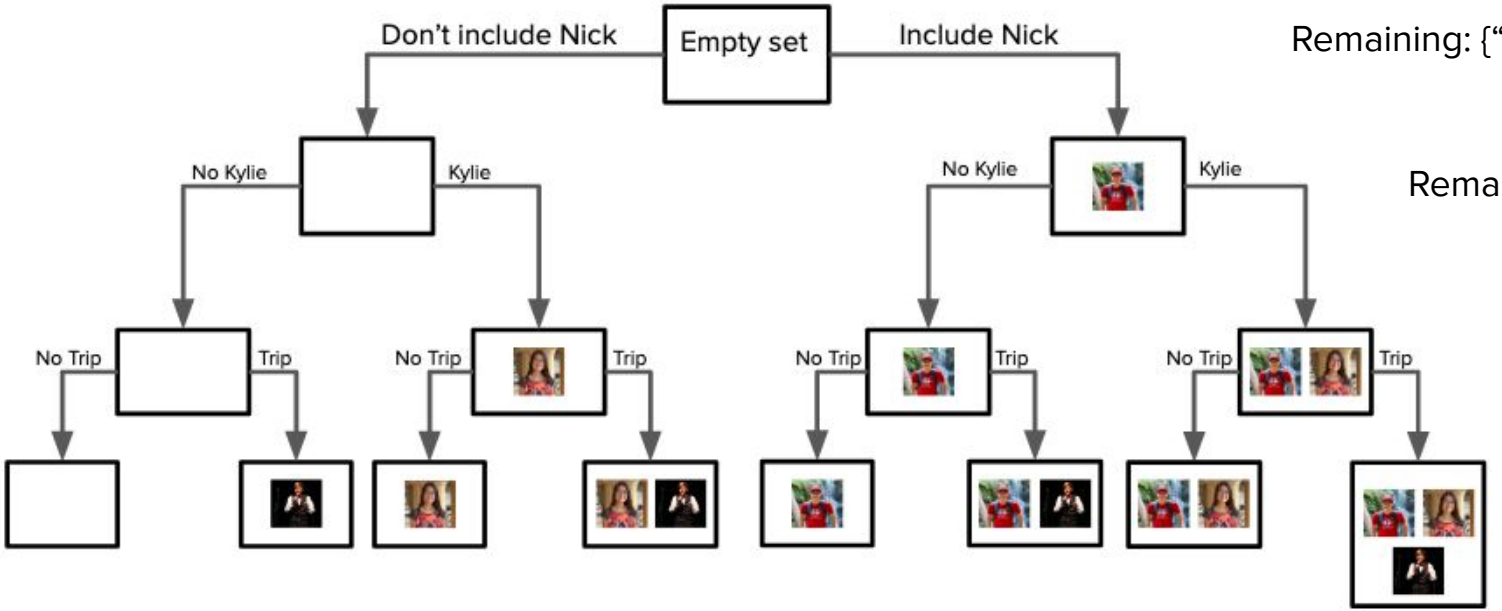
What defines our subsets decision tree?

- **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

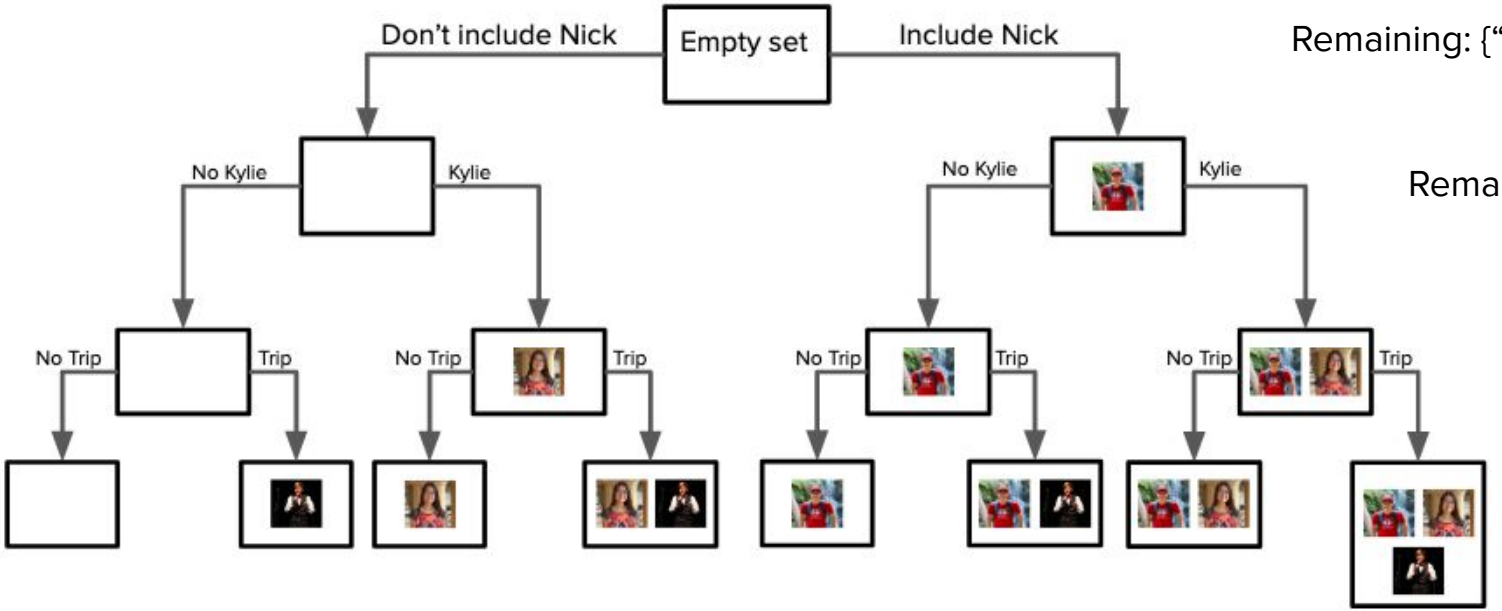
- Information we need to store along the way:
  - The set you’ve built so far
  - **The remaining elements in the original set**

Decision tree



Base case: No people remaining to choose from!

Decision tree



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

Let’s code it!

Takeaways

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

Choose

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

Takeaways

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

Explore (part 1)

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

Takeaways

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

Explore (part 2)

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

Takeaways

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

Explicit Unchoose (i.e. undo)

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

Takeaways

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

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

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

Takeaways

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

- 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

Application: Choosing an Unbiased Jury 陪审团

Jury Selection

- The process of jury selection involves processing a large pool of candidates that have been called for jury duty, and selecting some small subset of those candidates to serve on the jury.

- When selecting members of a jury, each individual person will come in with their own biases that might sway the case.

- Ideally, we would like to select a jury that is **unbiased (sum of all biases is 0)**

- **An unbiased jury is just a subset with a special property** – let's apply the code that we just wrote!

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

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

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

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

        allCandidates.insert(0, currentCandidate);
    }
}

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

Helper function: Extra variable to keep track of total bias

Jury Selection Code

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

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

        allCandidates.insert(0, currentCandidate);
    }
}

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

Base case: Only display juries with no total bias

Jury Selection Code

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

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

        allCandidates.insert(0, currentCandidate);
    }
}

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

Recursive case: Consider juries both with and without this person

Jury Selection Optimization

Jury Selection Code

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

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

        allCandidates.insert(0, currentCandidate);
    }
}

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

Vector addition/removal can be an expensive operation. Can we do better?

Optimizing Subset Creation

- The core component of subset generation includes visiting each element once, and making a decision about whether to include it or not

- Previously, we have done so by arbitrarily picking the "first" element in the collection as the one under consideration, and then removed it (expensive) from the collection for future recursive calls.

- **Key Idea:** Instead of modifying the collection of elements, let's just keep track of our current place in the collection (**index of the element that is currently under consideration**).

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);
}
```

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!

Takeaways

- 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
    - Hint: This will be important for those of you that attempt the backtracking challenge problem on Assignment 3!

Announcements

Announcements

- Assignment 3 was released last Thursday evening and is due on Thursday, July 16 at 11:59pm.

- Section leaders are currently working on grading and providing feedback on Assignment 2 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 digitally using BlueBook.
  - A practice diagnostic and review materials have been posted on the diagnostic page.

Revisiting mazes

Solving mazes with breadth-first search (BFS)

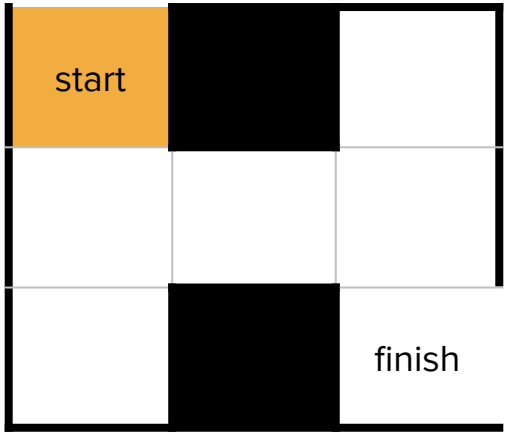
Solving mazes **recursively**

入口

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

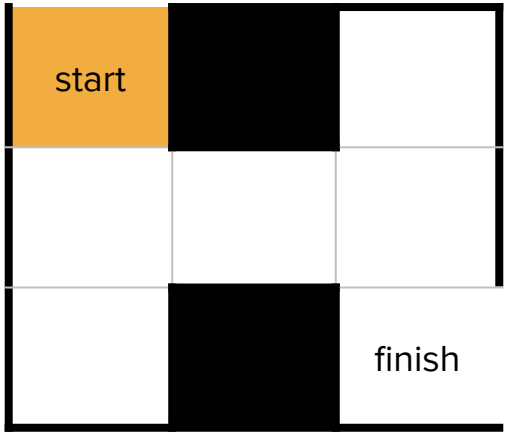
Solving mazes recursively

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



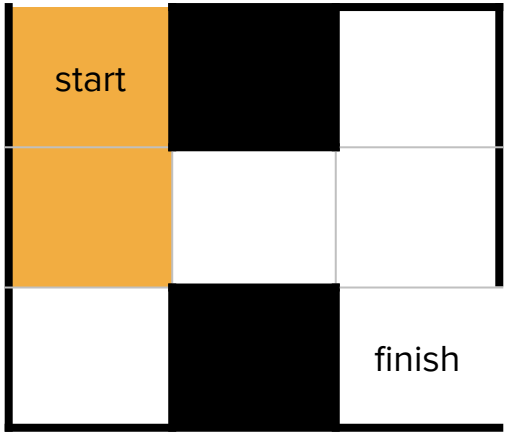
Solving mazes recursively

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



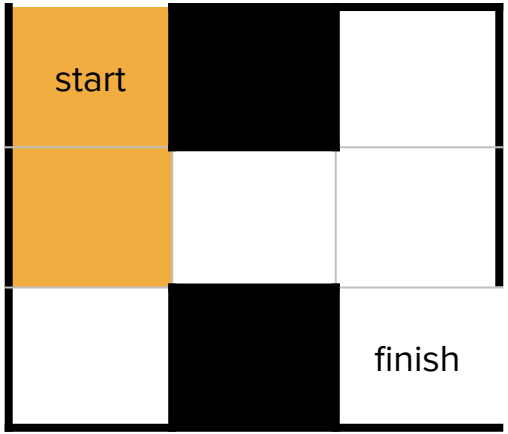
Solving mazes recursively

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



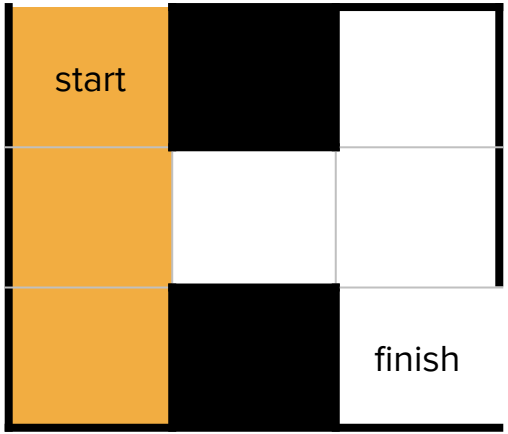
Solving mazes recursively

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



Solving mazes recursively

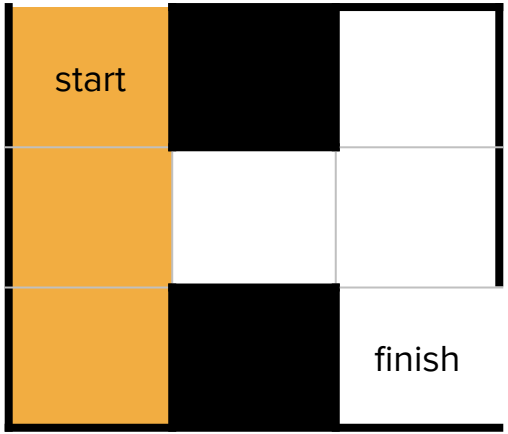
- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze



Solving mazes recursively

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

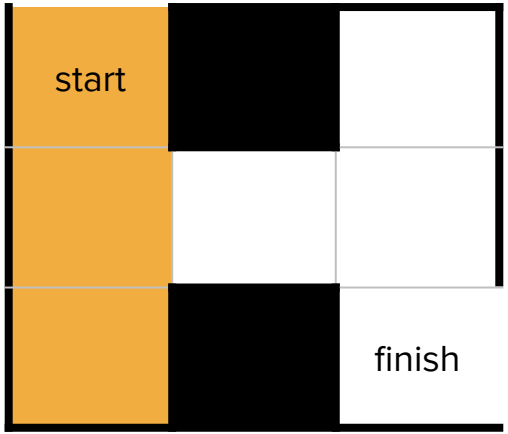
Dead end!
(cannot go North,
South, East, or West)



Solving mazes recursively

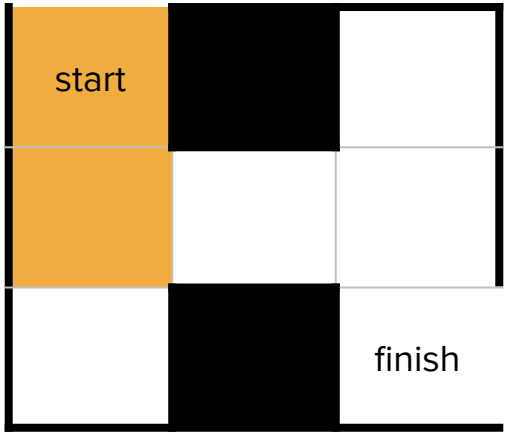
- Start at the entrance
- Take one step ~~North, South, East, or West~~

We must go back one step.



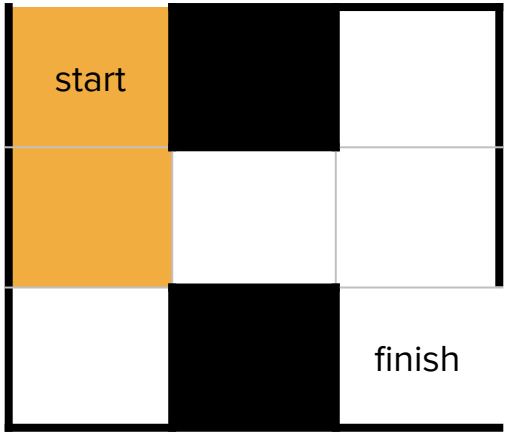
Solving mazes recursively

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



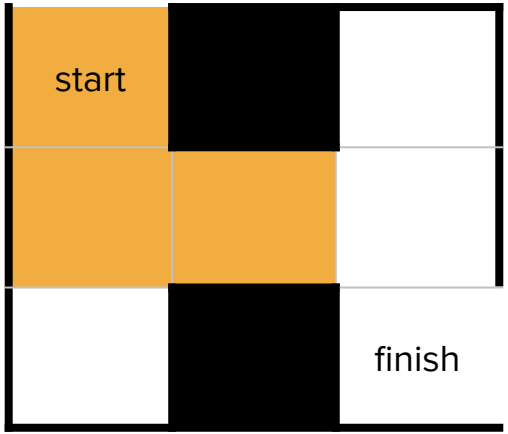
Solving mazes recursively

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



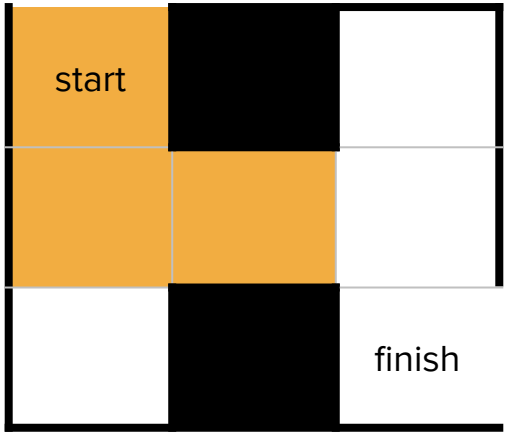
Solving mazes recursively

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



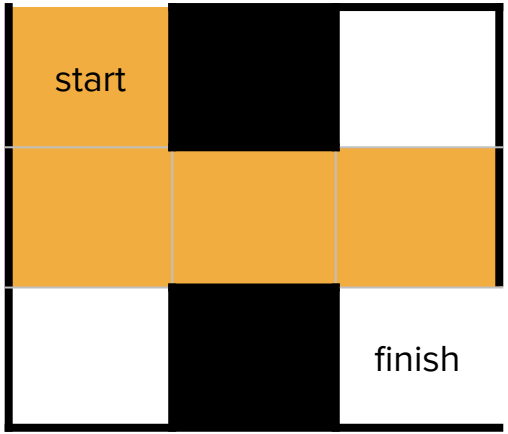
Solving mazes recursively

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



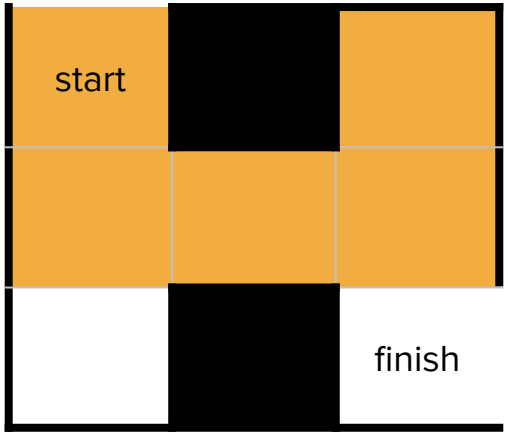
Solving mazes recursively

- 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

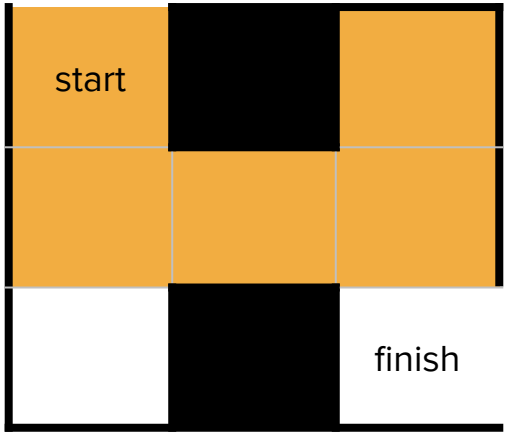
- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze



Solving mazes recursively

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

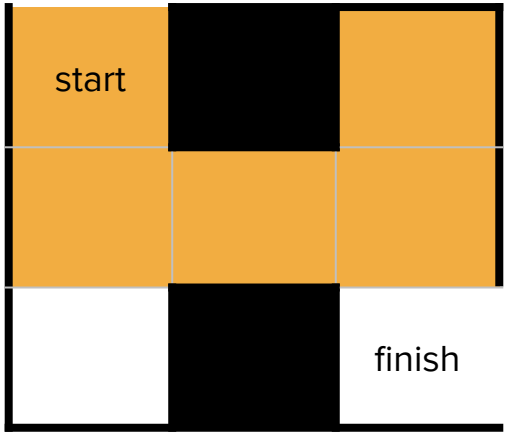
Dead end!
(cannot go North,
South, East, or West)



Solving mazes recursively

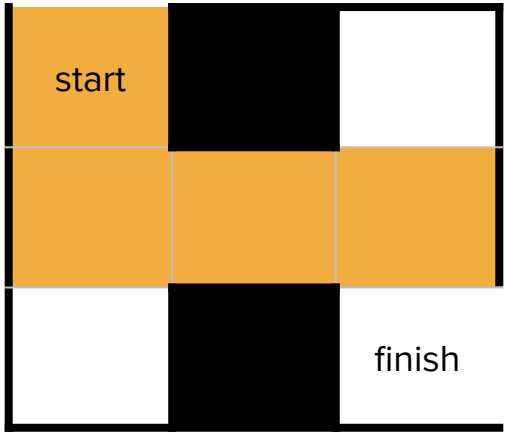
- Start at the entrance
- Take one step ~~North, South, East, or West~~

We must go back one step.



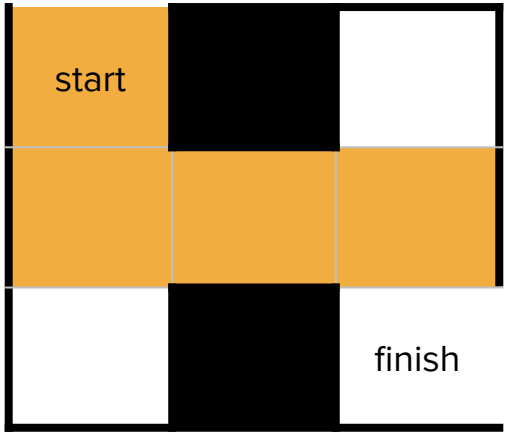
Solving mazes recursively

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



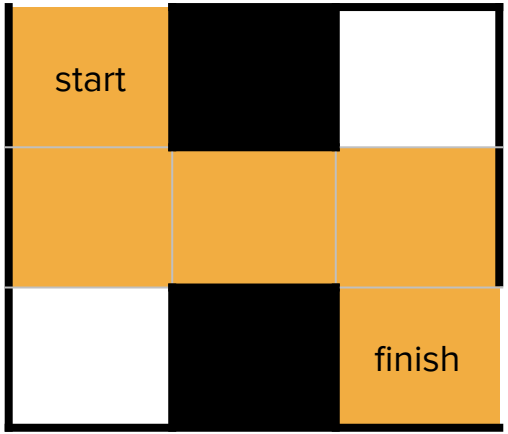
Solving mazes recursively

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



Solving mazes recursively

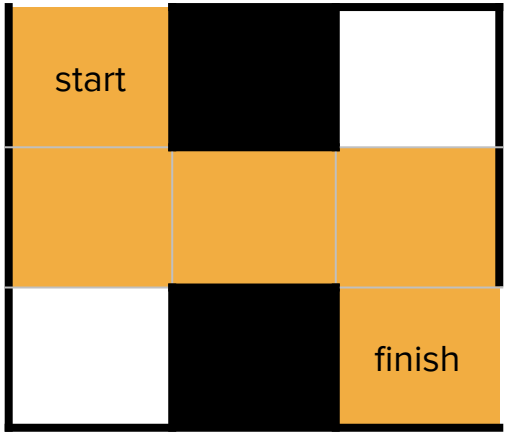
- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze



Solving mazes recursively

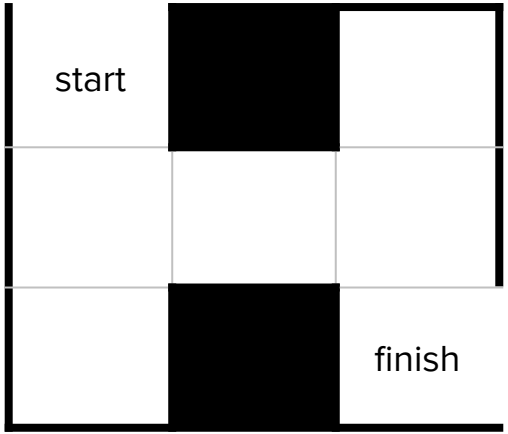
- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze

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?

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

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.
  - **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

翻转

- **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

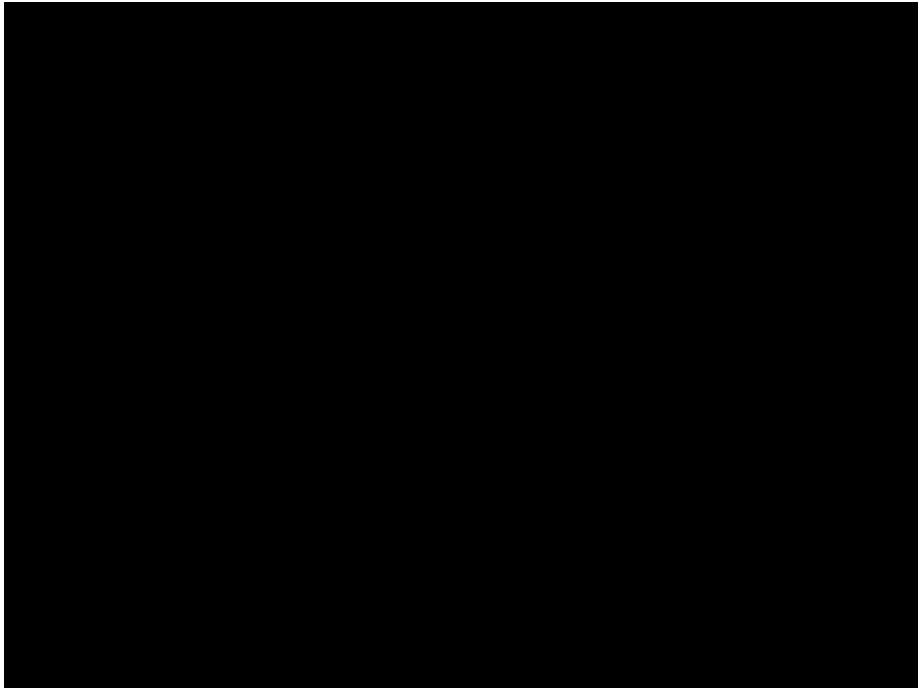
- **Base case**: We can stop exploring when we've reached the exit ➔ return true if the current location is the exit

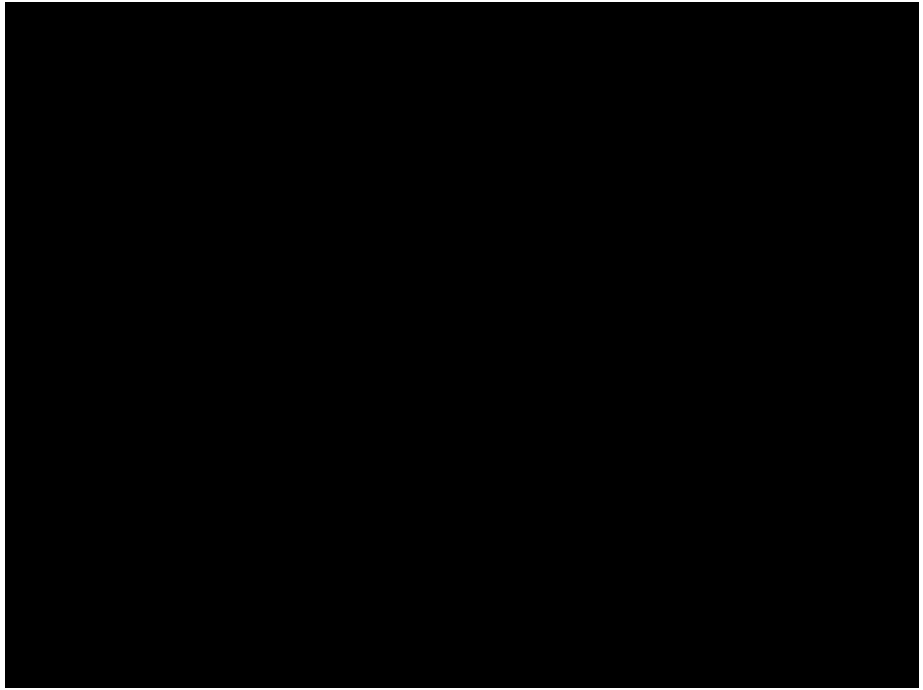
Let’s code it!

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.

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.

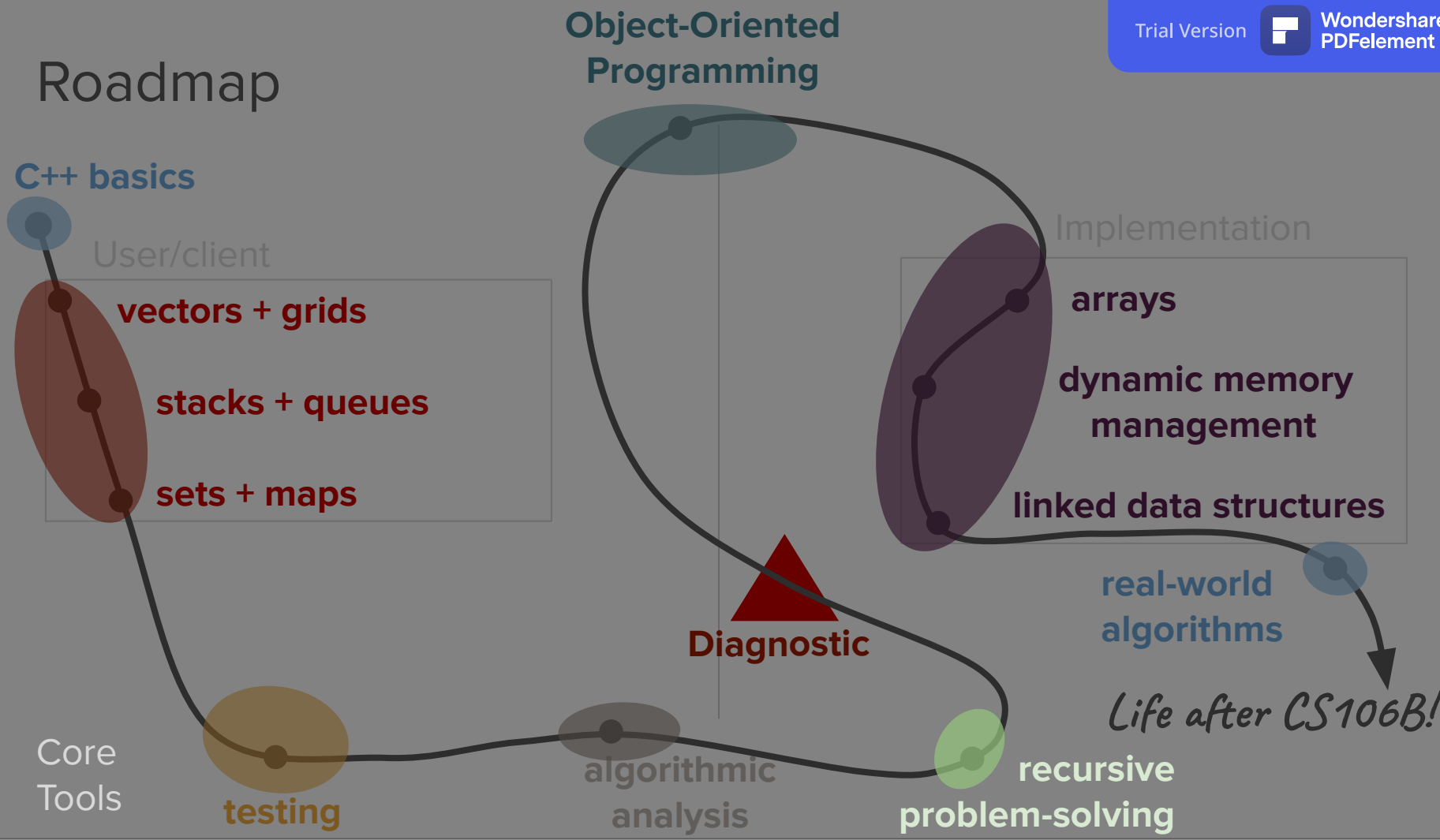
We've seen lots of different backtracking strategies...

Questions to ask yourself when planning your strategy:

- What does my decision tree look like? (decisions, options, what to keep track of)
- What are our base and recursive cases?
- What's the provided function prototype and requirements? Do we need a helper function?
- Do we care about returning or keeping track of the path we took to get to our solution?
- 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 returning as our solution? (a boolean, a final value, a set of results, etc.)
- What are we building up as our "many possibilities" in order to find our solution? (subsets, permutations, or something else)

What’s next?

Roadmap



Recursive optimization: Find the best solution