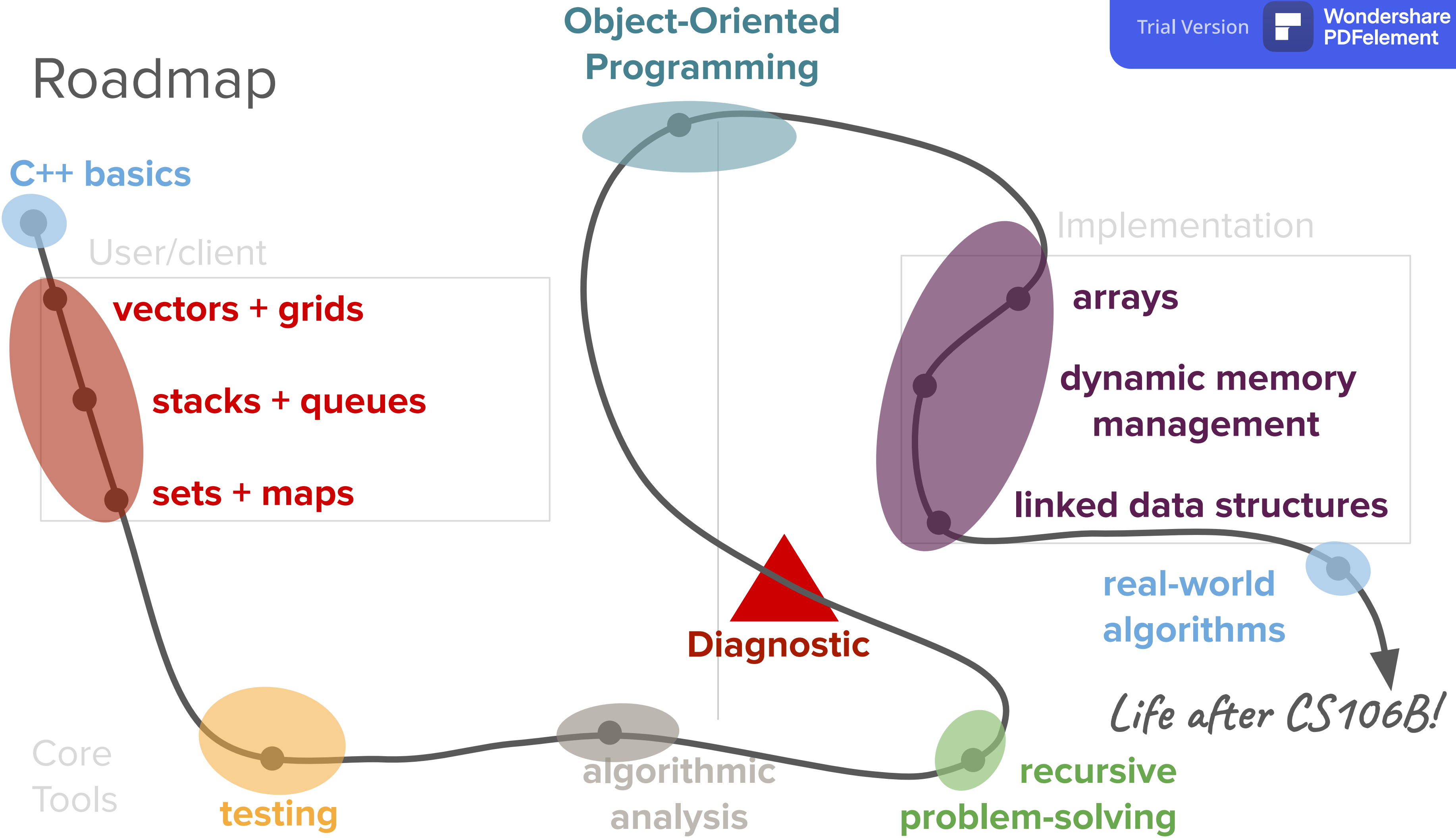
Why We Use Recursion

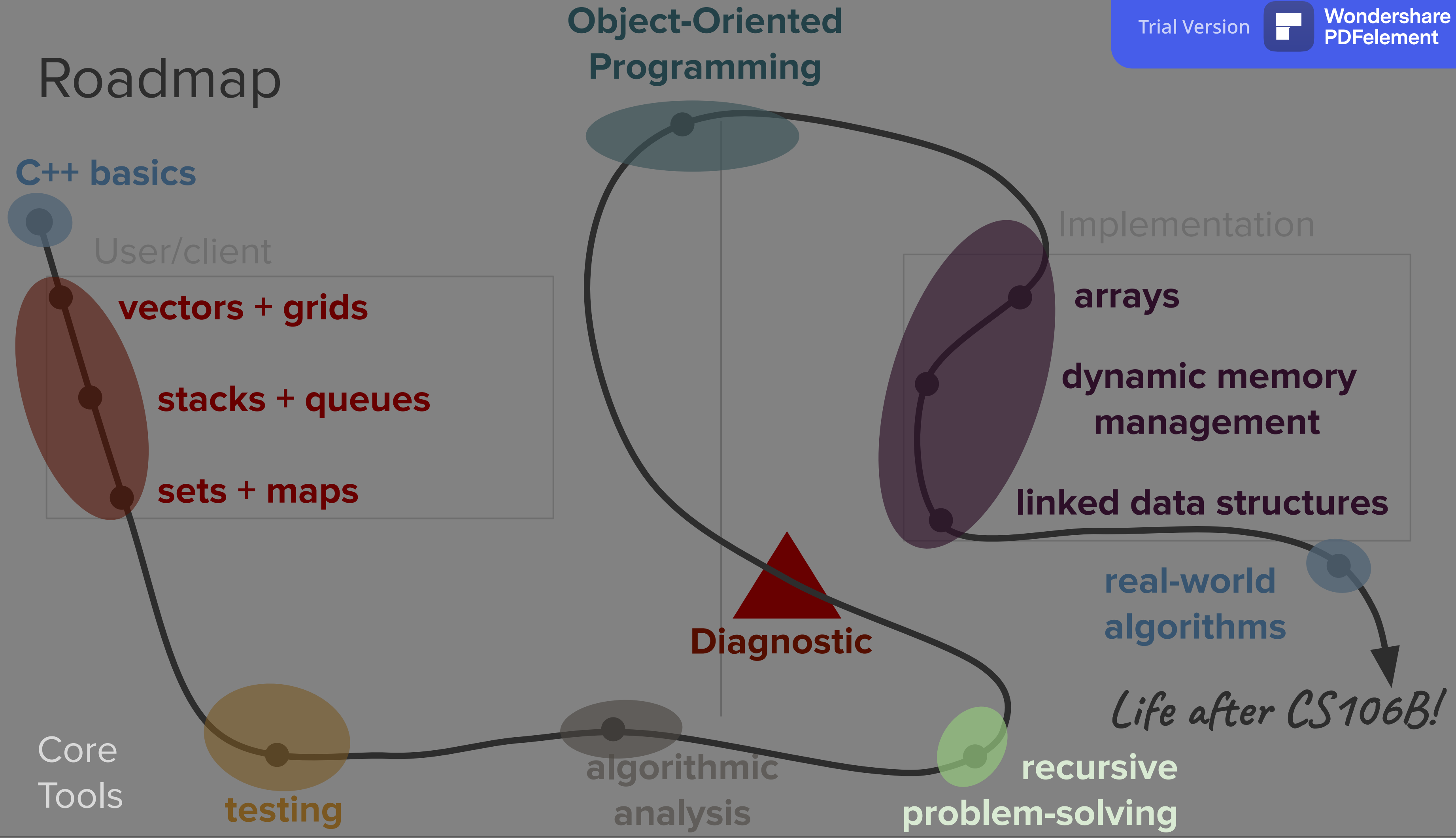
Which do you prefer: iteration or recursion? Include a short phrase explaining why.

(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

Why is recursion such a powerful problem-solving tool?

Today's topics

1. Review
2. Elegance
3. Efficiency
   (the return of Big O)
4. Recursive Backtracking

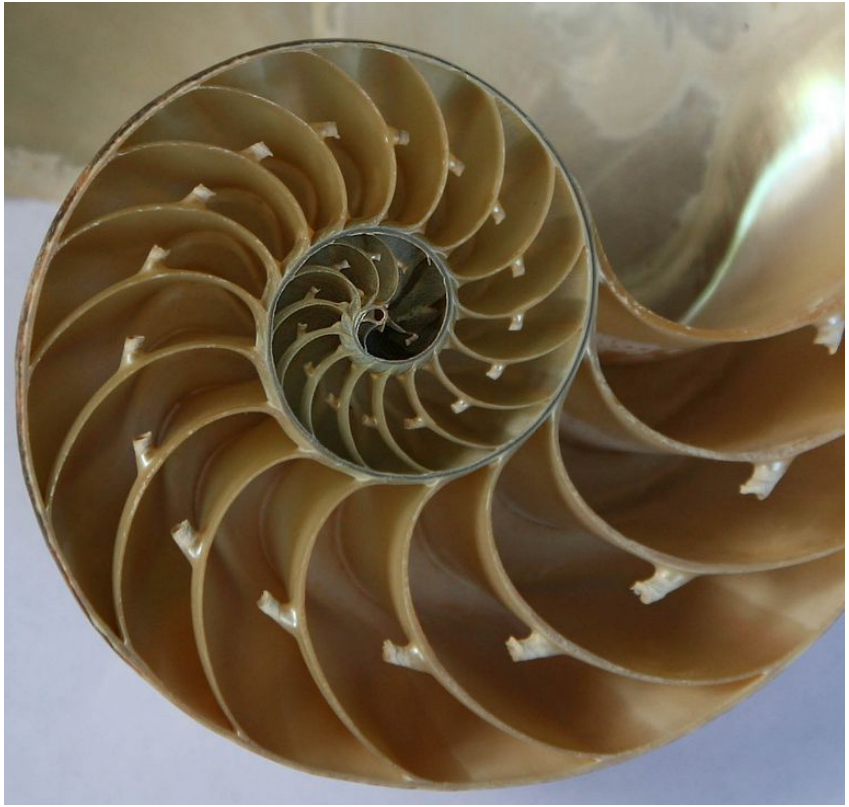
Review

(fractals)

Self-Similarity

- Solving problems recursively and analyzing recursive phenomena involves identifying **self-similarity**

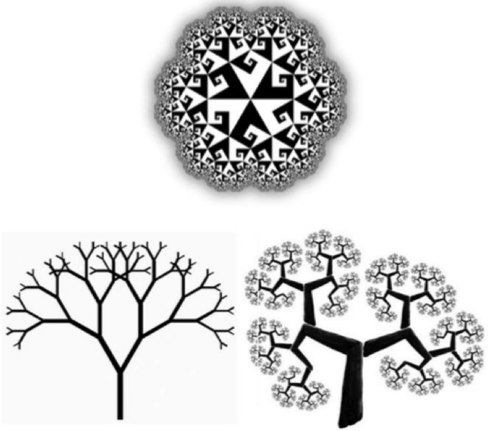
- An object is **self-similar** if it contains a smaller copy of itself.

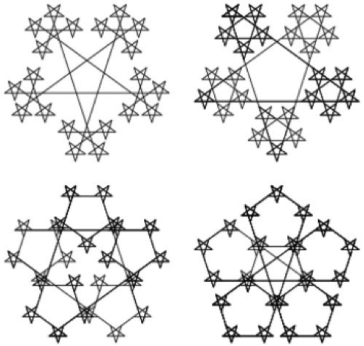


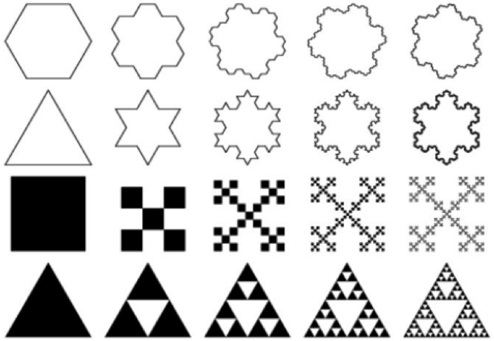
Fractals

- A **fractal** is any repeated, graphical pattern.

- A fractal is composed of **repeated instances of the same shape or pattern**, arranged in a structured way.



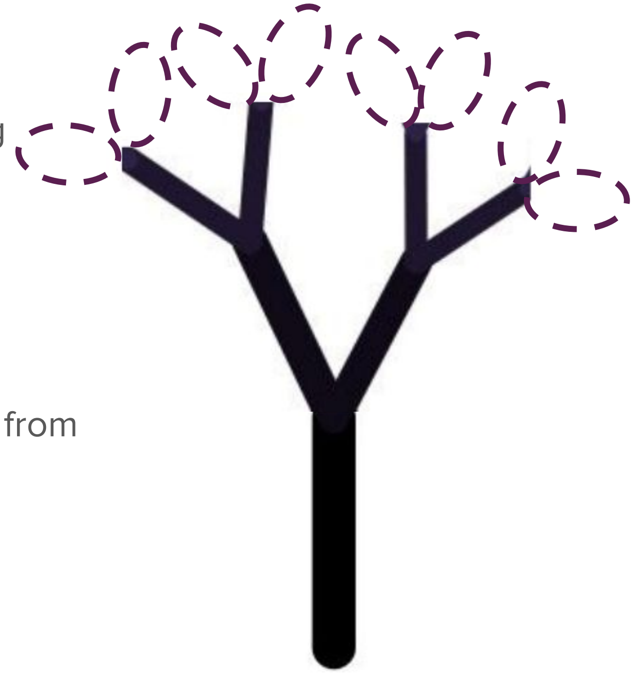




An order-3 tree

An order-0 tree is nothing at all.

An order-`n` tree is a line with two smaller order-`(n-1)` trees starting at the end of that line.



What differentiates the smaller tree from the bigger one?

1. It's at a different **position**.
2. It has a different **size**.
3. It has a different **orientation**.
4. It has a different **order**.

Fractals and self-similar structures are often defined in terms of some parameter called the **order**, which indicates the complexity of the overall structure.

Sierpinski Carpet

```
void drawSierpinskiCarpet(GWindow& window, double x, double y, double size, int order) {
   // Base case: A carpet of order 0 is a filled square.
   if (order == 0) {
       drawSquare(window, x, y, size);
   }  else {
       for (int row = 0; row < 3; row++) {
           for (int col = 0; col < 3; col++) {
               // The only square to skip is the very center one.
               if (row != 1 || col != 1) {
                   double newX = x + col * size / 3;
                   double newY = y + row * size / 3;
                   drawSierpinskiCarpet(window, newX, newY, size / 3, order - 1);
               }
           }
       }
   }
}
```

Iteration + Recursion

- It’s completely reasonable to mix iteration and recursion in the same function.
- Here, we’re firing off eight recursive calls, and the easiest way to do that is with a double for loop.
- Recursion doesn’t mean “the absence of iteration.” It just means “solving a problem by solving smaller copies of that same problem.”
- Iteration and recursion can be very powerful in combination!

Homework from yesterday

- Play Towers of Hanoi: https://www.mathsisfun.com/games/towerofhanoi.html

- Look for and write down patterns in how to solve the problem as you increase the number of disks. Try to get to at least 5 disks!

- **Extra challenge** (optional)**:** How would you define this problem recursively?
  - Don't worry about data structures here. Assume we have a function **`moveDisk(X, Y)`** that will handle moving a disk from the top of post **`X`** to the top of post **`Y`**.

Why do we use recursion?

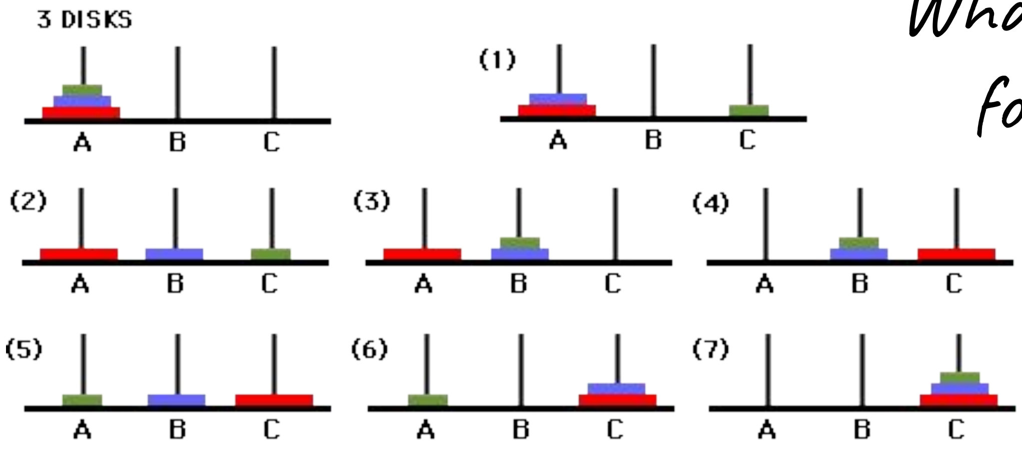
Why do we use recursion?

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

An **elegant** example: Towers of Hanoi

Pseudocode for 3 disks

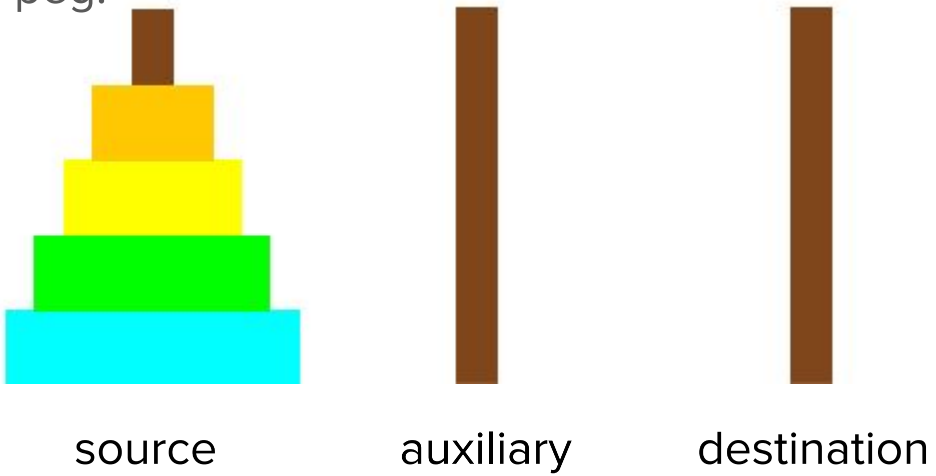
What if we add a fourth disk?



(1) Move disk 1 to destination
(2) Move disk 2 to auxiliary
(3) Move disk 1 to auxiliary
(4) Move disk 3 to destination
(5) Move disk 1 to source
(6) Move disk 2 to destination
(7) Move disk 1 to destination

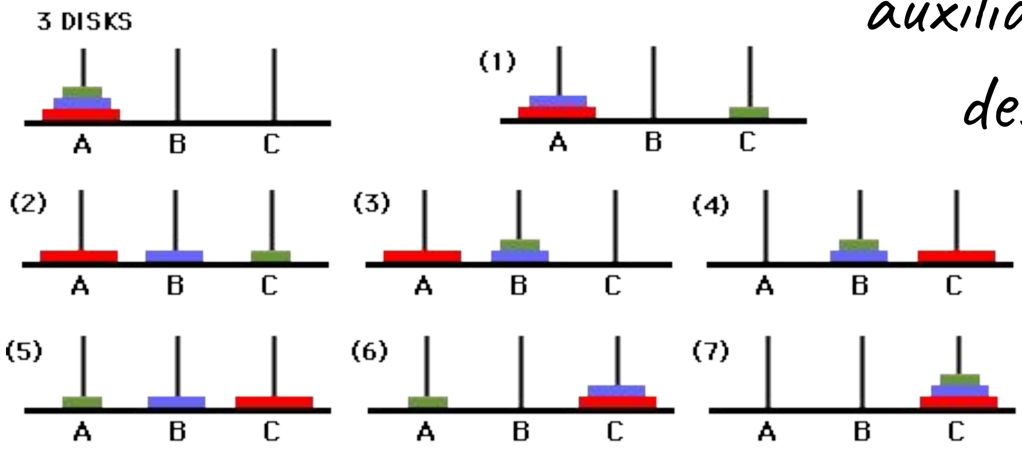
Towers of Hanoi with 4 disks

- We want to first move the biggest disk over to the destination peg.
  - We need to get the top three disks out of the way.
  - We already have an algorithm for moving three disks from a source peg to a destination peg!



Pseudocode for 3 disks

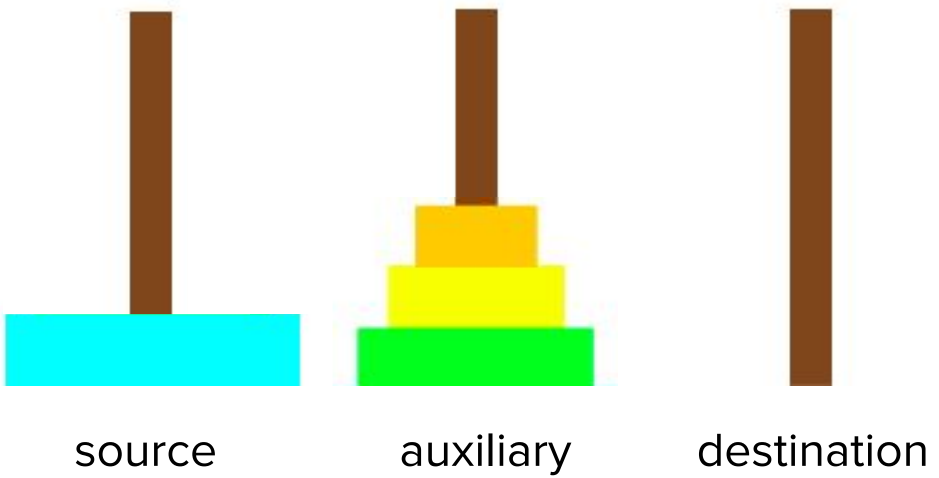
Idea: Move disks to auxiliary instead of destination!



(1) Move disk 1 to destination
(2) Move disk 2 to auxiliary
(3) Move disk 1 to auxiliary
(4) Move disk 3 to destination
(5) Move disk 1 to source
(6) Move disk 2 to destination
(7) Move disk 1 to destination

Towers of Hanoi with 4 disks

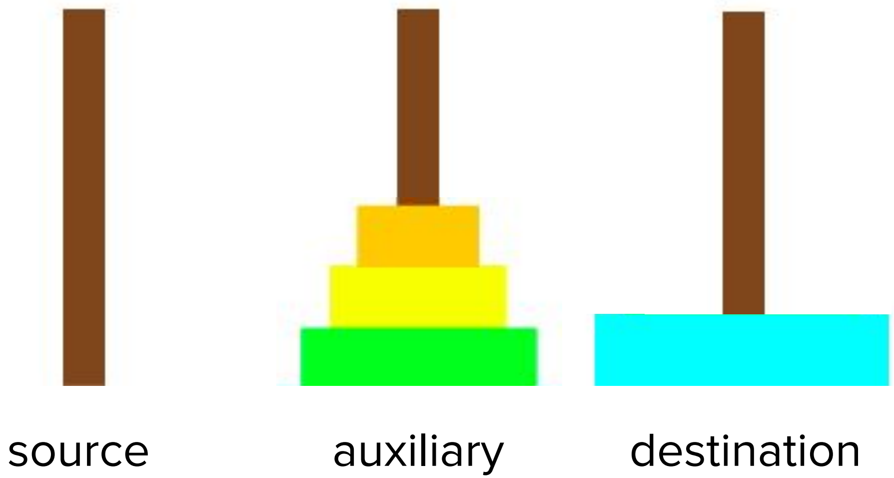
- We want to first move the biggest disk over to the destination peg.



Towers of Hanoi with 4 disks

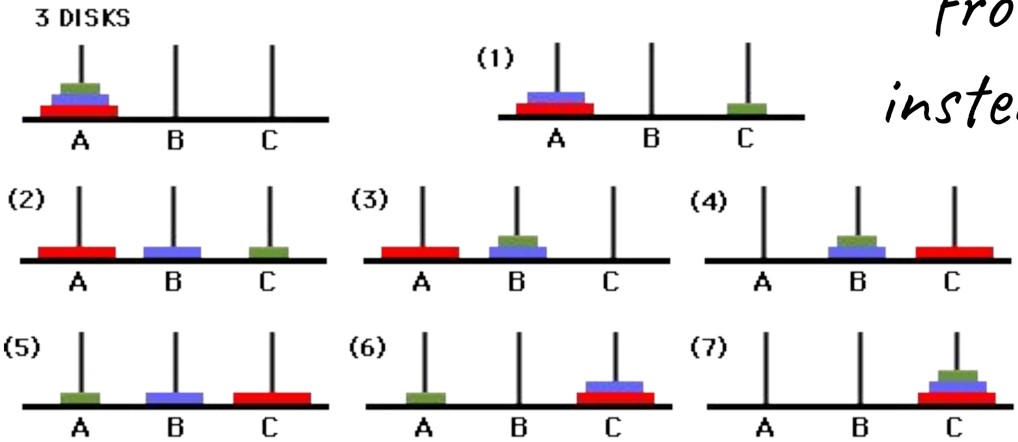
- We want to first move the biggest disk over to the destination peg.
- Now we need to move the stack of three from auxiliary to destination.

Use our existing 3-disk algorithm!



Pseudocode for 3 disks

Idea: Move disks from auxiliary instead of source!



(1) Move disk 1 to destination
(2) Move disk 2 to auxiliary
(3) Move disk 1 to auxiliary
(4) Move disk 3 to destination
(5) Move disk 1 to source
(6) Move disk 2 to destination
(7) Move disk 1 to destination

Discuss in breakouts:
How could we define the Towers of Hanoi solution recursively?

Towers of Hanoi solution

[live coding]

An **efficient** example: Binary Search

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Where is 89?

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Idea #1: We could just go through each element in order and do a linear search.

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Linear search is **O(n)**

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Can we do better? Can we take advantage of the structure of the data?

ADT Big-O Matrix

- Vectors
  - `.size() – O(1)`
  - `.add() – O(1)`
  - `v[i] – O(1)`
  - `.insert() – O(n)`
  - `.remove() – O(n)`
  - `.clear() - O(n)`
  - `traversal – O(n)`
- Grids
  - `.numRows()/.numCols() – O(1)`
  - `g[i][j] – O(1)`
  - `.inBounds() – O(1)`
  - traversal – $O(n^2)$
- Queues
  - `.size() – O(1)`
  - `.peek() – O(1)`
  - `.enqueue() – O(1)`
  - `.dequeue() – O(1)`
  - `.isEmpty() – O(1)`
  - `traversal – O(n)`
- Stacks
  - `.size() – O(1)`
  - `.peek() – O(1)`
  - `.push() – O(1)`
  - `.pop() – O(1)`
  - `.isEmpty() – O(1)`
  - `traversal – O(n)`
- Sets
  - `.size() – O(1)`
  - `.isEmpty() – O(1)`
  - `.add() – ???`
  - `.remove() – ???`
  - `.contains() – ???`
  - `traversal – O(n)`
- Maps
  - `.size() – O(1)`
  - `.isEmpty() – O(1)`
  - `m[key] – ???`
  - `.contains() – ???`
  - `traversal – O(n)`

Remember how their elements/keys always printed out in alphabetical order?

Note: Sets and Maps don't actually use a sorted list to store information, but the general idea of searching sorted data is similar.

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Where is 89?

Idea #2: Binary search

- Eliminate half of the data at each step.

- **Algorithm**: Check the middle element at **`(startIndex + endIndex) / 2`**
  - If the middle element is bigger than your desired value, eliminate the right half of the data and repeat.
  - If the middle element is smaller than your desired value, eliminate the left half of the data and repeat.
  - Otherwise, you've found your element!

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Where is 89?

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Start by looking at index:

```
(startIndex + endIndex) / 2
(0 + 9) / 2 –> 4
```

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Too small

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Eliminate left half

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

```
(startIndex + endIndex) / 2 =
(5 + 9) / 2 =
7
```

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

```
(startIndex + endIndex) / 2 =
(5 + 9) / 2 =
7
```

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Too small

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Eliminate left half

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

```
(startIndex + endIndex) / 2 =
(8 + 9) / 2 =
8
```

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

```
(startIndex + endIndex) / 2 =
(8 + 9) / 2 =
8
```

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Success!

Defining binary search recursively

- **Algorithm**: Check the middle element at `(startIndex + endIndex) / 2`
  - If the middle element is bigger than your desired value, eliminate the right half of the data and repeat.
  - If the middle element is smaller than your desired value, eliminate the left half of the data and repeat.
  - Otherwise, you've found your element!

- Recursive cases
  - Element at middle is too small ➔ binarySearch(right half of data)
  - Element at middle is too large ➔ binarySearch(left half of data)
- Base cases
  - Element at middle == desired element
  - Desired element is not in your data

Discuss in breakouts:

Read the code for **`binarySearch()`** and identify the base/recursive cases.

Binary search code

```
int binarySearch(Vector<int>& v, int targetVal, int startIndex, int endIndex) {
    if (startIndex > endIndex) {
        return -1;
    }

    int middleIndex = (startIndex + endIndex) / 2;
    int currentVal = v[middleIndex];
    if (targetVal == currentVal) {
        return middleIndex;
    } else if (targetVal < currentVal) {
        return binarySearch(v, targetVal, startIndex, middleIndex - 1);
    } else {
        return binarySearch(v, targetVal, middleIndex + 1, endIndex);
    }
}
```

Base cases

Binary search code

```
int binarySearch(Vector<int>& v, int targetVal, int startIndex, int endIndex) {
    if (startIndex > endIndex) {
        return -1;
    }

    int middleIndex = (startIndex + endIndex) / 2;
    int currentVal = v[middleIndex];
    if (targetVal == currentVal) {
        return middleIndex;
    } else if (targetVal < currentVal) {
        return binarySearch(v, targetVal, startIndex, middleIndex - 1);
    } else {
        return binarySearch(v, targetVal, middleIndex + 1, endIndex);
    }
}
```

Recursive cases

Binary search code

```
int binarySearch(Vector<int>& v, int targetVal, int startIndex, int endIndex) {
    if (startIndex > endIndex) {
        return -1;
    }

    int middleIndex = (startIndex + endIndex) / 2;
    int currentVal = v[middleIndex];
    if (targetVal == currentVal) {
        return middleIndex;
    } else if (targetVal < currentVal) {
        return binarySearch(v, targetVal, startIndex, middleIndex - 1);
    } else {
        return binarySearch(v, targetVal, middleIndex + 1, endIndex);
    }
}
```

We don't want the user to have to pass these in, but we need them to update our search range

Binary search code

```
int binarySearch(Vector<int>& v, int targetVal) {
    return binarySearchHelper(v, targetVal, 0, v.size() - 1);
}
```

```
int binarySearchHelper(Vector<int>& v, int targetVal, int startIndex, int endIndex) {
    ...
}
```

Use a **recursive helper function** for the extra parameters!

(`binarySearchHelper` would have the same code as the previous slide)

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

What's the runtime?

Binary search runtime

- For data of size **N**, it eliminates half until 1 element remains.

- Think of it from the other direction:
  - How many times do I have to multiply by 2 to reach **N**?

`1, 2, 4, 8, ..., N/4, N/2, N`

  - Call this number of multiplications **x**:

$$2^x = N$$
$$\mathbf{x = \log_2 N}$$

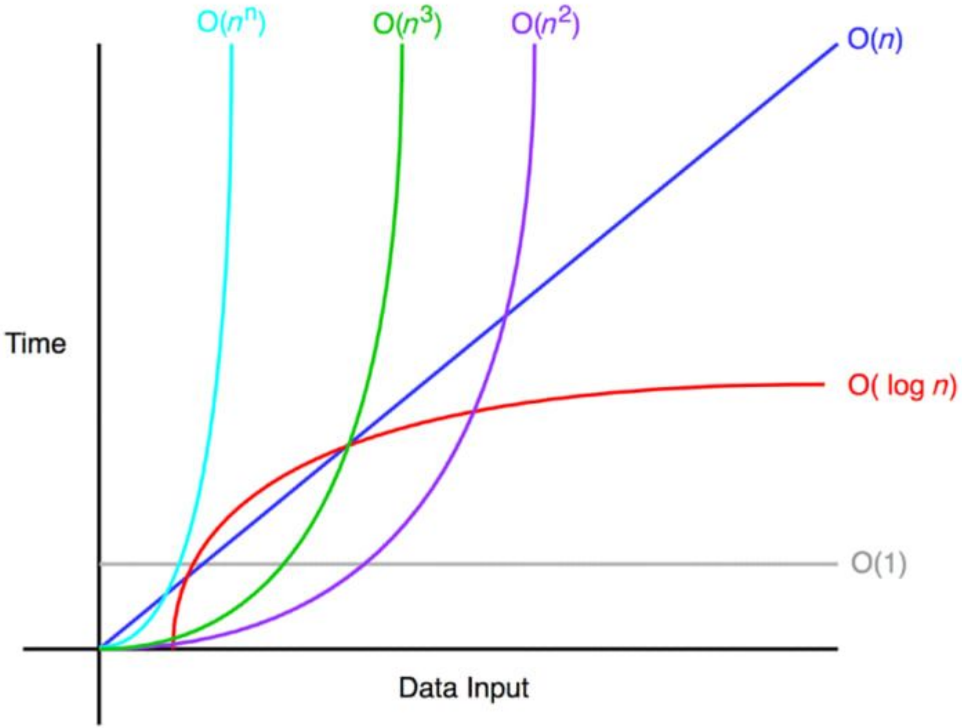
- Binary search has logarithmic Big-O: **`O(log N)`**

binarysearch.cpp

[demo]

Logarithmic runtime

- Better than linear
- A common runtime when you're able to "divide and conquer" in your algorithm, like with binary search



ADT Big-O Matrix

- Vectors
  - .size() - O(1)
  - .add() - O(1)
  - v[i] - O(1)
  - .insert() - O(n)
  - .remove() - O(n)
  - .clear() - O(n)
  - traversal - O(n)
- Grids
  - .numRows()/.numCols() - O(1)
  - g[i][j] - O(1)
  - .inBounds() - O(1)
  - traversal - O(n^2)
- Queues
  - .size() - O(1)
  - .peek() - O(1)
  - .enqueue() - O(1)
  - .dequeue() - O(1)
  - .isEmpty() - O(1)
  - traversal - O(n)
- Stacks
  - .size() - O(1)
  - .peek() - O(1)
  - .push() - O(1)
  - .pop() - O(1)
  - .isEmpty() - O(1)
  - traversal - O(n)
- Sets
  - .size() - O(1)
  - .isEmpty() - O(1)
  - .add() - ???
  - .remove() - ???
  - .contains() - ???
  - traversal - O(n)
- Maps
  - .size() - O(1)
  - .isEmpty() - O(1)
  - m[key] - ???
  - .contains() - ???
  - traversal - O(n)

ADT Big-O Matrix

- Vectors
  - `.size() - O(1)`
  - `.add() - O(1)`
  - `v[i] - O(1)`
  - `.insert() - O(n)`
  - `.remove() - O(n)`
  - `.clear() - O(n)`
  - `traversal - O(n)`
- Grids
  - `.numRows()/.numCols() - O(1)`
  - `g[i][j] - O(1)`
  - `.inBounds() - O(1)`
  - `traversal` - $O(n^2)$
- Queues
  - `.size() - O(1)`
  - `.peek() - O(1)`
  - `.enqueue() - O(1)`
  - `.dequeue() - O(1)`
  - `.isEmpty() - O(1)`
  - `traversal - O(n)`
- Stacks
  - `.size() - O(1)`
  - `.peek() - O(1)`
  - `.push() - O(1)`
  - `.pop() - O(1)`
  - `.isEmpty() - O(1)`
  - `traversal - O(n)`
- Sets
  - `.size() - O(1)`
  - `.isEmpty() - O(1)`
  - `.add() - O(log(n))`
  - `.remove() - O(log(n))`
  - `.contains() - O(log(n))`
  - `traversal - O(n)`
- Maps
  - `.size() - O(1)`
  - `.isEmpty() - O(1)`
  - `m[key] - O(log(n))`
  - `.contains() - O(log(n))`
  - `traversal - O(n)`

Announcements

Announcements

- Assignment 2 is due today at 11:59pm PDT.
- Assignment 3 will be released by the end of the day tomorrow.
- We will be releasing more concrete information about the diagnostic (including practice problems) over the weekend.

A **dynamic** example: Exploring many possibilities

Limits of iteration

- So far, we've seen problems that could be solved iteratively or recursively.
  - Depending on the problem, you could make the argument that one of the approaches was stylistically preferable or easier to understand.
  - But both got the job done!

Limits of iteration

- So far, we've seen problems that could be solved iteratively or recursively.

- However, there is a whole class of problems that are very difficult, or nearly impossible, to solve with an iterative approach.
    - These problems have the goal of exploring **many different possibilities or solutions.**
    - Because iteration is inherently linear (and not dynamic), it is usually used to build up a single solution without exploring many possible alternatives.
    - Recursion allows us to explore many potential possibilities at once via **the power of branching** that comes when we have multiple recursive calls.

Limits of iteration

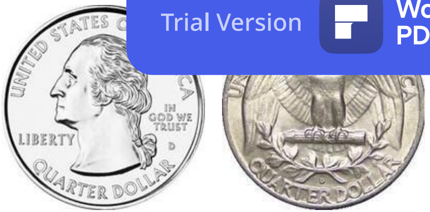
- So far, we've seen problems that could be solved iteratively or recursively.
- However, there is a whole class of problems that are very difficult, or nearly impossible, to solve with an iterative approach.
- To solve these problems and generate many possible solutions, we will have to learn a new problem-solving technique called **recursive backtracking**.
  - The key steps in recursive backtracking are that you make a choice about how to generate a solution, you use recursion to explore that choice, and then you might make a different choice and repeat the process.
  - This paradigm is called "**choose-explore-unchoose**."

Limits of iteration

- So far, we've seen problems that could be solved iteratively or recursively.
- However, there is a whole class of problems that are very difficult, or nearly impossible, to solve with an iterative approach.
- To solve these problems and generate many possible solutions, we will have to learn a new problem-solving technique called **recursive backtracking**.

Let's do an example!

Generating coin sequences

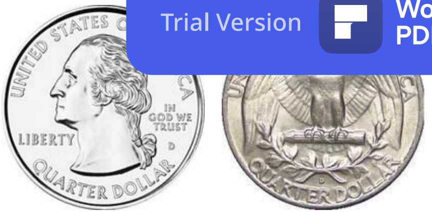


- Let's say that you're playing a game that involves flipping a coin a certain number of times in a row. Your success in the game depends on the exact sequence of "heads" and "tails" that you get.

- In the first version of this game, you get 2 coin flips on your turn. What are all the possible outcomes that you could get?

HH **HT** **TH** **TT**

Generating coin sequences



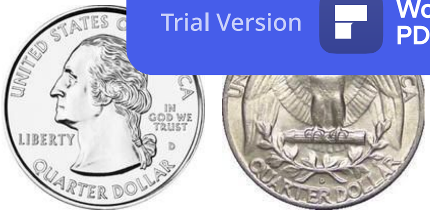
- Let's say that you're playing a game that involves flipping a coin a certain number of times in a row. Your success in the game depends on the exact sequence of "heads" and "tails" that you get.

- In a different version of the game, you instead get three flips of the coin on your turn. What are all the possible ways that your turn could go?

HHH HHT HTH HTT THH THT TTH TTT

How do we know that we got all the possibilities? How do we avoid repeats?

Generating coin sequences



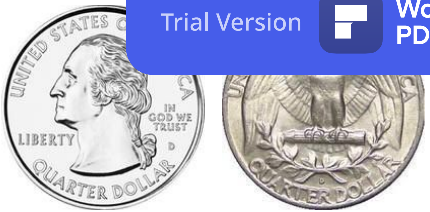
- Let's say that you're playing a game that involves flipping a coin a certain number of times in a row. Your success in the game depends on the exact sequence of "heads" and "tails" that you get.

- Can we observe any patterns between the outcomes in the game with 2 flips and the outcomes in the game with 3 flips?

HH **HT** **TH** **TT**

HHH **HHT** **HTH** **HTT** **THH** **THT** **TTH** **TTT**

Generating coin sequences



- Let's say that you're playing a game that involves flipping a coin a certain number of times in a row. Your success in the game depends on the exact sequence of "heads" and "tails" that you get.

- Can we observe any patterns between the outcomes in the game with 2 flips and the outcomes in the game with 3 flips?
  - There is a self-similar tree-like relationship between the possible outcomes of 2 flips and the possible outcomes of 3 flips.
  - The branching in the tree comes from deciding whether or not to add an H or a T to the existing sequence.
  - Together these branching sequences of decisions define a **decision tree.**

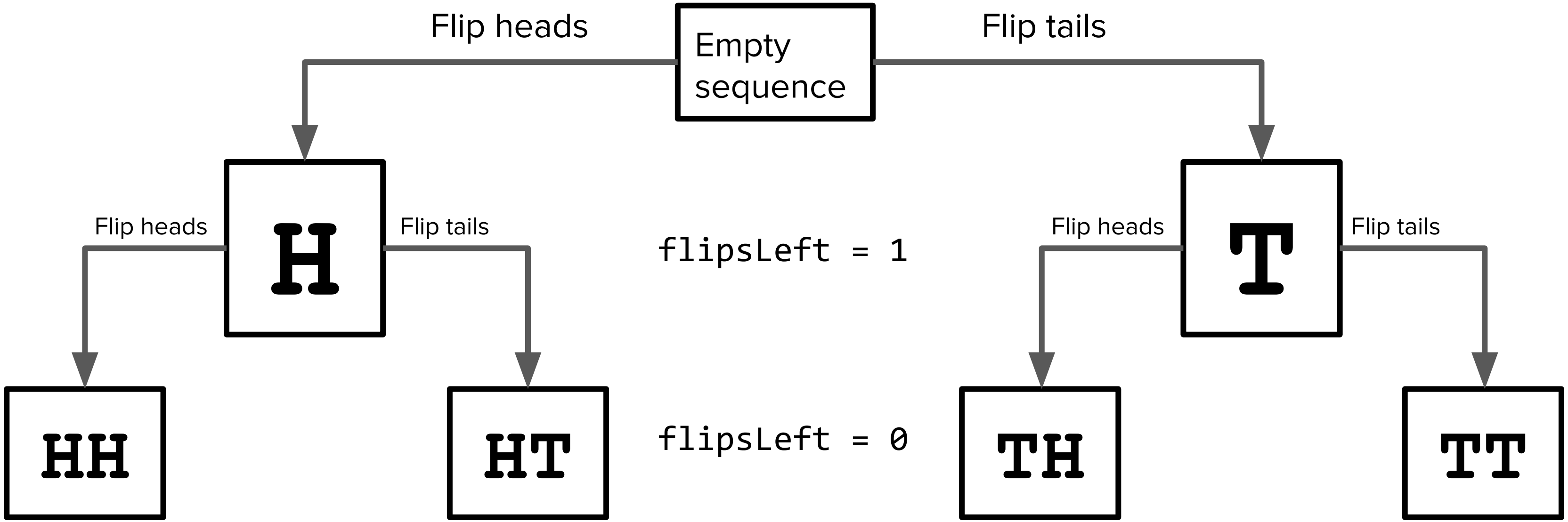
Why decision trees?

- We've seen trees in the context of fractals (drawing pretty shapes), but now we're going to apply meaningful context to these trees.

- In problems where we care about many possible outcomes, decision trees can help illustrate the recursive backtracking strategy for generating outcomes. They model the options we can choose from and the "decisions" we make along the way.

- Let's create a visualization of the possible space of outcomes that could result from N coin flips. Each decision is one flip, and the options for a single flip are either heads or tails.

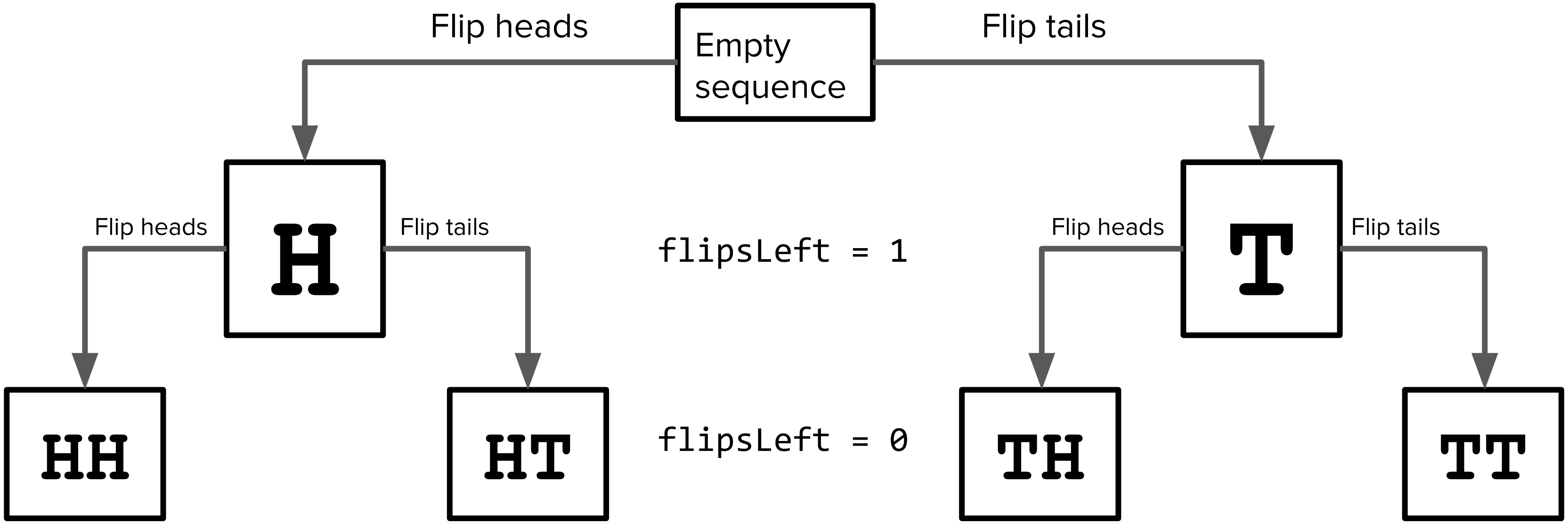
Example decision tree for N=2



Base case: when flipsLeft = 0

We reach the base case when we reach the leaves of our decision tree.

Example decision tree for N=2



Recursive cases: add 'H' **or** 'T' to the sequence

The branching points in our tree. We'll have a recursive call for each option.

Let’s code it!

```
void generateSequences(int length);
```

Takeaways: recursive backtracking + decision trees

- Unlike our previous recursion paradigm in which a solution gets built up as recursive calls return, in backtracking our final outputs occur at our base cases (leaves) and get built up as we go down the decision tree.

- The **height** of the tree corresponds to the **number of decisions** we have to make. The **width** at each decision point corresponds to the **number of options**.

- To exhaustively explore the entire search space, we must **try every possible option for every possible decision**. That can be a lot of paths to walk!
  - For the previous example, we have to make N decisions, with 2 choices for each decision. This means 2^N total possible outcomes!

Summary

Why do we use recursion?

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

Two types of recursion

Basic recursion

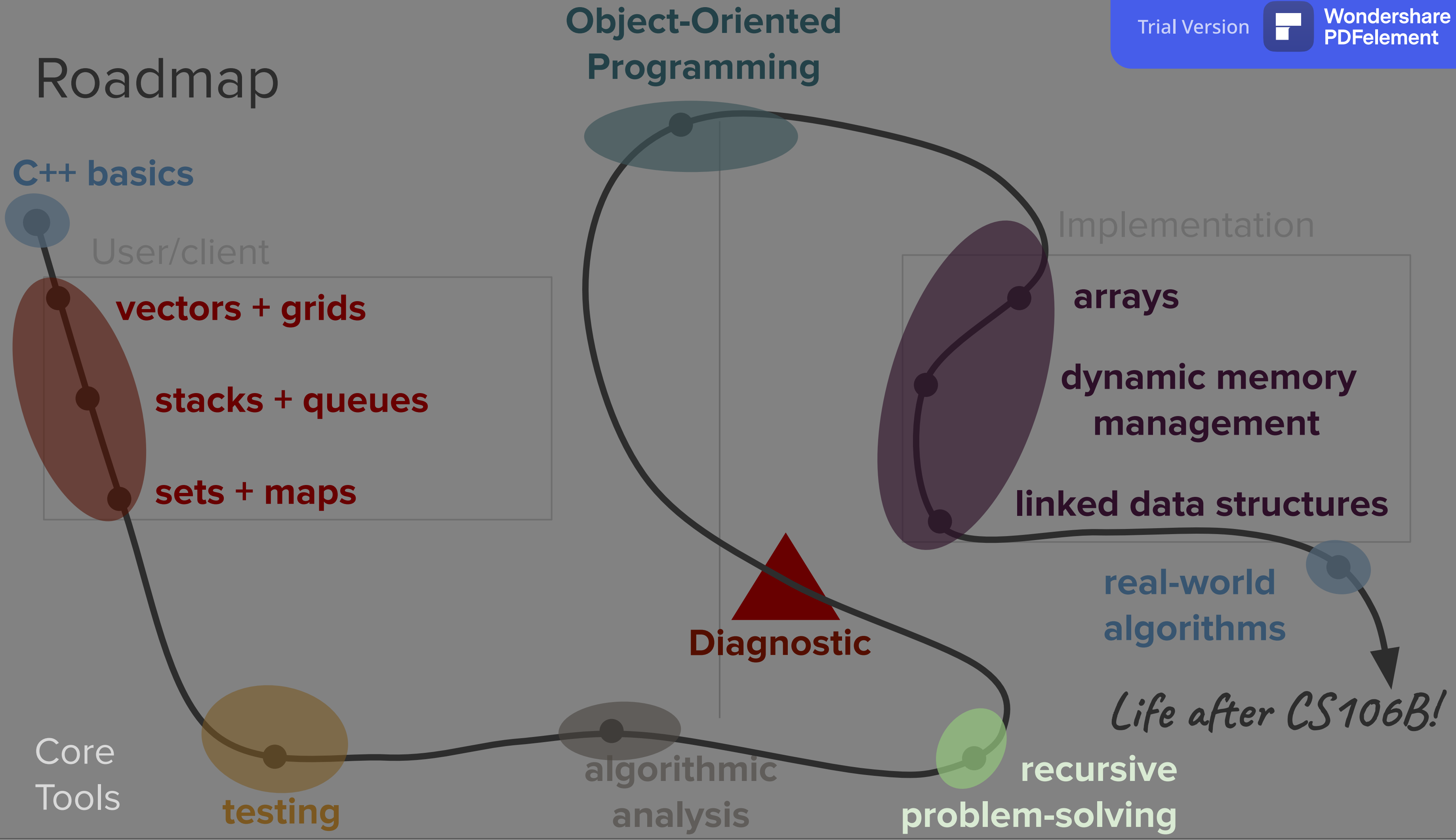
- One repeated task that builds up a solution as you come back up the call stack
- The final base case defines the initial seed of the solution and each call contributes a little bit to the solution
- Initial call to recursive function produces final solution

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

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

Recursive Backtracking