Recursive Fractals

What examples of recursion have you encountered in day-to-day life?

(put your answers in the chat)
Roadmap

Object-Oriented Programming

C++ basics

User/client

vectors + grids

stacks + queues

sets + maps

arrays
dynamic memory management
linked data structures
real-world algorithms
recursive problem-solving

Diagnostic

Life after CS106B!

Core Tools

testing

algorithmic analysis

Recursive problem-solving
Today’s question

How can we use visual representations to understand recursion?

How can we use recursion to make art?
Today’s topics

1. Review
2. Defining recursion in the context of fractals
3. The Cantor Set
4. The Sierpinski Carpet
5. Revisiting the Towers of Hanoi
Review
Definition

recursion

A problem-solving technique in which tasks are completed by reducing them into repeated, smaller tasks of the same form.
Recursion Review

- Recursion is a problem-solving technique in which tasks are completed by reducing them into **repeated, smaller tasks of the same form**.
  - A recursive operation (function) is defined in terms of itself (i.e. it calls itself).
Recursion Review

- Recursion is a problem-solving technique in which tasks are completed by reducing them into repeated, smaller tasks of the same form.

- Recursion has two main parts: the base case and the recursive case.
  - Base case: Simplest form of the problem that has a direct answer.
  - Recursive case: The step where you break the problem into a smaller, self-similar task.
Recursion Review

- Recursion is a problem-solving technique in which tasks are completed by reducing them into repeated, smaller tasks of the same form.

- Recursion has two main parts: the base case and the recursive case.

- The solution will get built up as you come back up the call stack.
  - The base case will define the “base” of the solution you’re building up.
  - Each previous recursive call contributes a little bit to the final solution.
  - The initial call to your recursive function is what will return the completely constructed answer.
Recursion Review

- Recursion is a problem-solving technique in which tasks are completed by reducing them into repeated, smaller tasks of the same form.

- Recursion has two main parts: the base case and the recursive case.

- The solution will get built up as you come back up the call stack.

- When solving problems recursively, look for self-similarity and think about what information is getting stored in each stack frame.
Recursion Review

- Recursion is a problem-solving technique in which tasks are completed by reducing them into repeated, smaller tasks of the same form.

- Recursion has two main parts: the base case and the recursive case.

- The solution will get built up as you come back up the call stack.

- When solving problems recursively, look for self-similarity and think about what information is getting stored in each stack frame.
Example:

isPalindrome()
Write a function that returns if a string is a palindrome

A string is a palindrome if it reads the same both forwards and backwards:

- `isPalindrome("level")` → true
- `isPalindrome("racecar")` → true
- `isPalindrome("step on no pets")` → true
- `isPalindrome("high")` → false
- `isPalindrome("hi")` → false
- `isPalindrome("palindrome")` → false
- `isPalindrome("X")` → true
- `isPalindrome(""")` → true
Approaching recursive problems

- Look for self-similarity.

- Try out an example and look for patterns.
  - Work through a simple example and then increase the complexity.
  - Think about what information needs to be “stored” at each step in the recursive case (like the current value of \( n \) in each \texttt{factorial} stack frame).

- Ask yourself:
  - What is the base case? (What is the simplest case?)
  - What is the recursive case? (What pattern of self-similarity do you see?)
Discuss:
What are the base and recursive cases?
(breakout rooms)
isPalindrome()

- Look for self-similarity: racecar
isPalindrome()

- Look for self-similarity: *racecar*
  - Look at the first and last letters of “racecar” → both are ‘r’
isPalindrome()

- Look for self-similarity: racecar
  - Look at the first and last letters of “racecar” → both are ‘r’
  - Check if “aceca” is a palindrome:
isPalindrome()

- Look for self-similarity:  racecar
  - Look at the first and last letters of “racecar” → both are ‘r’
  - Check if “aceca” is a palindrome:
    - Look at the first and last letters of “aceca” → both are ‘a’
    - Check if “cec” is a palindrome:
isPalindrome()

- Look for self-similarity: racecar
  - Look at the first and last letters of “racecar” → both are ‘r’
  - Check if “aceca” is a palindrome:
    - Look at the first and last letters of “aceca” → both are ‘a’
    - Check if “cec” is a palindrome:
      - Look at the first and last letters of “cec” → both are ‘c’
      - Check if “e” is a palindrome:
isPalindrome()

- Look for self-similarity: `racecar`
  - Look at the first and last letters of “racecar” → both are ‘r’
  - Check if “aceca” is a palindrome:
    - Look at the first and last letters of “aceca” → both are ‘a’
    - Check if “cec” is a palindrome:
      - Look at the first and last letters of “cec” → both are ‘c’
      - Check if “e” is a palindrome:
        - **Base case**: “e” is a palindrome
isPalindrome() 

- Look for self-similarity: racecar 
  - Look at the first and last letters of “racecar” → both are ‘r’ 
  - Check if “aceca” is a palindrome: 
    - Look at the first and last letters of “aceca” → both are ‘a’ 
    - Check if “cec” is a palindrome: 
      - Look at the first and last letters of “cec” → both are ‘c’ 
      - Check if “e” is a palindrome: 
        - Base case: “e” is a palindrome

What about the false case?
isPalindrome()

- Look for self-similarity: high
isPalindrome()

- Look for self-similarity: high
  - Look at the first and last letters of “high” → both are ‘h’
isPalindrome()

- Look for self-similarity: **high**
  - Look at the first and last letters of “high” → both are ‘h’
  - Check if “ig” is a palindrome:
isPalindrome()

- Look for self-similarity: high
  - Look at the first and last letters of “high” ➔ both are ‘h’
  - Check if “ig” is a palindrome:
    - Look at the first and last letters of “ig” ➔ not equal
    - **Base case:** Return false
isPalindrome()

- **Base cases:**
  - isPalindrome("") → **true**
  - isPalindrome(string of length 1) → **true**
  - If the first and last letters are not equal → **false**

- **Recursive case:** If the first and last letters are equal, isPalindrome(string) = isPalindrome(string minus first and last letters)
isPalindrome()  

- **Base cases:**
  - isPalindrome("") → **true**
  - isPalindrome(string of length 1) → **true**
  - If the first and last letters are not equal → **false**

- **Recursive case:** If the first and last letters are equal,  
  isPalindrome(string) = isPalindrome(string minus first and last letters)

*There can be multiple base (or recursive) cases!*
isPalindrome()

```cpp
bool isPalindrome (string s) {
    if (s.length() < 2) {
        return true;
    } else {
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        return isPalindrome(s.substr(1, s.length() - 2));
    }
}
```
isPalindrome() in action

```cpp
int main() {
    cout << boolalpha <<
        isPalindrome("racecar")
    << noboolalpha << endl;
    return 0;
}
```
isPalindrome() in action

```cpp
int main() {
    cout << boolalpha <<
        isPalindrome("racecar")
    << noboolalpha << endl;
    return 0;
}
```
isPalindrome() in action

```cpp
int main() {
    string s = "racecar";
    bool isPalindrome (string s) {
        if (s.length() < 2) {
            return true;
        } else {
            if (s[0] != s[s.length() - 1]) {
                return false;
            }
            return isPalindrome(s.substr(1, s.length() - 2));
        }
    }
    cout << boolalpha << isPalindrome(s) << noboolalpha << endl;
    return 0;
}
```
isPalindrome() in action

```cpp
int main() {
    string s = "racecar";
    bool isPalindrome(string s) {
        if (s.length() < 2) {
            return true;
        } else {
            if (s[0] != s[s.length() - 1]) {
                return false;
            }
            return isPalindrome(s.substr(1, s.length() - 2));
        }
    }
    cout << boolalpha << isPalindrome(s) << noboolalpha << endl;
    return 0;
}
```
isPalindrome() in action

```cpp
int main() {
    bool isPalindrome (string s) {
        if (s.length() < 2) {
            return true;
        } else {
            if (s[0] != s[s.length() - 1]) {
                return false;
            }
            return isPalindrome(s.substr(1, s.length() - 2));
        }
    }
    cout << boolalpha << isPalindrome("racecar") << noboolalpha << endl;
    return 0;
}
```
isPalindrome() in action

```cpp
int main() {
    bool isPalindrome(string s) {
        if (s.length() < 2) {
            return true;
        } else {
            if (s[0] != s[s.length() - 1]) {
                return false;
            }
            return isPalindrome(s.substr(1, s.length() - 2));
        }
    }
    cout << boolalpha << isPalindrome("racecar") << noboolalpha << endl;
    return 0;
}
```
isPalindrome() in action

```cpp
int main() {
    bool isPalindrome (string s) {
        if (s.length() < 2) {
            return true;
        } else {
            if (s[0] != s[s.length() - 1]) {
                return false;
            }
            return isPalindrome(s.substr(1, s.length() - 2));
        }
    }
    string S = "racecar"
    return isPalindrome(S);
}
```
isPalindrome() in action

```cpp
int main() {
    bool isPalindrome (string s) {
        if (s.length() < 2) {
            return true;
        }
        else {
            if (s[0] != s[s.length() - 1]) {
                return false;
            }
            return isPalindrome(s.substr(1, s.length() - 2));
        }
    }
    cout << boolalpha << isPalindrome("racecar") << noboolalpha << endl;
    return 0;
}
```
isPalindrome() in action

```cpp
int main() {
    cout << boolalpha <<
    isPalindrome("racecar") << noboolalpha << endl;
    return 0;
}

bool isPalindrome (string s) {
    if (s.length() < 2) {
        return true;
    } else {
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        return isPalindrome(s.substr(1, s.length() - 2));
    }
}
```
isPalindrome() in action

```cpp
int main() {
    cout << boolalpha <<
    isPalindrome("racecar") << noboolalpha << endl;
    return 0;
}

bool isPalindrome (string s) {
    if (s.length() < 2) {
        return true;
    } else {
        if (s[0] != s[s.length() - 1]) {
            return false;
        } else {
            return isPalindrome(s.substr(1, s.length() - 2));
        }
    }
}
```
isPalindrome() in action

```cpp
int main() {
    bool isPalindrome(string s) {
        if (s.length() < 2) {
            return true;
        } else {
            if (s[0] != s[s.length() - 1]) {
                return false;
            }
            return isPalindrome(s.substr(1, s.length() - 2));
        }
    }
    cout << boolalpha << isPalindrome("racecar") << noboolalpha << endl;
    return 0;
}
```
isPalindrome() in action

```cpp
int main() {
    bool isPalindrome (string s) {
        if (s.length() < 2) {
            return true;
        } else {
            if (s[0] != s[s.length() - 1]) {
                return false;
            } else {
                return isPalindrome(s.substr(1, s.length() - 2));
            }
        }
        return s == s;
    }

    cout << boolalpha << isPalindrome("racecar") << noboolalpha << endl;
    return 0;
}
```
isPalindrome() in action

```cpp
bool isPalindrome (string s) {
    if (s.length() < 2) {
        return true;
    } else {
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        return isPalindrome(s.substr(1, s.length() - 2));
    }
}
```
isPalindrome() in action

```cpp
int main() {
    cout << boolalpha <<
    isPalindrome("racecar") << noboolalpha << endl;
    return 0;
}

bool isPalindrome (string s) {
    if (s.length() < 2) {
        return true;
    }
    else {
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        return isPalindrome(s.substr(1, s.length() - 2));
    }
}
```
isPalindrome() in action

```cpp
int main() {
    bool isPalindrome (string s) {
        if (s.length() < 2) {
            return true;
        } else {
            if (s[0] != s[s.length() - 1]) {
                return false;
            }
            return isPalindrome(s.substr(1, s.length() - 2));
        }
    }
    cout << boolalpha << isPalindrome("racecar") << noboolalpha << endl;
    return 0;
}
```

true
isPalindrome() in action

```cpp
int main() {
    bool isPalindrome(string s) {
        if (s.length() < 2) {
            return true;
        } else {
            if (s[0] != s[s.length() - 1]) {
                return false;
            }
            return isPalindrome(s.substr(1, s.length() - 2));
        }
    }
    cout << boolalpha << isPalindrome("racecar") << noboolalpha << endl;
    return 0;
}
```

The function `isPalindrome` checks if a string is a palindrome. It handles the base case where the length is less than 2 by returning `true`. For strings of length 2 or more, it compares the first and last characters. If they are not equal, it returns `false`. Otherwise, it recursively calls `isPalindrome` with the substring excluding the first and last characters. The main function demonstrates the usage of `isPalindrome` with the string "racecar", which is a palindrome, and prints the result as `true`. 
isPalindrome() in action

```cpp
int main() {
    cout << boolalpha <<
    isPalindrome("racecar")
    << noboolalpha << endl;
    return 0;
}

Prints true!
How can we use visual representations to understand recursion?
Self-Similarity
Self-Similarity

- Solving problems recursively and analyzing recursive phenomena involves identifying **self-similarity**
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.
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.
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.
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.
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.
Self-similarity shows up in many real-world objects and phenomena, and is the key to truly understanding their formation and existence.
Graphical Representations of Recursion
Graphical Representations of Recursion

- Our first exposure to recursion yesterday was graphical in nature!
  - "Vee" is a recursive program that traces the path of a sprite in Scratch
  - The sprite draws out a funky tree-like structure as it goes along its merry way
Graphical Representations of Recursion

- Our first exposure to recursion yesterday was graphical in nature!
  - "Vee" is a recursive program that traces the path of a sprite in Scratch
  - The sprite draws out a funky tree-like structure as it goes along its merry way

- Graphical representations of recursion allow us to visualize the result of having multiple recursive calls
  - Understanding this "branching" of the tree is critical to solving challenging problems with recursion
Fractals
Fractals

- A **fractal** is any repeated, graphical pattern.
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.
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.
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.
Understanding Fractal Structure
What differentiates the smaller tree from the bigger one?
What differentiates the smaller tree from the bigger one?
1. It's at a different position.
What differentiates the smaller tree from the bigger one?
1. It's at a different position.
2. It has a different size.
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.
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**.
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.
An order-0 tree

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.
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.
An order-2 tree

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.
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.
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.
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.
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.
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.
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.
In Summary
In Summary

We drew this tree recursively.
In Summary

Each recursive call just draws one branch. The sum total of all the recursive calls draws the whole tree.
An Awesome Website!

http://recursivedrawing.com/
Announcements
Announcements

● Assignment 3 will be released by the end of the day today.
  ○ The YEAH session will take place tomorrow from 11:30am-12:30pm PDT!

● Make sure to check out our weekly announcement posts on Ed – there's lot of important info contained there!

● Assignment 1 Revisions are due today at 11:59pm PDT.

● The Mid-Quarter Diagnostic will be administered next week between 12:30pm on Wednesday, July 21 and 11:30am PDT on Friday, July 23.
  ○ More details (logistics, software, etc.) and practice materials posted Wed.
How can we use recursion to make art?
C++ Stanford graphics library
Graphics in CS106B

● Creating graphical programs is not one of our main focuses in this class, but a brief crash course in working with graphical programs is necessary to be able to code up some fractals of our own.

● The Stanford C++ libraries provide extensive capabilities to create custom graphical programs. The full documentation of these capabilities can be found in the official documentation.

● We will abstract away almost all of the complexity for you via provided helper functions.
  ○ There are two main classes/components of the library you need to know: GWindow and GPoint
**GWindow**

- A **GWindow** is an abstraction for the graphical window upon which we will do all of our drawing.
**GWindow**

- A **GWindow** is an abstraction for the graphical window upon which we will do all of our drawing.
- The window defines a coordinate system of x-y values
  - The top left corner is \((0, 0)\)
  - The bottom right corner is \((\text{windowWidth}-1, \text{windowHeight}-1)\)
**GWindow**

- A **GWindow** is an abstraction for the graphical window upon which we will do all of our drawing.
- The window defines a coordinate system of x-y values
  - The top left corner is \((0, 0)\)
  - The bottom right corner is \((\text{windowWidth}-1, \text{windowHeight}-1)\)
- All lines and shapes drawn on the window are defined by their \((x, y)\) coordinates
GPoint

- A GPoint is a handy way to bundle up the x-y coordinates for a specific point in the window.
  - Very similar in functionality to the GridLocation struct we learned about before!
**GPoint**

- A **GPoint** is a handy way to bundle up the x-y coordinates for a specific point in the window.
  - Very similar in functionality to the **GridLocation** struct we learned about before!

```cpp
GPoint topLeft(200, 100);
GPoint bottomRight(400, 250);
drawFilledRect(topLeft, bottomRight);

GPoint midpoint = {
    (topLeft.x + bottomRight.x) / 2,
    (topLeft.y + bottomRight.y) / 2
};
```
Cantor Set example
The first fractal we will code is called the "Cantor" fractal, named after the late-19th century German mathematician Georg Cantor.

The Cantor fractal is a set of lines where there is one main line, and below that there are two other lines: each 1/3 of the width of the original line, with one on the left and one on the right (with a 1/3 separation of whitespace between them).

Below each of the other lines is an identical situation: two 1/3 lines.

This repeats until the lines are no longer visible.
An order-0 Cantor Set
An order-1 Cantor Set
An order-2 Cantor Set
An order-6 Cantor Set
An order-6 Cantor Set

Another Cantor Set
An order-6 Cantor Set

Another Cantor Set

Also a Cantor Set
How to draw an order-n Cantor Set
How to draw an order-n Cantor Set

1. Draw a line from start to end.
How to draw an order-n Cantor Set

1. Draw a line from **start** to **end**.

2. Underneath the left third, draw a Cantor Set of order-\((n - 1)\).
How to draw an order-$n$ Cantor Set

1. Draw a line from **start** to **end**.

2. Underneath the left third, draw a Cantor Set of order-$(n - 1)$.

3. Underneath the right third, draw a Cantor Set of order-$(n - 1)$.
How to draw an order-n Cantor Set

1. Draw a line from \textbf{start} to \textbf{end}.

2. Underneath the left third, draw a Cantor Set of order-(n - 1).

3. Underneath the right third, draw a Cantor Set of order-(n - 1).

Base case:
\begin{align*}
\text{order} &= 0
\end{align*}
Cantor Set demo

[Qt Creator]
Real-world application of the Cantor Set
Sierpinski Carpet example
Sierpinski Carpet

- First described by Wacław Sierpiński in 1916
- A generalization of the Cantor Set to two dimensions!
- Defined by the subdivision of a shape (a square in this case) into smaller copies of itself.
  - The same pattern applied to a triangle yields a Sierpinski triangle, which you will code up on the next assignment.
An order-0 Sierpinski Carpet
An order-1 Sierpinski Carpet

An order-1 carpet is subdivided into eight order-0 carpets arranged in this grid pattern.
An order-2 Sierpinski Carpet
An order-2 Sierpinski Carpet
Sierpinski Carpet Formalized

- Base Case (order-0)
  - Draw a filled square at the appropriate location
Sierpinski Carpet Formalized

- **Base Case (order-0)**
  - Draw a filled square at the appropriate location

- **Recursive Case (order-n, n ≠ 0)**
  - Draw 8 order n-1 Sierpinski carpets, arranged in a 3x3 grid, omitting the center location
Sierpinski Carpet Formalized

- **Base Case (order-0)**
  - Draw a filled square at the appropriate location

- **Recursive Case (order-n, n ≠ 0)**
  - Draw 8 order n-1 Sierpinski carpets, arranged in a 3x3 grid, omitting the center location

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>(0,1)</td>
<td>(0,2)</td>
</tr>
<tr>
<td>(1,0)</td>
<td></td>
<td>(1,2)</td>
</tr>
<tr>
<td>(2,0)</td>
<td>(2,1)</td>
<td>(2,2)</td>
</tr>
</tbody>
</table>
Sierpinski Carpet Formalized

- **Base Case (order-0)**
  - Draw a filled square at the appropriate location

- **Recursive Case (order-n, n ≠ 0)**
  - Draw 8 order n-1 Sierpinski carpets, arranged in a 3x3 grid, omitting the center location
    - i.e. Draw an n-1 fractal at (0,0), draw an n-1 fractal at (0,1), draw an n-1 fractal at (0,2)...

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>(0,1)</td>
<td>(0,2)</td>
</tr>
<tr>
<td>(1,0)</td>
<td></td>
<td>(1,2)</td>
</tr>
<tr>
<td>(2,0)</td>
<td>(2,1)</td>
<td>(2,2)</td>
</tr>
</tbody>
</table>
Sierpinski Carpet Pseudocode (Take 1)

drawSierpinskiCarpet (x, y, order):
    if (order == 0)
        drawFilledSquare(x, y, BASE_SIZE)
    else
        drawSierpinskiCarpet(newX(x, y, 0, 0), newY(x, y, 0, 0), order -1)
        drawSierpinskiCarpet(newX(x, y, 0, 1), newY(x, y, 0, 1), order -1)
        drawSierpinskiCarpet(newX(x, y, 0, 2), newY(x, y, 0, 2), order -1)
        drawSierpinskiCarpet(newX(x, y, 1, 0), newY(x, y, 1, 0), order -1)
        drawSierpinskiCarpet(newX(x, y, 1, 2), newY(x, y, 1, 2), order -1)
        drawSierpinskiCarpet(newX(x, y, 2, 0), newY(x, y, 2, 0), order -1)
        drawSierpinskiCarpet(newX(x, y, 2, 1), newY(x, y, 2, 1), order -1)
        drawSierpinskiCarpet(newX(x, y, 2, 2), newY(x, y, 2, 2), order -1)
Sierpinski Carpet Pseudocode (Take 1)

drawSierpinskiCarpet (x, y, order):
    if (order == 0)
        drawFilledSquare(x, y, size, size);
    else
        drawSierpinskiCarpet(newX(x, y, 0, 0), newY(x, y, 0, 0), order -1)
        drawSierpinskiCarpet(newX(x, y, 0, 1), newY(x, y, 0, 1), order -1)
        drawSierpinskiCarpet(newX(x, y, 0, 2), newY(x, y, 0, 2), order -1)
        drawSierpinskiCarpet(newX(x, y, 1, 0), newY(x, y, 1, 0), order -1)
        drawSierpinskiCarpet(newX(x, y, 1, 2), newY(x, y, 1, 2), order -1)
        drawSierpinskiCarpet(newX(x, y, 2, 0), newY(x, y, 2, 0), order -1)
        drawSierpinskiCarpet(newX(x, y, 2, 1), newY(x, y, 2, 1), order -1)
        drawSierpinskiCarpet(newX(x, y, 2, 2), newY(x, y, 2, 2), order -1)

This isn’t very pretty, can we do better?
Sierpinski Carpet Pseudocode (Take 2)

drawSierpinskiCarpet (x, y, order):
    if (order == 0)
        drawFilledSquare(x, y, BASE_SIZE)
    else
        for row = 0 to row = 2:
            for col = 0 to col = 2:
                if (col != 1 || row != 1):
                    x_i = newX(x, y, row, col)
                    y_i = newY(x, y, row, col)
                    drawSierpinskiCarpet(x_i, y_i, 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!
Revisiting the Towers of Hanoi

[Recursive Part 2: Electric Boogaloo]
Pseudocode for 3 disks

(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
Homework before tomorrow’s lecture

- 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.
What’s next?
Roadmap

C++ basics

User/client

vectors + grids
stacks + queues
sets + maps

Object-Oriented Programming

arrays
dynamic memory management
linked data structures

Implementation

real-world algorithms

Diagnostic

recursive problem-solving

Life after CS106B!

Core Tools

testing
algorithmic analysis
Advanced Recursion Examples