Recursive Fractals

Amrita Kaur

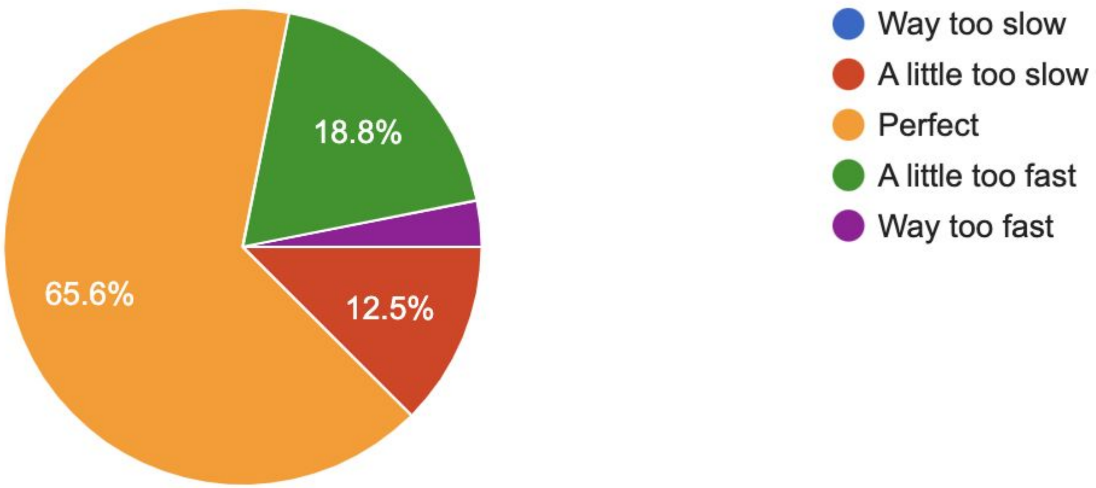
July 11, 2023

Contributions made from previous CS106B Instructors

Week 2 Feedback

Rate the pace of lecture

96 responses



Week 2 Feedback

Things you liked:

“**Everything** has been so fun so far! Keep up the same stuff.”

“The **live coding** examples were extremely helpful”

“I like that you guys have started to do **more recapping**. That makes it easier to catch up and is a good reminder to what we did last time.”

“I love the **slides** and I would love if you can keep doing it”

“increased **opportunities to discuss** with each other”

“Haven't ran into any issues that needed help but I feel like all the **resources** are there.”

Week 2 Feedback

Places we can improve:

“occasional **interactive larger pieces of code** where the class can give suggestions”

“It would be good if you could **repeat the question** that students asked so that those watching the recording can hear the questions more clearly.”

Week 2 Feedback

We hear you…

“Feedback form is really long :(”

“Not running too much over 2:30”

Week 2 Feedback

Assignment Feedback:

“Solving the first assignment was super fulfilling & I didn't require any additional assistance or clarification. I think that goes to show the effort that’s gone into setting up the assignment.”

“I loved how, almost immediately, I was able to use my C++ skills to **implement algorithms that are commonly used in the real world**.”

“I wish there was **more required coding than short answer**”

Week 2 Feedback

Anything else you would like us to know:

“Keep up the good work, the course is super engaging and interesting so far”

“You rock!”

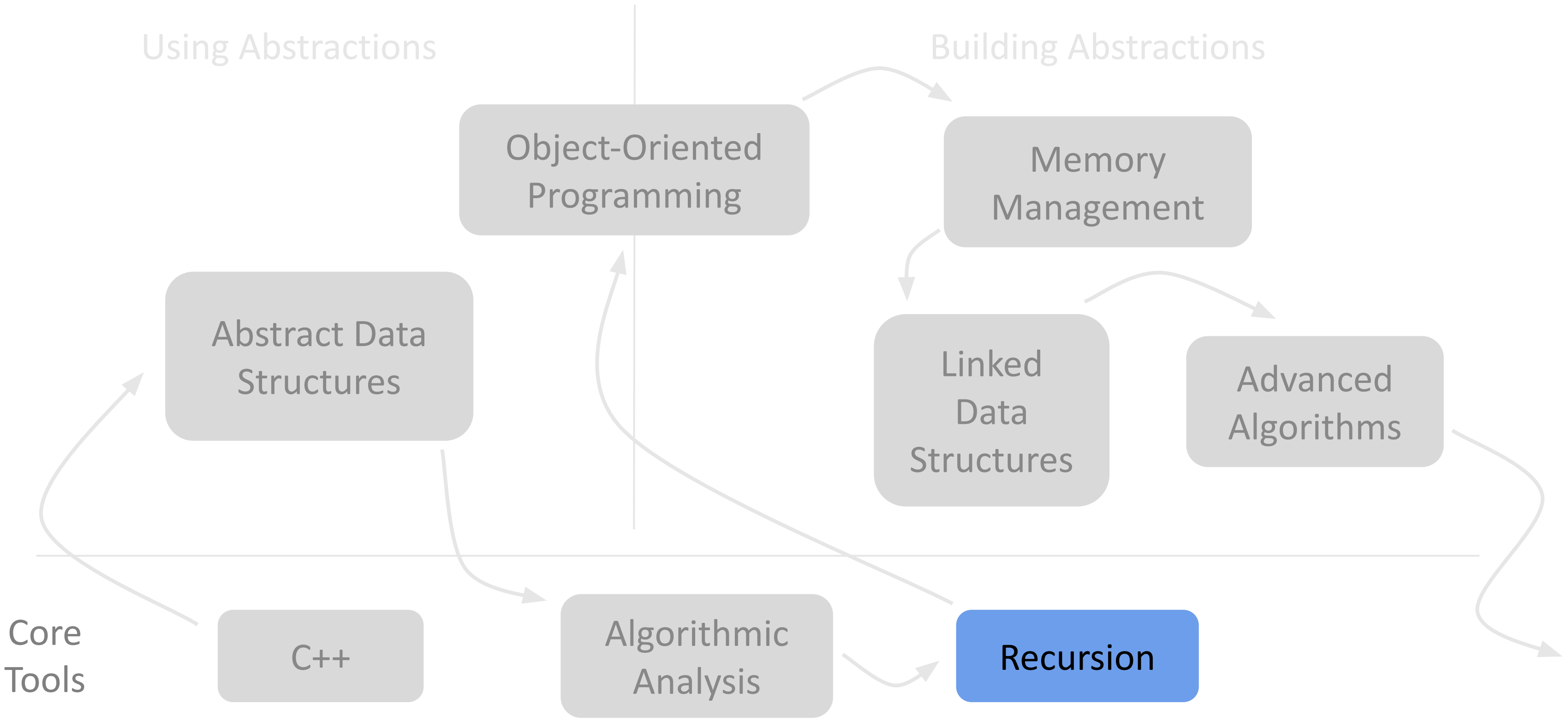
“I ate pancakes this morning.”

“the fairlife protein shakes are the best I've ever had. They taste just like the boxed horizon milk I used to drink.”

Announcements and Reminders

- Assignment 2 due Friday at 11:59pm
  - Use your help resources!

Roadmap


Using Abstractions
Building Abstractions
Object-Oriented Programming
Memory Management
Abstract Data Structures
Linked Data Structures
Advanced Algorithms
Core Tools
C++
Algorithmic Analysis
Recursion

What is recursion?

- A problem-solving technique in which tasks are completed by reducing them into **repeated, smaller tasks of the same form**.
- Powerful substitution for iteration (loops)
  - Start by seeing the difference between iterative vs. recursive solutions
  - Later will see problems that can only be solved by recursion
- Results in elegant, often shorter code
- In programming, it means that the function calls itself
  - Every time the function is called, the problem becomes a little smaller

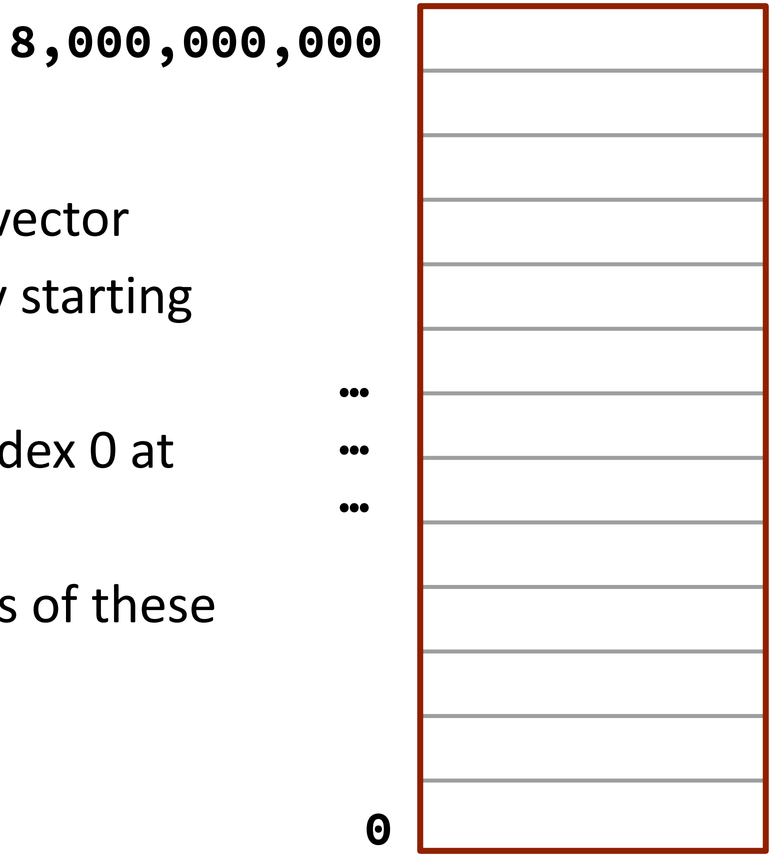
Two main components

- Base case
  - The simplest version of your problem that all other cases reduce to
  - An occurrence that can be answered directly
- Recursive case
  - More complex version of the problem that cannot be directly answered
  - Break down the task into smaller occurrences
  - Take the “recursive leap of faith” and trust the smaller tasks will solve the problem for you!

Three "Musts" of Recursion

1. Your code must have a case for all valid inputs.
2. You must have a base case that does not make recursive calls.
3. When you make a recursive call it should be to a simpler instance of the same problem, and make progress towards the base case.

Computer Memory

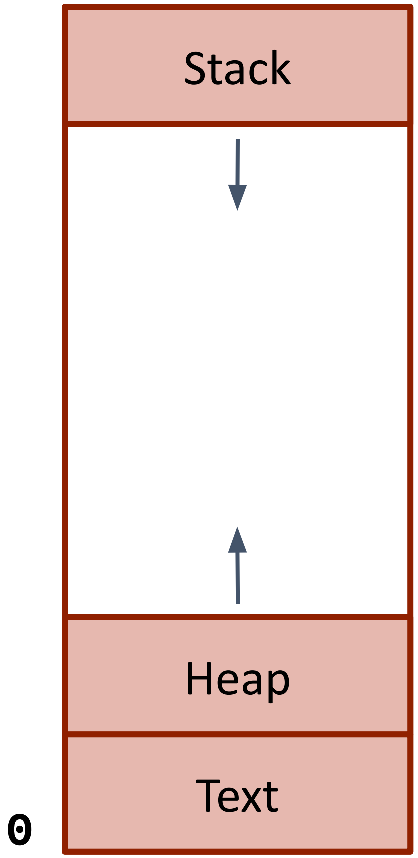


- Computer's memory is like a giant vector
- Like a vector, we can index memory starting from 0.
- We draw memory vertically with index 0 at the bottom
- Typical laptop's memory has billions of these indexed slots (one byte each)

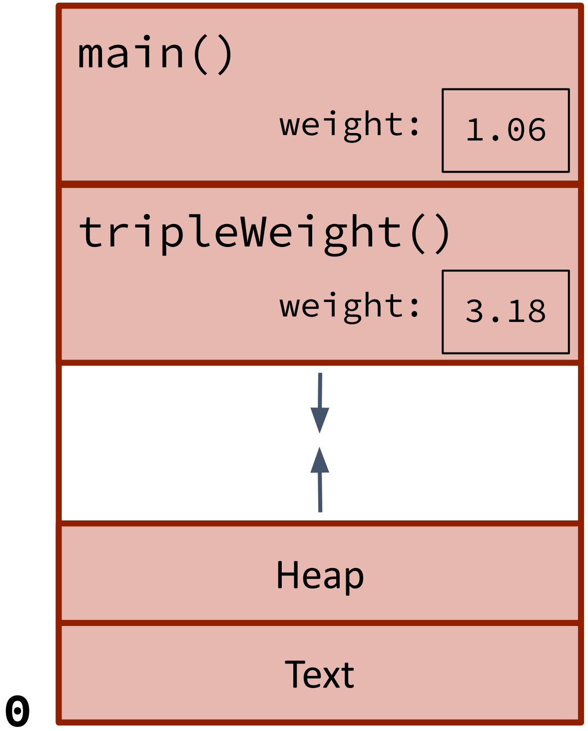
Computer Memory

Divide memory in a few main regions

- Text: program's own code
- Heap: where dynamically allocated memory resides
- Stack: where local variables for each function are stored



Stack Frames



The “stack” part of memory is a stack!

- A function call pushes a stack frame onto the stack
- A function return pops a stack frame from the stack

Recursive vs Iterative Methods

```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```

```
int factorialIterative (int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result = result * i;
    }
    return result;
}
```

`n = 5,` time = 5.823 ms

`n = 100,000,` time = 8.703 ms

`n = 1,000,000,` "segmentation fault"

`n = 5,` time = 5.485 ms

`n = 100,000,` time = 5.589 ms

`n = 1,000,000,` time = 7.501 ms

Approaching recursive problems

- Look for self-similarity.
- Try out an example.
  - 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
- 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?)

Palindromes

Is it a Palindrome?

- Write a function **`isPalindrome()`** that returns `true` or `false` based on if a string is a palindrome or not.
- A string is a palindrome if it reads the same forwards and backwards.
  - `isPalindrome(“racecar”) = true`
  - `isPalindrome(“noon”) = true`
  - `isPalindrome(“step on no pets”) = true`
  - `isPalindrome(“pindrop”) = false`
  - `isPalindrome(“yo”) = false`
  - `isPalindrome(“palindrome”) = false`
  - `isPalindrome(“X”) = true`
  - `isPalindrome(“”) = true`

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’

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’

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
            - “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: “noon”

isPalindrome()

Look for self-similarity: “noon”

- Look at the first and last letters of “noon” → both are ‘n’

isPalindrome()

Look for self-similarity: “noon”

- Look at the first and last letters of “noon” → both are ‘n’
- Check if “oo” is a palindrome

isPalindrome()

Look for self-similarity: “noon”

- Look at the first and last letters of “noon” → both are ‘n’
- Check if “oo” is a palindrome
    - Look at the first and last letters of “oo” → both are ‘o’

isPalindrome()

Look for self-similarity: “noon”

- Look at the first and last letters of “noon” → both are ‘n’
- Check if “oo” is a palindrome
    - Look at the first and last letters of “oo” → both are ‘o’
    - Check if “” is a palindrome

isPalindrome()

Look for self-similarity: “noon”

- Look at the first and last letters of “noon” → both are ‘n’
- Check if “oo” is a palindrome
  - Look at the first and last letters of “oo” → both are ‘o’
  - Check if “” is a palindrome
    - “” is a palindrome

isPalindrome()

Look for self-similarity: “noon”

- Look at the first and last letters of “noon” → both are ‘n’
- Check if “oo” is a palindrome
    - Look at the first and last letters of “oo” → both are ‘o’
    - Check if “” is a palindrome
        - **Base Case:** “” is a palindrome

isPalindrome()

Base Case:

Odd number of letters:

isPalindrome(string of length 1) = true

Even number of letters:

isPalindrome(“”) = true

isPalindrome()

Base Case:

Odd number of letters:

isPalindrome(string of length 1) = true

Even number of letters:

isPalindrome(“”) = true

Recursive Case:

isPalindrome()

Base Case:

Odd number of letters:

isPalindrome(string of length 1) = true

Even number of letters:

isPalindrome(“”) = true

Recursive Case:

If the first and last letters are the same,

isPalindrome(string) = isPalindrome(string minus first and last letters)

isPalindrome()

Look for self-similarity: “pindrop”

isPalindrome()

Look for self-similarity: “pindrop”

- Look at the first and last letters of “pindrop” → both are ‘p’

isPalindrome()

Look for self-similarity: “pindrop”

- Look at the first and last letters of “pindrop” → both are ‘p’
- Check if “indro” is a palindrome

isPalindrome()

Look for self-similarity: “pindrop”

- Look at the first and last letters of “pindrop” $\longrightarrow$ both are ‘p’
- Check if “indro” is a palindrome
    - Look at the first and last letters of “indro” $\longrightarrow$ not equal

isPalindrome()

Look for self-similarity: “pindrop”

- Look at the first and last letters of “pindrop” → both are ‘p’
- Check if “indro” is a palindrome
  - Look at the first and last letters of “indro” → not equal
  - Return `false`

isPalindrome()

Look for self-similarity: “pindrop”

- Look at the first and last letters of “pindrop” $\longrightarrow$ both are ‘p’
- Check if “indro” is a palindrome
  - Look at the first and last letters of “indro” $\longrightarrow$ not equal
  - **Base Case:** Return `false`

isPalindrome()

Base Case:

isPalindrome(string of length 1) = true

isPalindrome(“”) = true

isPalindrome(string where first and last letters aren’t equal) = false

Recursive Case:

If the first and last letters are the same,

isPalindrome(string) = isPalindrome(string minus first and last letters)

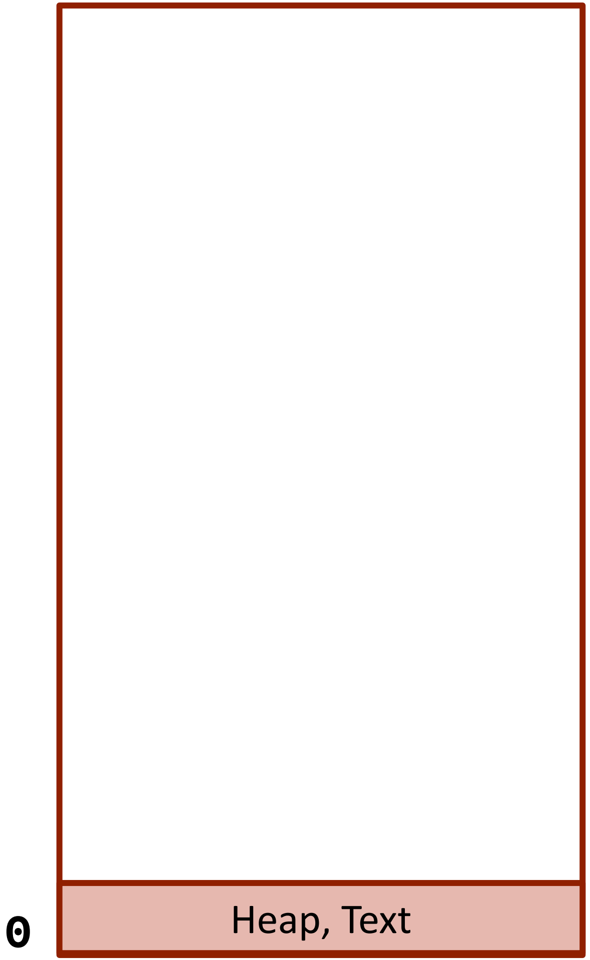
isPalindrome()

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

code here

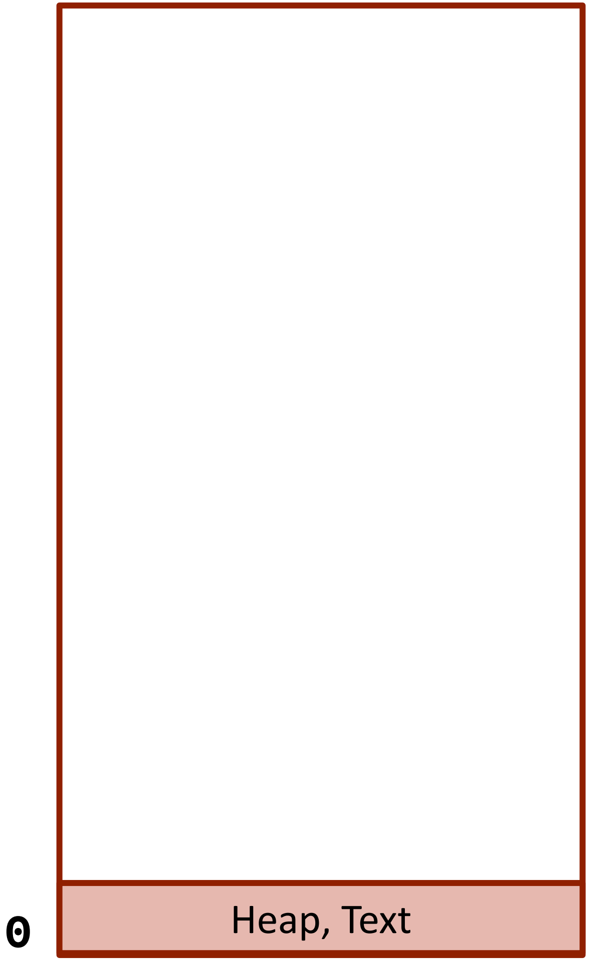
isPalindrome() in action

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



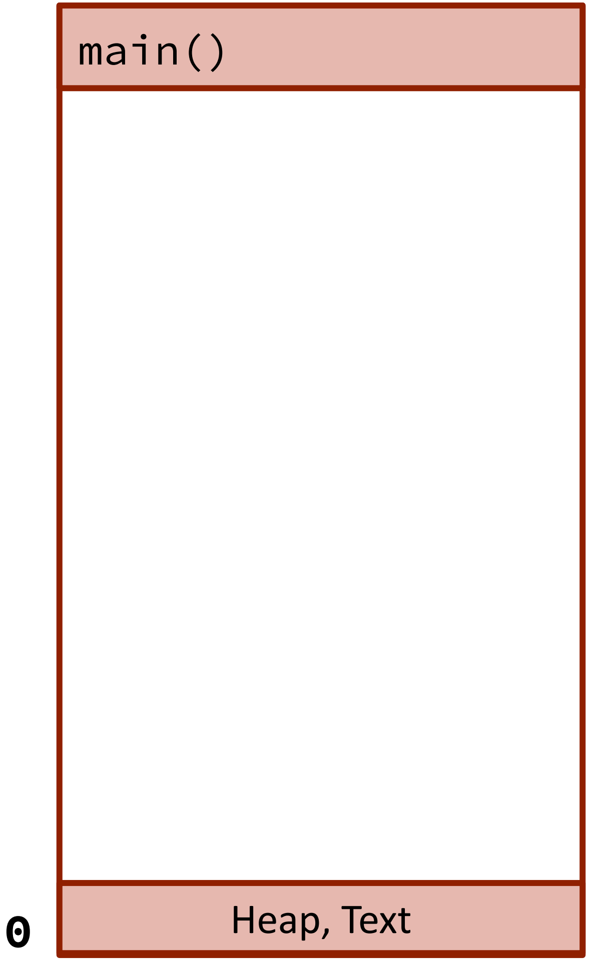
isPalindrome() in action

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



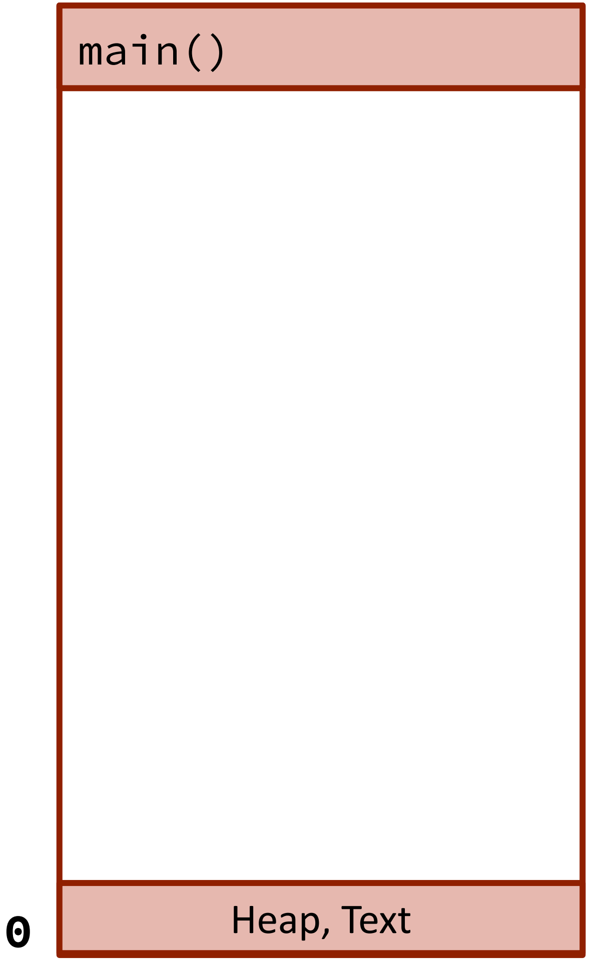
isPalindrome() in action

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



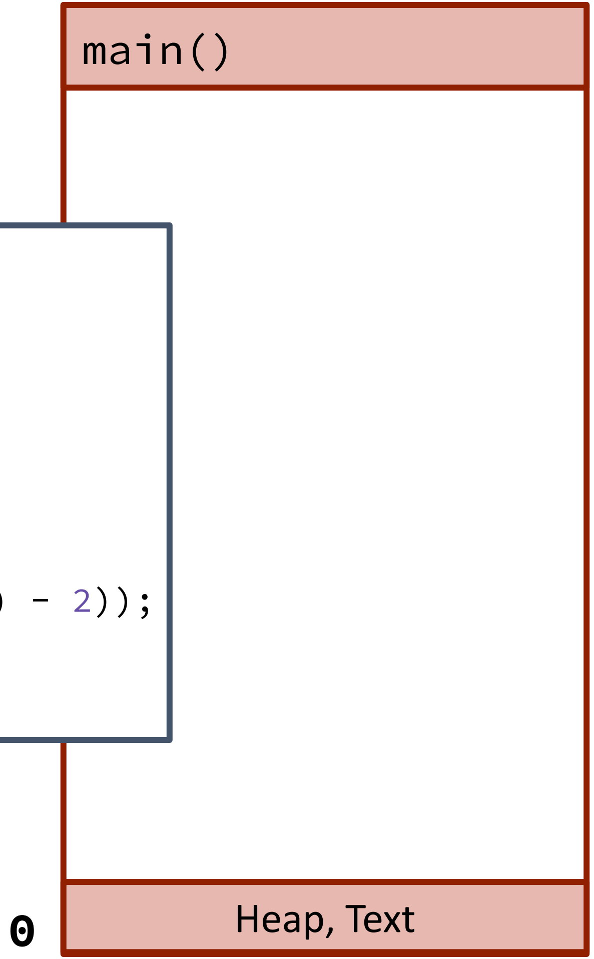
isPalindrome() in action

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



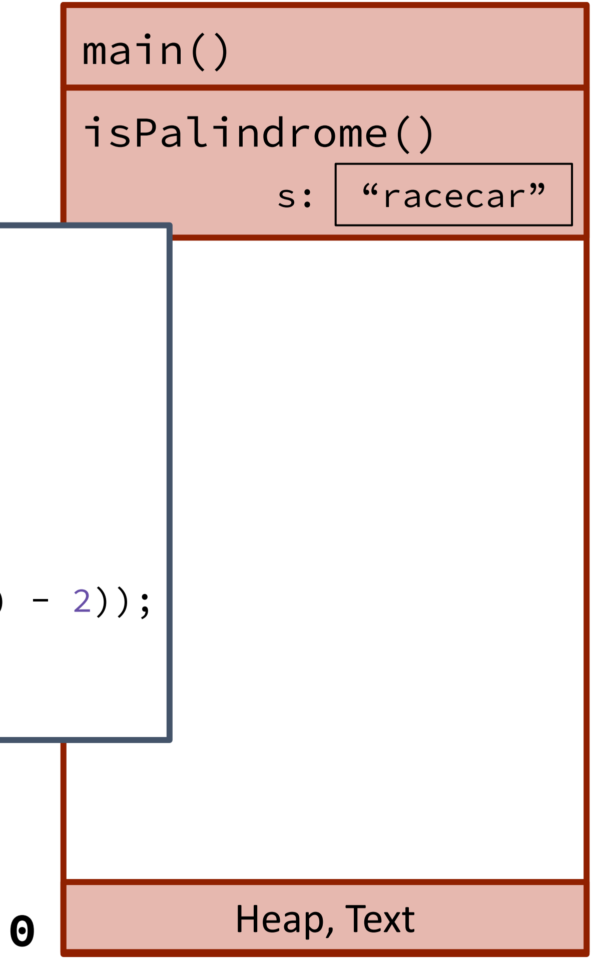
isPalindrome() in action

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



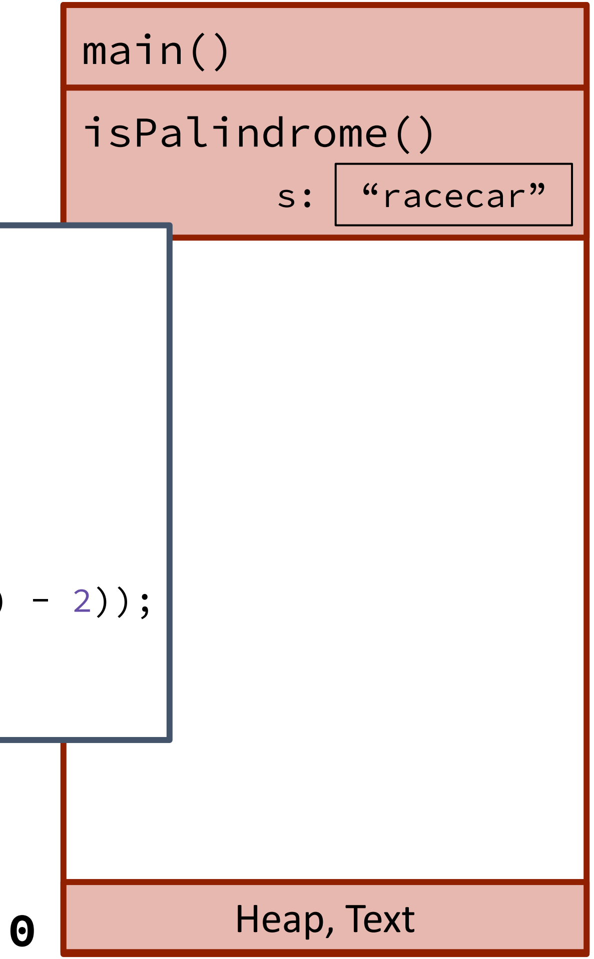
isPalindrome() in action

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



isPalindrome() in action

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



isPalindrome() in action

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

main()

isPalindrome()

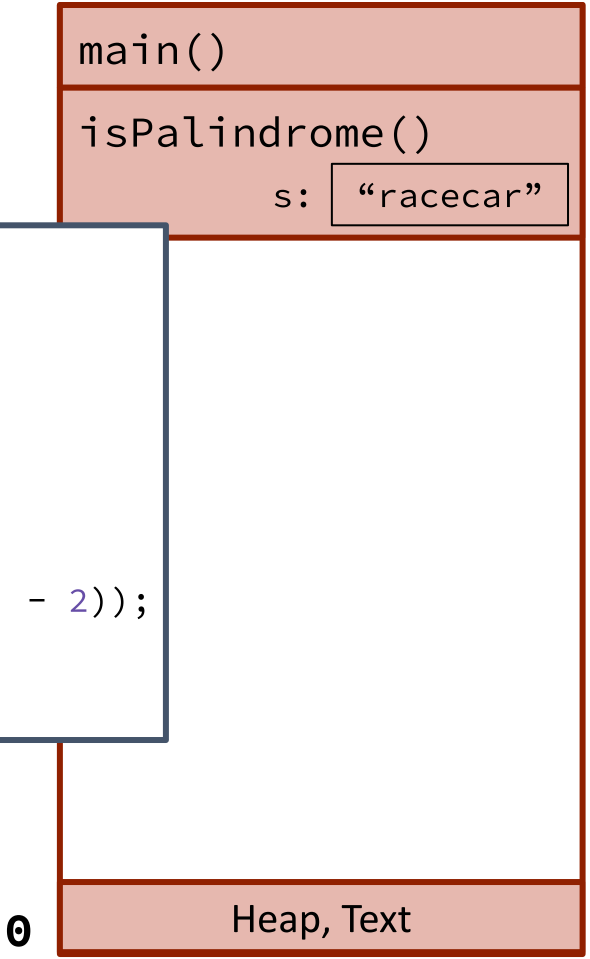
s: “racecar”

0

Heap, Text

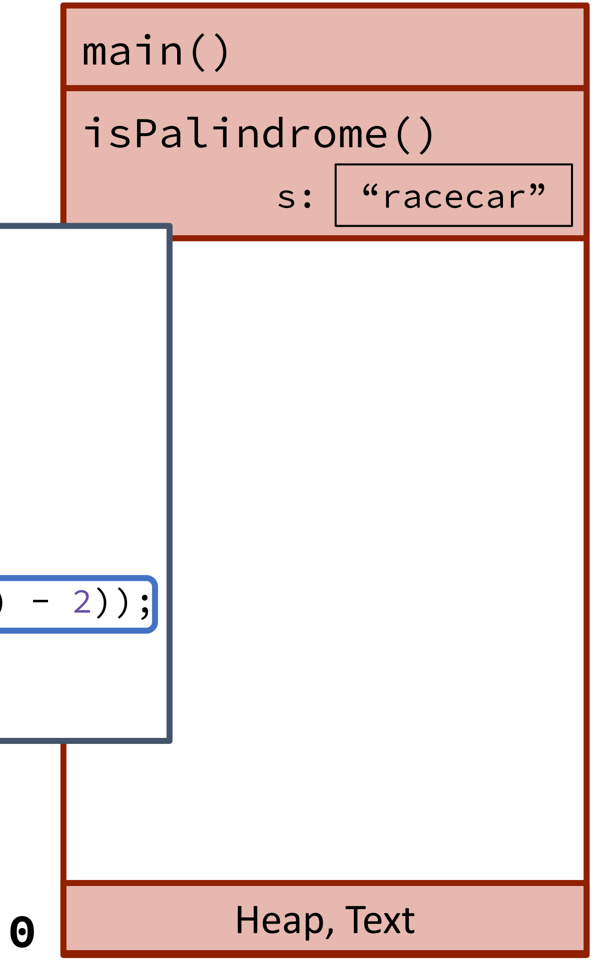
isPalindrome() in action

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



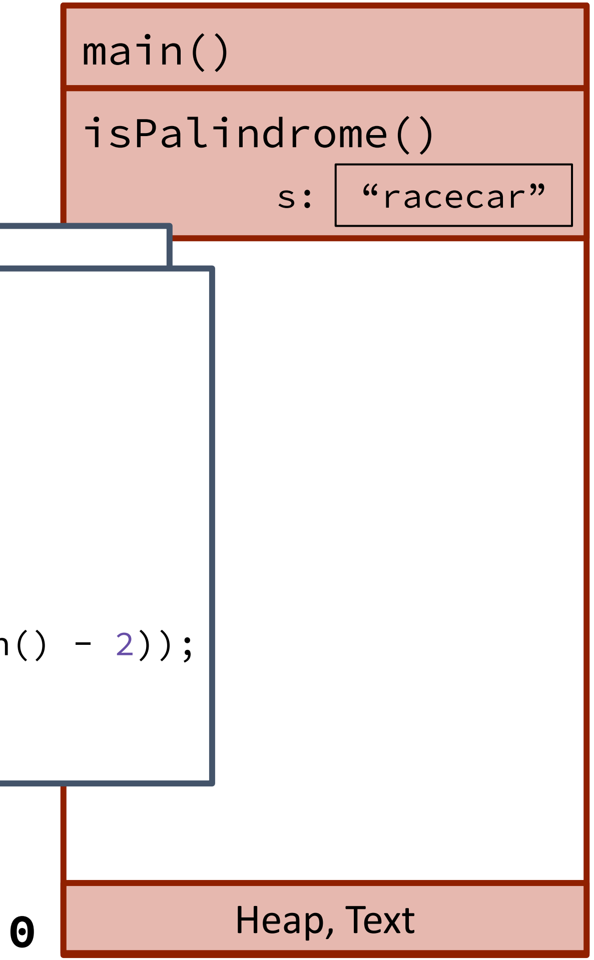
isPalindrome() in action

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



isPalindrome() in action

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



isPalindrome() in action

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

main()

isPalindrome()

s: “racecar”

isPalindrome()

s: “aceca”

0

Heap, Text

isPalindrome() in action

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

main()

isPalindrome()

s: "racecar"

indrome()

s: "aceca"

0

Heap, Text

isPalindrome() in action

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

main()

isPalindrome()

s: "racecar"

indrome()

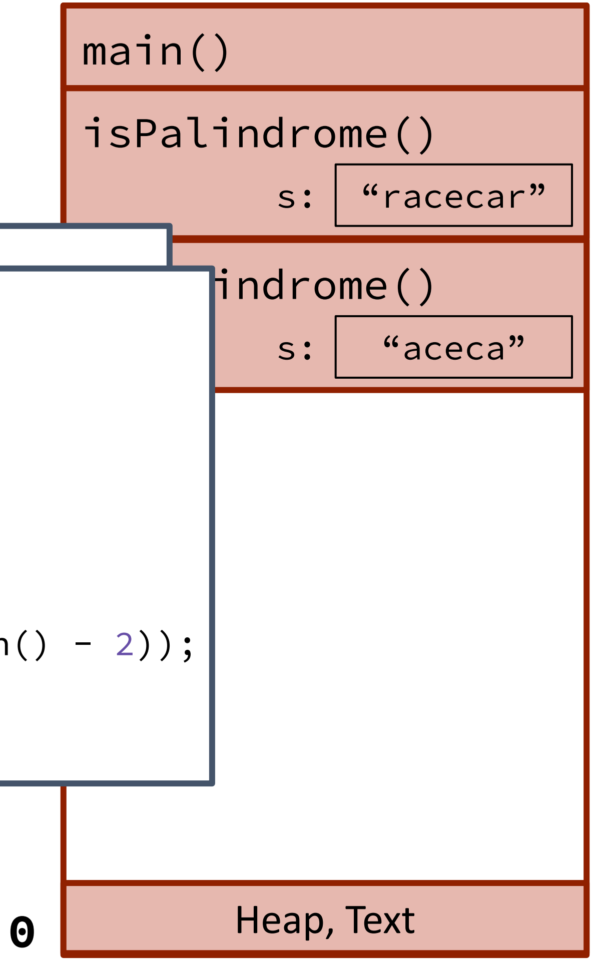
s: "aceca"

0

Heap, Text

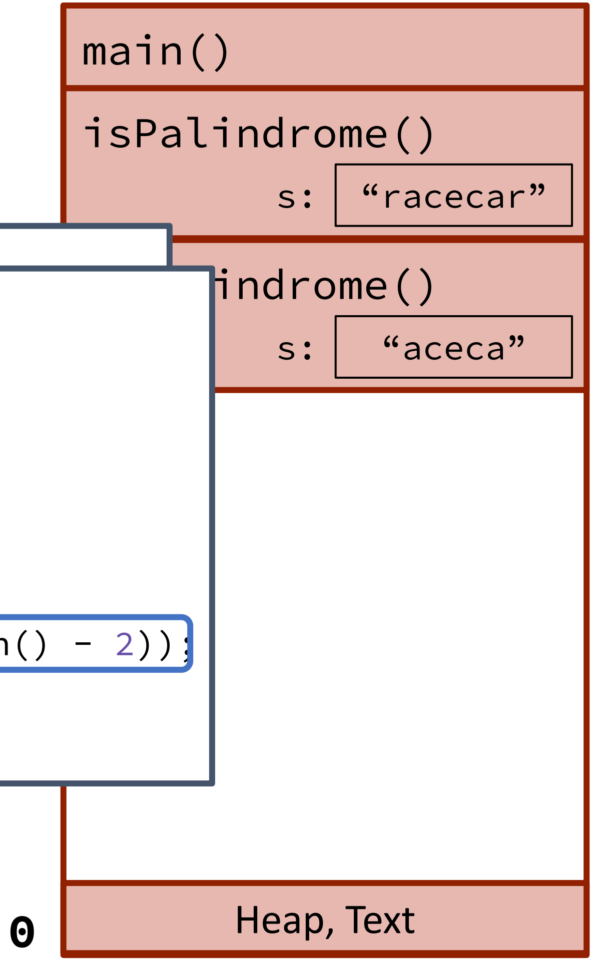
isPalindrome() in action

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



isPalindrome() in action

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



isPalindrome() in action

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

main()

isPalindrome()

s: “racecar”

indrome()

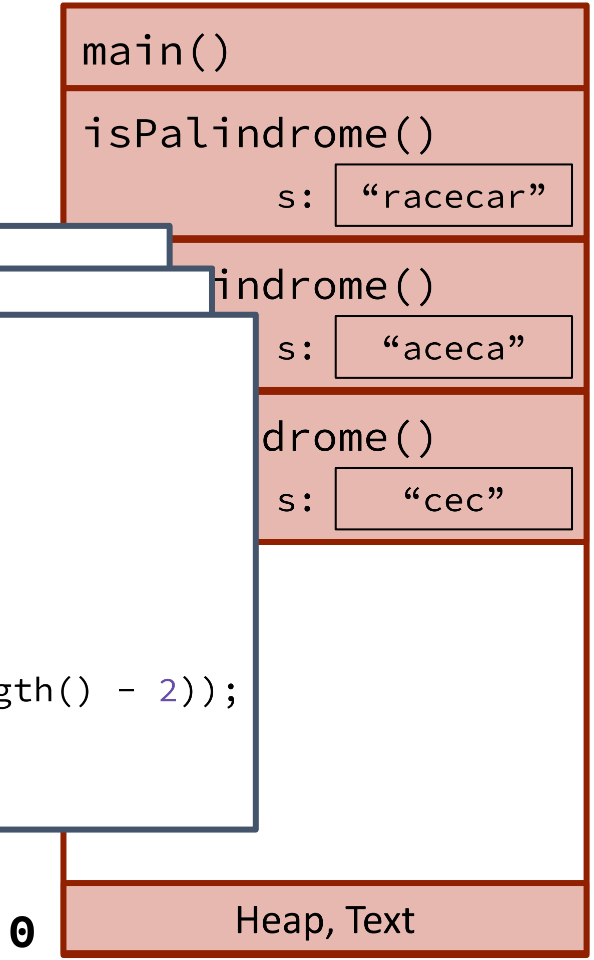
s: “aceca”

0

Heap, Text

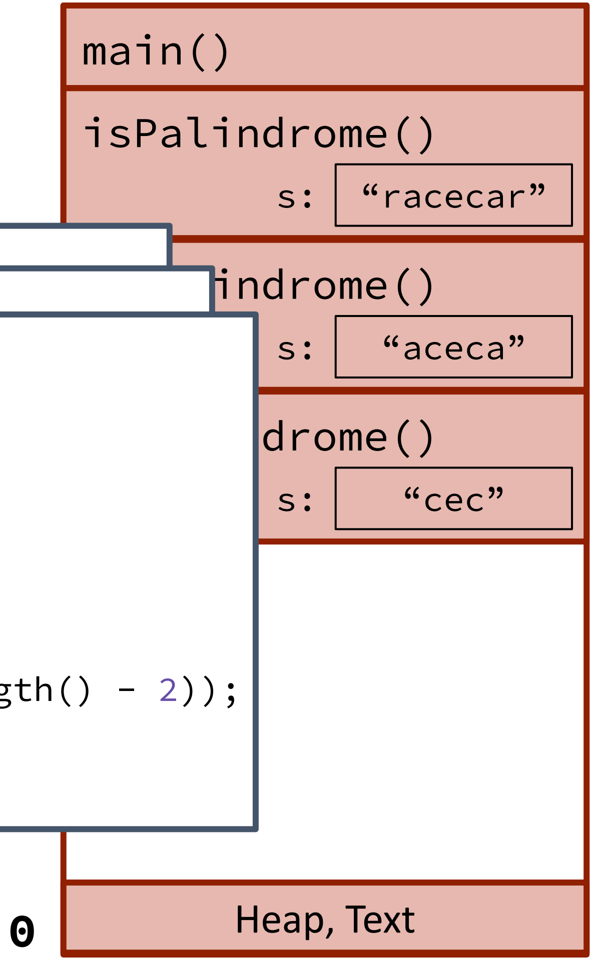
isPalindrome() in action

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



isPalindrome() in action

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



isPalindrome() in action

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

main()

isPalindrome()

s: "racecar"

indrome()

s: "aceca"

drome()

s: "cec"

0

Heap, Text

isPalindrome() in action

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

main()

isPalindrome()

s: "racecar"

indrome()

s: "aceca"

drome()

s: "cec"

0

Heap, Text

isPalindrome() in action

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

main()

isPalindrome()
s: “racecar”

indrome()
s: “aceca”

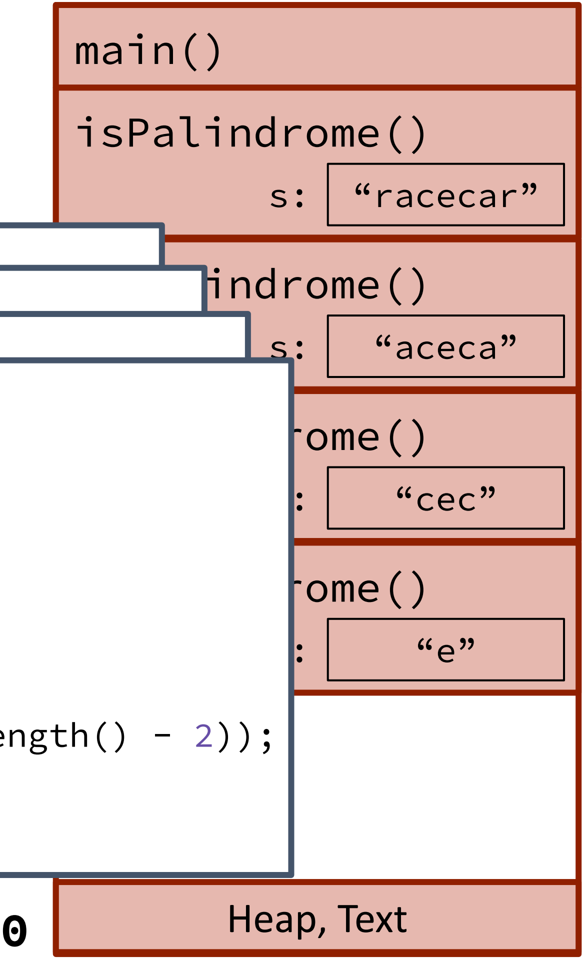
rome()
: “cec”

0

Heap, Text

isPalindrome() in action

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



isPalindrome() in action

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

main()

isPalindrome()
s: "racecar"

…indrome()
s: "aceca"

…ome()
…: "cec"

…ome()
…: "e"

0

Heap, Text

isPalindrome() in action

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

main()

isPalindrome()
s: “racecar”

indrome()
s: “aceca”

rome()
: “cec”

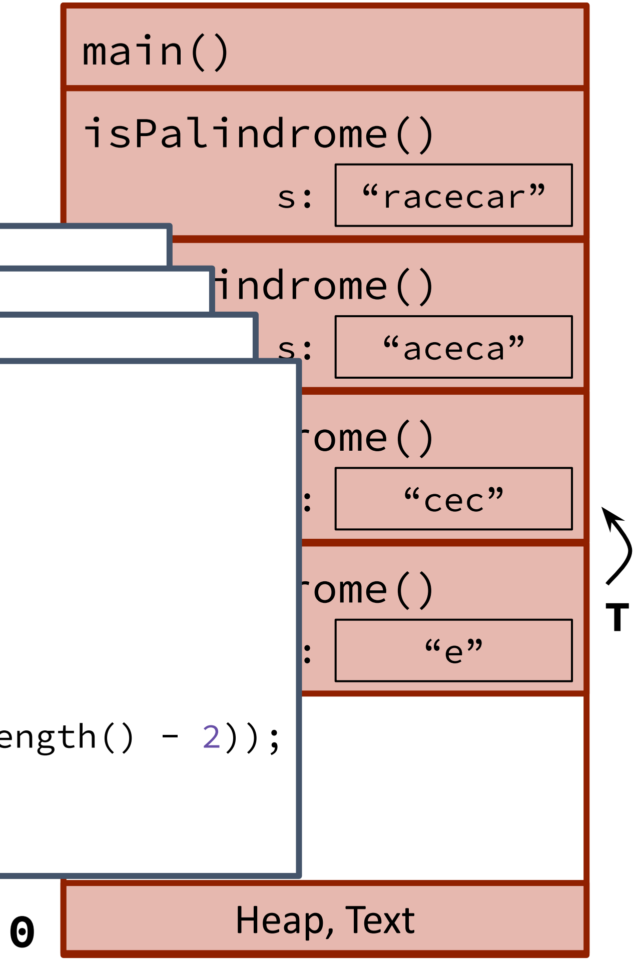
rome()
: “e”

0

Heap, Text

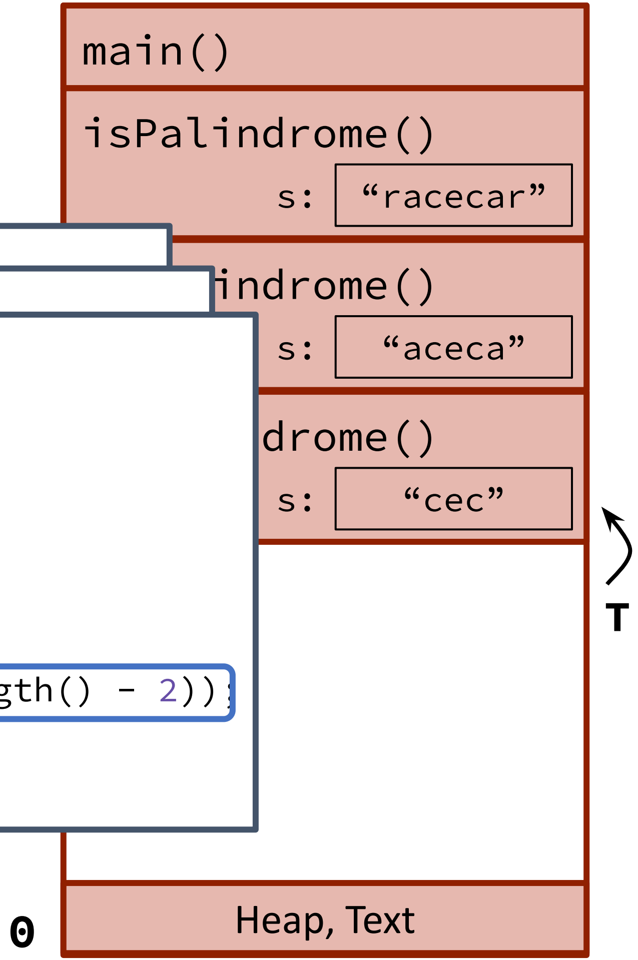
isPalindrome() in action

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



isPalindrome() in action

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



isPalindrome() in action

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

true

main()

isPalindrome()
s: "racecar"

s: "aceca"

s: "cec"

T

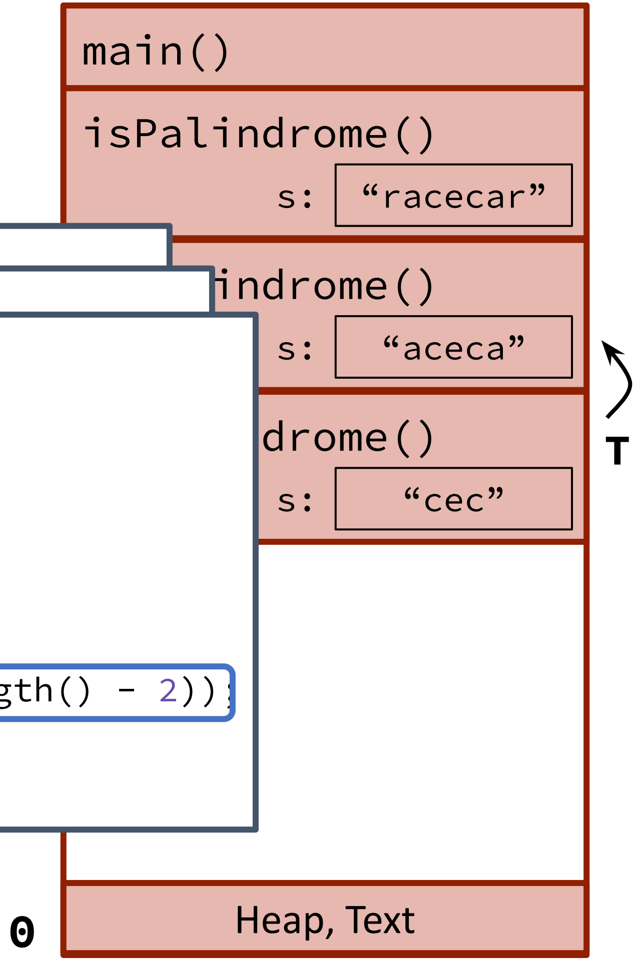
0

Heap, Text

isPalindrome() in action

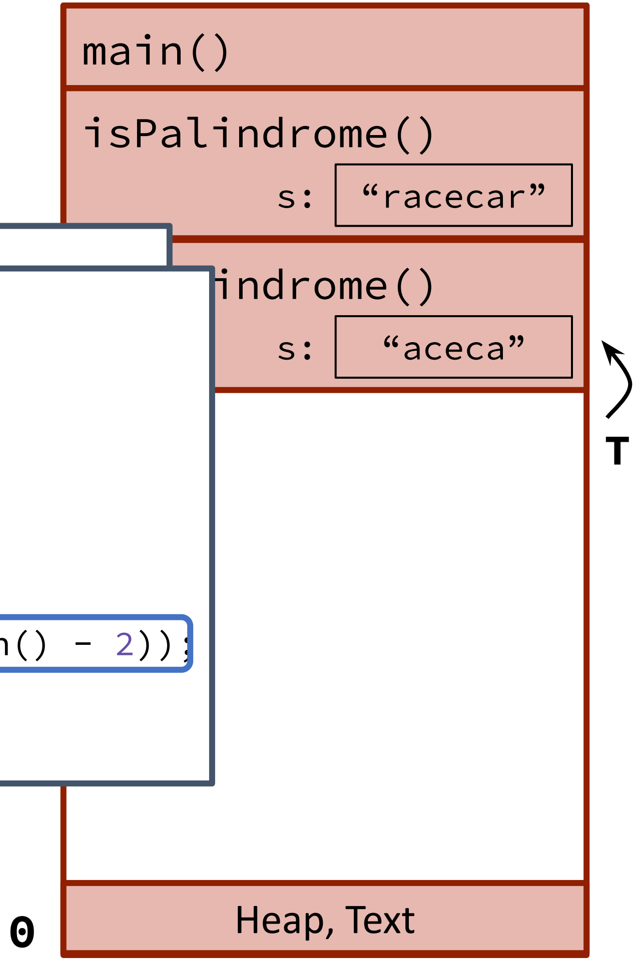
```
bool isPalindrome(string s) {
    if (s.length() <= 1) {
        return true;
    } else {
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        return isPalindrome(s.substr(1, s.length() - 2));
    }
}
```

true



isPalindrome() in action

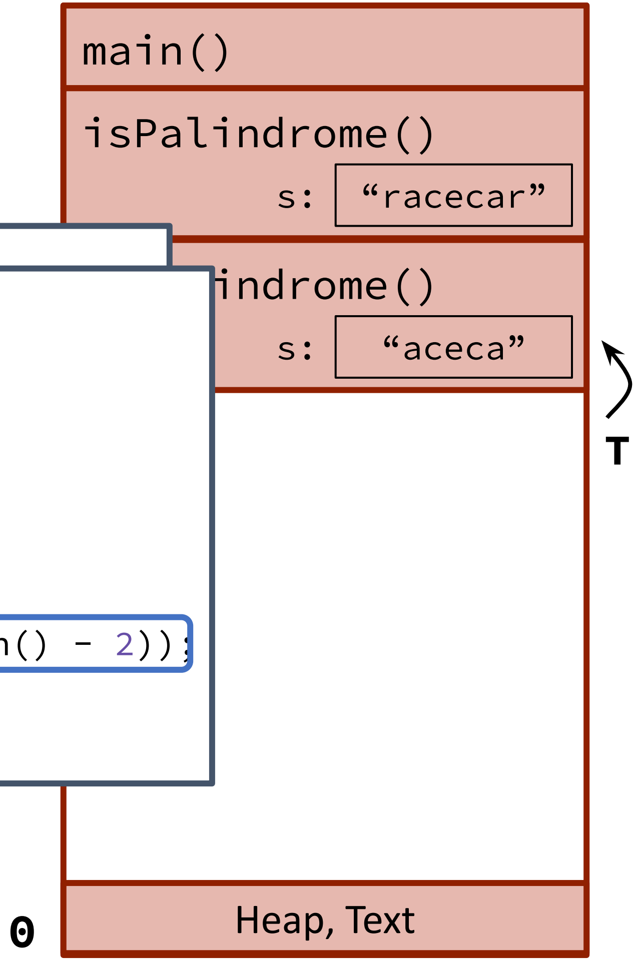
```
bool isPalindrome(string s) {
    if (s.length() <= 1) {
        return true;
    } else {
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        return isPalindrome(s.substr(1, s.length() - 2));
    }
}
```



isPalindrome() in action

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

true



isPalindrome() in action

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

true

main()

isPalindrome()

s: “racecar”

indrome()

s: “aceca”

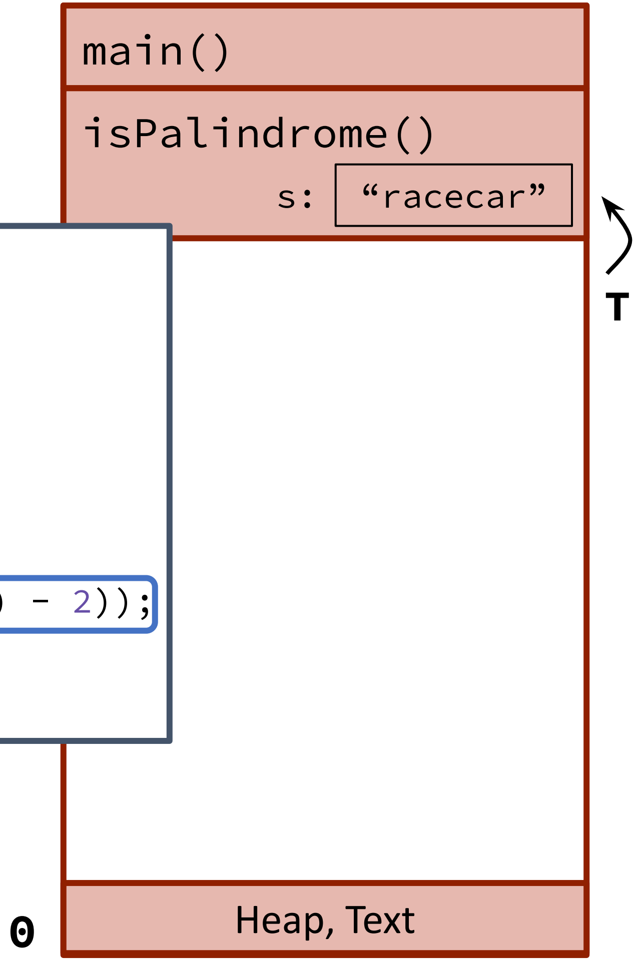
T

0

Heap, Text

isPalindrome() in action

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



isPalindrome() in action

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

true

main()

isPalindrome()

s: "racecar"

T

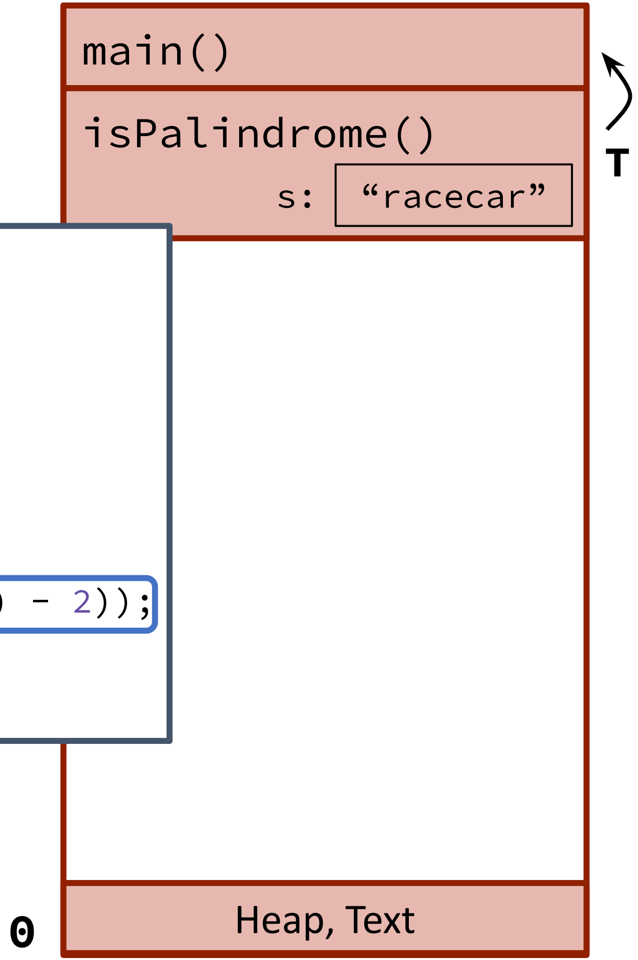
0

Heap, Text

isPalindrome() in action

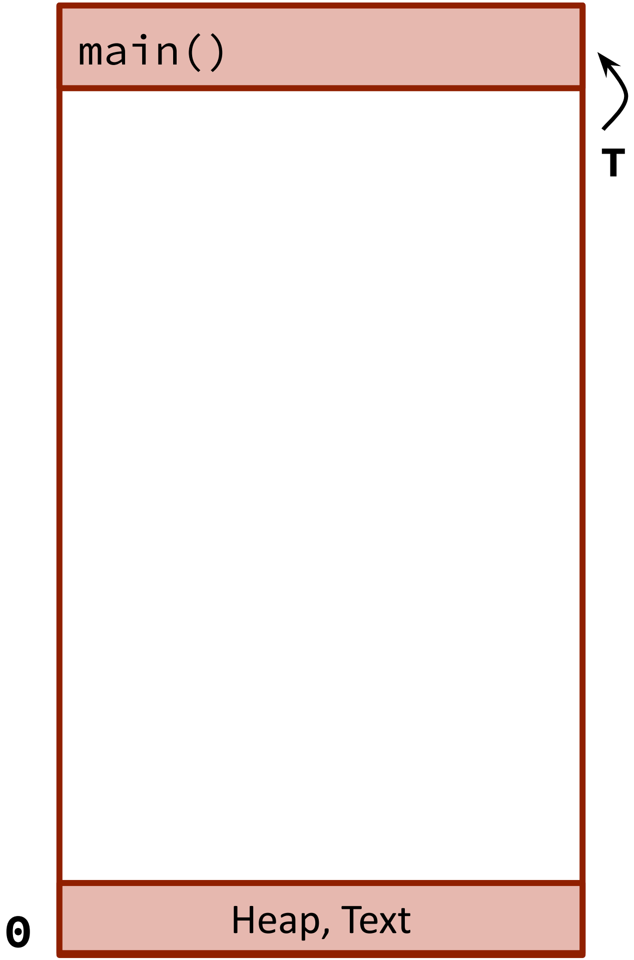
```
bool isPalindrome(string s) {
    if (s.length() <= 1) {
        return true;
    } else {
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        return isPalindrome(s.substr(1, s.length() - 2));
    }
}
```

true



isPalindrome() in action

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

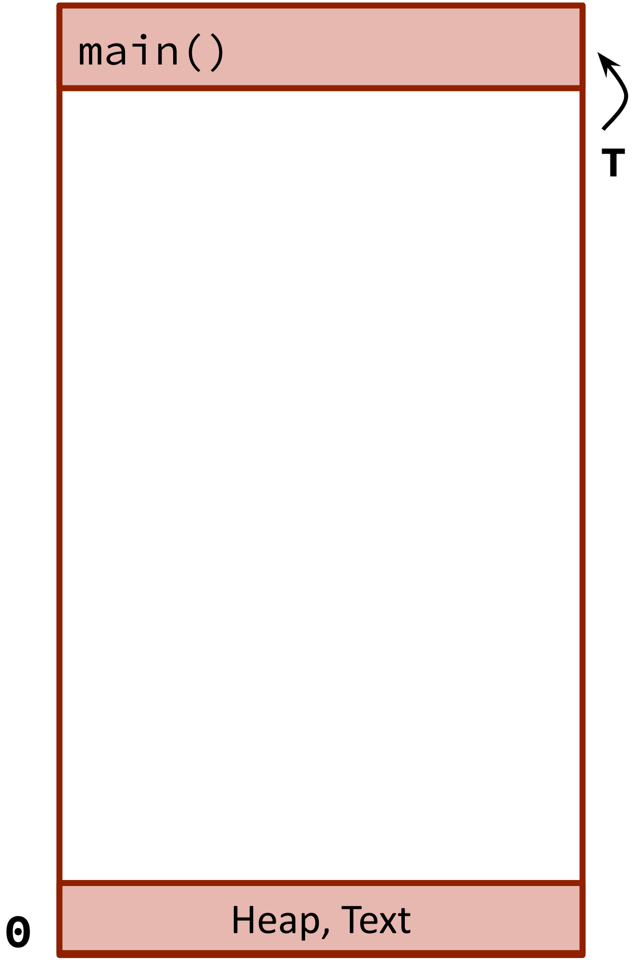


isPalindrome() in action

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

Console:

```
true
```

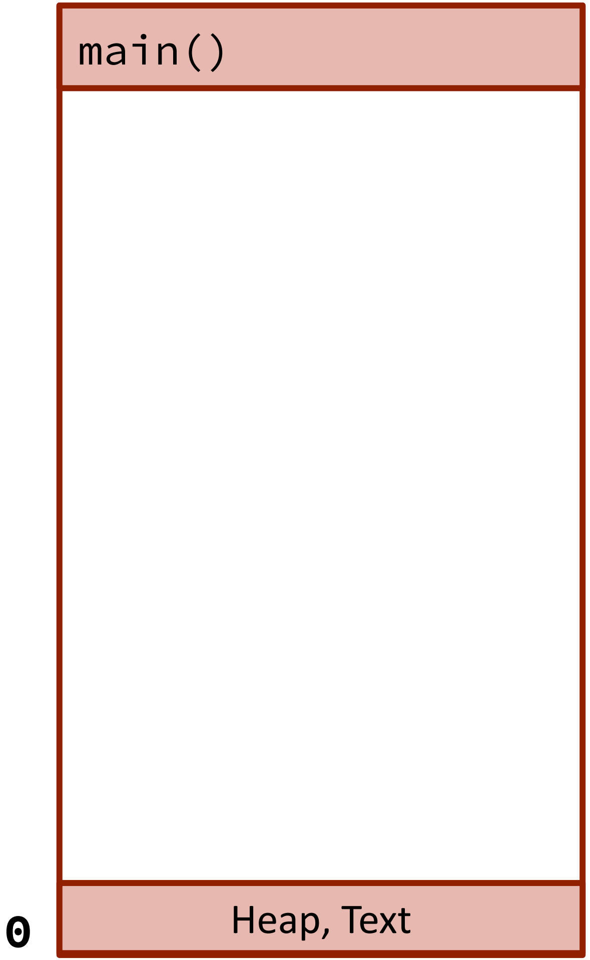


isPalindrome() in action

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

Console:

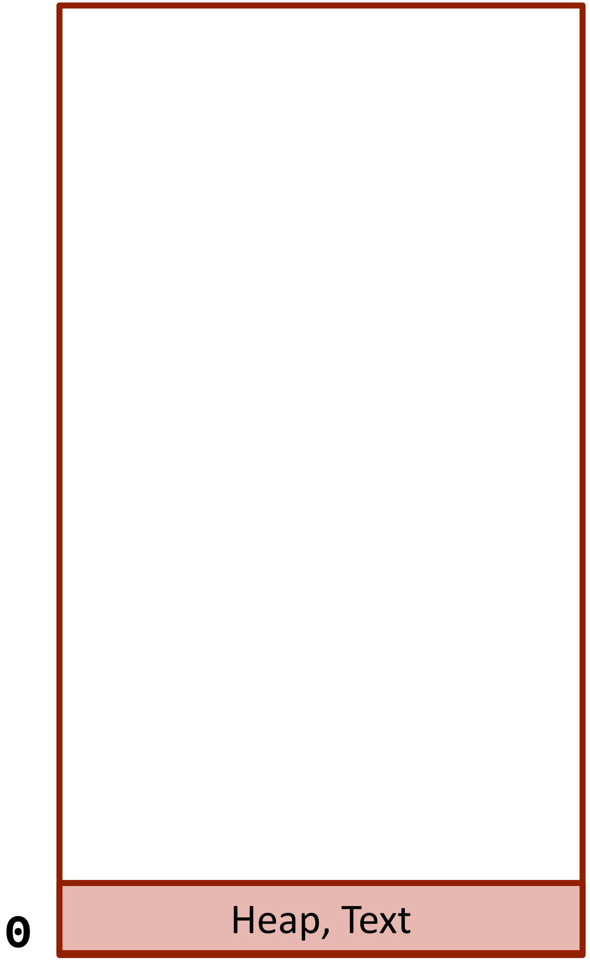
```
true
```



isPalindrome() in action

Console:

```
true
```

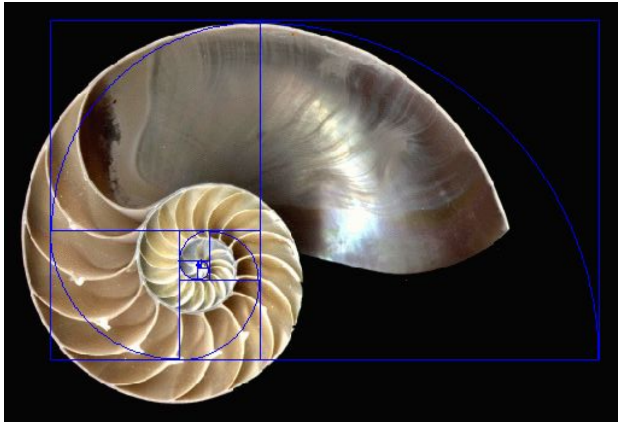


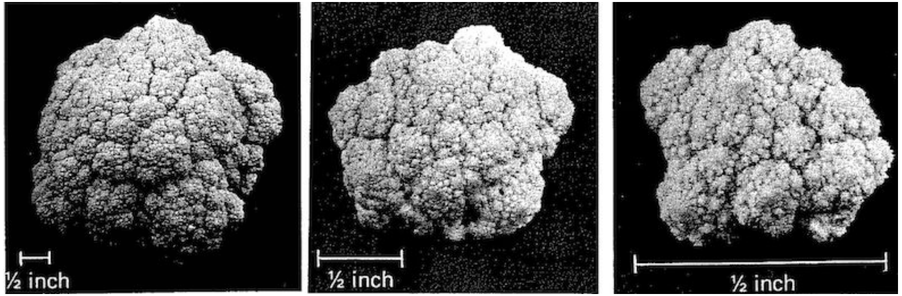
Visual Representations of Recursion

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
- Shows up in many real-world objects and phenomena

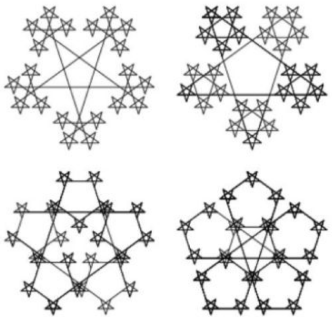
Recursion in nature

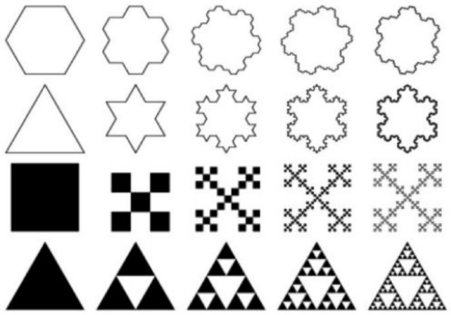




Fractal

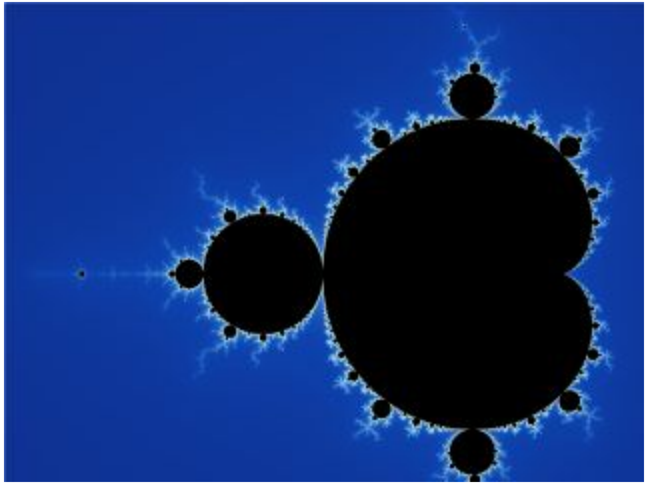
- Any repeated, graphical pattern
- Composed of repeated instances of the same shape or pattern, arranged in a structured way





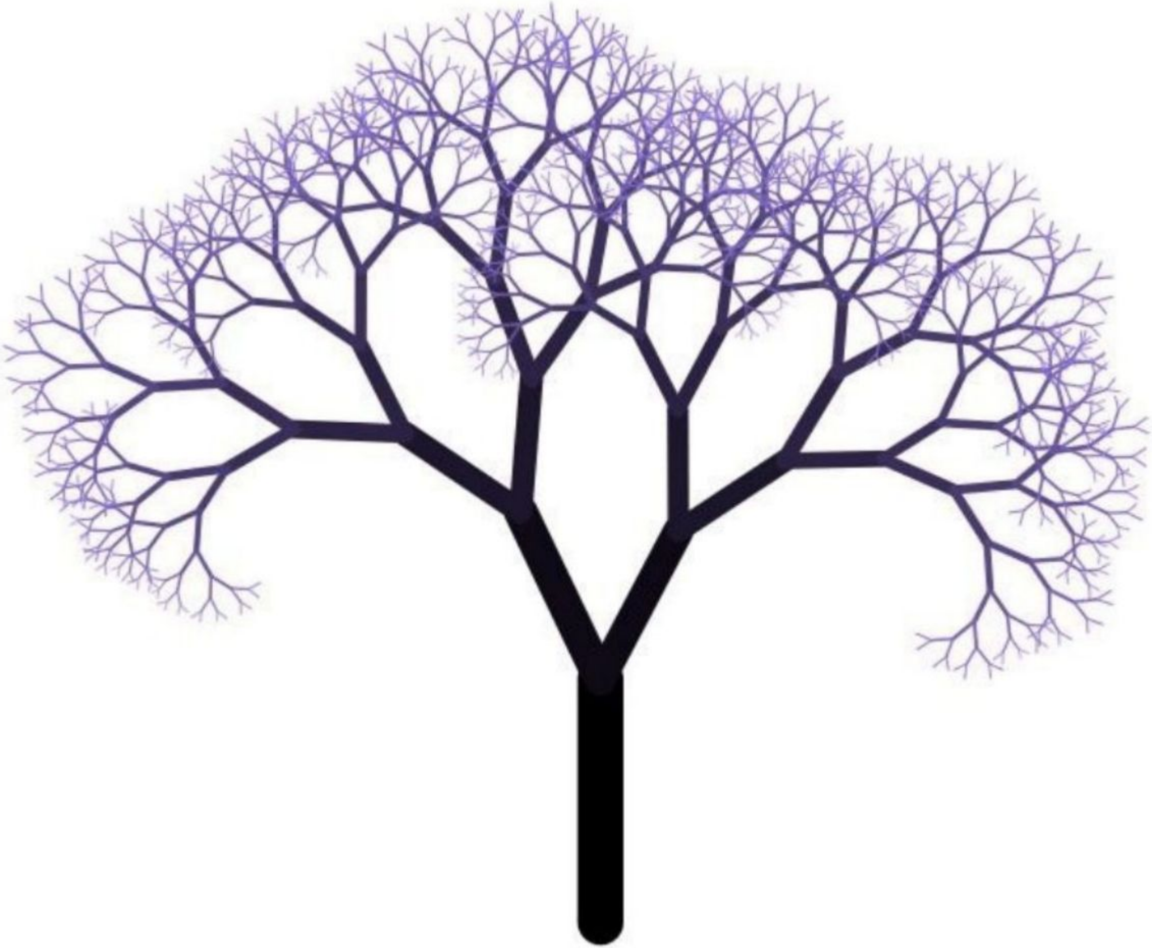
Fractal

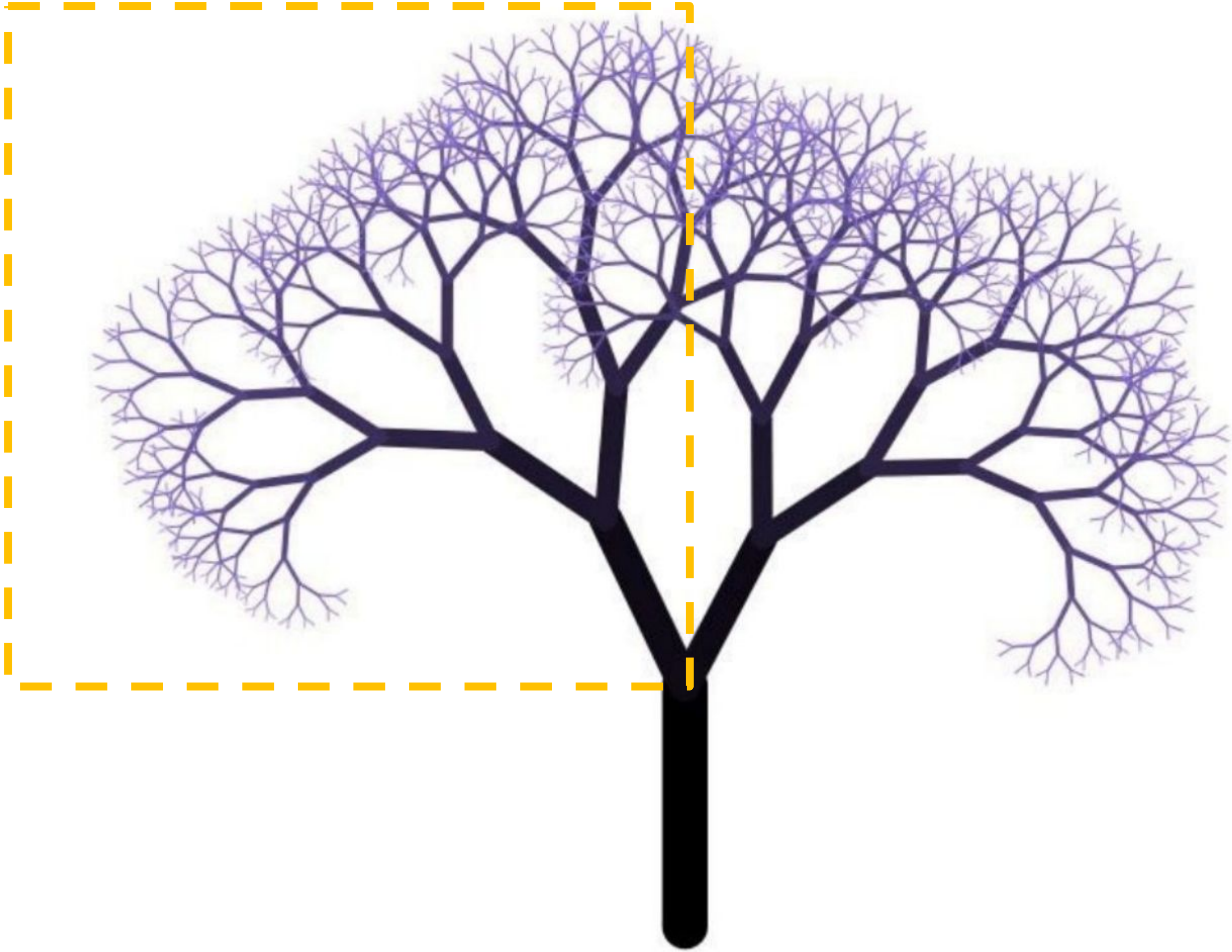
- Any repeated, graphical pattern
- Composed of repeated instances of the same shape or pattern, arranged in a structured way

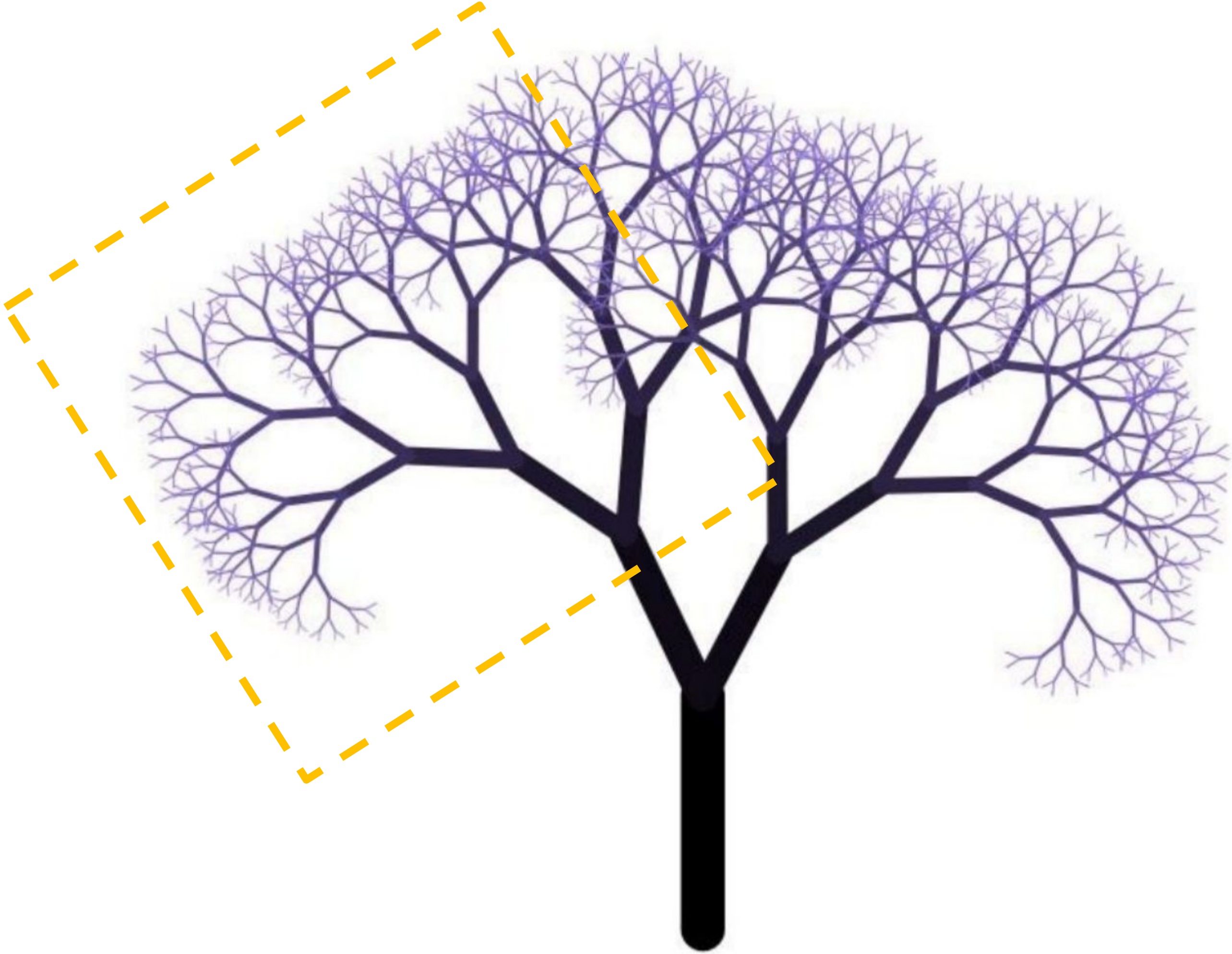


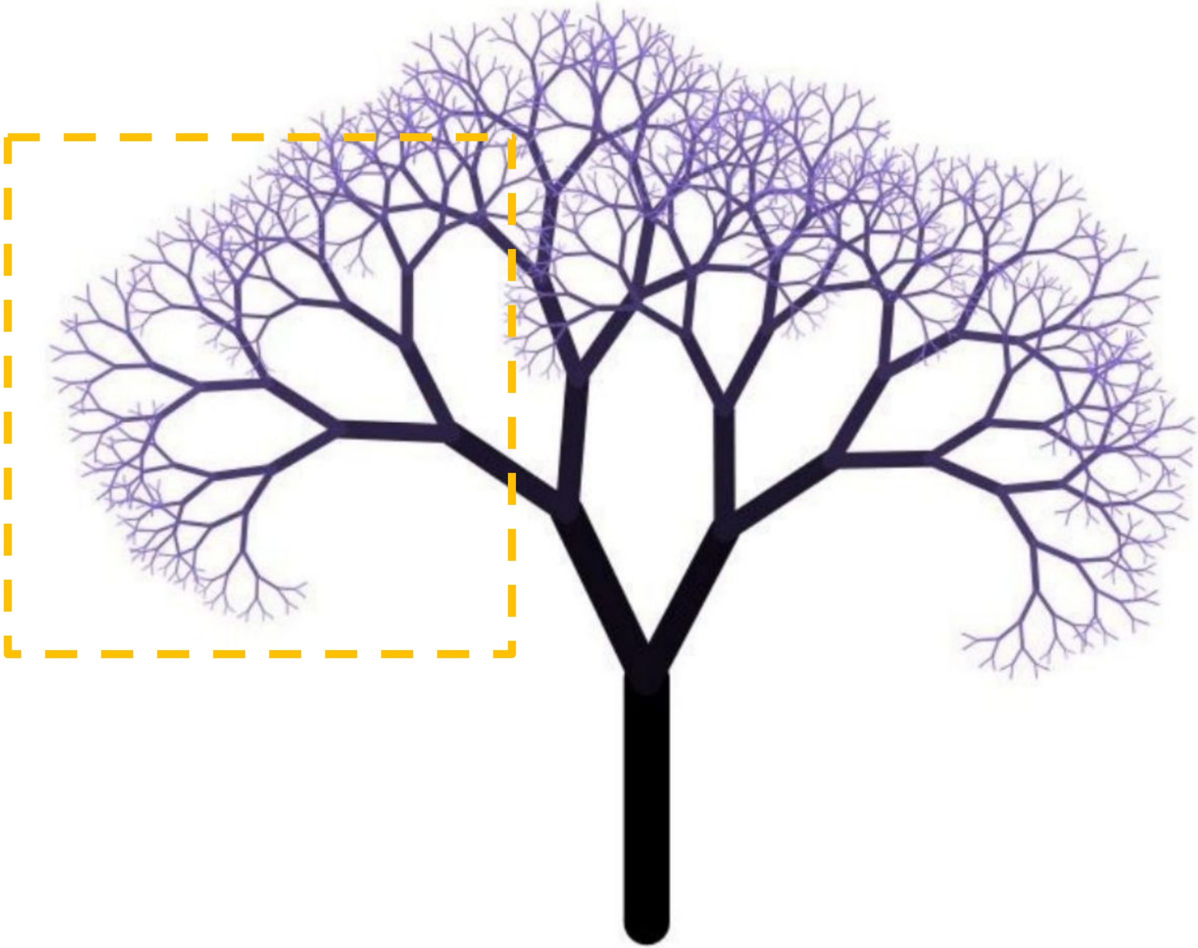
The set is defined in the complex plane as complex numbers c for which the function f_c(z) = z^2 + c does not diverge to infinity when iterated starting at z=0

Understanding Fractal Structure

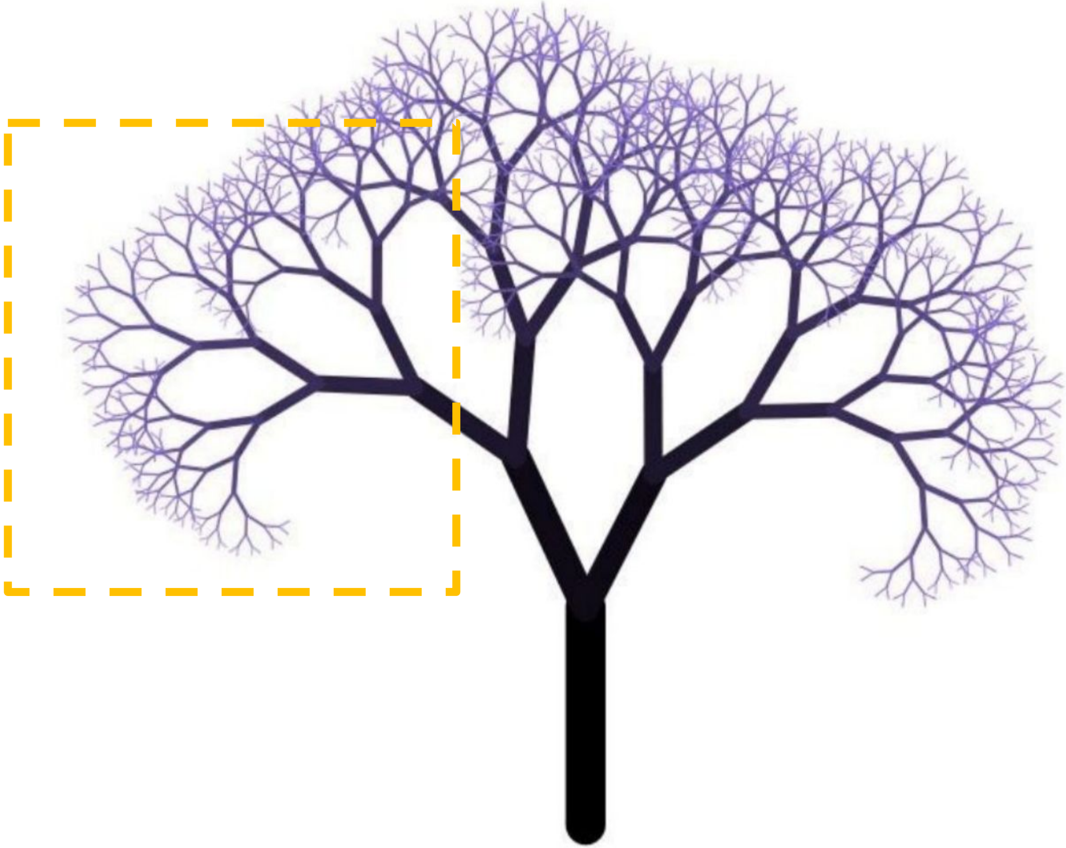






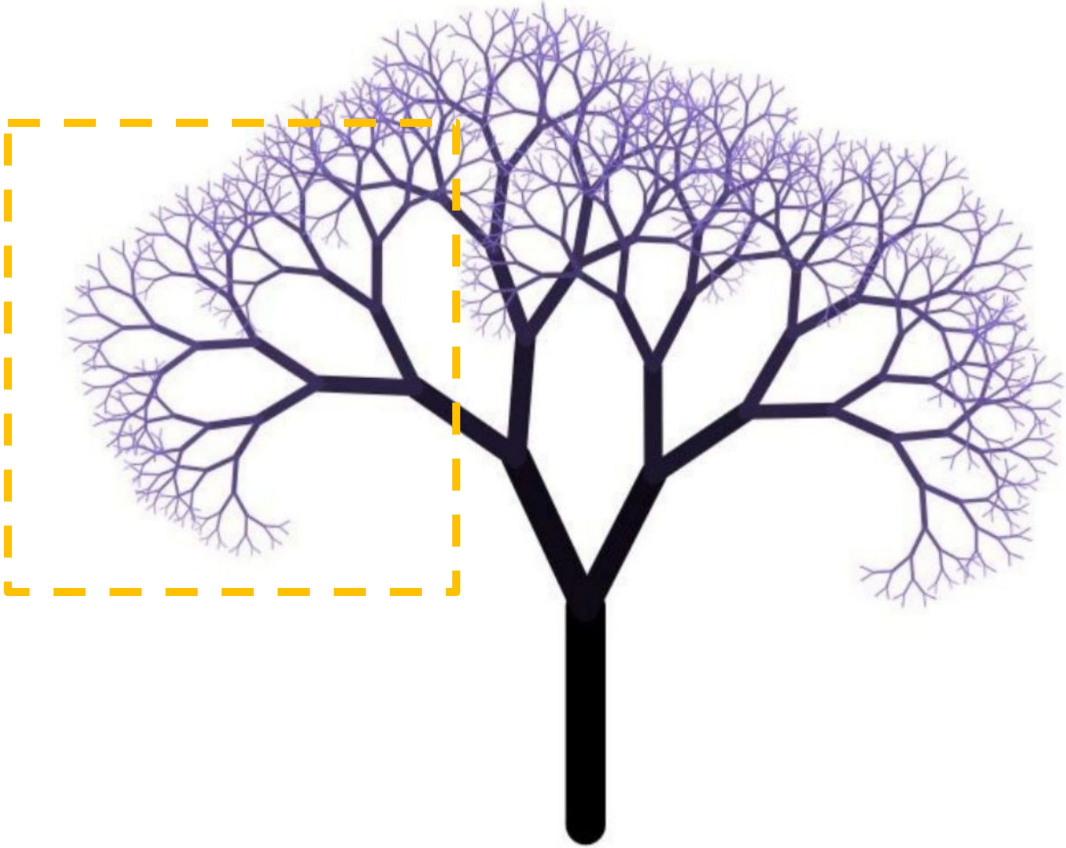


What differentiates the smaller tree from the bigger one?



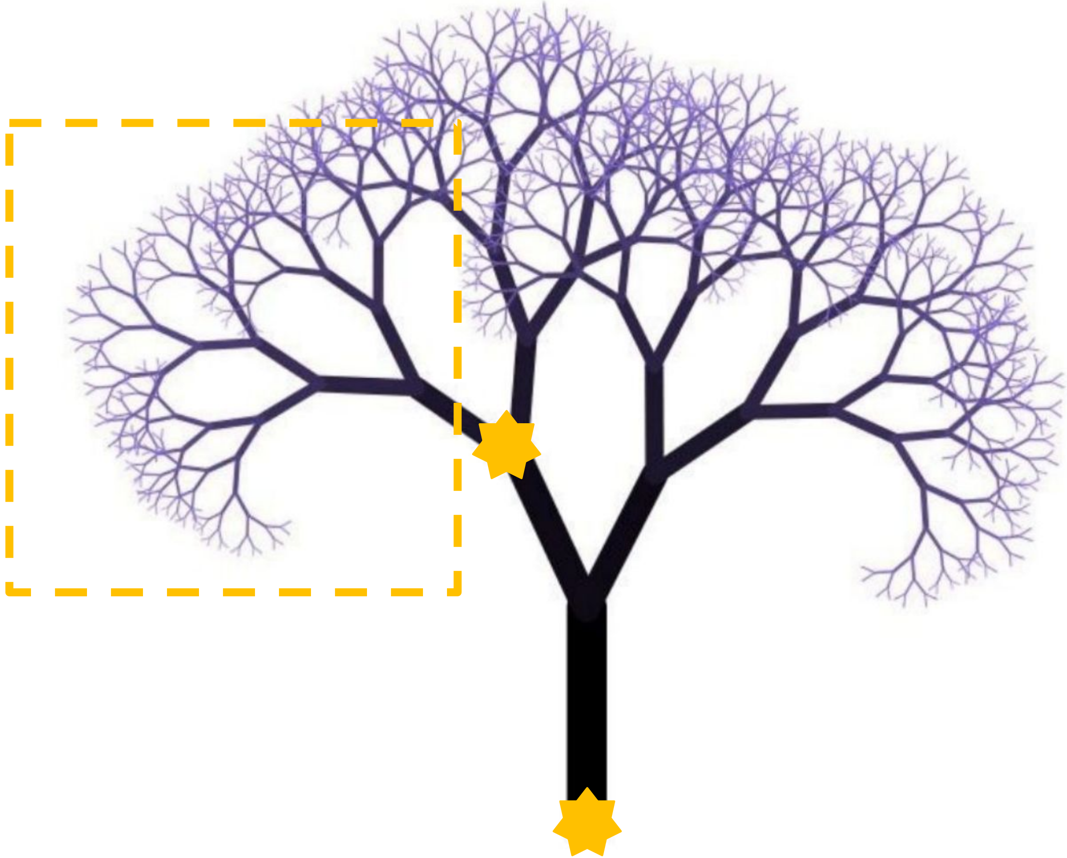
What differentiates the smaller tree from the bigger one?

1. It's at a different **position**



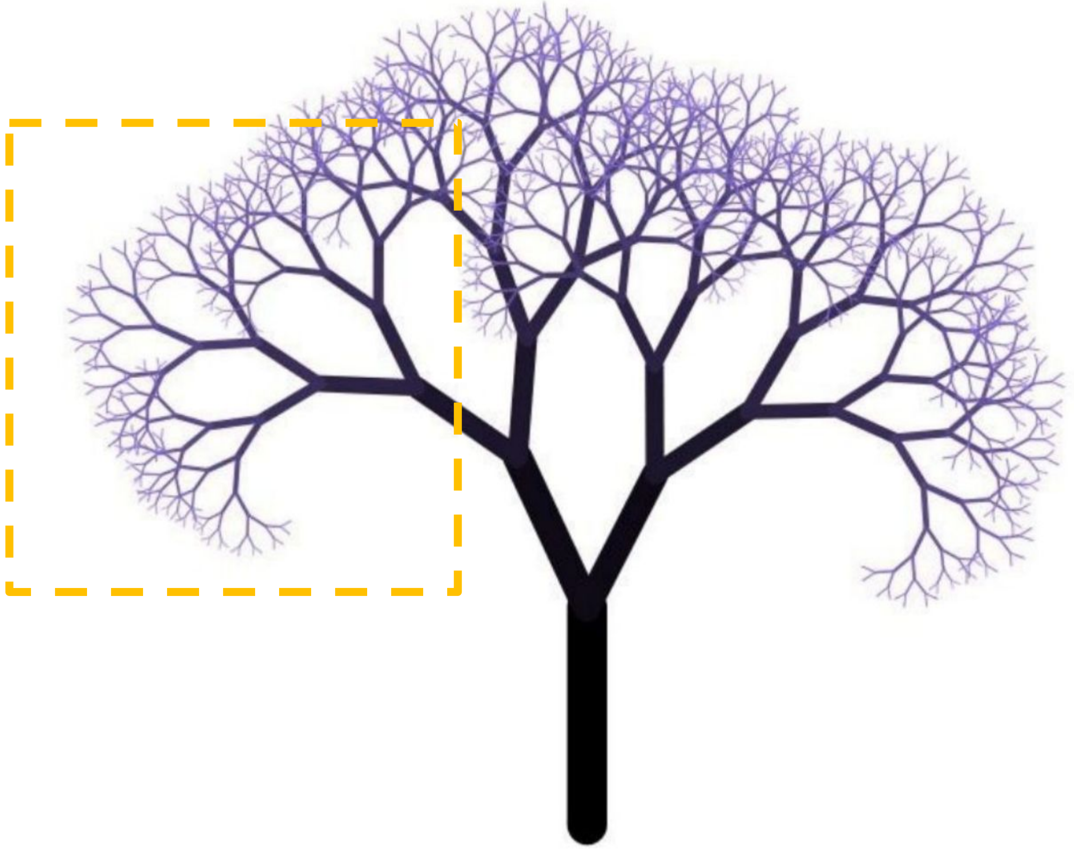
What differentiates the smaller tree from the bigger one?

1. It's at a different **position**



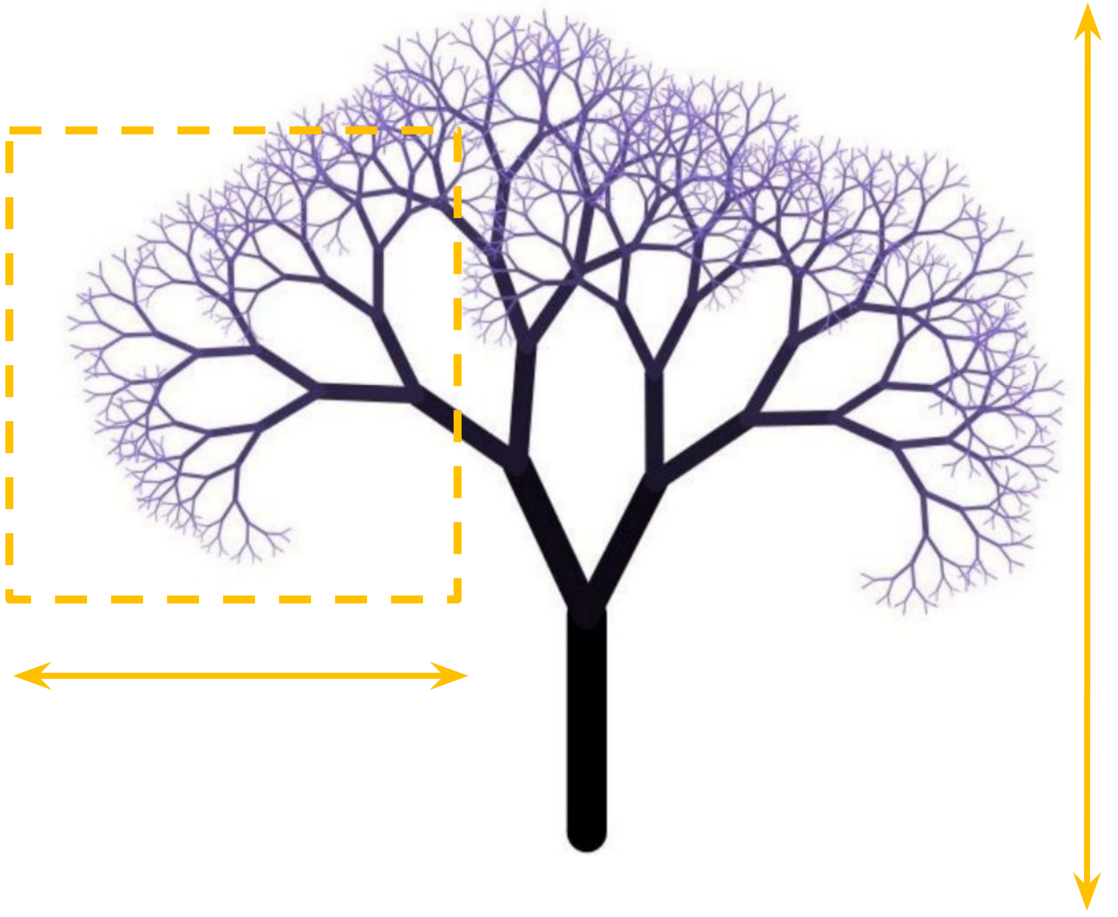
What differentiates the smaller tree from the bigger one?

1. It's at a different **position**
2. It has a different size



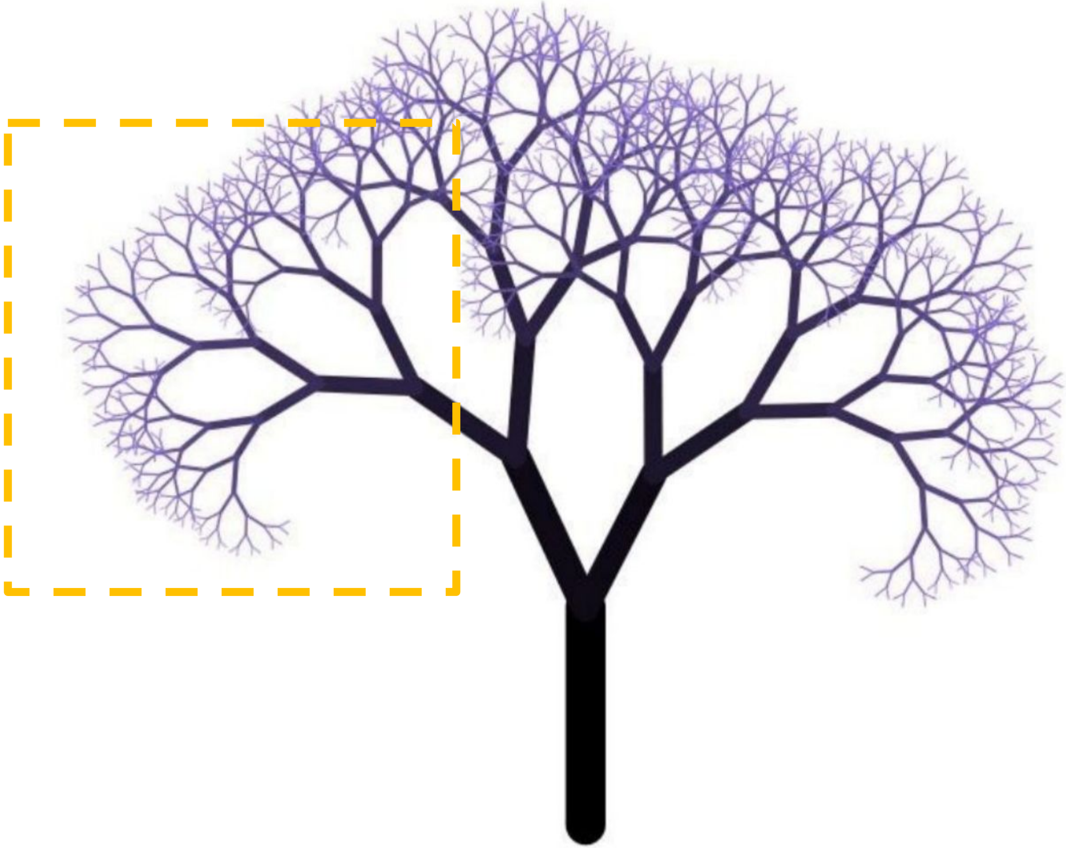
What differentiates the smaller tree from the bigger one?

1. It’s at a different **position**
2. It has a different size



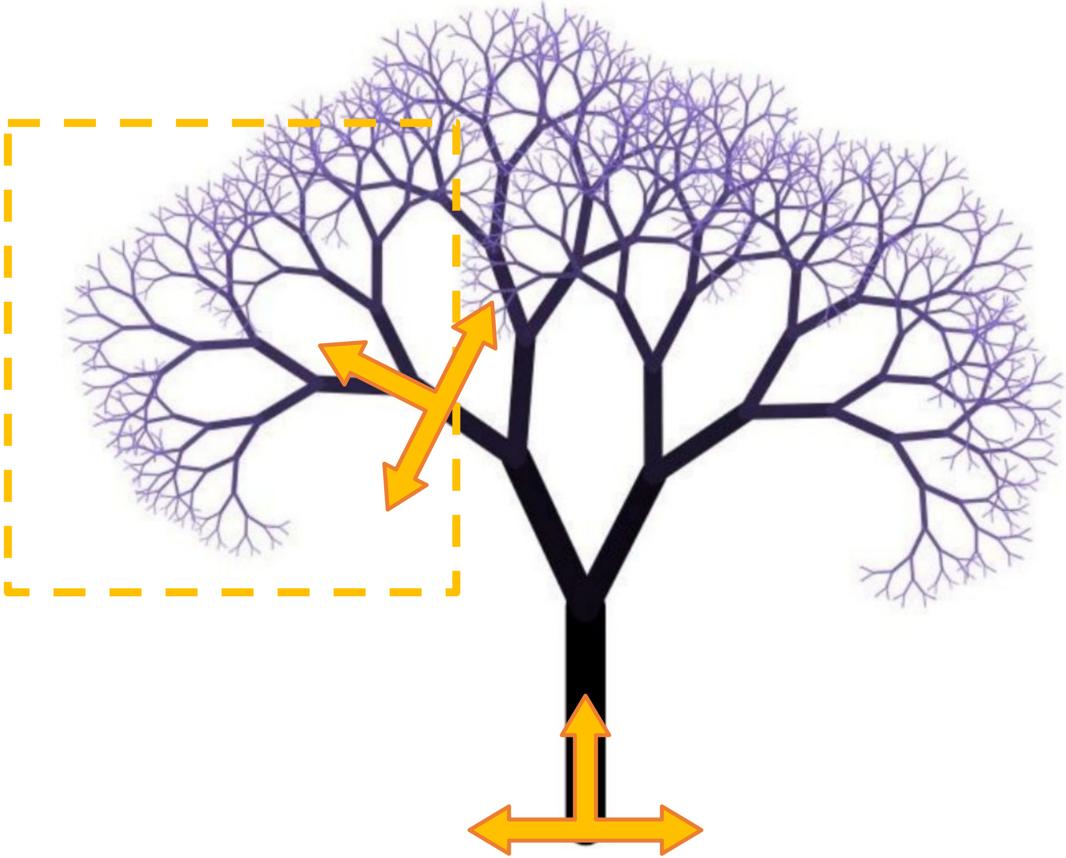
What differentiates the smaller tree from the bigger one?

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



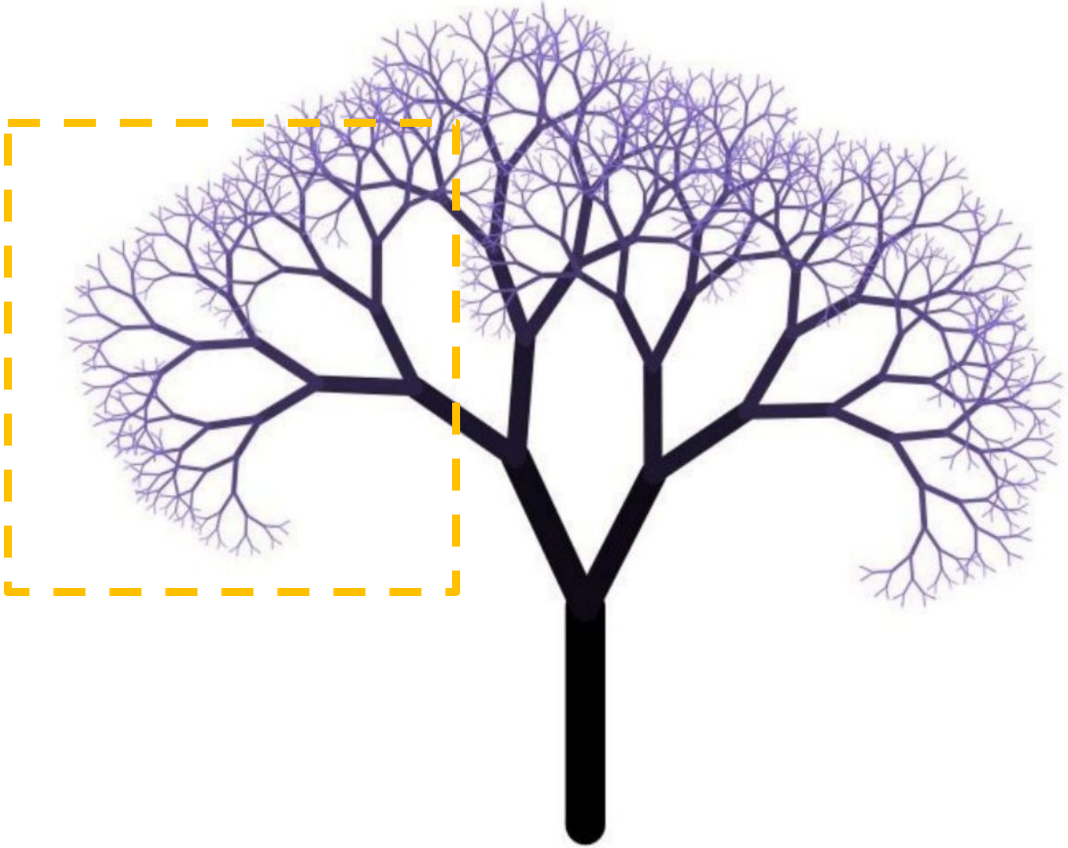
What differentiates the smaller tree from the bigger one?

1. It’s at a different **position**
2. It has a different **size**
3. It has a different orientation



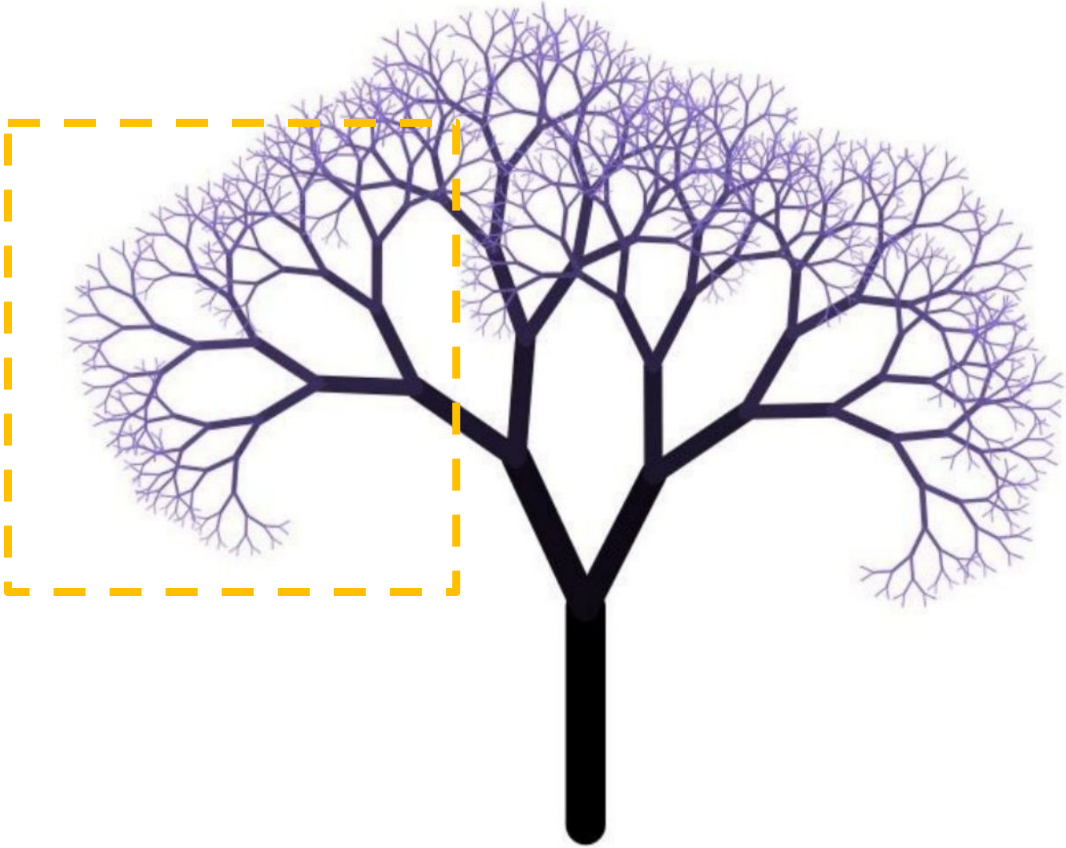
What differentiates the smaller tree from the bigger one?

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.

Order-0 tree

An order-0 tree is nothing.

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

Order-1 tree

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

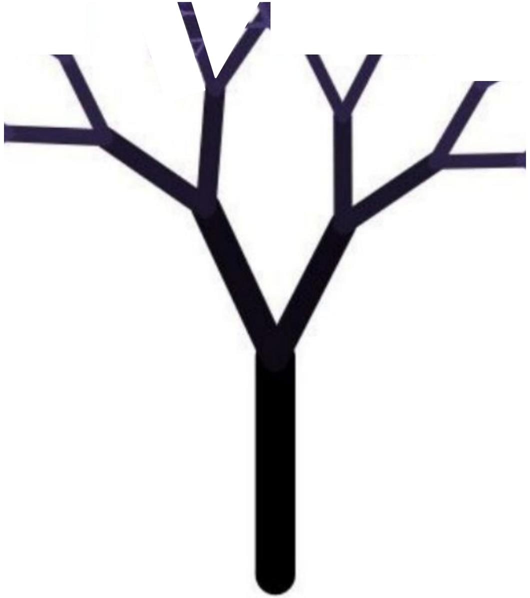
Order-2 tree

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

Order-3 tree

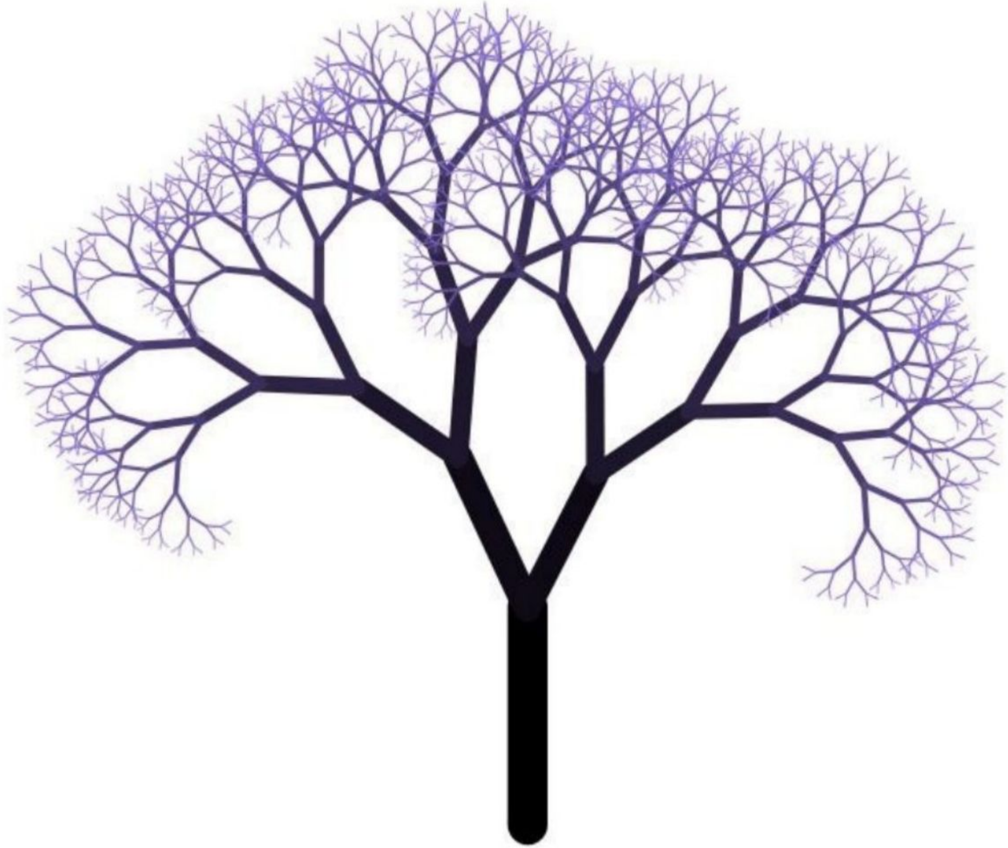
Fractals and self-similar structures are often defined in terms of some parameter called the **order**, which indicates the complexity of the overall structure.

Order-4 tree



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

Order-11 tree



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

Order-3 tree

An order-0 tree is nothing.

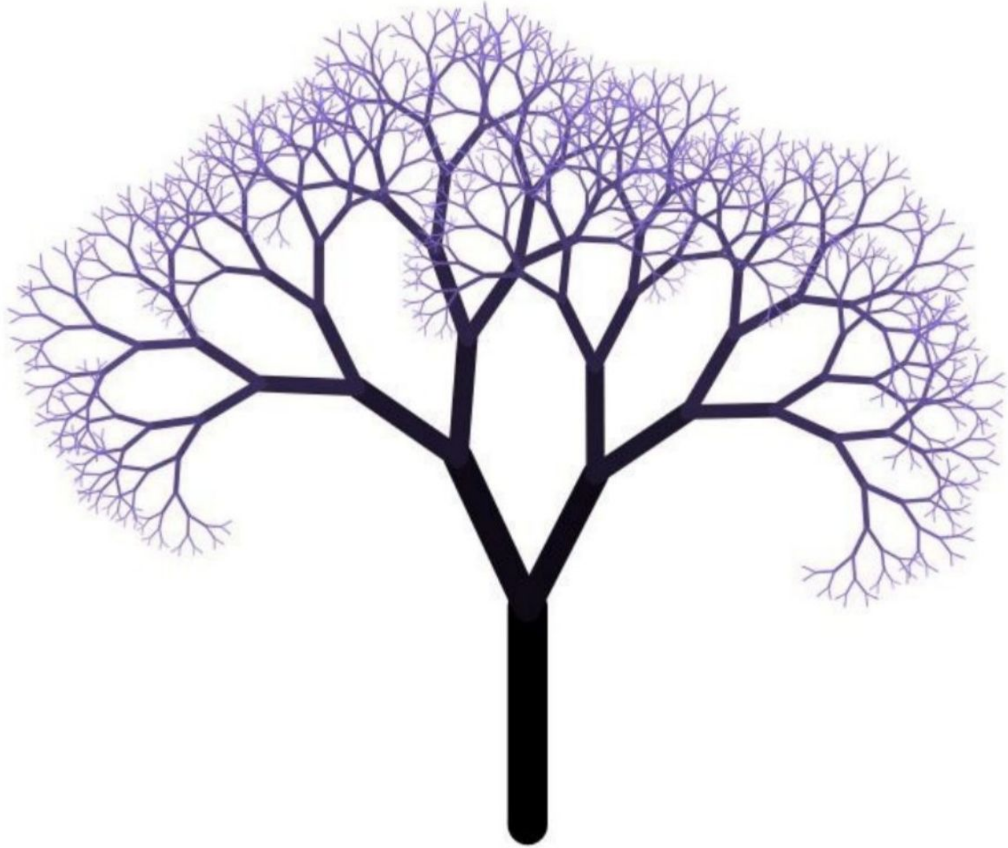
An order-n tree is a line with two smaller order-(n-1) trees starting at the end of the line.

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

Order-11 tree

We drew this tree recursively!

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



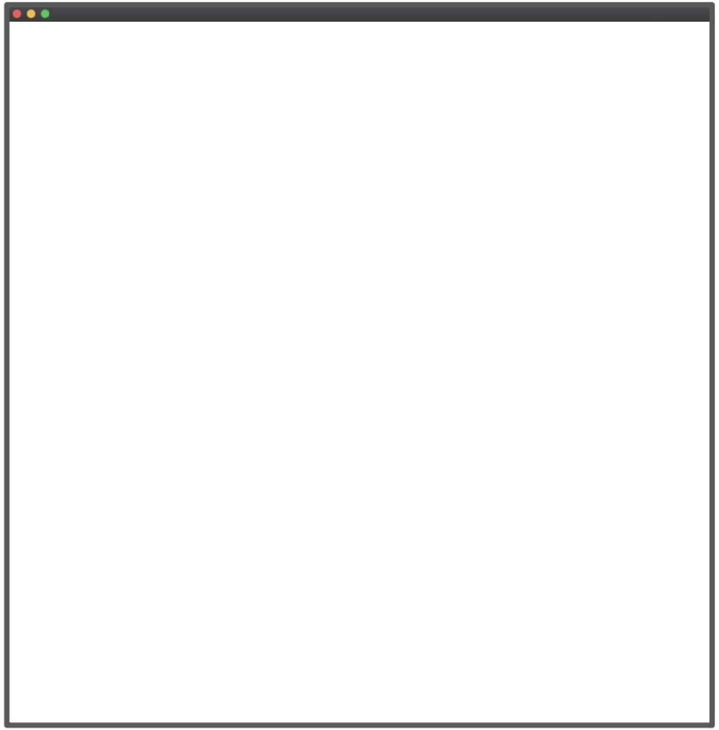
Aside on Graphics

Graphics in CS106B

- Creating graphical programs is not one of our main focuses in this class, but we need to know how to work with graphical programs to code up some fractals of our own
- Stanford C++ libraries provide extensive capabilities to create custom graphical programs
  - Full documentation can be found here
- We will abstract away almost all of the complexity for you via provided helper functions
  - Main components you need to know: `GWindow` and `GPoint`

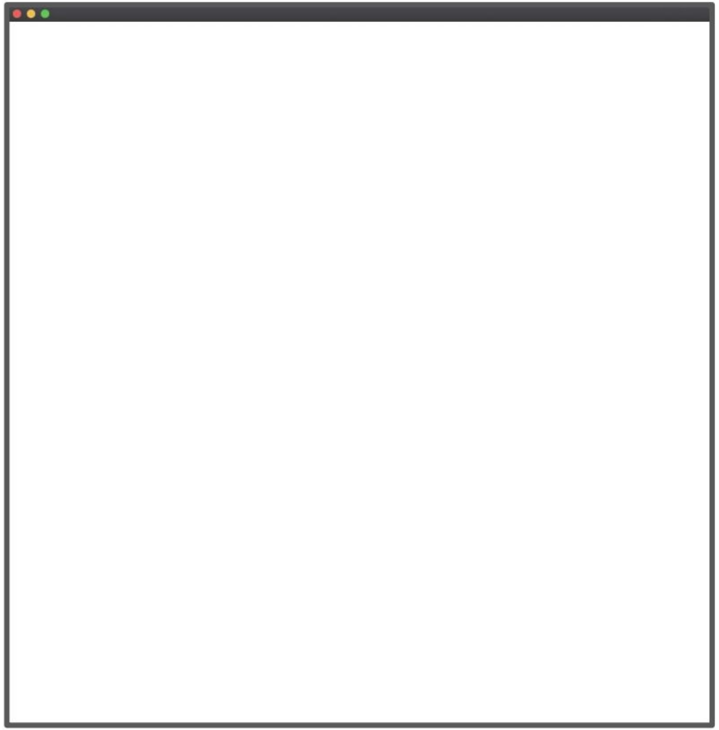
GWindow

- An abstraction for the graphical window upon which we will do all of our drawing.



GWindow

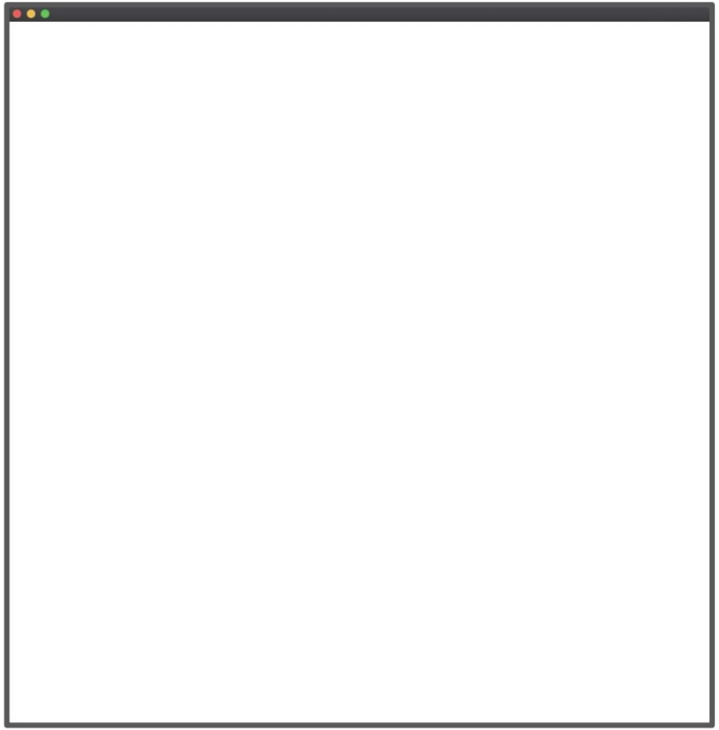
- An abstraction for the graphical window upon which we will do all of our drawing.
- The window defines a coordinate system of x-y values



`GWindow`

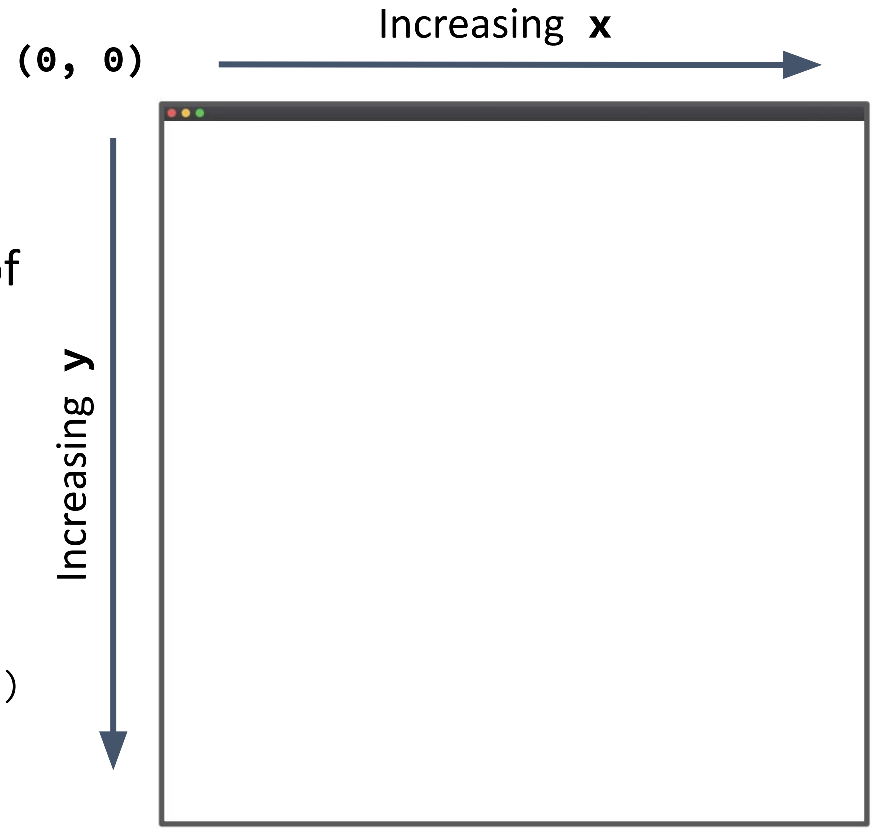
- An abstraction for the graphical window upon which we will do all of our drawing.
- The window defines a coordinate system of x-y values
  - Top left corner is `(0, 0)`

`(0, 0)`



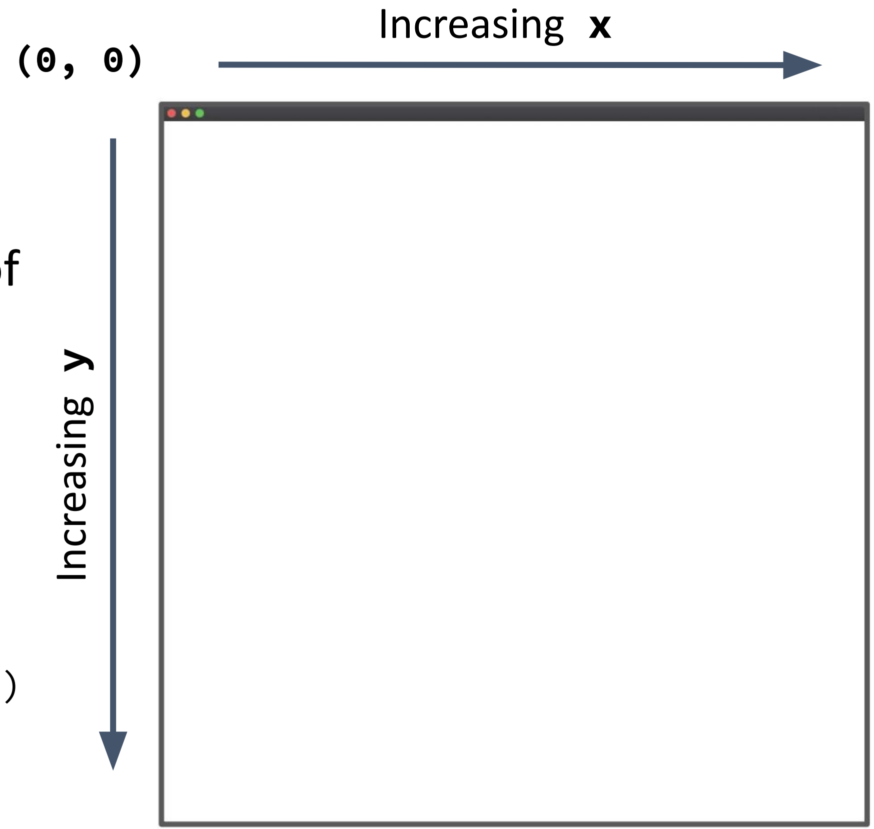
GWindow

- 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
  - Top left corner is `(0, 0)`
  - Bottom right corner is `(windowWidth-1, windowHeight-1)`



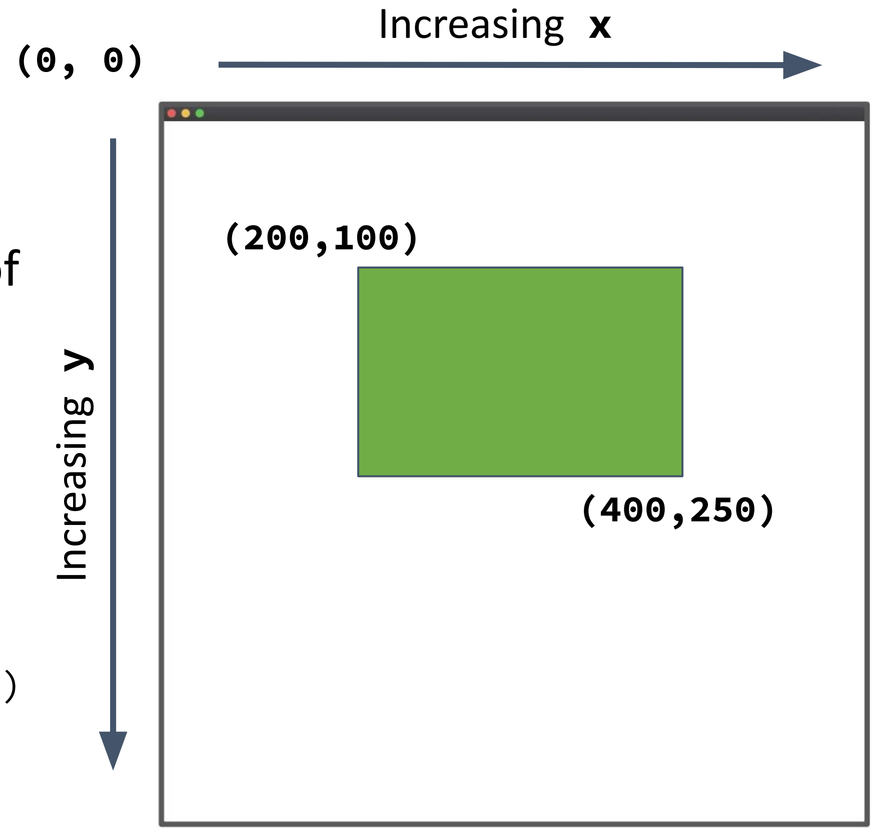
GWindow

- 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
  - Top left corner is `(0, 0)`
  - Bottom right corner is `(windowWidth-1, windowHeight-1)`
- All lines and shapes drawn on the window are defined by their `(x,y)` coordinates



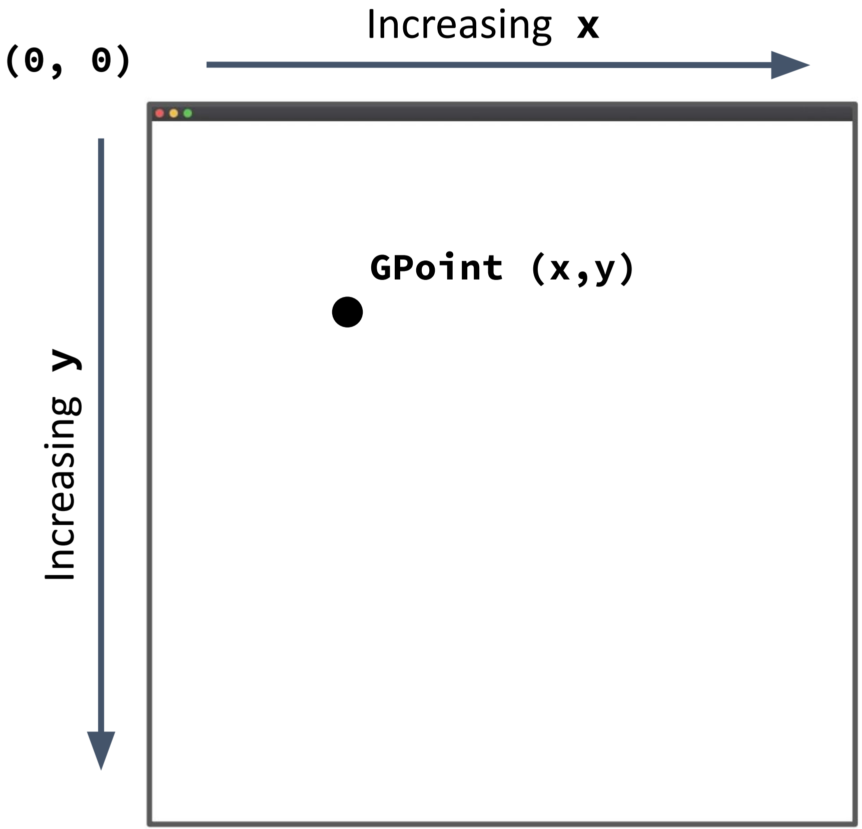
GWindow

- 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
  - Top left corner is `(0, 0)`
  - Bottom right corner is `(windowWidth-1, windowHeight-1)`
- All lines and shapes drawn on the window are defined by their `(x,y)` coordinates



GPoint

- Handy way to bundle up the `(x,y)` coordinates for a specific point in the window

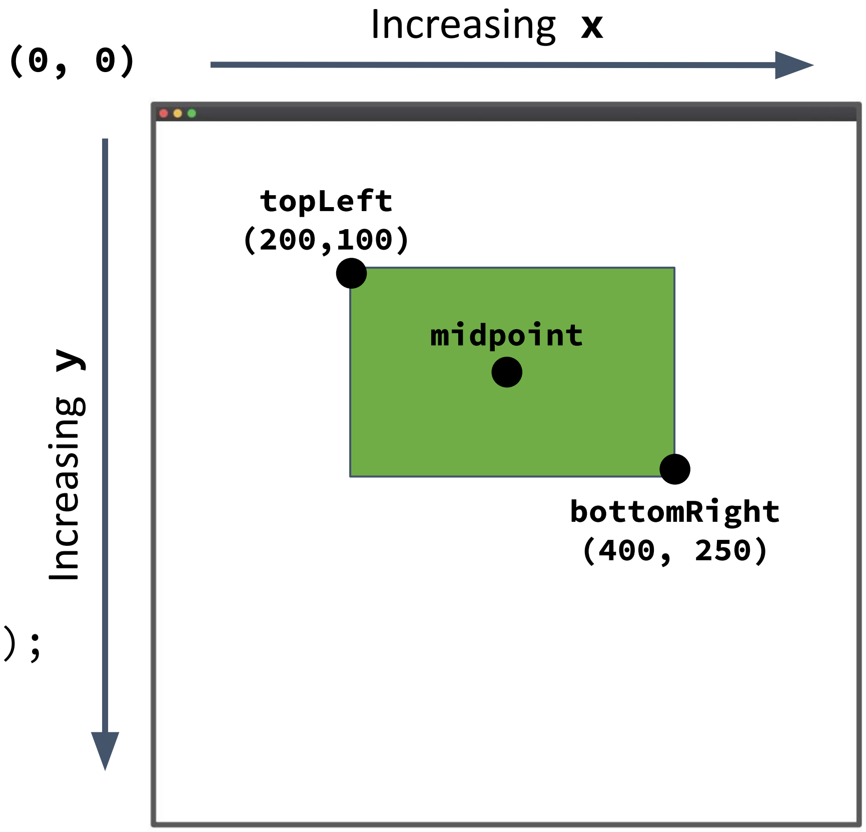


GPoint

- Handy way to bundle up the (x,y) coordinates for a specific point in the window

```
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

Cantor Set

- Set of lines where there is one main line, and below that there are two other lines: each ⅓ of the width of the original line, with one on the left and one on the right (with a ⅓ separation of whitespace between them)
- Below each of the other lines is an identical situation: two ⅓ lines.
- This repeats until the lines are no longer visible.

Order-0 Cantor Set

Order-1 Cantor Set

Order-2 Cantor Set

Order-3 Cantor Set

Order-6 Cantor Set

Order-6 Cantor Set

Another Cantor Set!

Another Cantor Set!

Approaching recursive problems

- Look for self-similarity.
- Try out an example.
  - 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
- 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?)

Drawing an order-n Cantor Set

Drawing an order-n Cantor Set

1. Draw a line from `left` to `right`

Drawing an order-n Cantor Set

1. Draw a line from `left` to `right`

Drawing an order-n Cantor Set

1. Draw a line from `left` to `right`

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

Drawing an order-n Cantor Set

1. Draw a line from `left` to `right`

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)

drawCantor()

```
drawCantor(GWindow &w, int level, GPoint left, GPoint right)
```

Base Case:

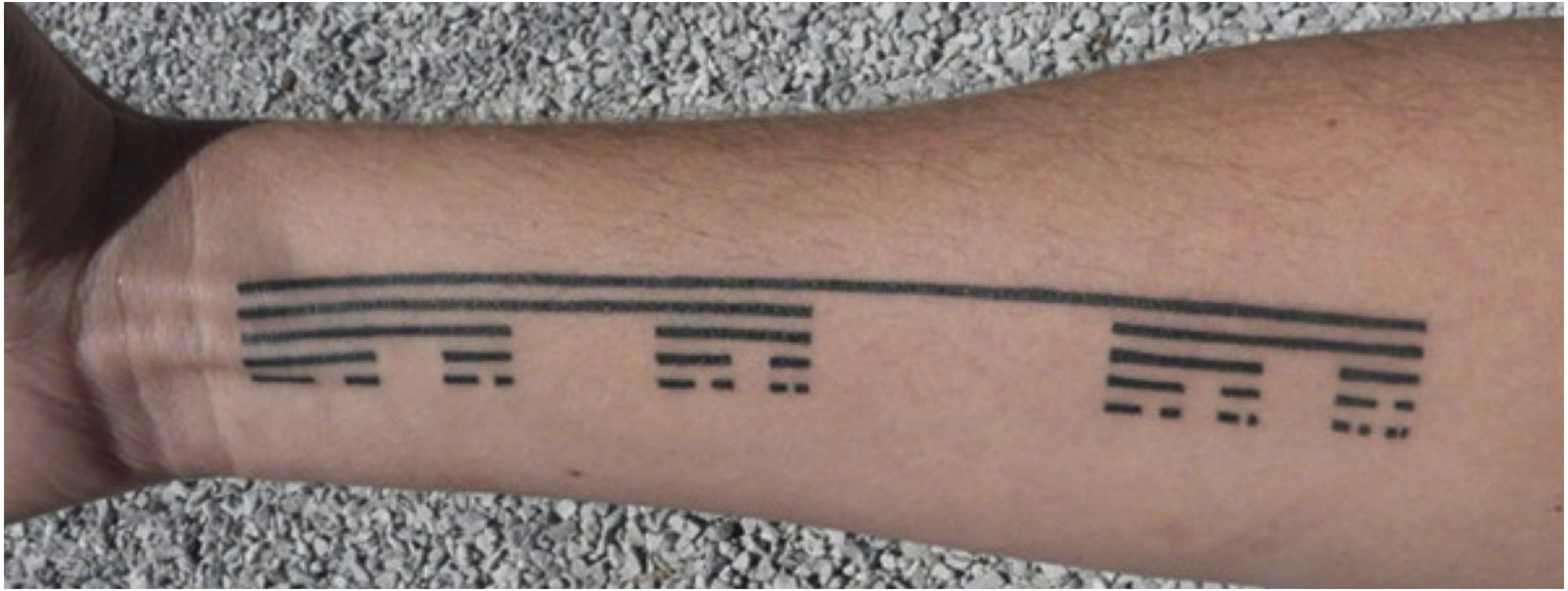
drawCantor(order is 0) ⟶ draw nothing

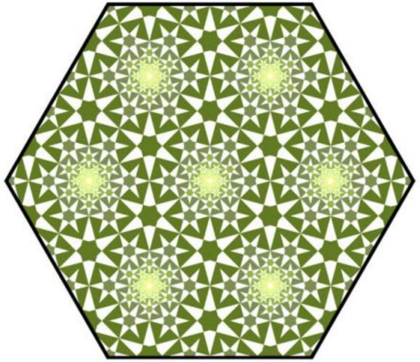
Recursive Case:

drawCantor(order) ⟶ draw a line on top, and then drawCantor(order-1) on left and right

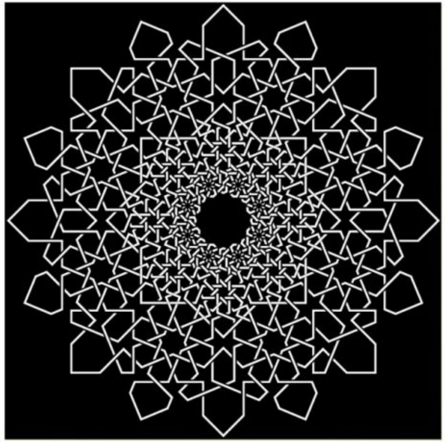
Let’s Code It Up!

Real-world applications

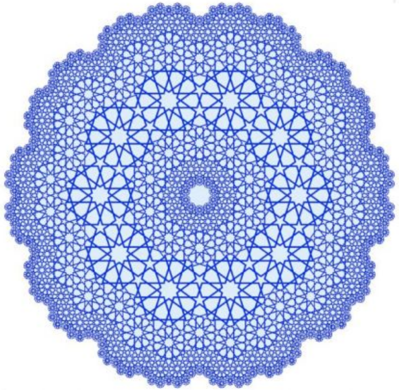




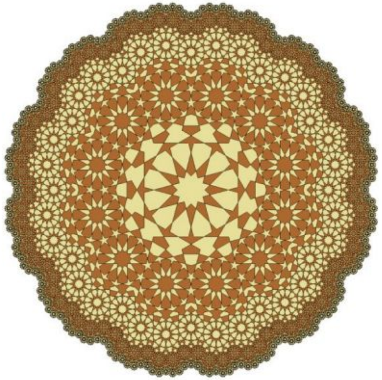
n=6, narrowed stars, radial repeating, mosaic w/color by level



n=8, scaled rosettes, radial inward, interlace w/equal band width

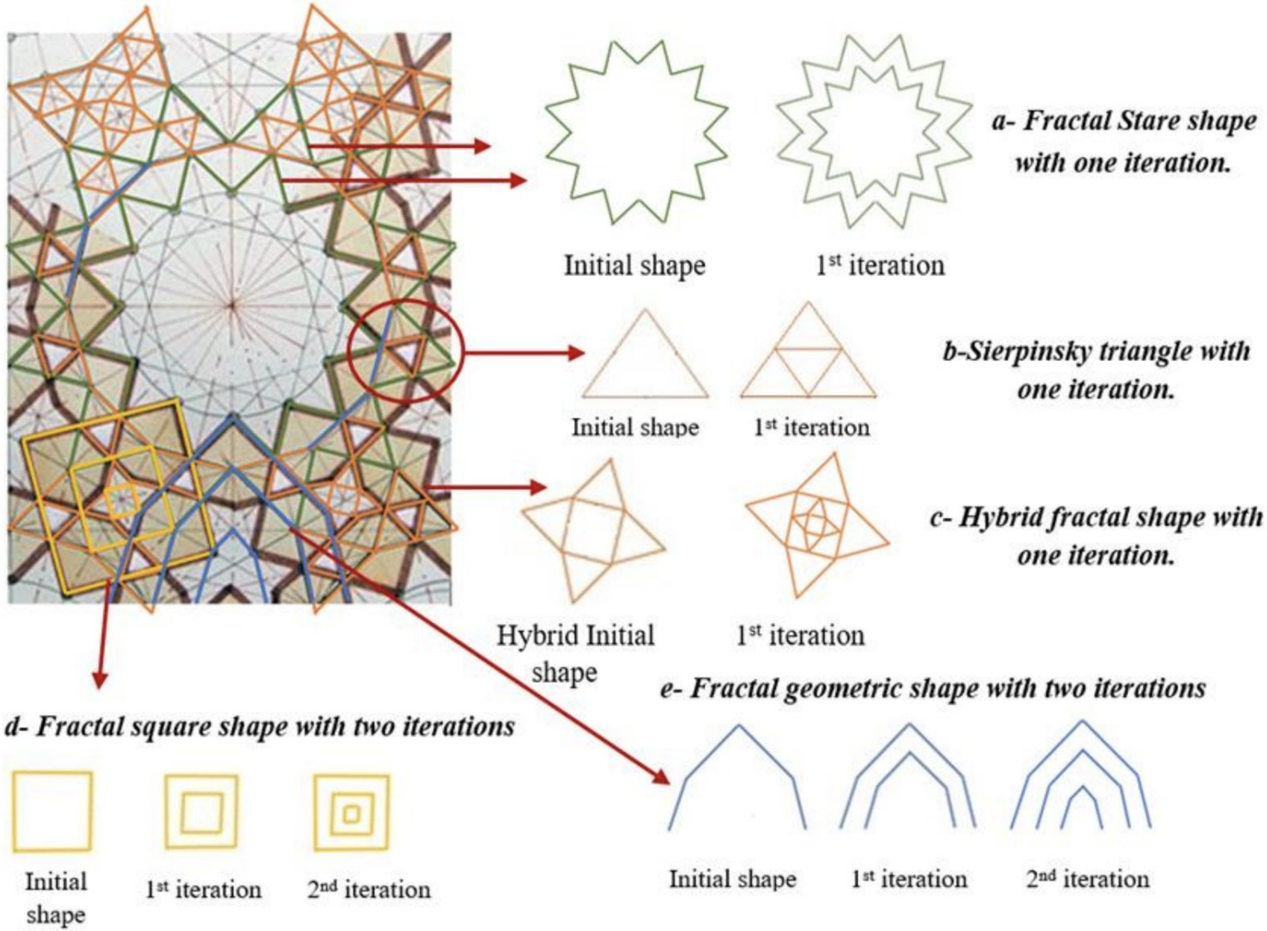


n=10, scaled rosettes, radial combined, outline w/variable band width



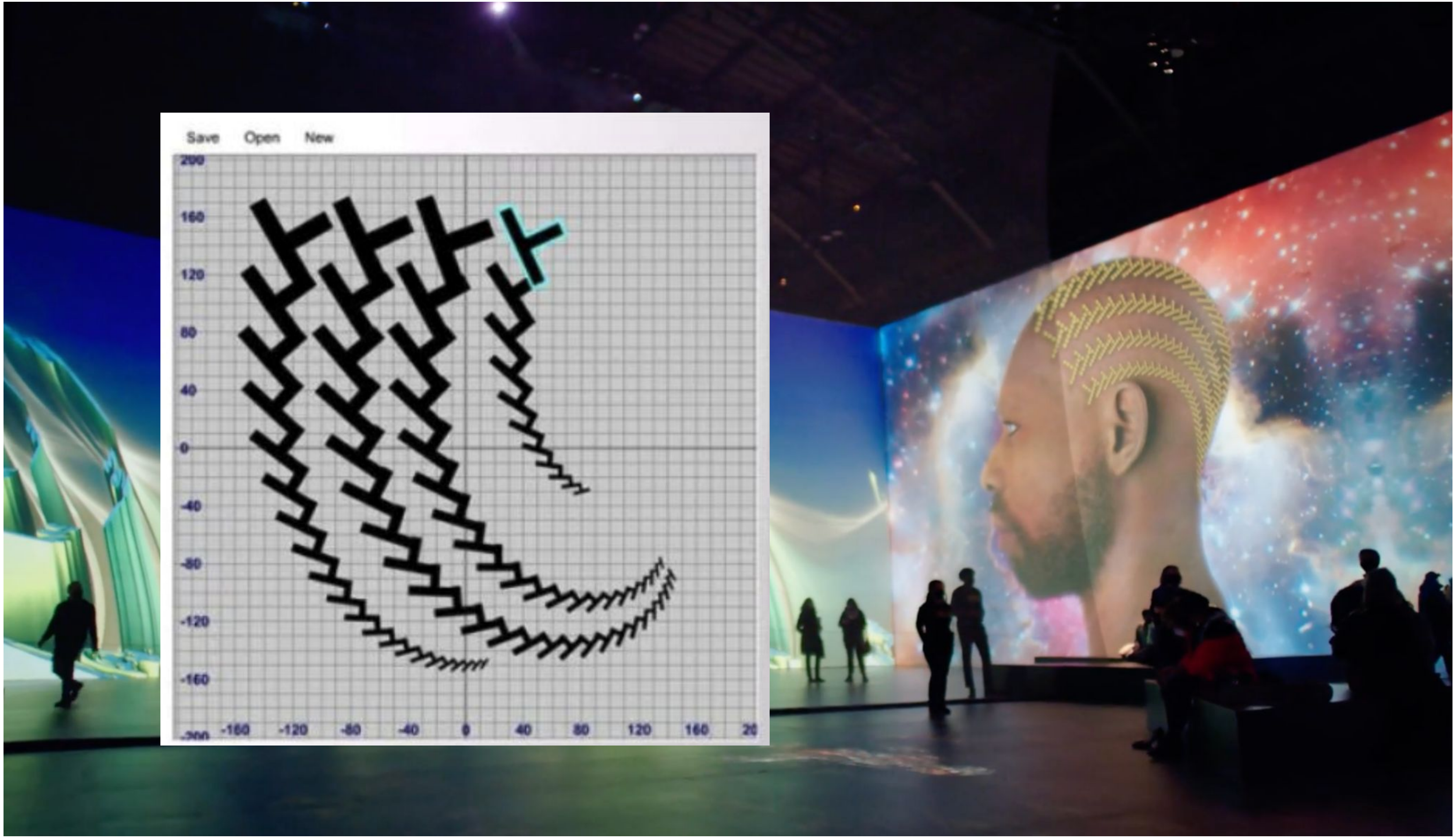
n=12, scaled extended rosettes, radial outward, mosaic 2-color

Source: [Fractal Islamic Geometric Patterns Based on Arrangements of { n /2} Stars]

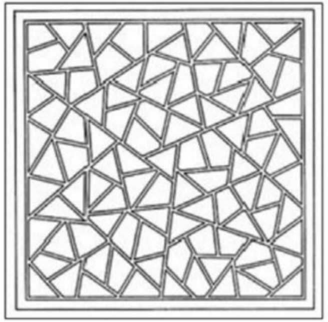


Source: [Fractal Shapes in Islamic Design]

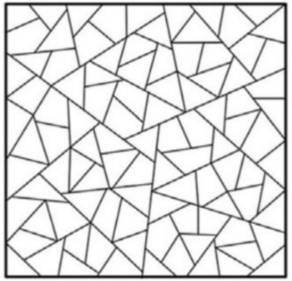
Source: Rashaad Newsome, Ron Eglash



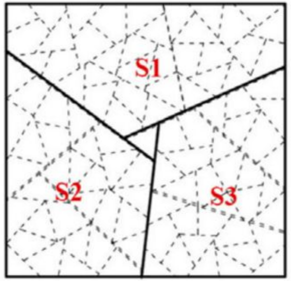
Source: Rashaad Newsome, Ron Eglash



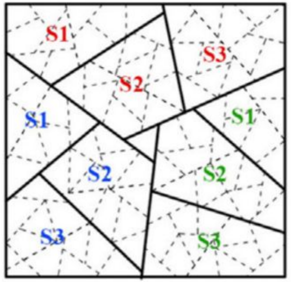
traditional cracked-ice lattice



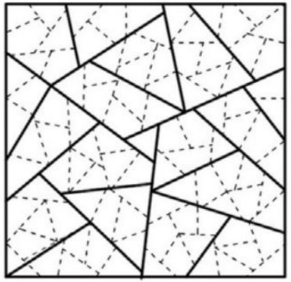
single-lines transformation



1st order segments

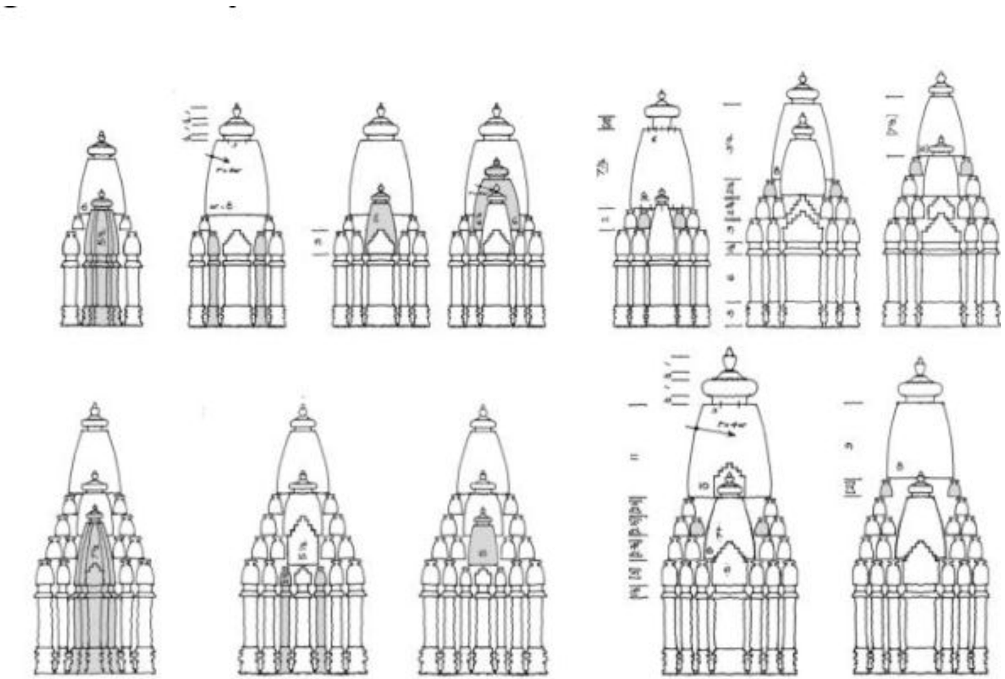


2nd order segments



3rd order segments

Source: [Chinese lattice geometry]



Source: [Indian Hindu Temple Architecture]

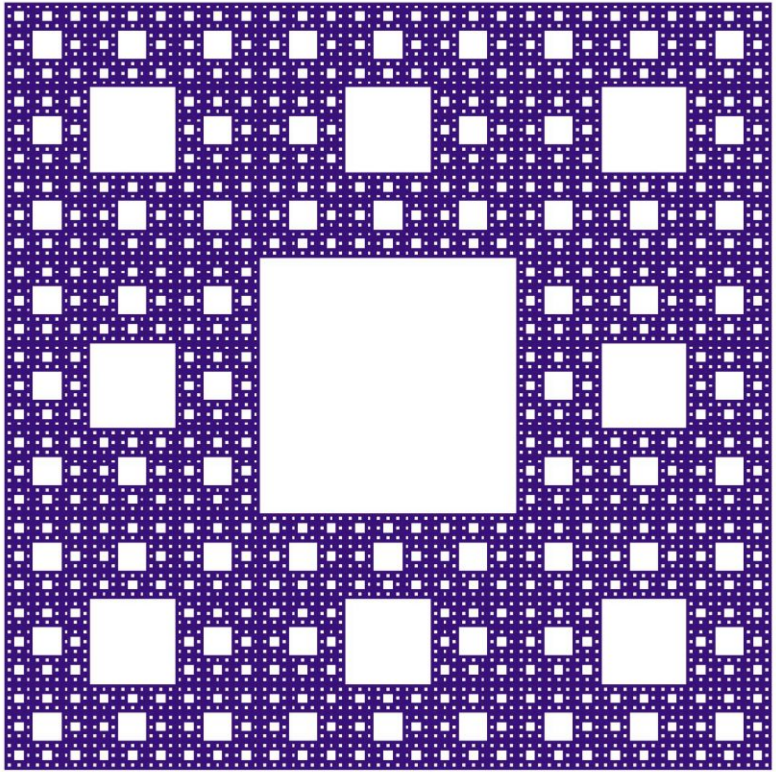
Fun Generative Art to Try!

- http://recursivedrawing.com/
- https://csdt.org/culture/africanfractals/science.html
- https://p5js.org/ / https://processing.org/
- The Coding Train youtube tutorials

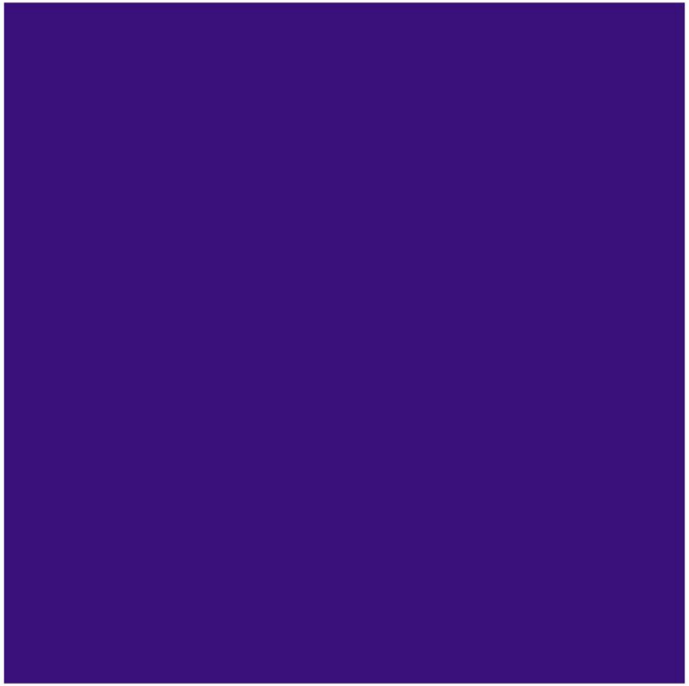
Sierpinski Carpet

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!

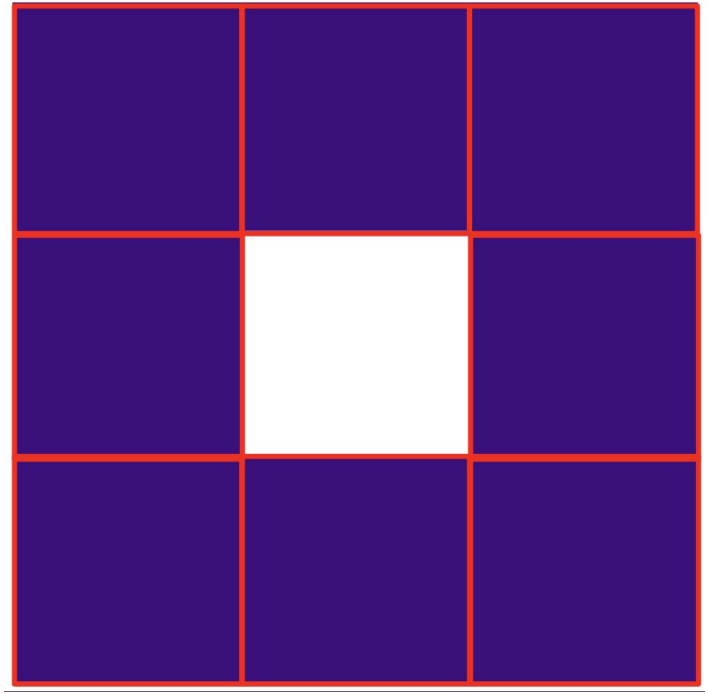


Order-0 Sierpinski Carpet

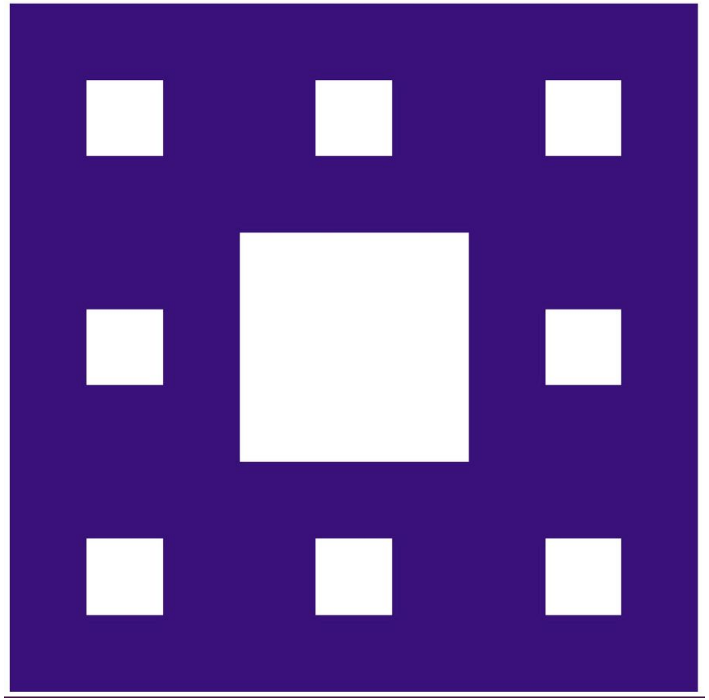


Order-1 Sierpinski Carpet

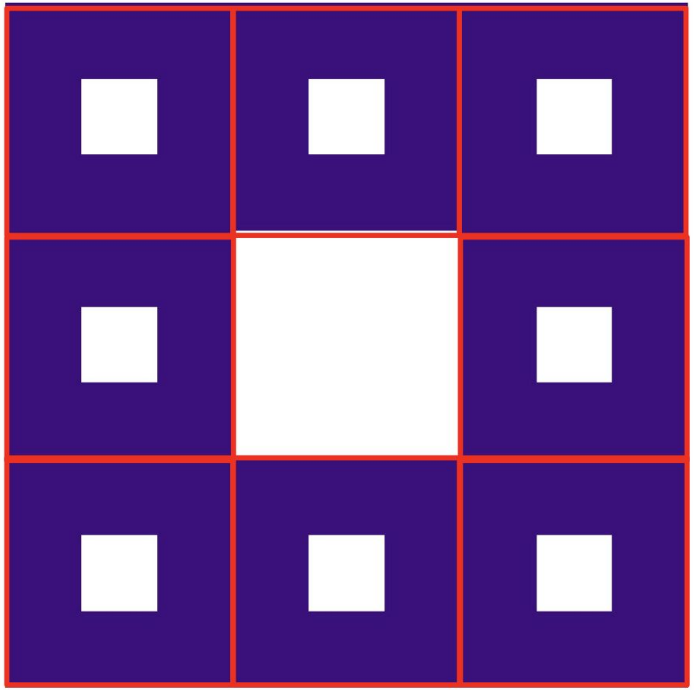
An order-1 carpet is subdivided into eight order-0 carpets arranged in this grid pattern



Order-2 Sierpinski Carpet



Order-2 Sierpinski Carpet



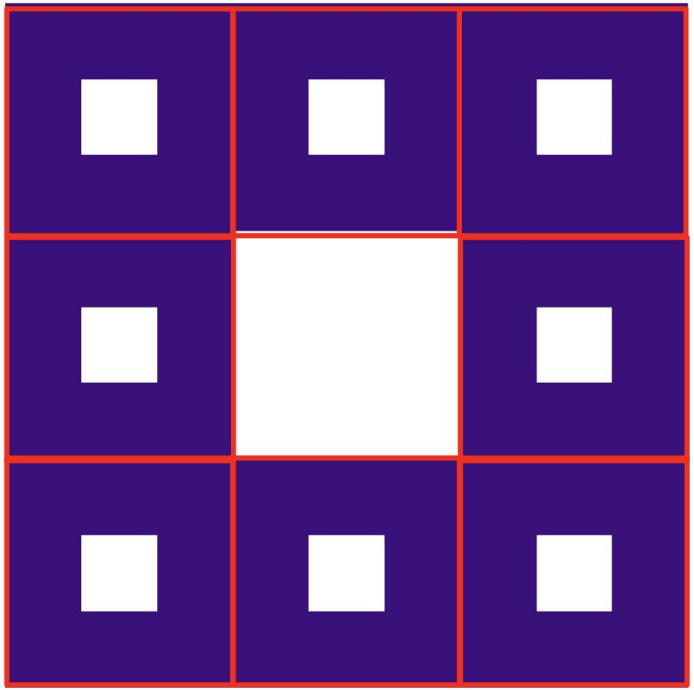
Sierpinski Carpet

Base Case: Order-0

- Draw a filled square at the appropriate location

Recursive Case: Order-n, n is not 0

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



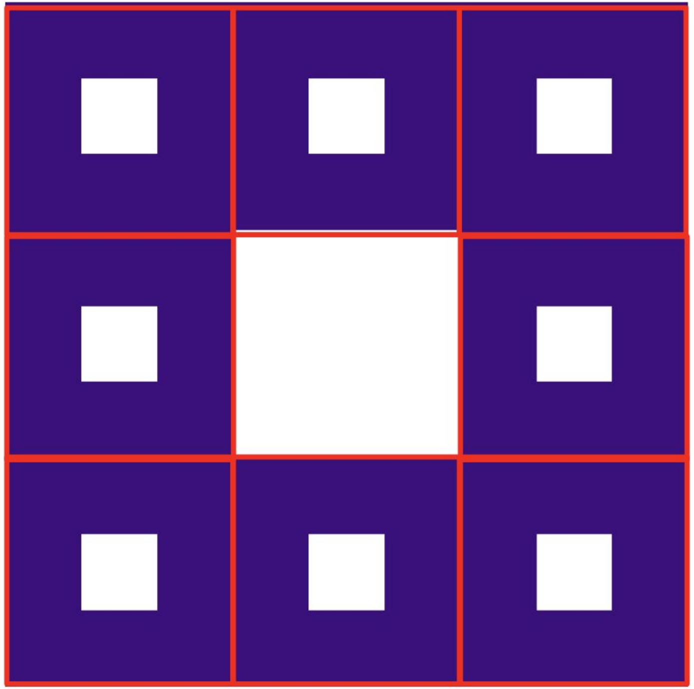
Sierpinski Carpet

Base Case: Order-0

- Draw a filled square at the appropriate location

Recursive Case: Order-n, n is not 0

- Draw 8 order-(n-1) Sierpinski carpets, arranged in a 3x3 grid, omitting the center location
- **Use loops!**



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 loops.
- 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!

Recap

- Fractal - any repeated, graphical pattern
  - Composed of repeated instances of the same shape or pattern, arranged in a structured way
  - Used almost universally across the world and across history
- More advanced recursion
  - Multiple base cases
  - Multiple recursive cases
  - Use iteration