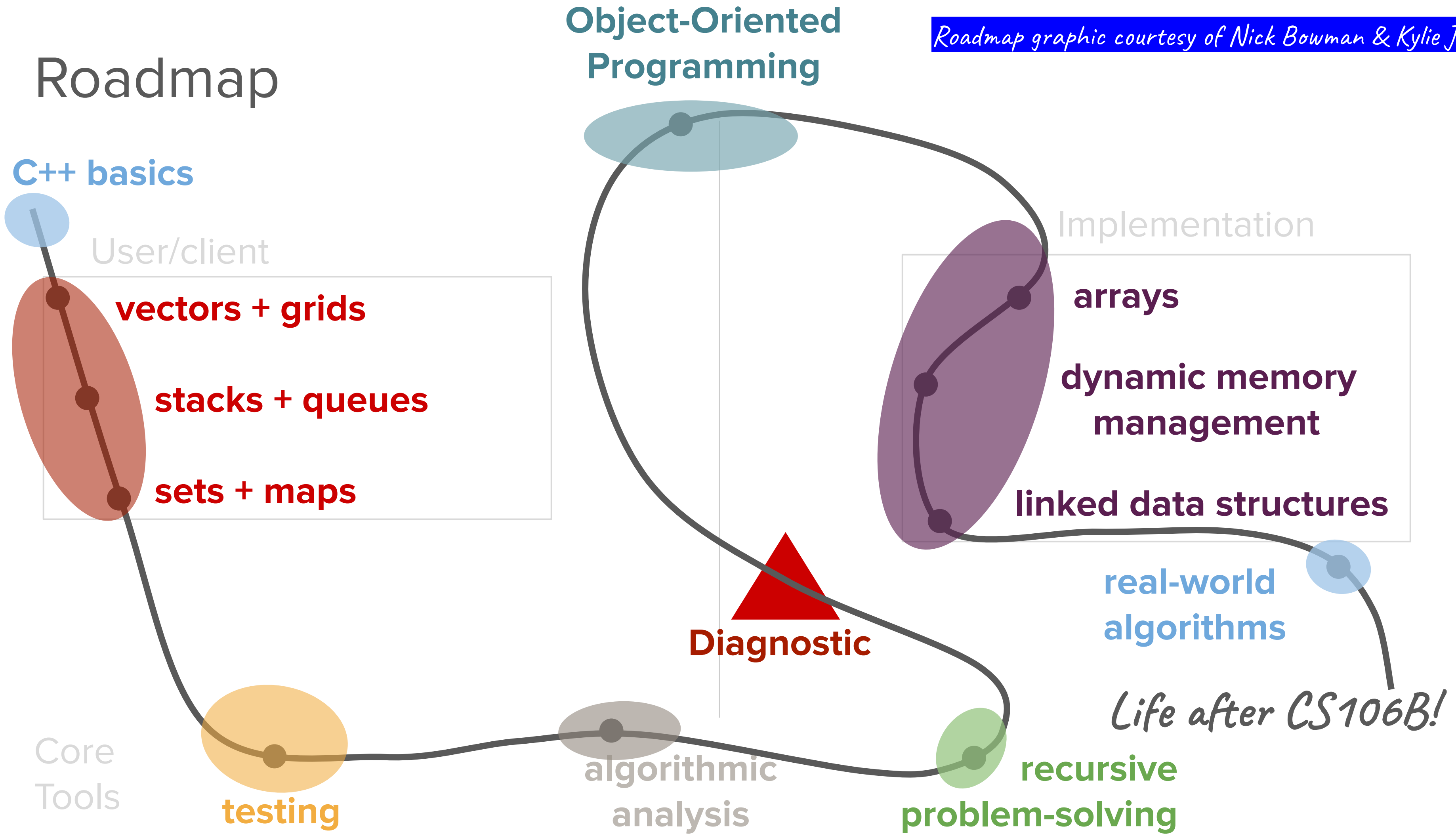
Introduction to Recursion

What's been the most challenging part of Assignment 2 for you so far?

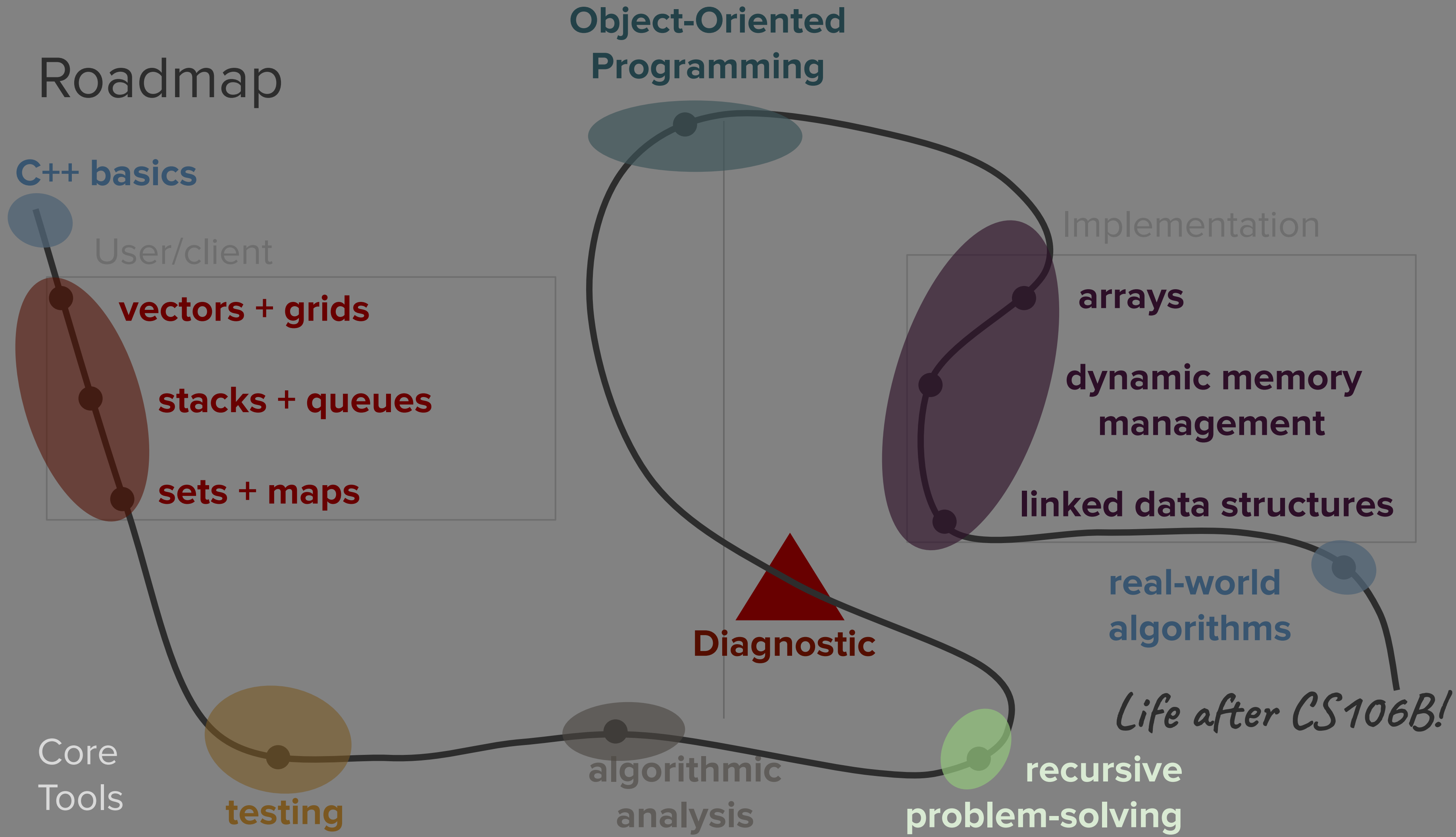
(put your answers the chat)

Roadmap

Roadmap graphic courtesy of Nick Bowman & Kylie Jue



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 topics

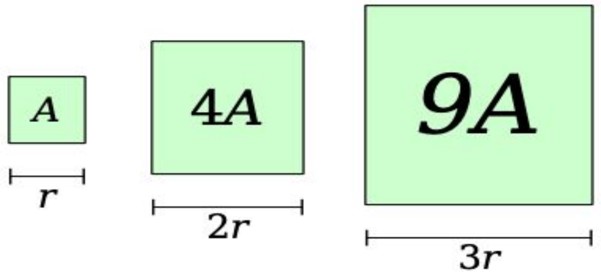
1. Review
2. Defining recursion
3. Recursion + Stack Frames (e.g. factorials)
4. Recursive Problem-Solving (e.g. string reversal)
5. Time permitting, introduction to Fractals

Review

Big O

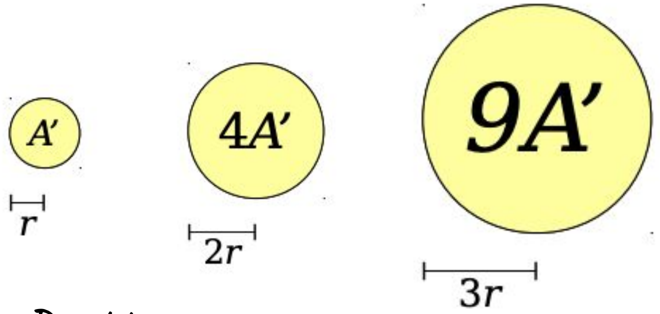
Big-O Notation

- **Big-O notation** is a way of quantifying the **rate at which some quantity grows**.
- Example:
  - A square of side length `r` has area `O(r`^{2}`)`.
  - A circle of radius `r` has area `O(r`^{2}`)`.



Doubling r increases area 4x

Tripling r increases area 9x



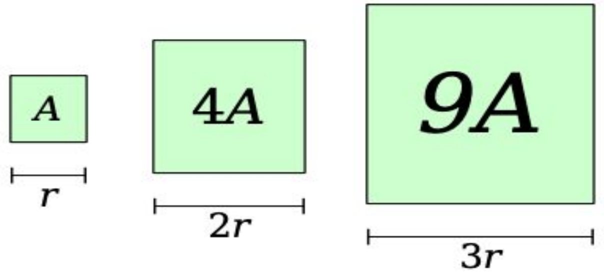
Doubling r increases area 4x

Tripling r increases area 9x

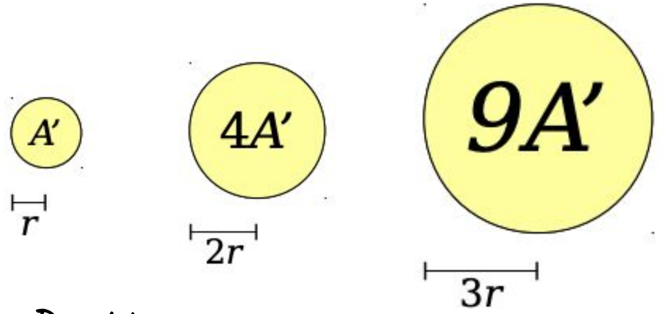
Big-O Notation

- **Big-O notation** is a way of quantifying the rate at which some quantity grows.
- Example:
  - A square of side length **`r`** has area **`O(r²)`**.
  - A circle of radius **`r`** has area **`O(r²)`**.

This just says that these quantities grow at the same relative rates. It does not say that they're equal!



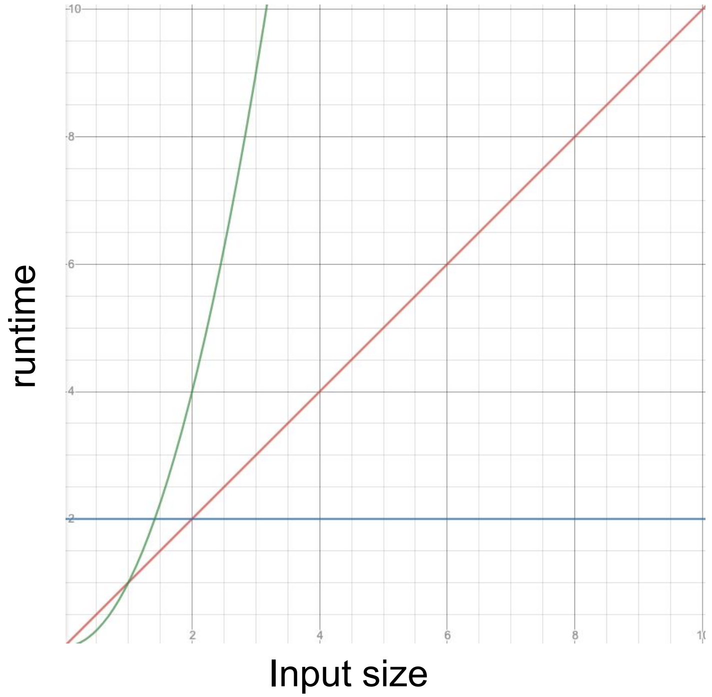
Doubling r increases area 4x
Tripling r increases area 9x



Doubling r increases area 4x
Tripling r increases area 9x

Efficiency Categorizations So Far

- Constant Time – O(1)
    - Super fast, this is the best we can hope for!
    - example: Euclid's Algorithm for Perfect Numbers
- Linear Time – O(n)
    - This is okay; we can live with this
- Quadratic Time – O(n^2)
    - This can start to slow down really quickly
    - example: Exhaustive Search for Perfect Numbers



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)`

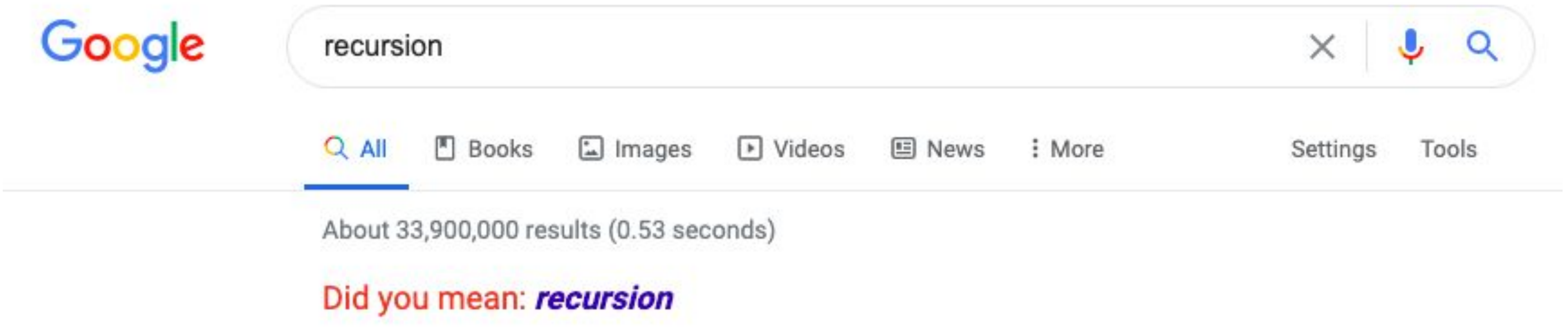
- Sets
  - `.size() - O(1)`
  - `.isEmpty() - O(1)`
  - `.add() - ???`
  - `.remove() - ???`
  - `.contains() - ???`
  - `traversal - O(n)`

- Maps
  - `.size() - O(1)`
  - `.isEmpty() - O(1)`
  - `m[key] - ???`
  - `.contains() - ???`
  - `traversal - O(n)`

What is recursion?

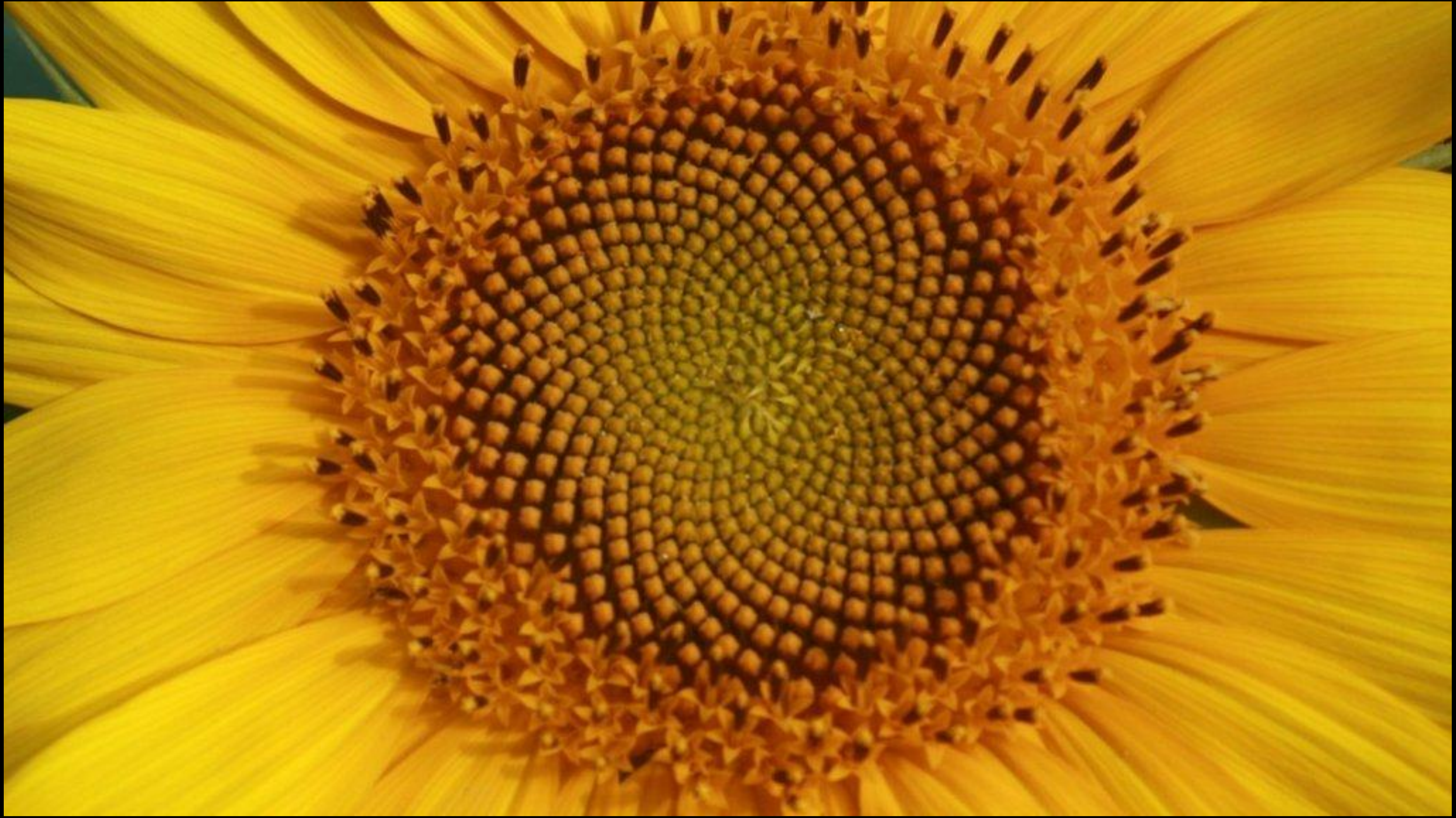
What is recursion?

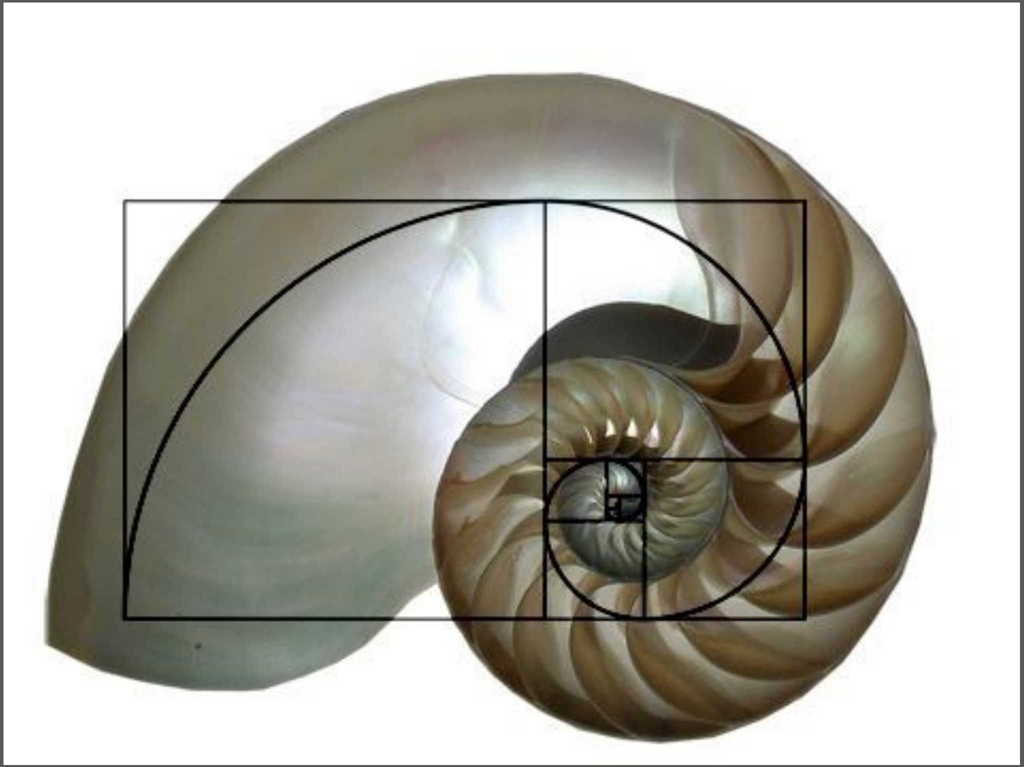
Wikipedia: “Recursion occurs when a thing is defined in terms of itself.”

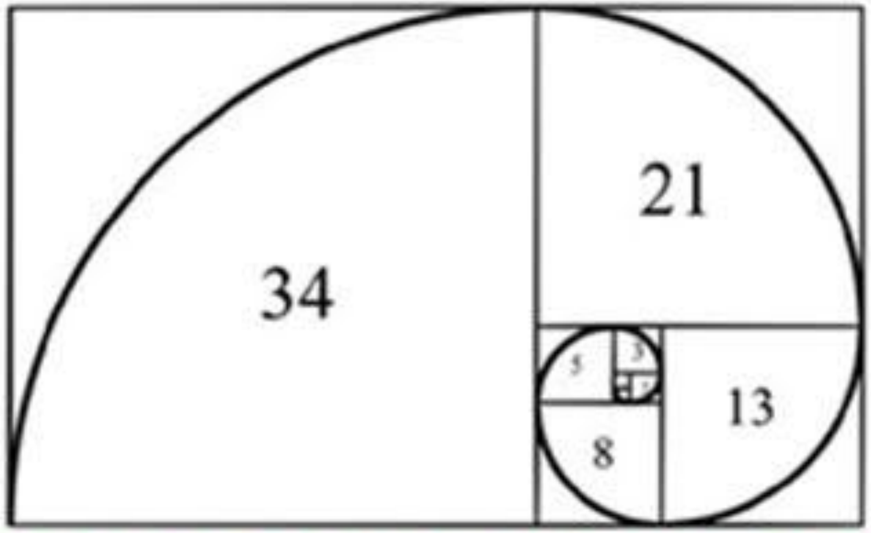


About 33,900,000 results (0.53 seconds)

Did you mean: *recursion*







0, 1, 1, 2, 3, 5, 8, 13, 21, ,34, 55, 89, 144...

0 + 1 = 1
1 + 1 = 2
2 + 1 = 3
3 + 2 = 5
5 + 3 = 8
8 + 5 = 13
13 + 8 = 21
21 + 13 = 34
34 + 21 = 55
55 + 34 = 89
89 + 55 = 144

Today's question

How can we take advantage of self-similarity within a problem to solve it more elegantly?

Definition

recursion

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

What is recursion?

- A powerful substitute for iteration (loops)
    - We’ll start off with seeing the difference between iterative vs. recursive solutions
    - Later in the week we’ll see problems/tasks that can only be solved using recursion

What is recursion?

- A powerful substitute for iteration (loops)
    - We'll start off with seeing the difference between iterative vs. recursive solutions
    - Later in the week we'll see problems/tasks that can only be solved using recursion

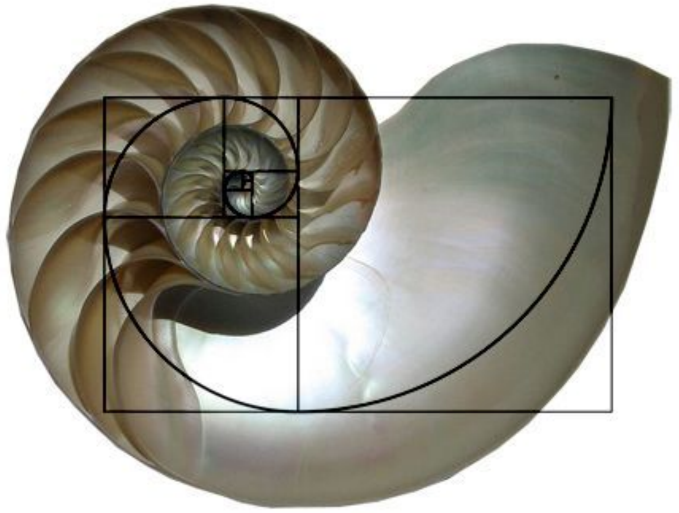
- Results in elegant, often shorter code when used well

What is recursion?

- A powerful substitute for iteration (loops)
  - We'll start off with seeing the difference solutions
  - Later in the week we'll see problems/ta recursion

- Results in elegant, often shorter code when

- Often applied to sorting and searching problems and can be used to express patterns seen in nature



What is recursion?

- A powerful substitute for iteration (loops)
  - We’ll start off with seeing the difference between iterative vs. recursive solutions
  - Later in the week we’ll see problems/tasks that can only be solved using recursion

- Results in elegant, often shorter code when used well

- Often applied to sorting and searching problems and can be used to express patterns seen in nature

- Will be part of many of our future assignments!

How many students are in a lecture hall?

a [non-Zoom] analogy

How many students are in the lecture hall?

- Let’s suppose I want to find out how many people are at lecture today, but I don’t want to walk around and count each person.

- I want to recruit your help, but I also want to minimize each individual’s amount of work.

How many students are in the lecture hall?

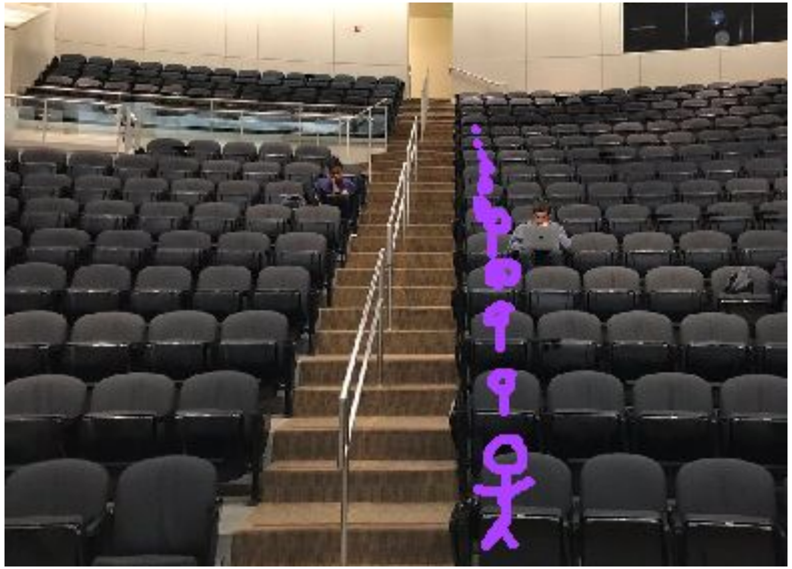
- Let’s suppose I want to find out how many people are at lecture today, but I don’t want to walk around and count each person.

- I want to recruit your help, but I also want to minimize each individual’s amount of work.

We can solve this problem recursively!

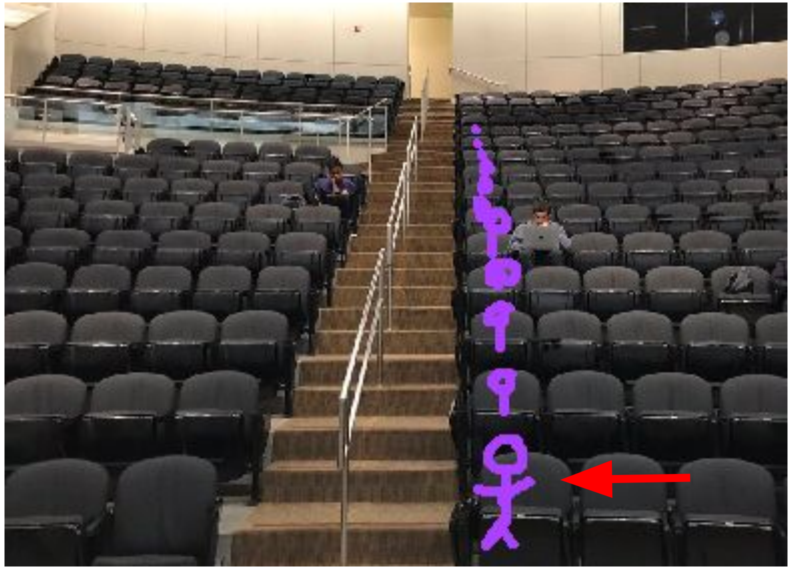
How many students are in the lecture hall?

- We’ll focus on solving the problem for single “column” of students.



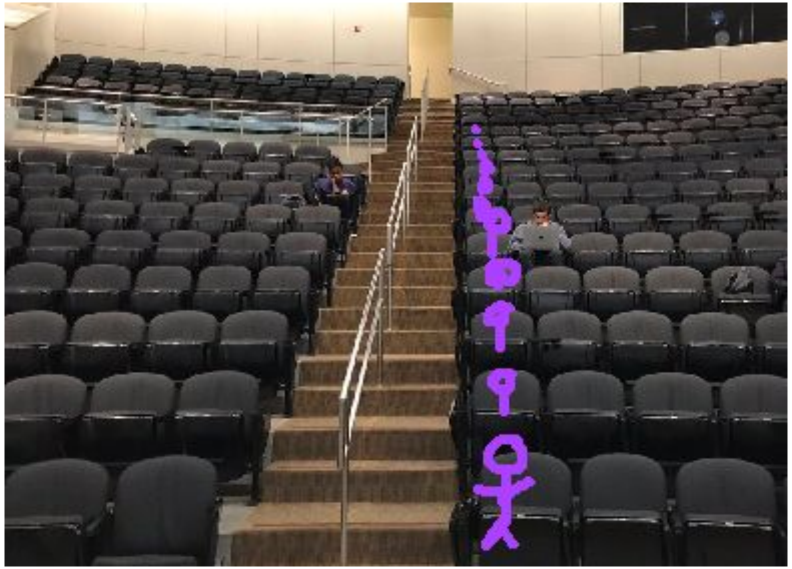
How many students are in the lecture hall?

- We’ll focus on solving the problem for single “column” of students.
  - I go to the first person in the front row and ask: “How many people are sitting directly behind you in your ‘column’?”



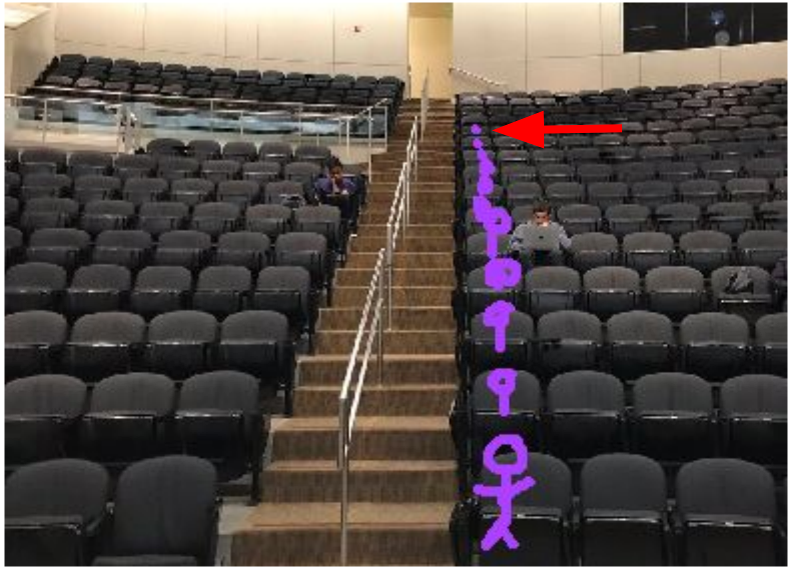
How many students are in the lecture hall?

- We'll focus on solving the problem for single "column" of students.
    - I go to the first person in the front row and ask: "How many people are sitting directly behind you in your 'column'?"
    - Student's algorithm:
        - If there is no one behind me, answer 0.
        - If someone is sitting behind me:
            - Ask that person: How many people are sitting directly behind you in your "column"?
            - When they respond with a value N, respond (N + 1) to the person who asked me.



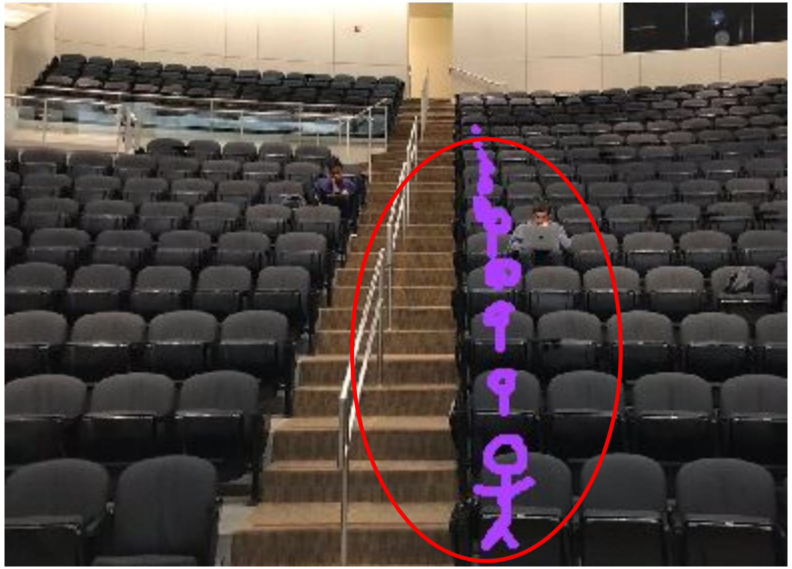
How many students are in the lecture hall?

- We'll focus on solving the problem for single "column" of students.
    - I go to the first person in the front row and ask: "How many people are sitting directly behind you in your 'column'?"
    - Student's algorithm:
        - **If there is no one behind me, answer 0.**
        - If someone is sitting behind me:
            - Ask that person: How many people are sitting directly behind you in your "column"?
            - When they respond with a value N, respond (N + 1) to the person who asked me.



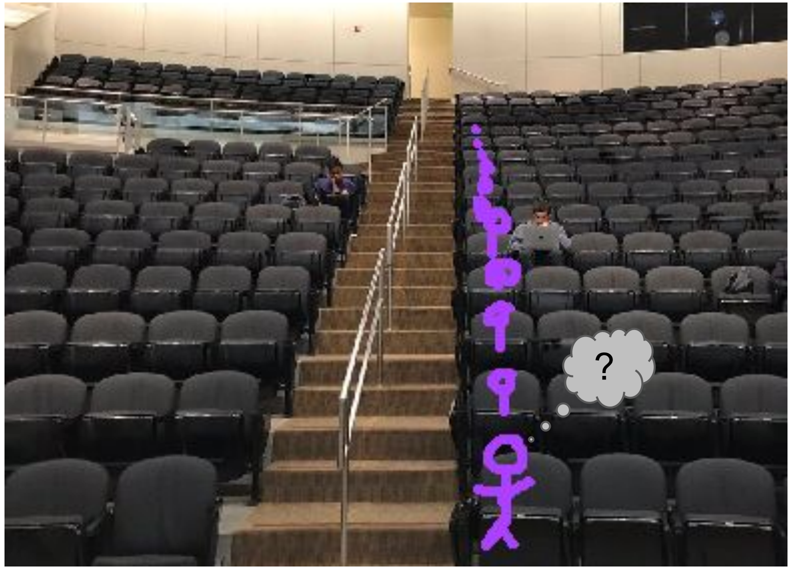
How many students are in the lecture hall?

- We'll focus on solving the problem for single "column" of students.
    - I go to the first person in the front row and ask: "How many people are sitting directly behind you in your 'column'?"
    - Student's algorithm:
        - If there is no one behind me, answer 0.
        - **If someone is sitting behind me:**
            - Ask that person: How many people are sitting directly behind you in your "column"?
            - When they respond with a value N, respond (N + 1) to the person who asked me.



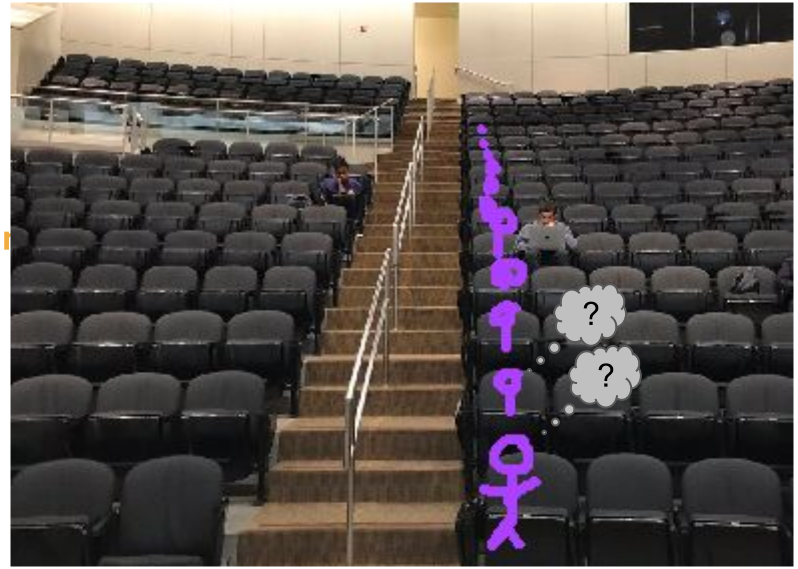
How many students are in the lecture hall?

- We'll focus on solving the problem for single "column" of students.
  - **I go to the first person in the front row and ask: "How many people are sitting directly behind you in your 'column'?"**
  - Student's algorithm:
    - If there is no one behind me, answer 0.
    - If someone is sitting behind me:
      - Ask that person: How many people are sitting directly behind you in your "column"?
      - When they respond with a value N, respond (N + 1) to the person who asked me.



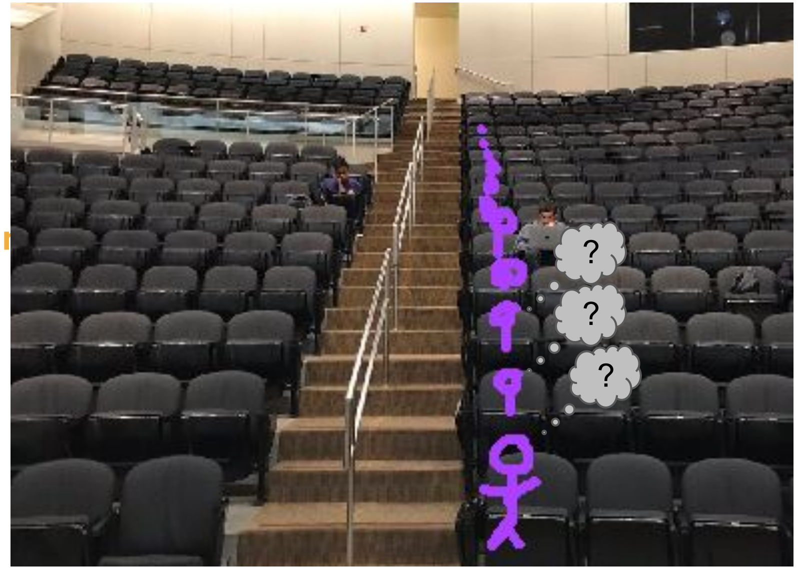
How many students are in the lecture hall?

- We'll focus on solving the problem for single "column" of students.
  - I go to the first person in the front row and ask: "How many people are sitting directly behind you in your 'column'?"
  - Student's algorithm:
    - If there is no one behind me, answer 0.
    - **If someone is sitting behind me:**
      - **Ask that person: How many people are sitting directly behind you in your "column"?**
      - When they respond with a value N, respond (N + 1) to the person who asked me.



How many students are in the lecture hall?

- We'll focus on solving the problem for single "column" of students.
  - I go to the first person in the front row and ask: "How many people are sitting directly behind you in your 'column'?"
  - Student's algorithm:
    - If there is no one behind me, answer 0.
    - **If someone is sitting behind me:**
      - **Ask that person: How many people are sitting directly behind you in your "column"?**
      - When they respond with a value N, respond (N + 1) to the person who asked me.



How many students are in the lecture hall?

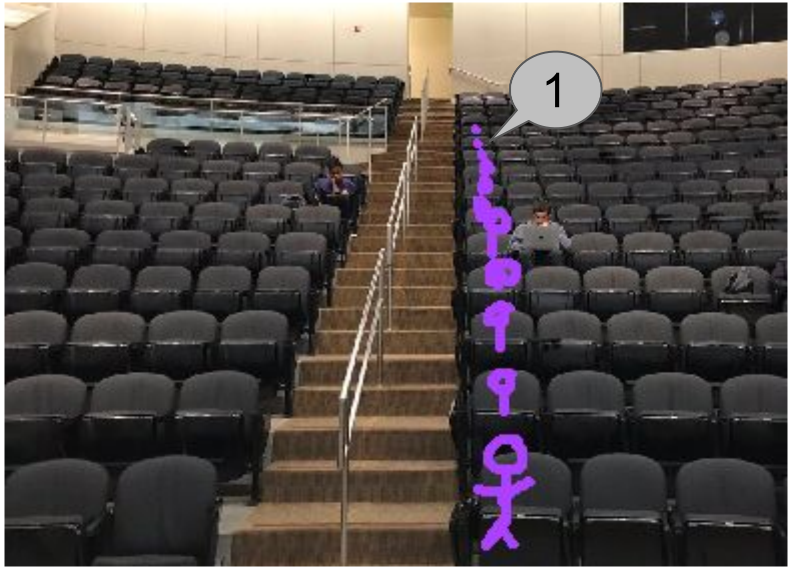
- We'll focus on solving the problem for single "column" of students.
  - I go to the first person in the front row and ask: "How many people are sitting directly behind you in your 'column'?"
  - Student's algorithm:
    - If there is no one behind me, answer 0.
    - **If someone is sitting behind me:**
      - **Ask that person: How many people are sitting directly behind you in your "column"?**
      - When they respond with a value N, respond (N + 1) to the person who asked me.

How many students are in the lecture hall?

- We'll focus on solving the problem for single "column" of students.
  - I go to the first person in the front row and ask: "How many people are sitting directly behind you in your 'column'?"
  - Student's algorithm:
    - **If there is no one behind me, answer 0.**
    - If someone is sitting behind me:
      - Ask that person: How many people are sitting directly behind you in your "column"?
      - When they respond with a value N, respond (N + 1) to the person who asked me.

How many students are in the lecture hall?

- We'll focus on solving the problem for single "column" of students.
  - I go to the first person in the front row and ask: "How many people are sitting directly behind you in your 'column'?"
  - Student's algorithm:
    - If there is no one behind me, answer 0.
    - **If someone is sitting behind me:**
      - Ask that person: How many people are sitting directly behind you in your "column"?
      - When they respond with a value N, respond (N + 1) to the person who asked me.

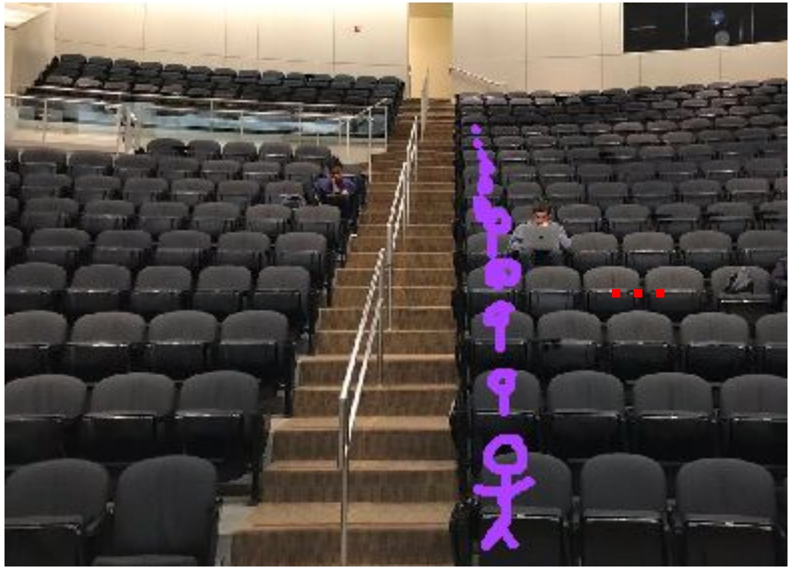


How many students are in the lecture hall?

- We'll focus on solving the problem for single "column" of students.
  - I go to the first person in the front row and ask: "How many people are sitting directly behind you in your 'column'?"
  - Student's algorithm:
    - If there is no one behind me, answer 0.
    - **If someone is sitting behind me:**
      - Ask that person: How many people are sitting directly behind you in your "column"?
      - When they respond with a value N, respond (N + 1) to the person who asked me.

How many students are in the lecture hall?

- We'll focus on solving the problem for single "column" of students.
  - I go to the first person in the front row and ask: "How many people are sitting directly behind you in your 'column'?"
  - Student's algorithm:
    - If there is no one behind me, answer 0.
    - **If someone is sitting behind me:**
      - Ask that person: How many people are sitting directly behind you in your "column"?
      - When they respond with a value N, respond (N + 1) to the person who asked me.



How many students are in the lecture hall?

- We'll focus on solving the problem for single "column" of students.
  - I go to the first person in the front row and ask: "How many people are sitting directly behind you in your 'column'?"
  - Student's algorithm:
    - If there is no one behind me, answer 0.
    - **If someone is sitting behind me:**
      - Ask that person: How many people are sitting directly behind you in your "column"?
      - When they respond with a value N, respond (N + 1) to the person who asked me.

How many students are in the lecture hall?

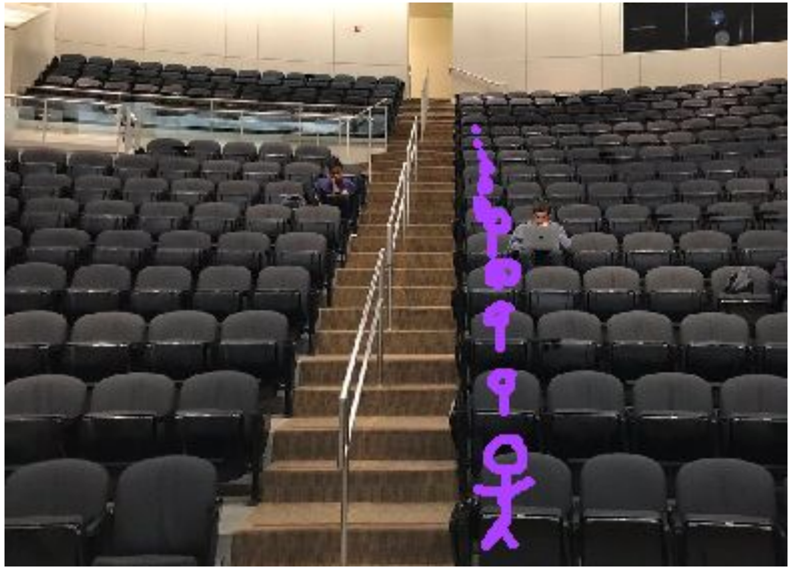
- We’ll focus on solving the problem for single “column” of students.
  - I go to the first person in the front row and ask: “How many people are sitting directly behind you in your ‘column’?”
  - Student’s algorithm:
    - If there is no one behind me, answer 0.
    - **If someone is sitting behind me:**
      - Ask that person: How many people are sitting directly behind you in your “column”?
      - When they respond with a value N, respond (N + 1) to the person who asked me.

How many students are in the lecture hall?

- We'll focus on solving the problem for single "column" of students.
  - I go to the first person in the front row and ask: "How many people are sitting directly behind you in your 'column'?"
  - Student's algorithm:
    - If there is no one behind me, answer 0.
    - **If someone is sitting behind me:**
      - Ask that person: How many people are sitting directly behind you in your "column"?
      - When they respond with a value N, respond (N + 1) to the person who asked me.

How many students are in the lecture hall?

- We'll focus on solving the problem for single "column" of students.
    - I go to the first person in the front row and ask: "How many people are sitting directly behind you in your 'column'?"
    - Student's algorithm:
        - If there is no one behind me, answer 0.
        - If someone is sitting behind me:
            - Ask that person: How many people are sitting directly behind you in your "column"?
            - When they respond with a value N, respond (N + 1) to the person who asked me.



- Can generalize to the entire lecture hall!

Definition

recursion

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

Two main cases (components) of recursion

- Base case
    - The simplest version(s) of your problem that all other cases reduce to
    - An occurrence that can be answered directly

Two main cases (components) of recursion

- Base case
  - The simplest version(s) of your problem that all other cases reduce to
  - An occurrence that can be answered directly

“If there is no one behind me, answer 0.”

Two main cases (components) of recursion

- Base case
  - The simplest version(s) of your problem that all other cases reduce to
  - An occurrence that can be answered directly

- Recursive case
  - The step at which you break down more complex versions of the task into smaller occurrences
  - Cannot be answered directly
  - Take the “recursive leap of faith” and trust the smaller tasks will solve the problem for you!

Two main cases (components) of recursion

- Base case
    - The simplest version(s) of your problem that all other cases reduce to
    - An occurrence that can be answered directly

- Recursive case
    - The step at which you break down more complex versions of the task into smaller occurrences
    - Cannot be answered directly
    - Take the “recursive leap of faith” and trust the smaller tasks will solve the problem for you!

“If someone is sitting behind me…”

Two main cases (components) of recursion

- Base case
  - The simplest version(s) of your problem that all other cases reduce to
  - An occurrence that can be answered directly

- Recursive case
  - The step at which you break down more complex versions of the task into smaller occurrences
  - Cannot be answered directly
  - Take the “recursive leap of faith” and trust the smaller tasks will solve the problem for you!

Factorial example

Factorials

- The number **n factorial**, denoted **n!**, is

$$n \times (n - 1) \times \ldots \times 3 \times 2 \times 1$$

Factorial slides based on an example by Keith Schwarz

Factorials

- The number **n factorial**, denoted **n!**, is

$$n \times (n - 1) \times \ldots \times 3 \times 2 \times 1$$

- For example,
  - `3! = 3 × 2 × 1 = 6.`
  - `4! = 4 × 3 × 2 × 1 = 24.`
  - `5! = 5 × 4 × 3 × 2 × 1 = 120.`
  - `0! = 1. (by definition)`

Factorials

- The number **n factorial**, denoted **n!**, is

$$n \times (n - 1) \times \ldots \times 3 \times 2 \times 1$$

- For example,
    - `3! = 3 × 2 × 1 = 6.`
    - `4! = 4 × 3 × 2 × 1 = 24.`
    - `5! = 5 × 4 × 3 × 2 × 1 = 120.`
    - `0! = 1. (by definition)`
- Factorials show up in unexpected places. We'll see one later this quarter when we talk about sorting algorithms.

Factorials

- The number **n factorial**, denoted **n!**, is

$$n \times (n - 1) \times \ldots \times 3 \times 2 \times 1$$

- For example,
  - `3! = 3 × 2 × 1 = 6.`
  - `4! = 4 × 3 × 2 × 1 = 24.`
  - `5! = 5 × 4 × 3 × 2 × 1 = 120.`
  - `0! = 1. (by definition)`
- Factorials show up in unexpected places. We'll see one later this quarter when we talk about sorting algorithms.
- Let's implement a function to compute factorials!

Computing factorials

```
5! = 5 x 4 x 3 x 2 x 1
```

Computing factorials

```
5! = 5 x 4 x 3 x 2 x 1
```

Computing factorials

$$5! = 5 \times \underbrace{4 \times 3 \times 2 \times 1}_{4!}$$

Computing factorials

```
5! = 5 x 4!
```

Computing factorials

```
5! = 5 x 4!
```

Computing factorials

```
5! = 5 x 4!
4! = 4 x 3 x 2 x 1
```

Computing factorials

```
5! = 5 x 4!
4! = 4 x 3 x 2 x 1
```

Computing factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times \underbrace{3 \times 2 \times 1}_{3!}$$

Computing factorials

```
5! = 5 x 4!
4! = 4 x 3!
```

Computing factorials

```
5! = 5 x 4!
4! = 4 x 3!
```

Computing factorials

```
5! = 5 x 4!
4! = 4 x 3!
3! = 3 x 2 x 1
```

Computing factorials

```
5! = 5 x 4!
4! = 4 x 3!
3! = 3 x 2 x 1
```

Computing factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times \underbrace{2 \times 1}_{2!}$$

Computing factorials

```
5! = 5 x 4!
4! = 4 x 3!
3! = 3 x 2!
```

Computing factorials

```
5! = 5 x 4!
4! = 4 x 3!
3! = 3 x 2!
```

Computing factorials

```
5! = 5 x 4!
4! = 4 x 3!
3! = 3 x 2!
2! = 2 x 1!
```

Computing factorials

```
5! = 5 x 4!
4! = 4 x 3!
3! = 3 x 2!
2! = 2 x 1!
1! = 1 x 0!
```

Computing factorials

```
5! = 5 x 4!
4! = 4 x 3!
3! = 3 x 2!
2! = 2 x 1!
1! = 1 x 0!
0! = 1
```

Computing factorials

```
5! = 5 x 4!
4! = 4 x 3!
3! = 3 x 2!
2! = 2 x 1!
1! = 1 x 0!
0! = 1
```

By definition!

Another view of factorials

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

Another view of factorials

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

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

Recursion in action

```
int main() {
    int n = factorial(5);
    cout << "5! = " << n << endl;
    return 0;
}
```

Recursion in action

```
int main() {
    int n = factorial(5);
    cout << "5! = " << n << endl;
    return 0;
}
```

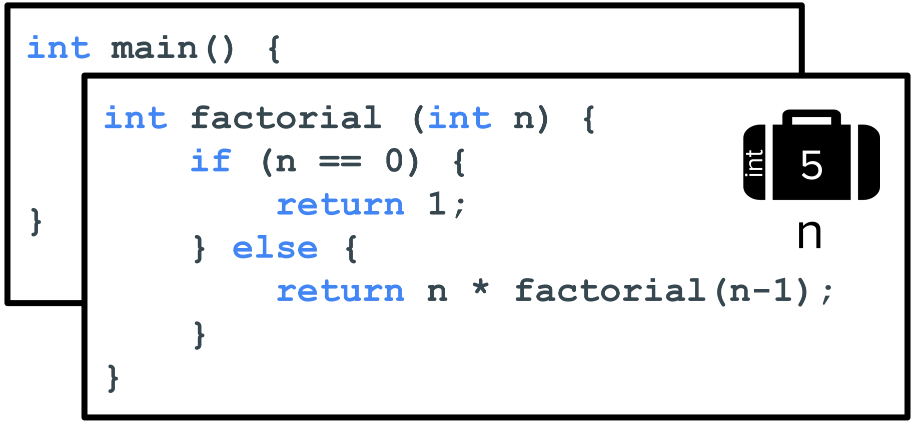
This is a "**stack frame**." One gets created each time a function is called.

- The "stack" is where in your computer's memory the information is stored.
- A "frame" stores all of the data (variables) for that particular function call.

Recursion in action

```
int main() {
    int n = factorial(5);
    cout << "5! = " << n << endl;
    return 0;
}
```

Recursion in action



```
int main() {

}
```

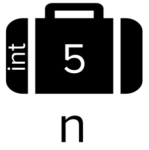
```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```

When a function gets called, a new stack frame gets created.

Recursion in action

```
int main() {

}
```

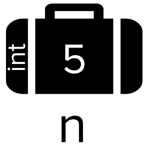
```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```



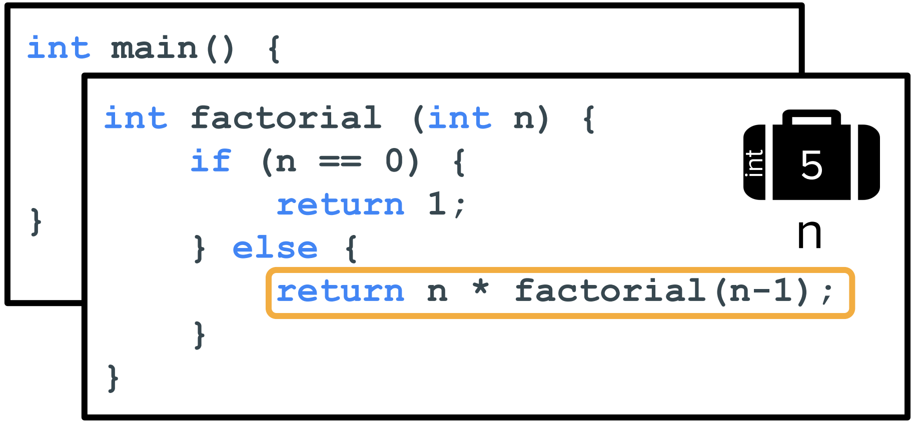
Recursion in action

```
int main() {


}
```

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



Recursion in action



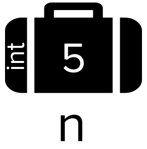
```
int main() {

}
```

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

Recursion in action

```
int main() {

}
```

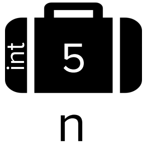
```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }          5
}
```



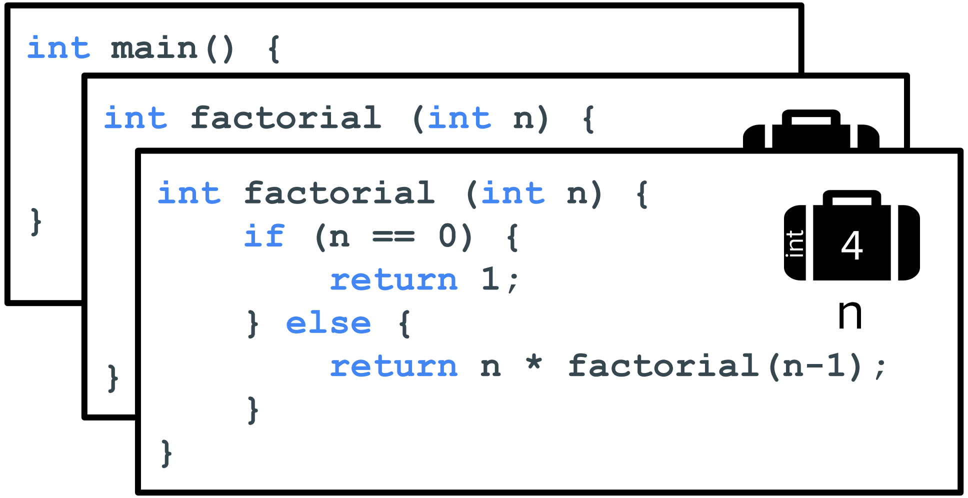
Recursion in action

```
int main() {


}
```

```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }           5
}
```



Recursion in action



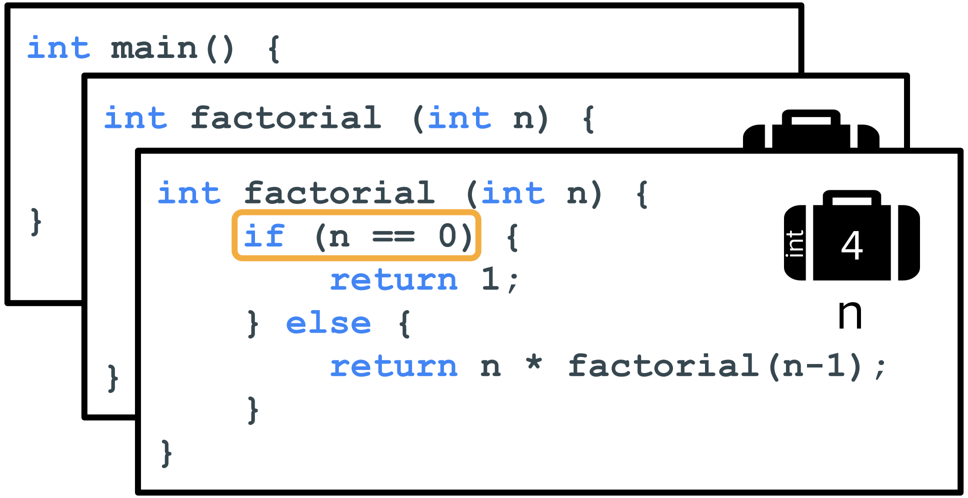
```
int main() {

}
```

```
int factorial (int n) {

}
```

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

Every time we call **`factorial()`**, we get a new copy of the local variable **n** that's independent of all the previous copies because it exists inside the new frame.

Recursion in action

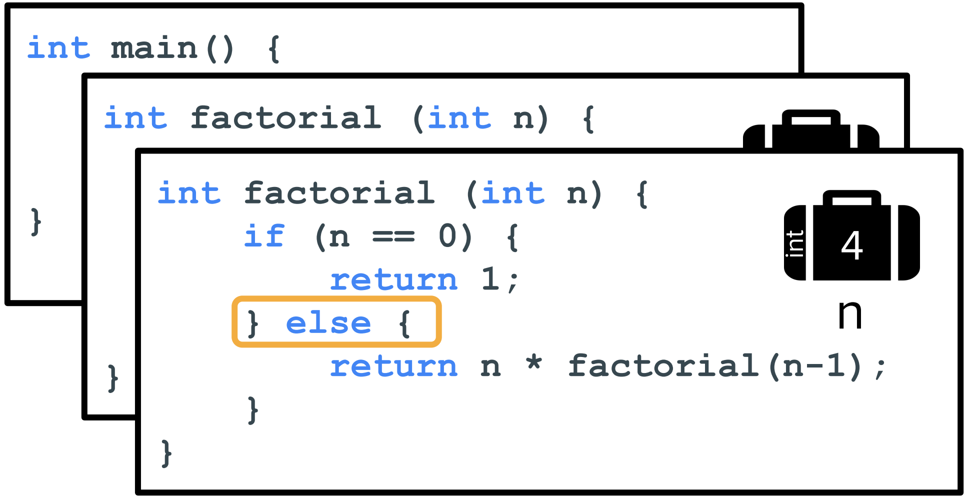


```
int main() {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action

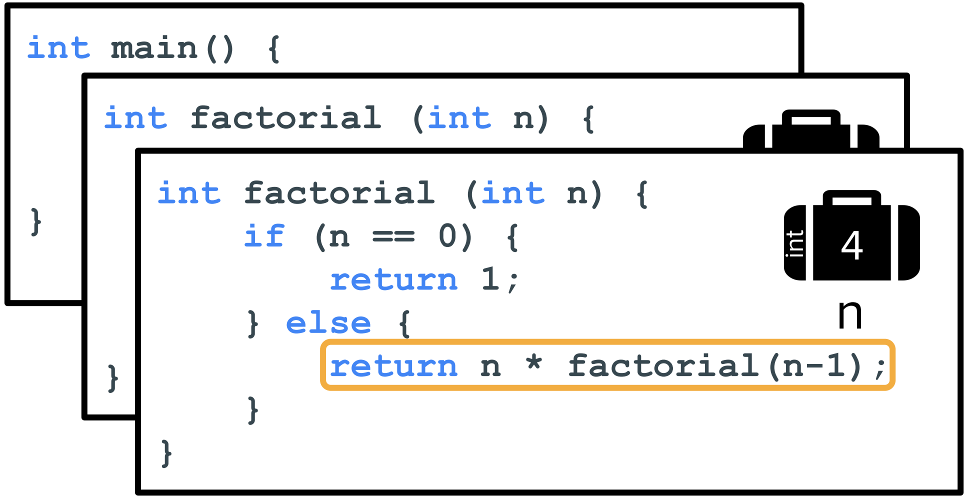


```
int main() {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action

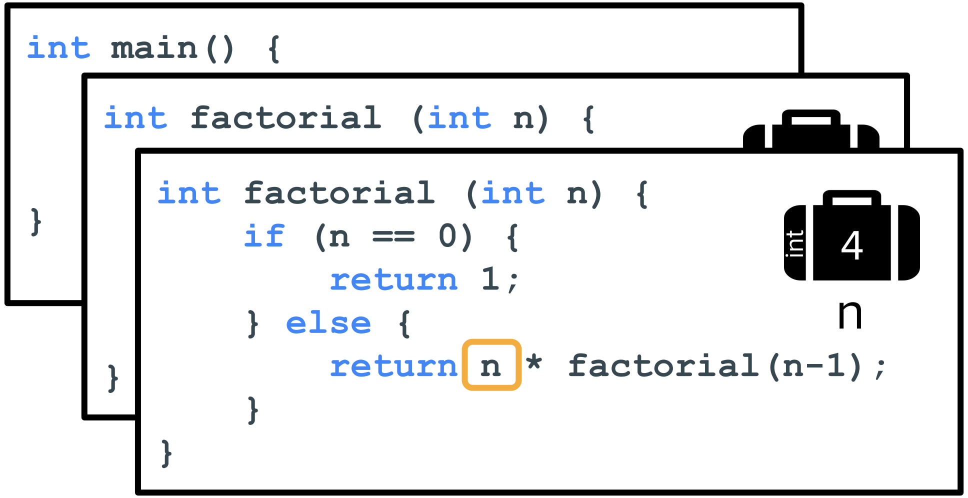


```
int main() {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action

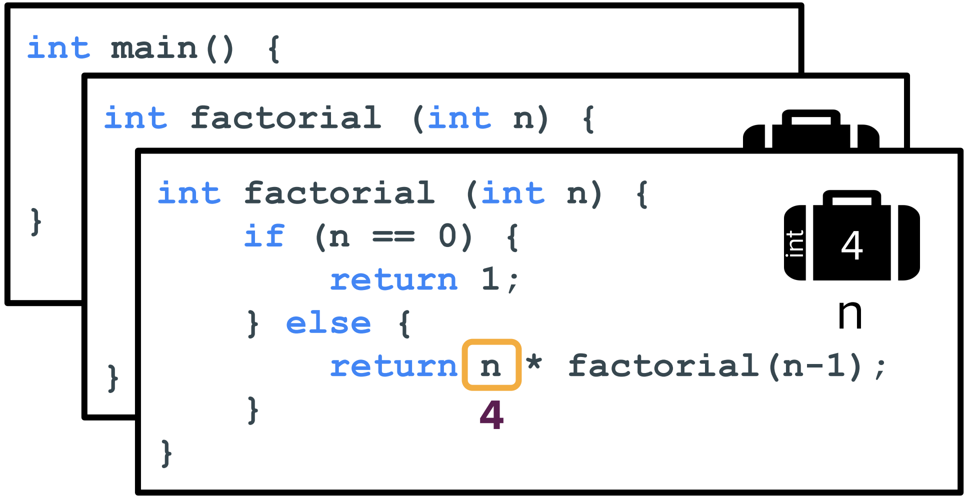


```
int main() {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action

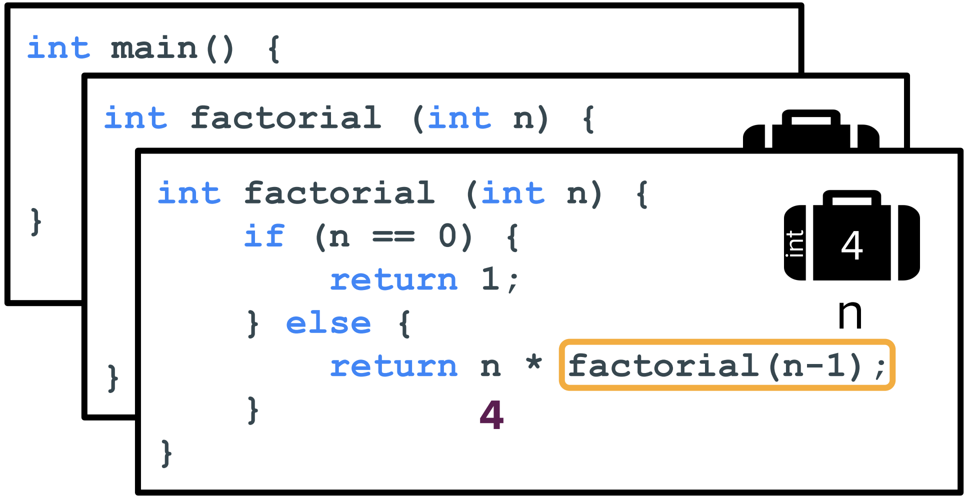


```
int main() {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action

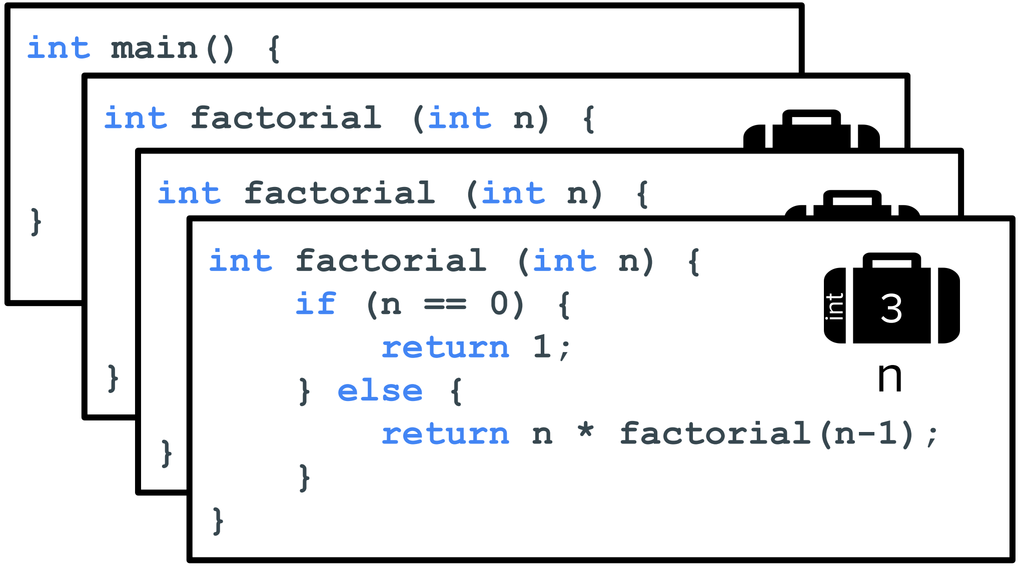


```
int main() {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action



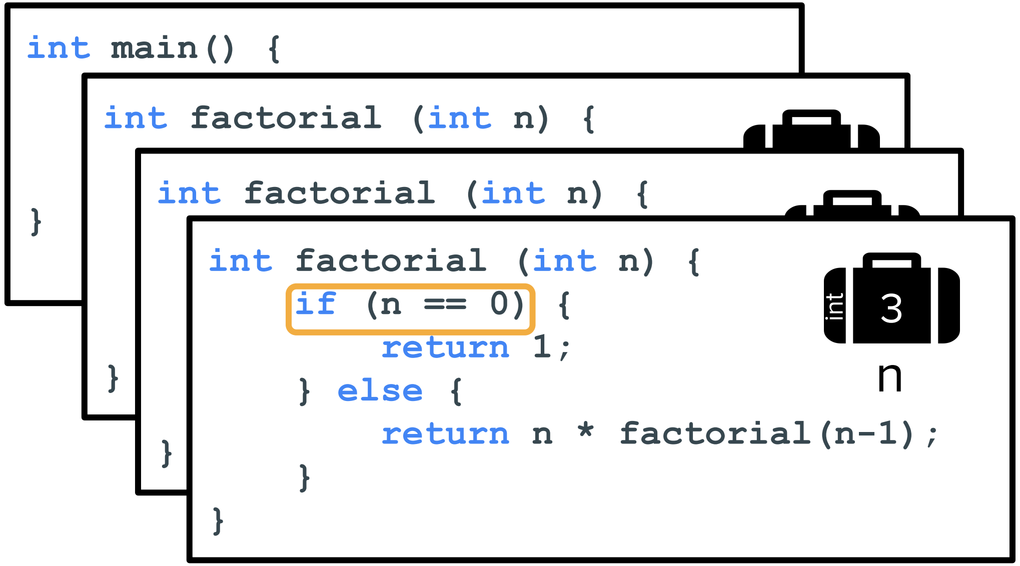
```
int main() {
}
```

```
int factorial (int n) {
}
```

```
int factorial (int n) {
}
```

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

Recursion in action



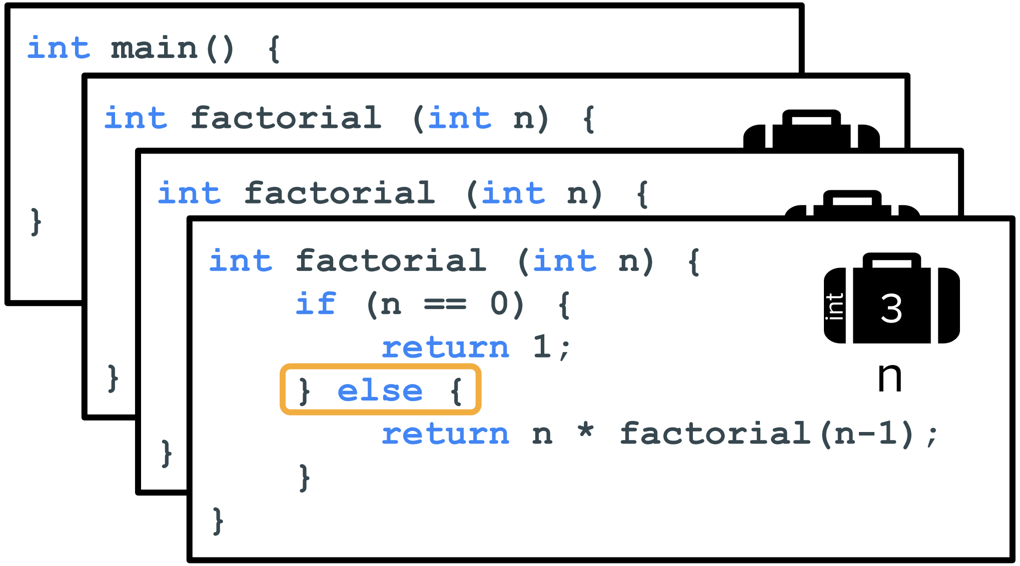
```
int main() {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action



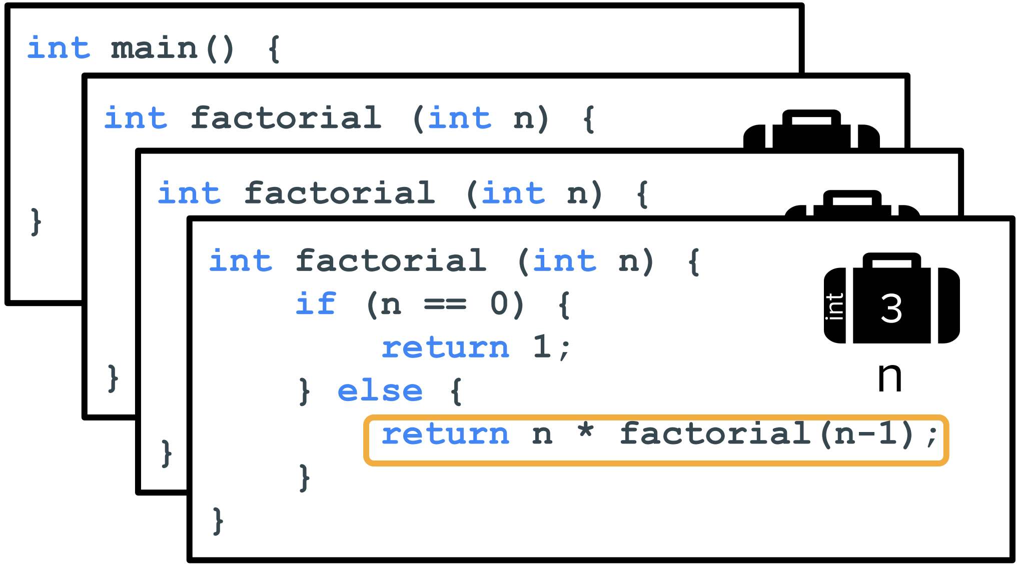
```
int main() {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action



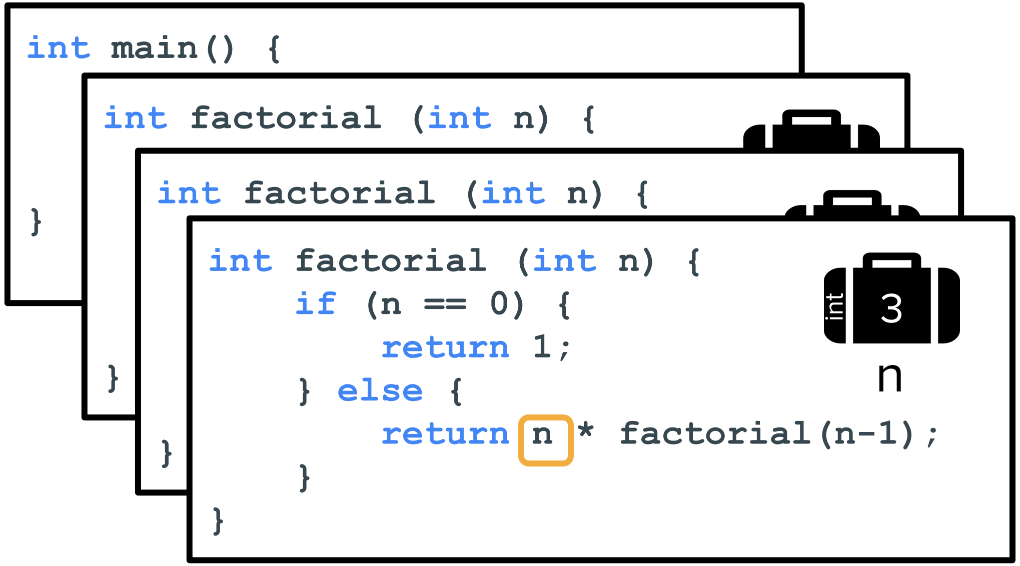
```
int main() {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action



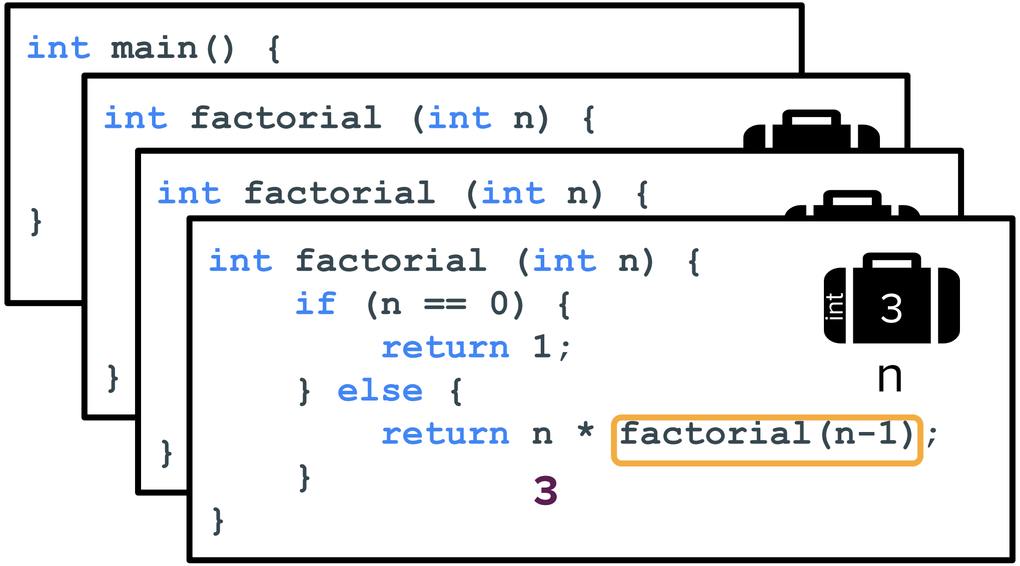
```
int main() {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action



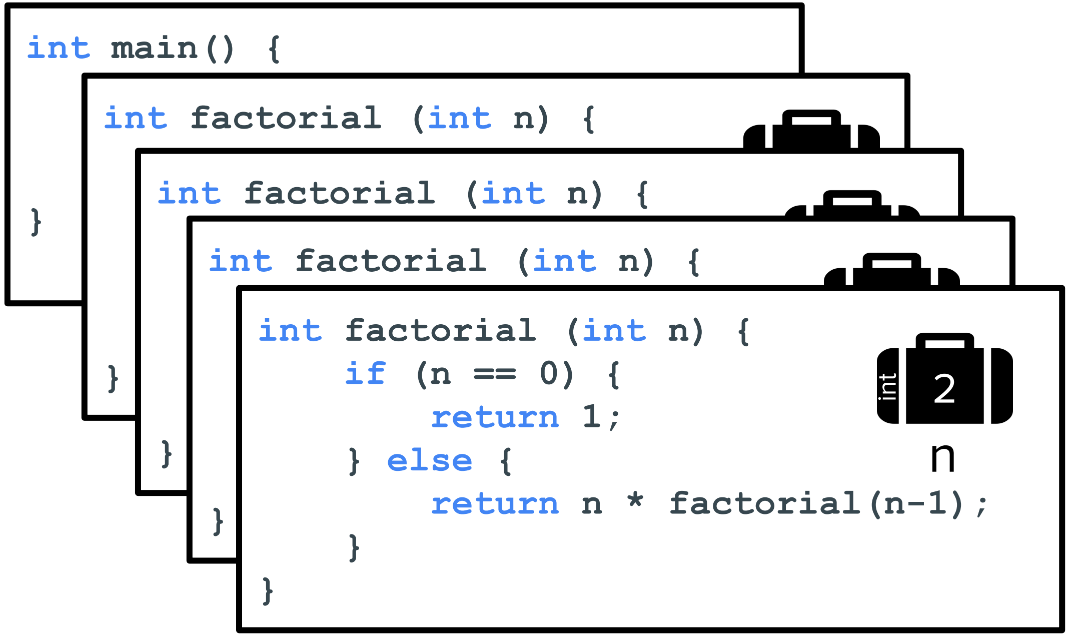
```
int main() {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }          3
}
```

Recursion in action



```
int main() {
}
```

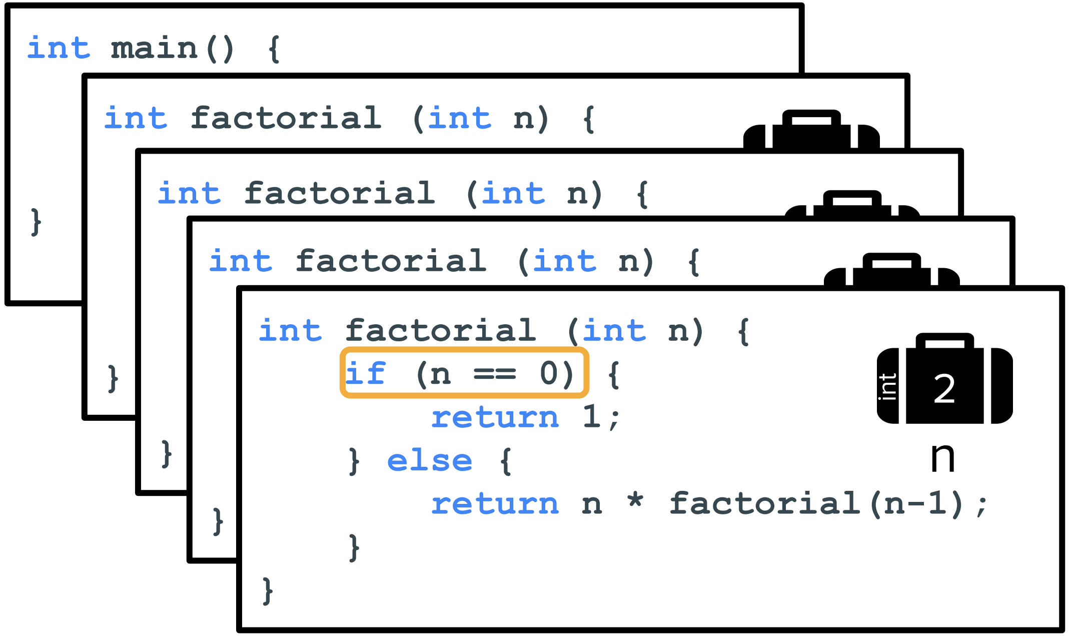
```
int factorial (int n) {
}
```

```
int factorial (int n) {
}
```

```
int factorial (int n) {
}
```

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

Recursion in action



```
int main() {

}
```

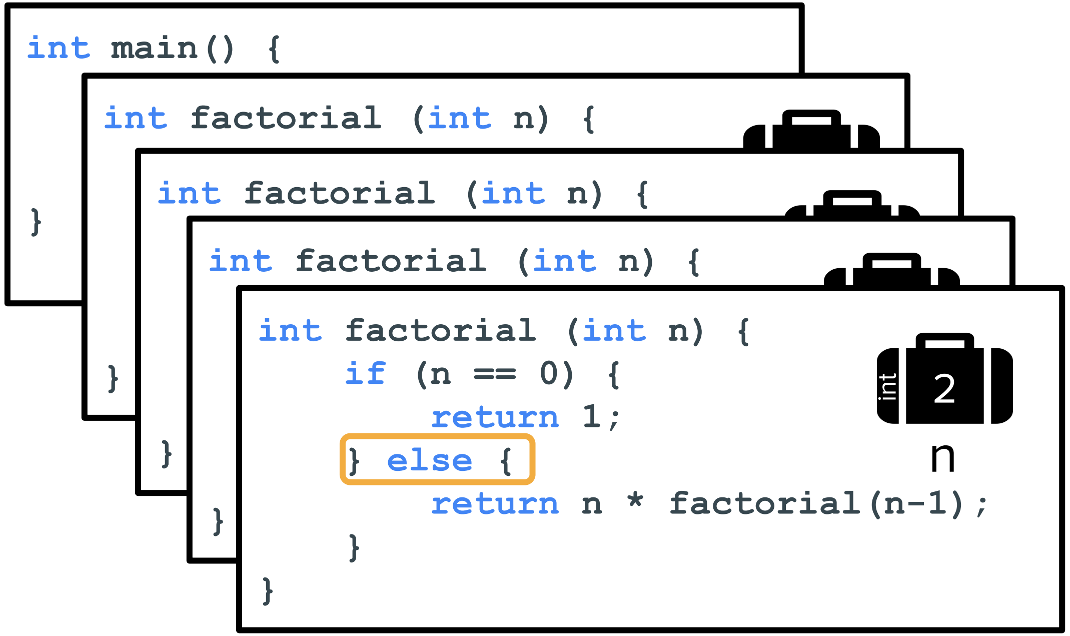
```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action



```
int main() {

}
```

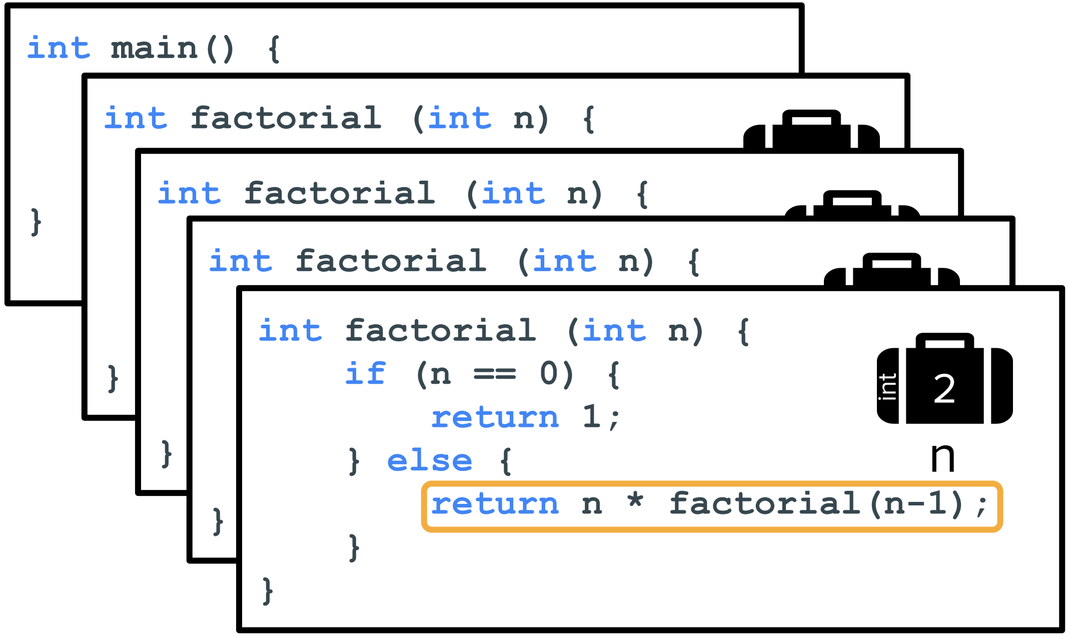
```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action



```
int main() {

}
```

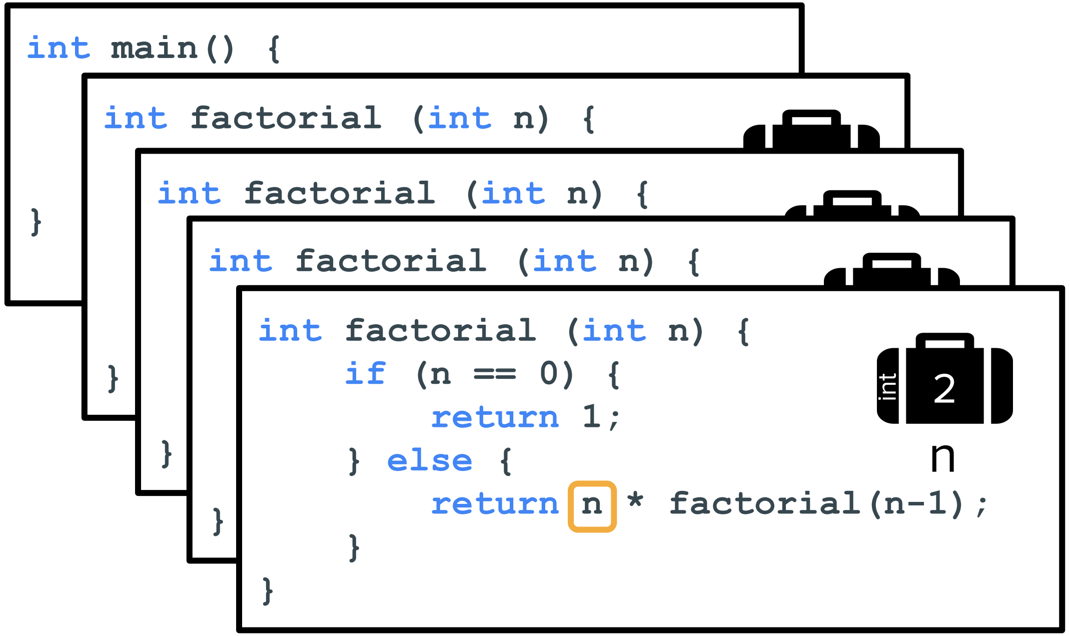
```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action



```
int main() {

}
```

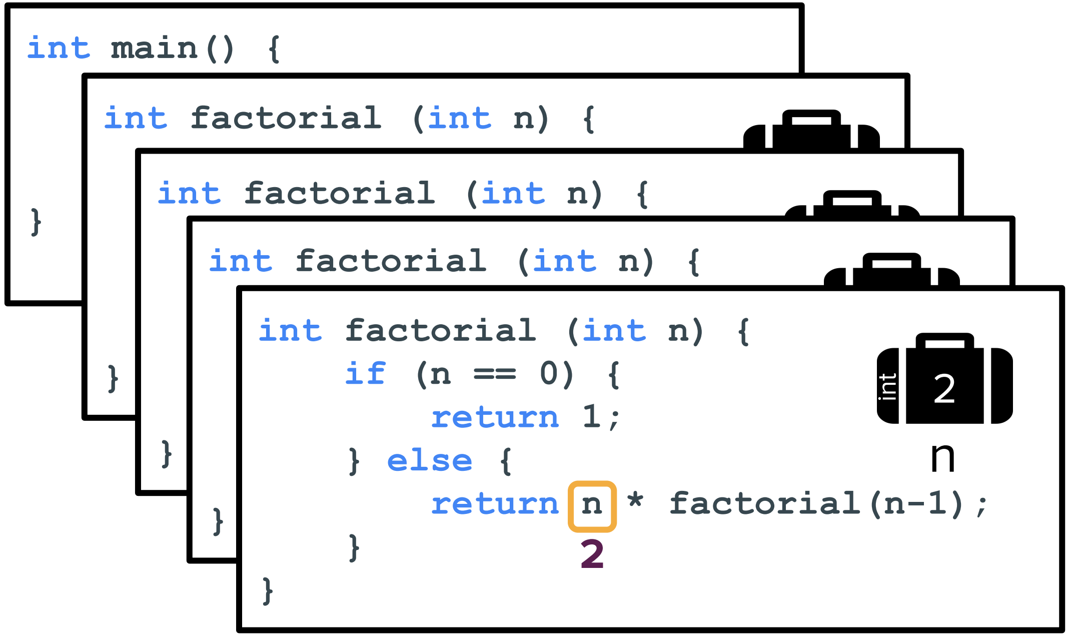
```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

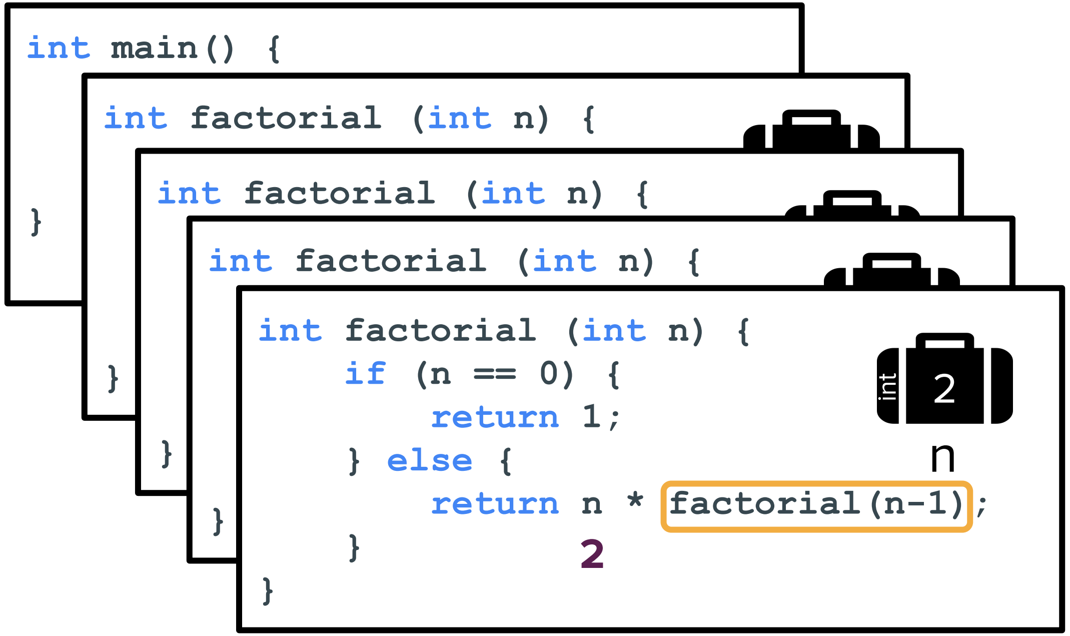
```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```

Recursion in action



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

Recursion in action



```
int main() {

}
```

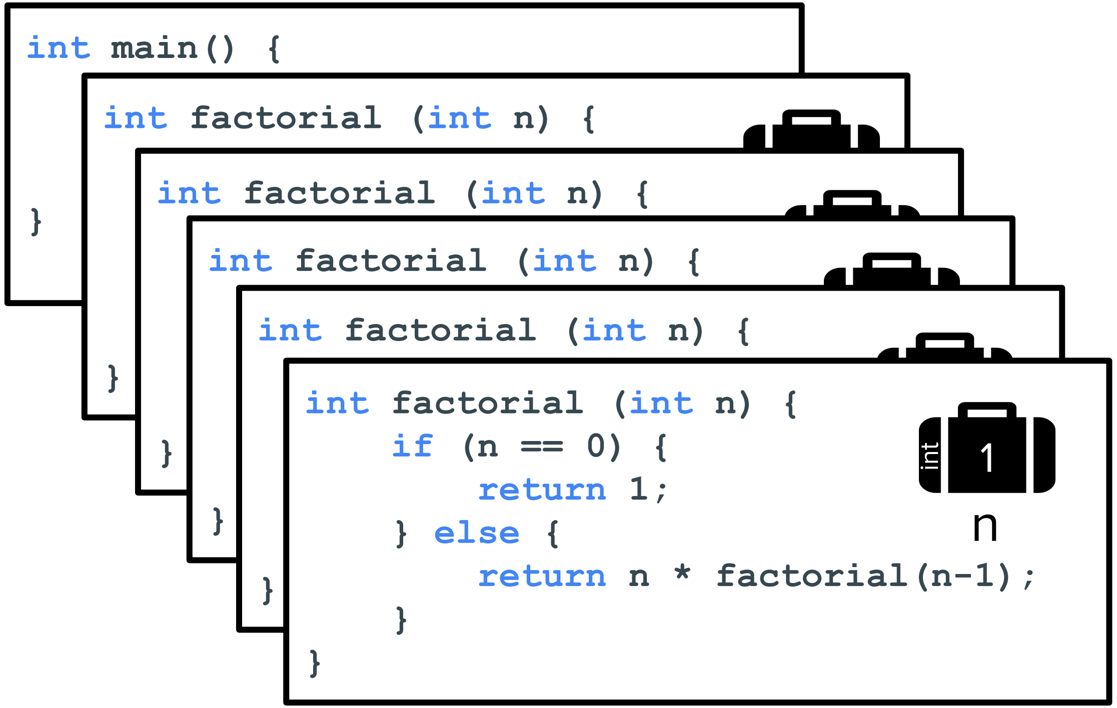
```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }           2
}
```

Recursion in action



```
int main() {
}
```

```
int factorial (int n) {
}
```

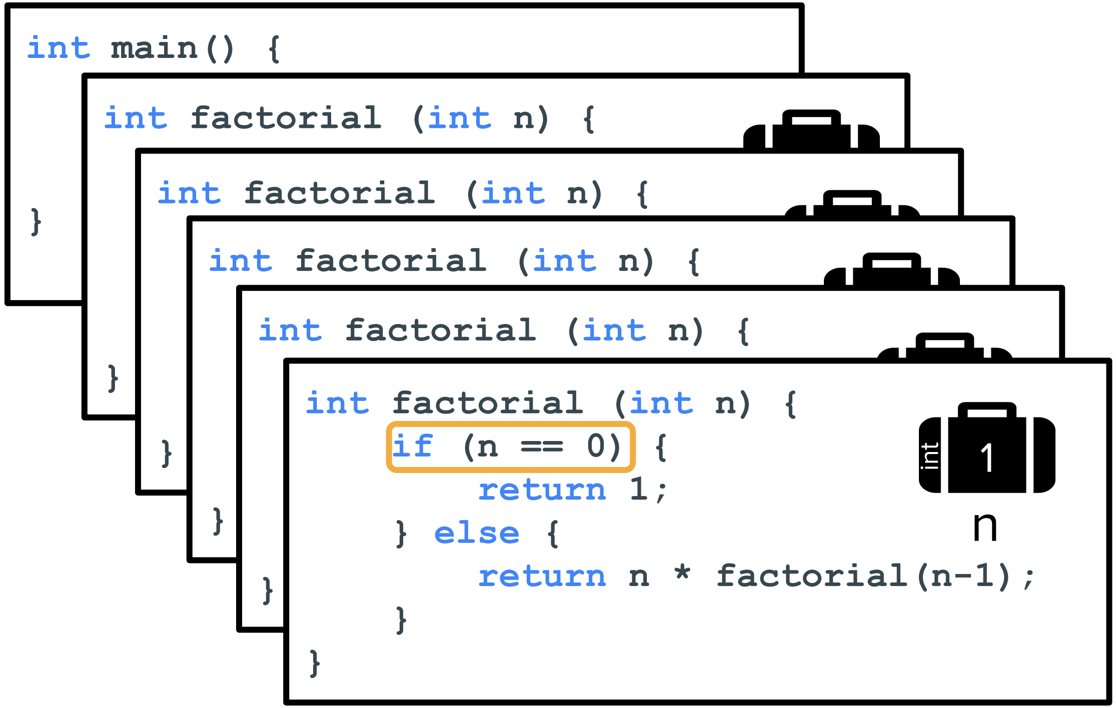
```
int factorial (int n) {
}
```

```
int factorial (int n) {
}
```

```
int factorial (int n) {
}
```

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

Recursion in action



```
int main() {

}
```

```
int factorial (int n) {

}
```

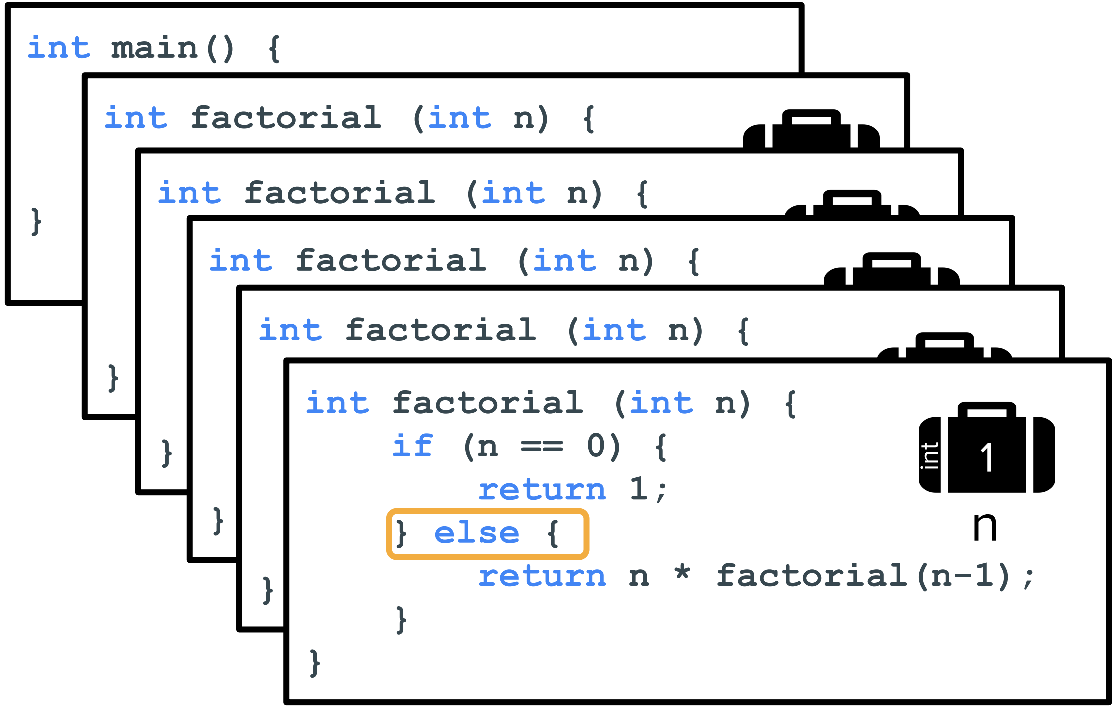
```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

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

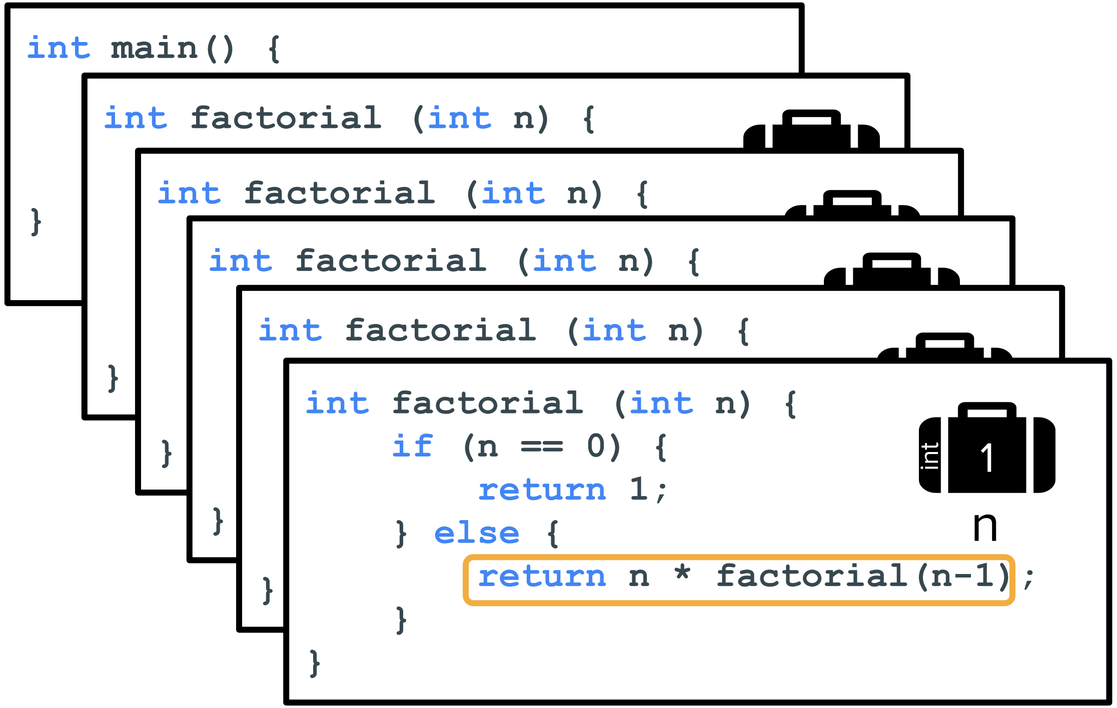
Recursion in action



```
int main() {

}
```

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

Recursion in action

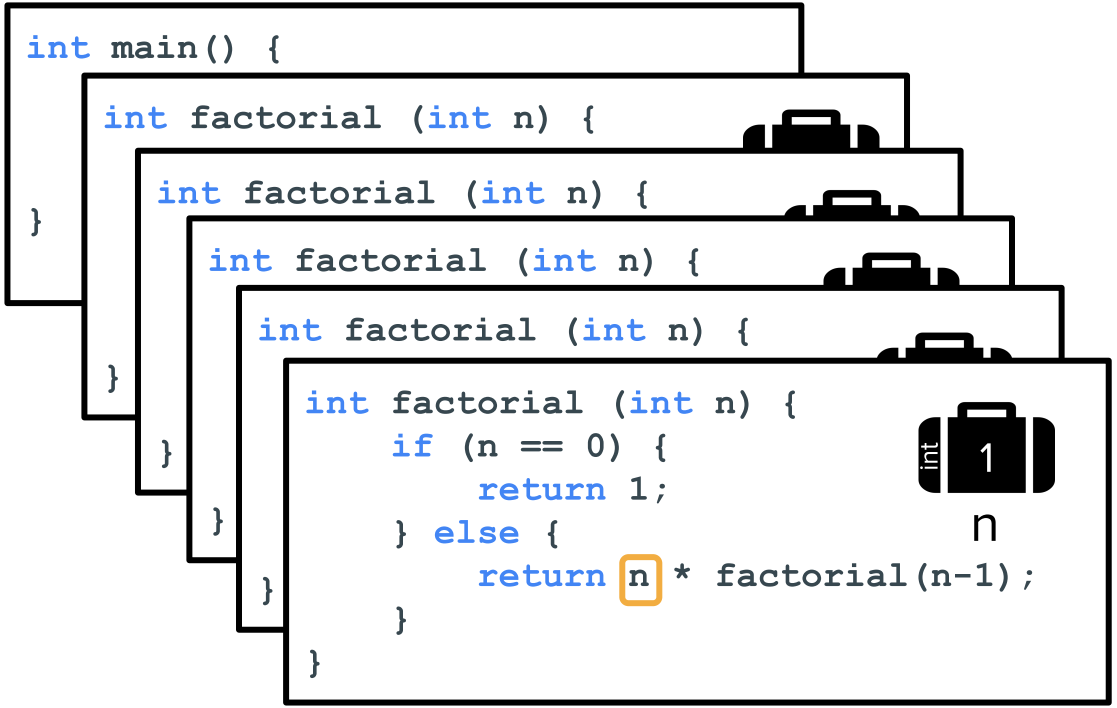


```
int main() {
}
```

```
int factorial (int n) {
}
```

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

Recursion in action



```
int main() {

}
```

```
int factorial (int n) {

}
```

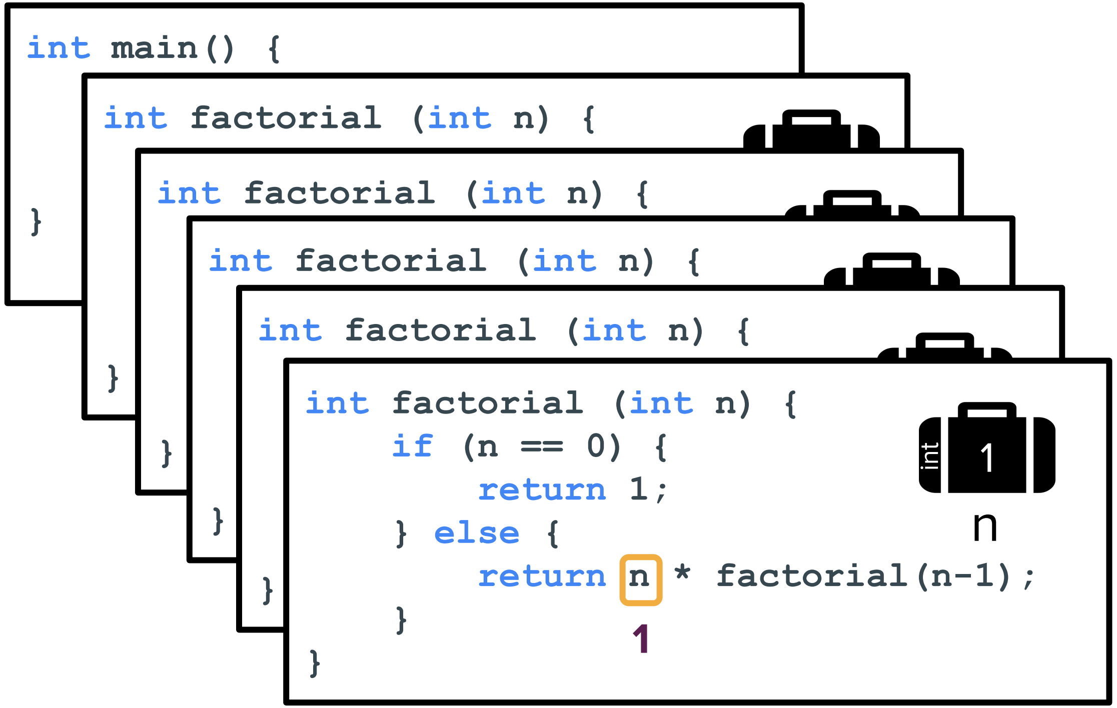
```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action



```
int main() {
}
```

```
int factorial (int n) {
}
```

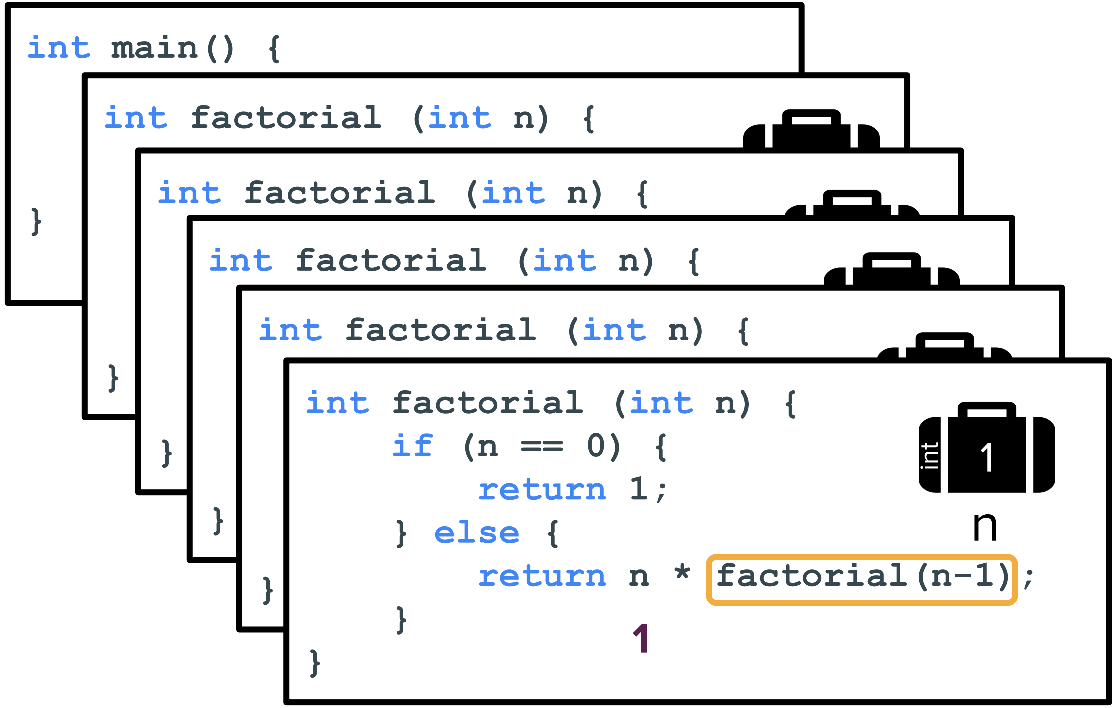
```
int factorial (int n) {
}
```

```
int factorial (int n) {
}
```

```
int factorial (int n) {
}
```

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

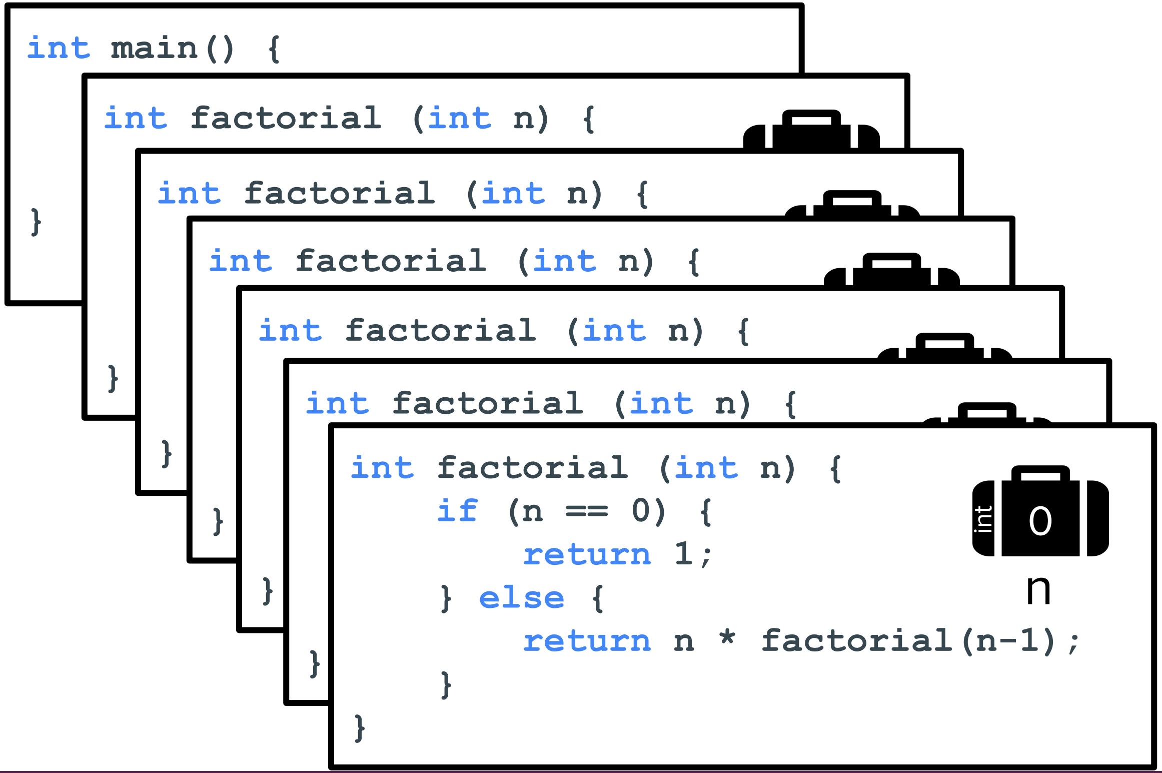
Recursion in action



```
int main() {

}
```

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

Recursion in action



```
int main() {
}
```

```
int factorial (int n) {
}
```

```
int factorial (int n) {
}
```

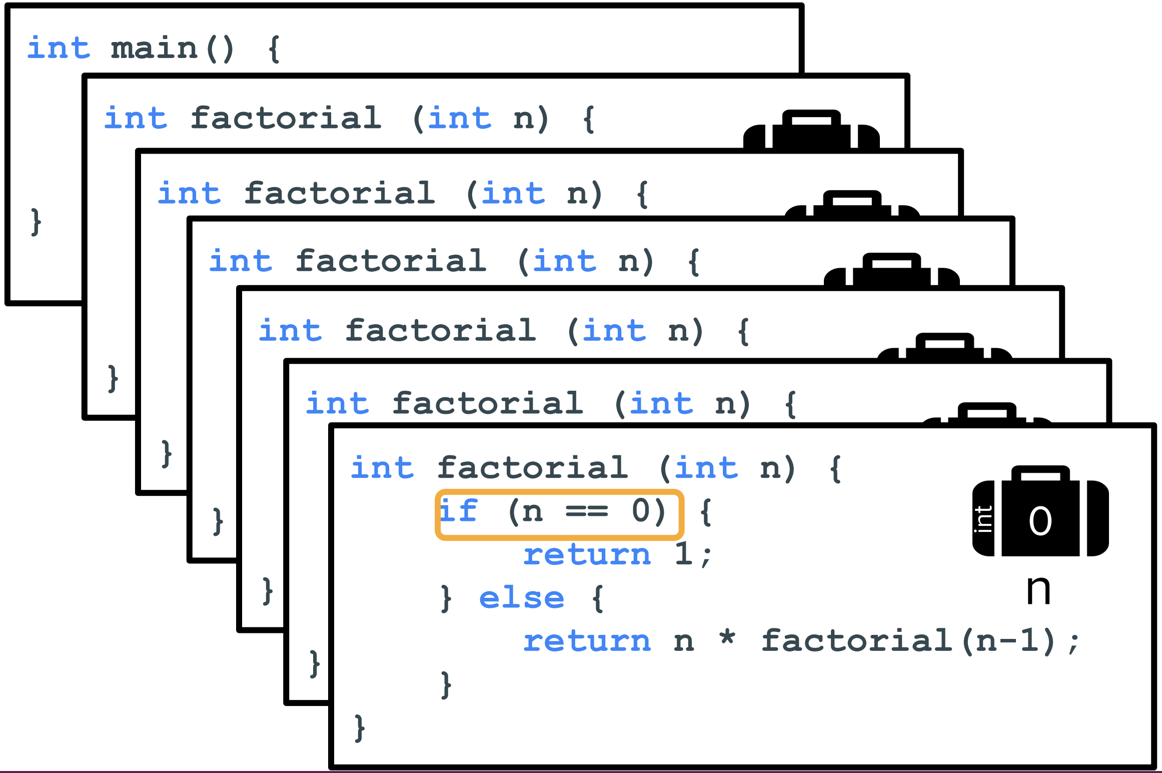
```
int factorial (int n) {
}
```

```
int factorial (int n) {
}
```

```
int factorial (int n) {
}
```

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

Recursion in action



```
int main() {
}
```

```
int factorial (int n) {
}
```

```
int factorial (int n) {
}
```

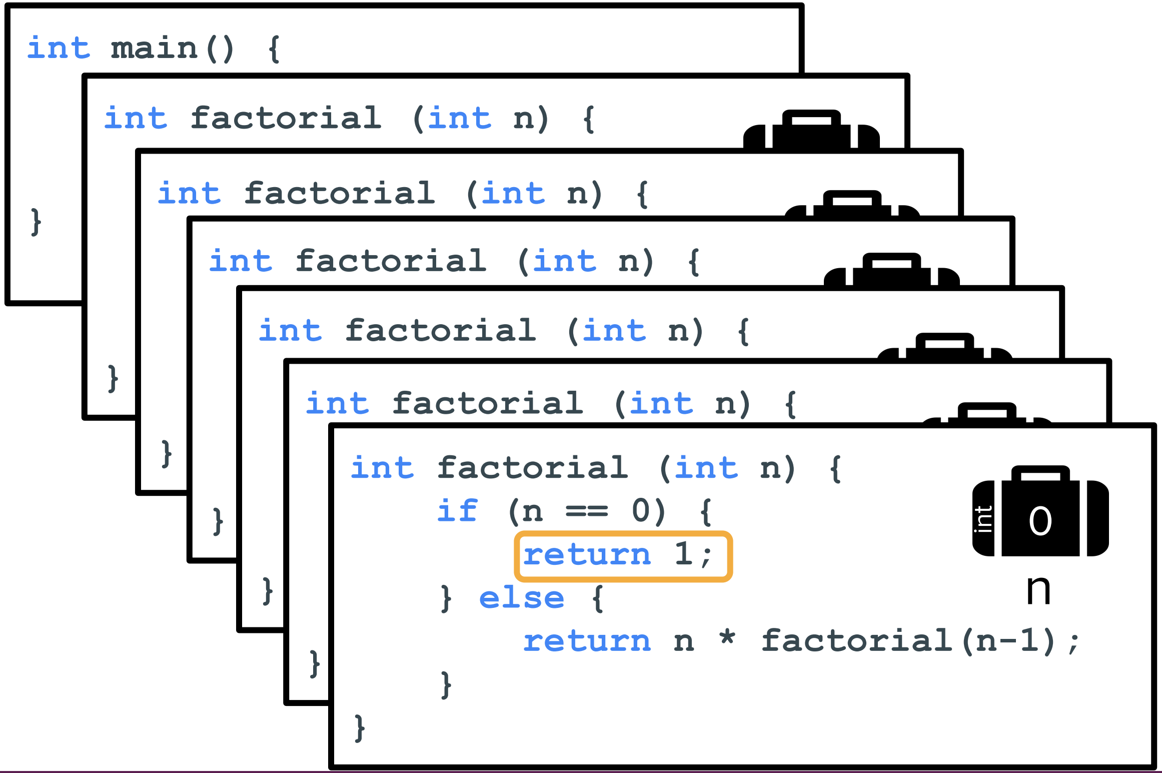
```
int factorial (int n) {
}
```

```
int factorial (int n) {
}
```

```
int factorial (int n) {
}
```

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

Recursion in action



```
int main() {
}
```

```
int factorial (int n) {
}
```

```
int factorial (int n) {
}
```

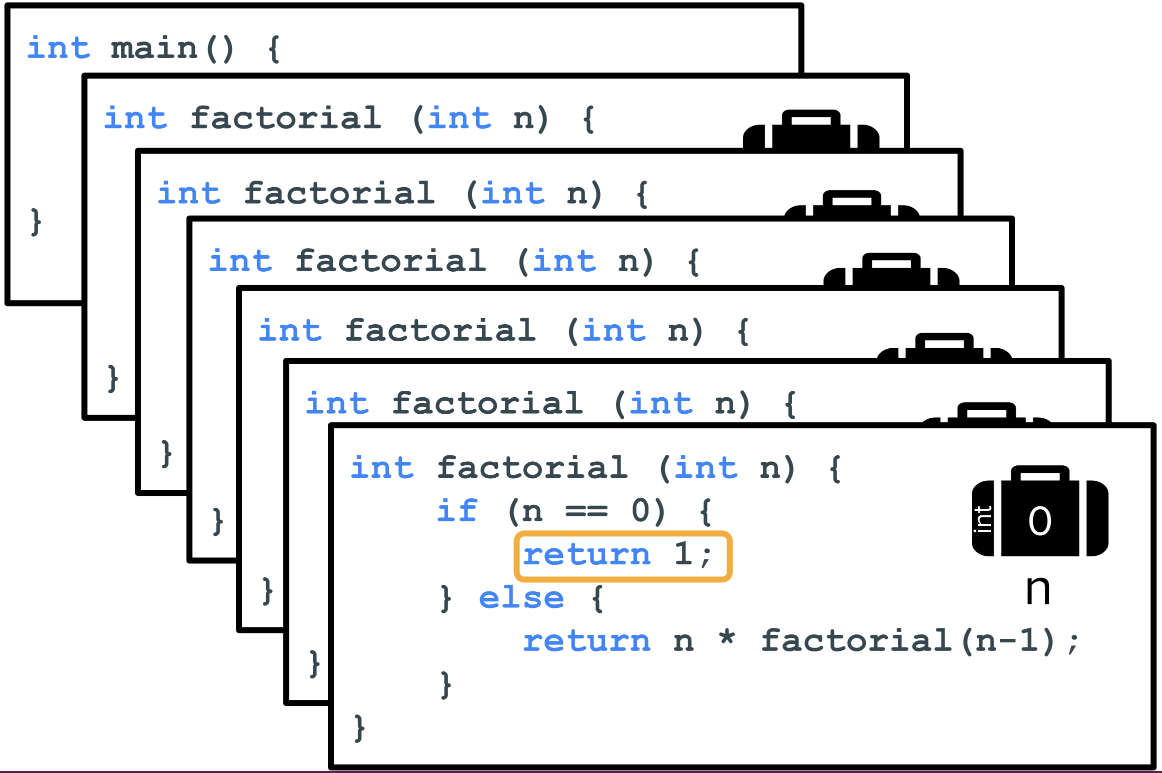
```
int factorial (int n) {
}
```

```
int factorial (int n) {
}
```

```
int factorial (int n) {
}
```

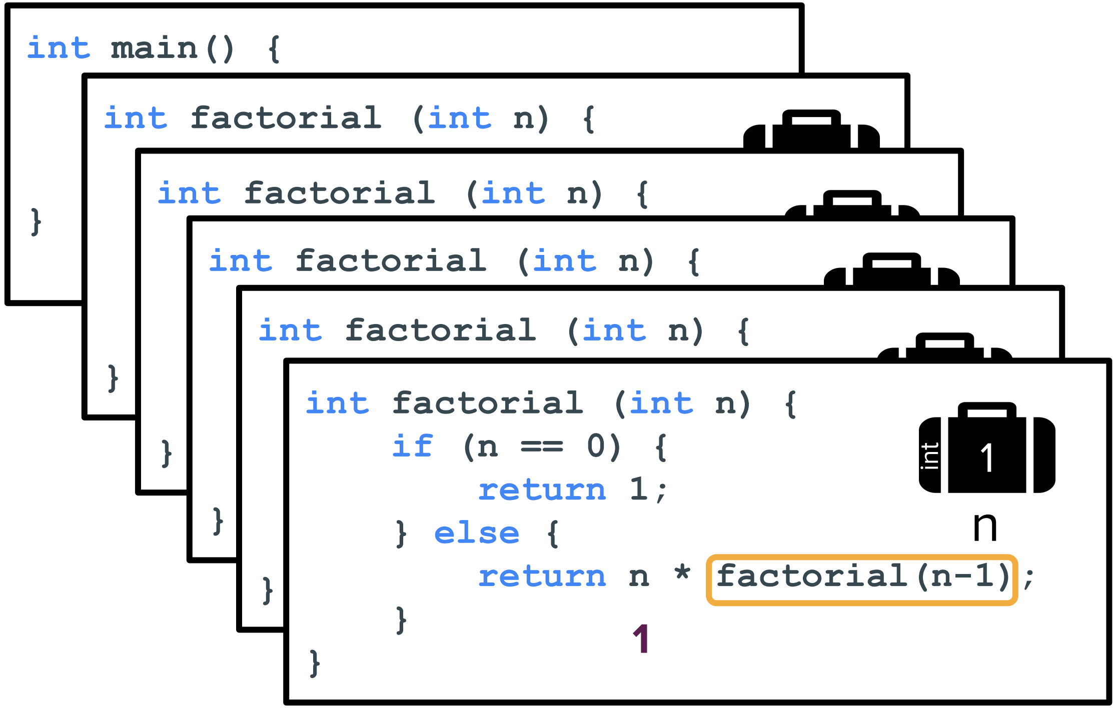
```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```

Recursion in action


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

Stack frames go away (get cleared from memory) once they return.

Recursion in action



```
int main() {

}
```

```
int factorial (int n) {

}
```

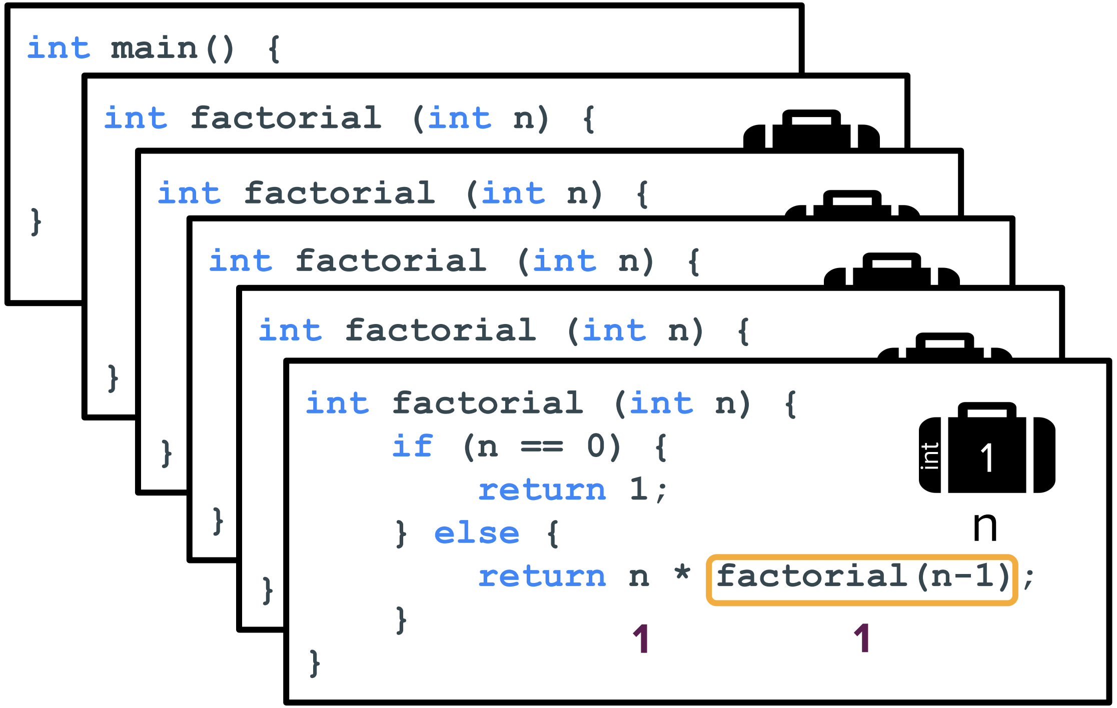
```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action



```
int main() {

}
```

```
int factorial (int n) {

}
```

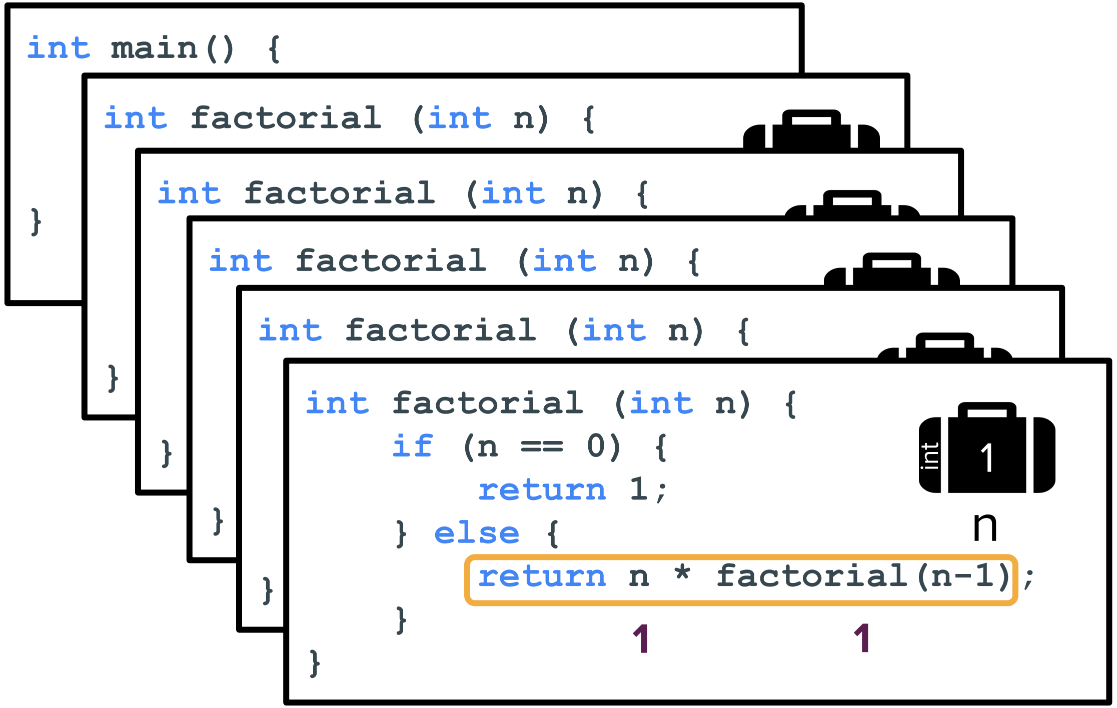
```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action



```
int main() {

}
```

```
int factorial (int n) {

}
```

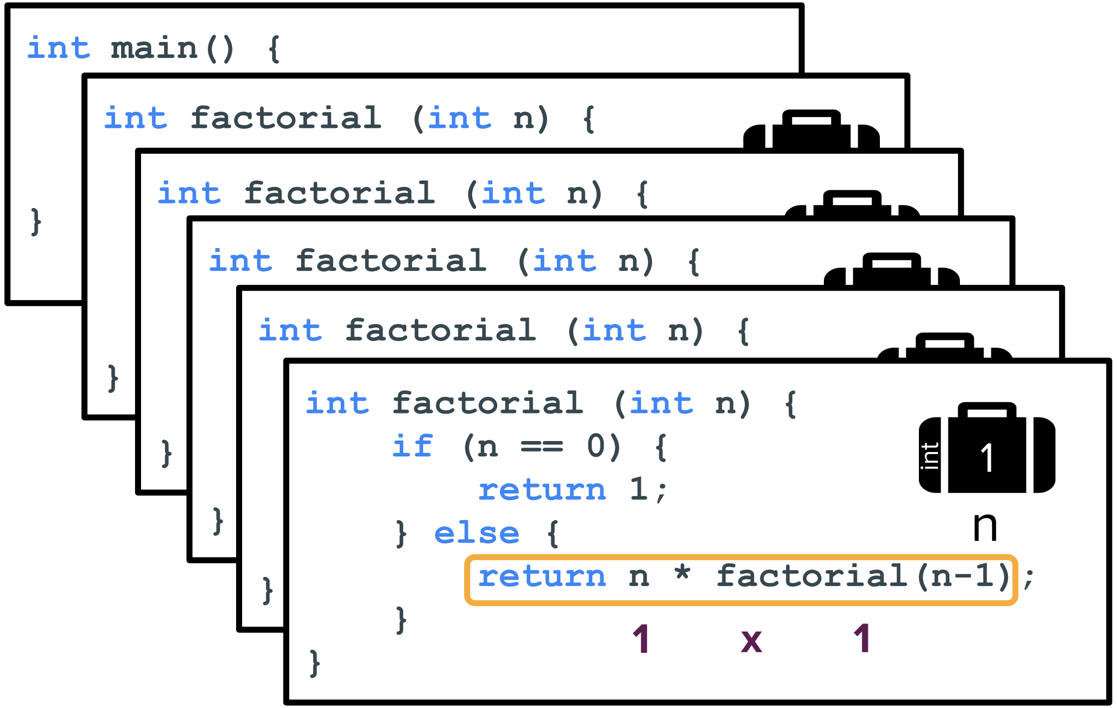
```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

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

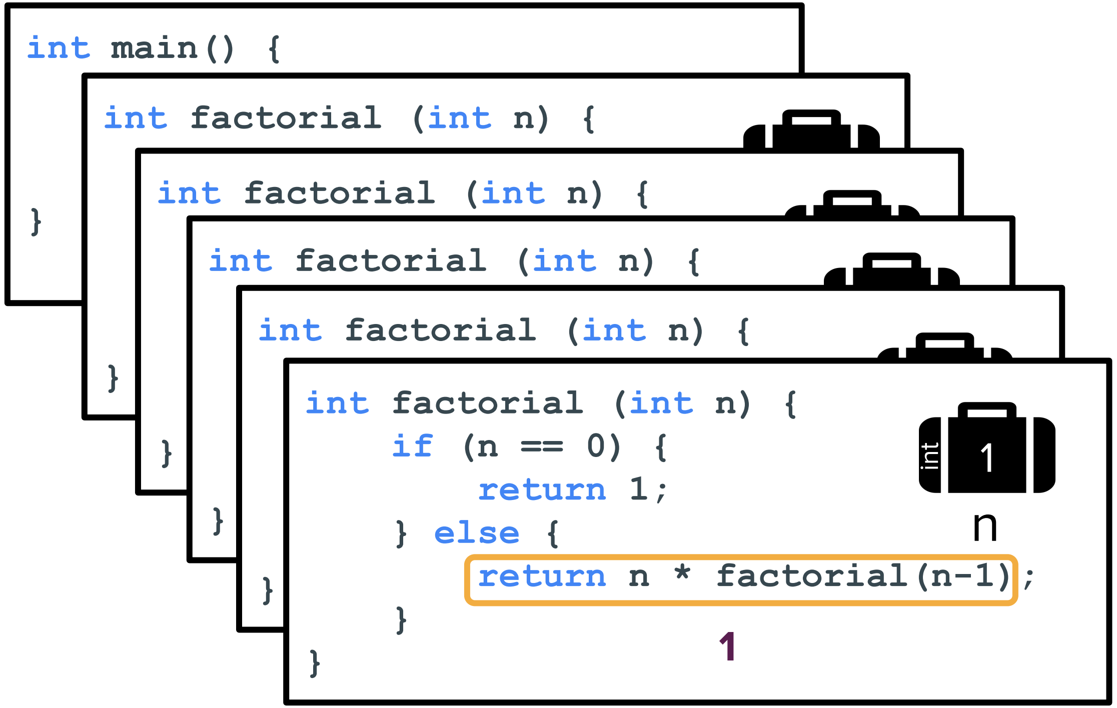
Recursion in action



```
int main() {

}
```

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

Recursion in action



```
int main() {
}
```

```
int factorial (int n) {
}
```

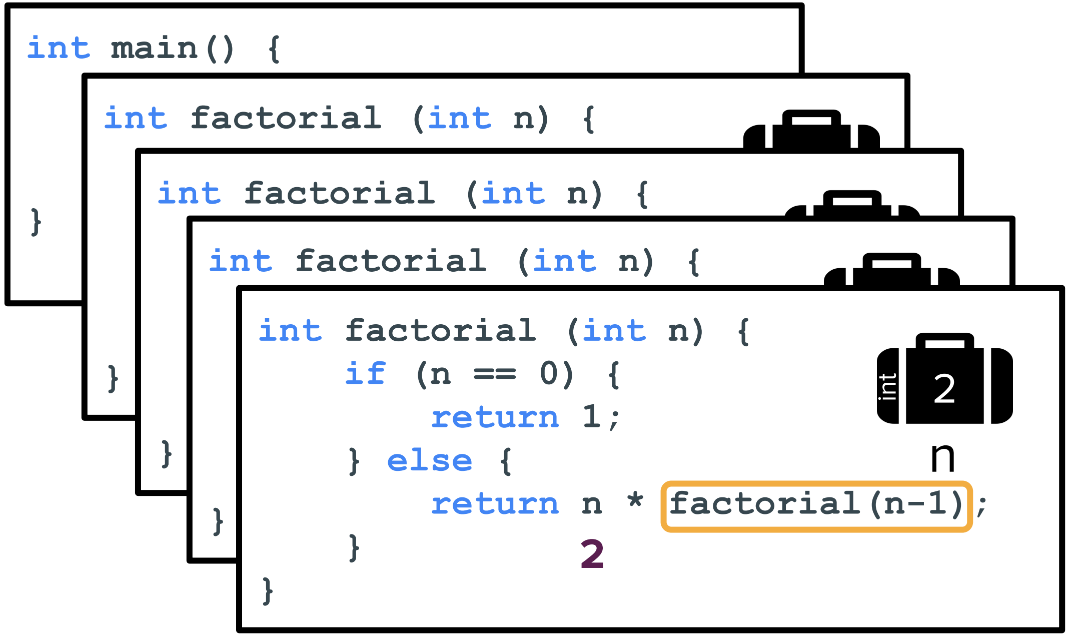
```
int factorial (int n) {
}
```

```
int factorial (int n) {
}
```

```
int factorial (int n) {
}
```

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

Recursion in action



```
int main() {

}
```

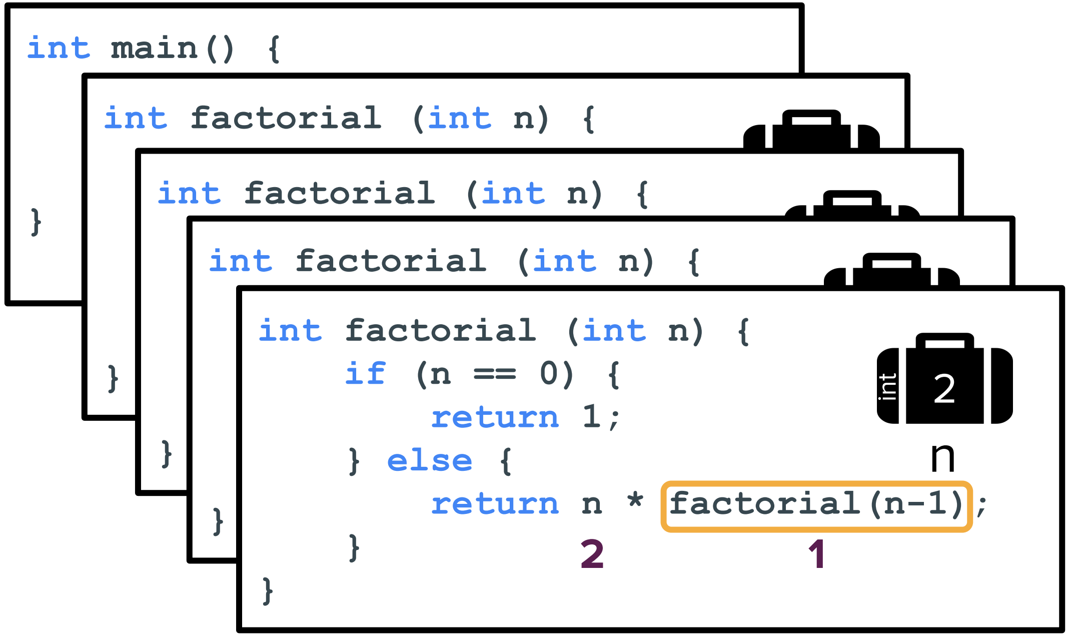
```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action



```
int main() {

}
```

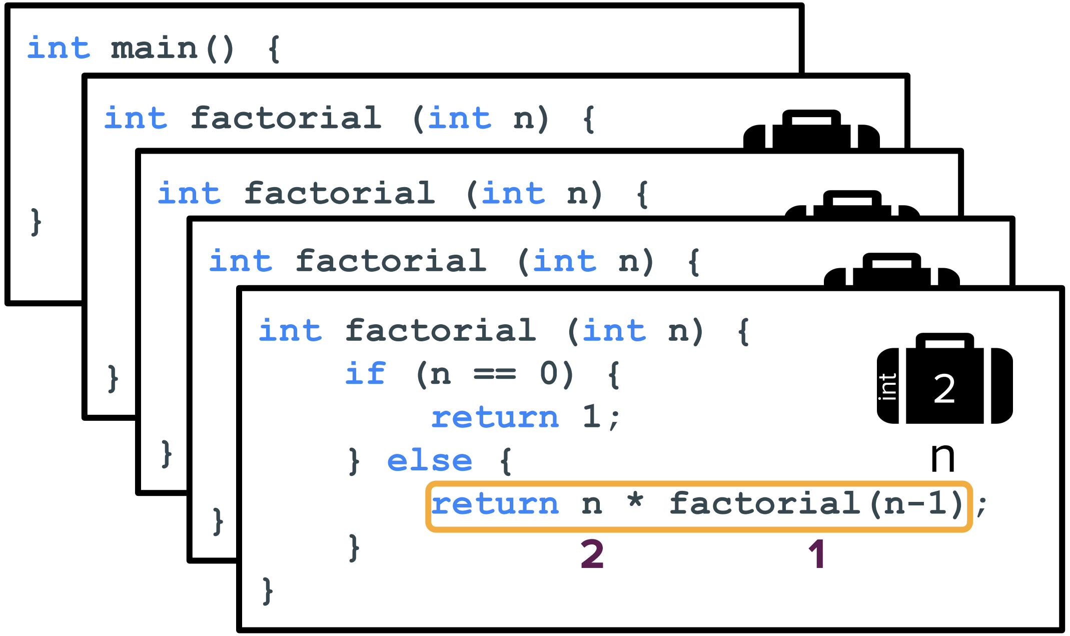
```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

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

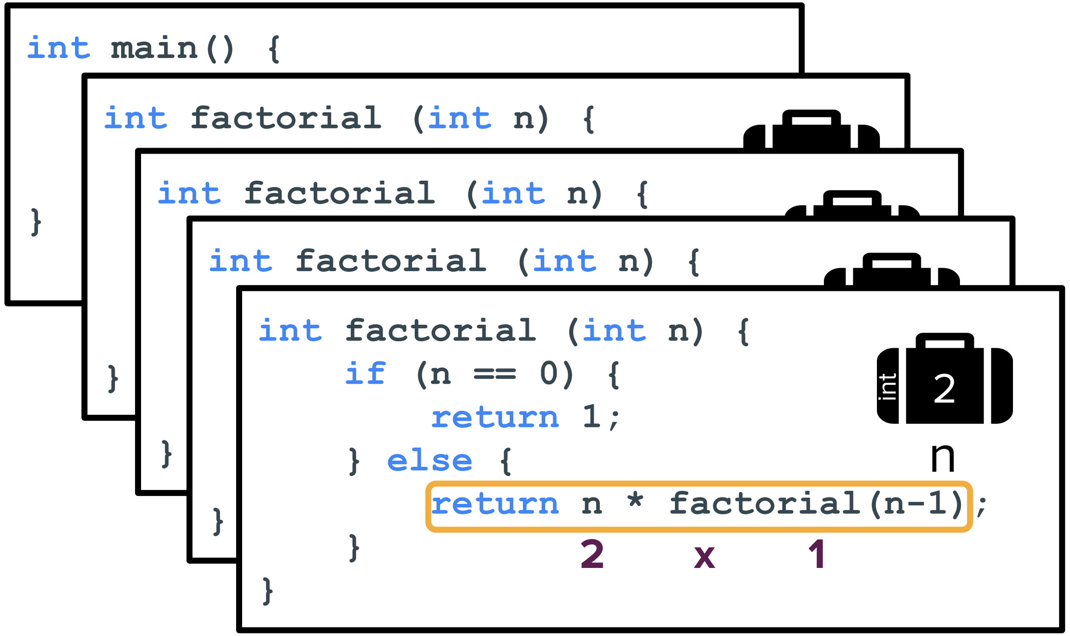
Recursion in action



```
int main() {

}
```

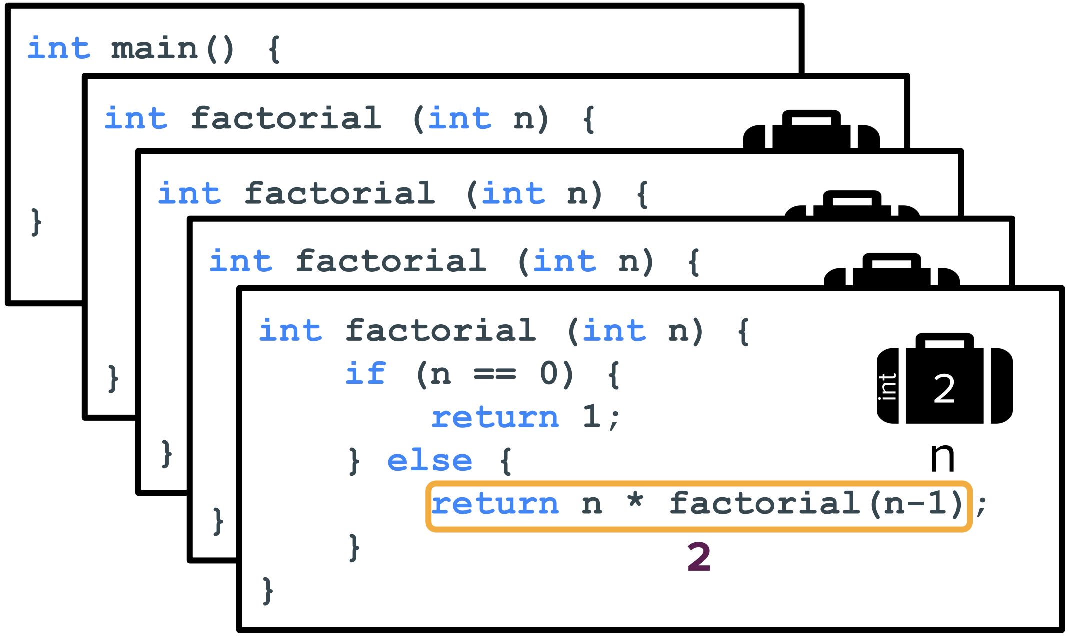
```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```

Recursion in action



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

Recursion in action



```
int main() {

}
```

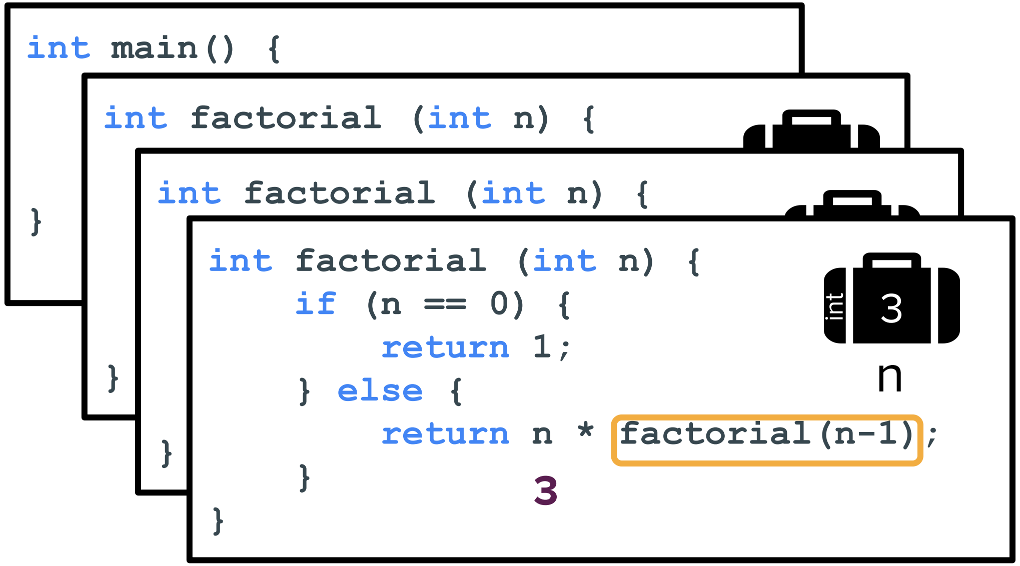
```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action



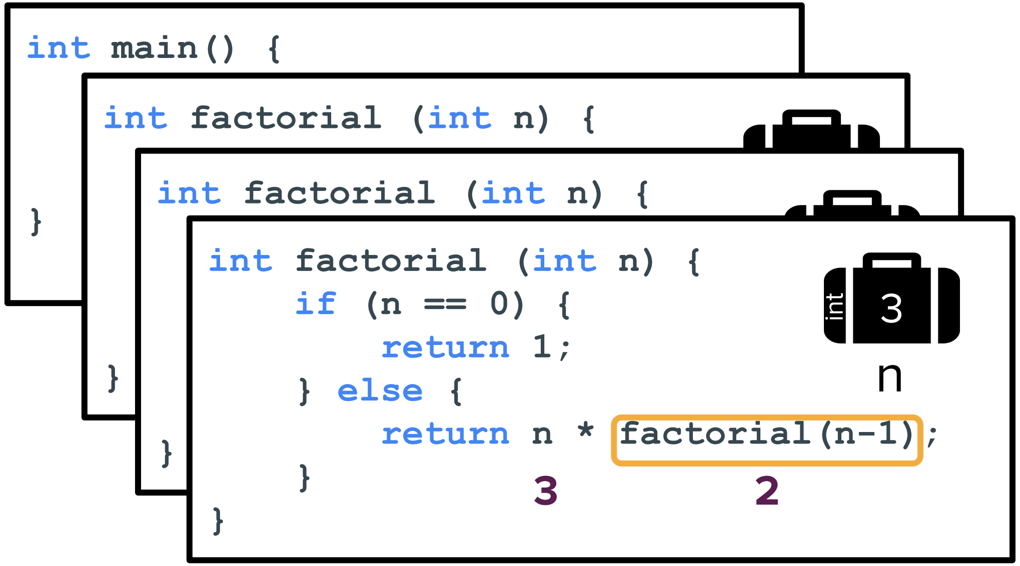
```
int main() {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action



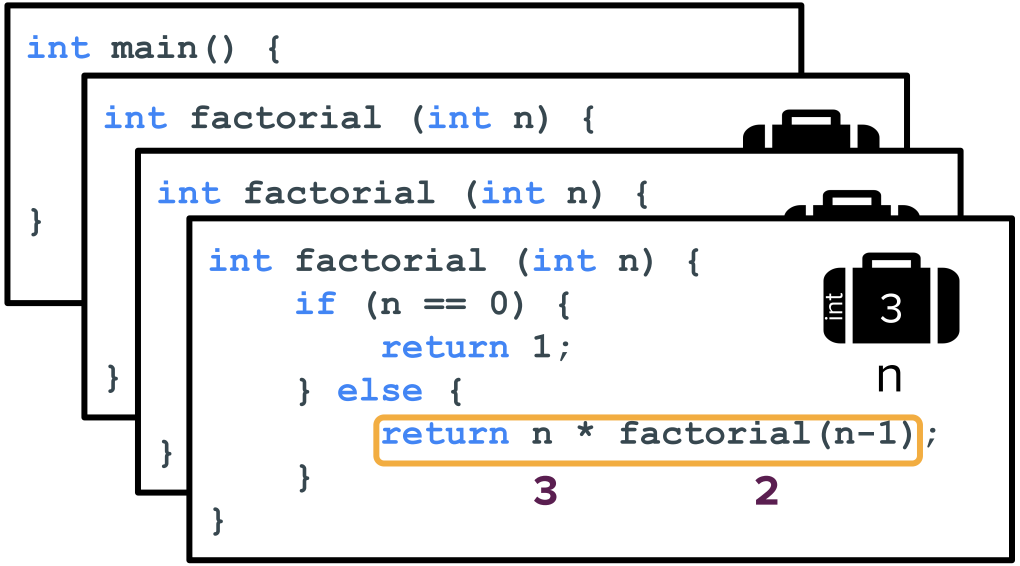
```
int main() {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action



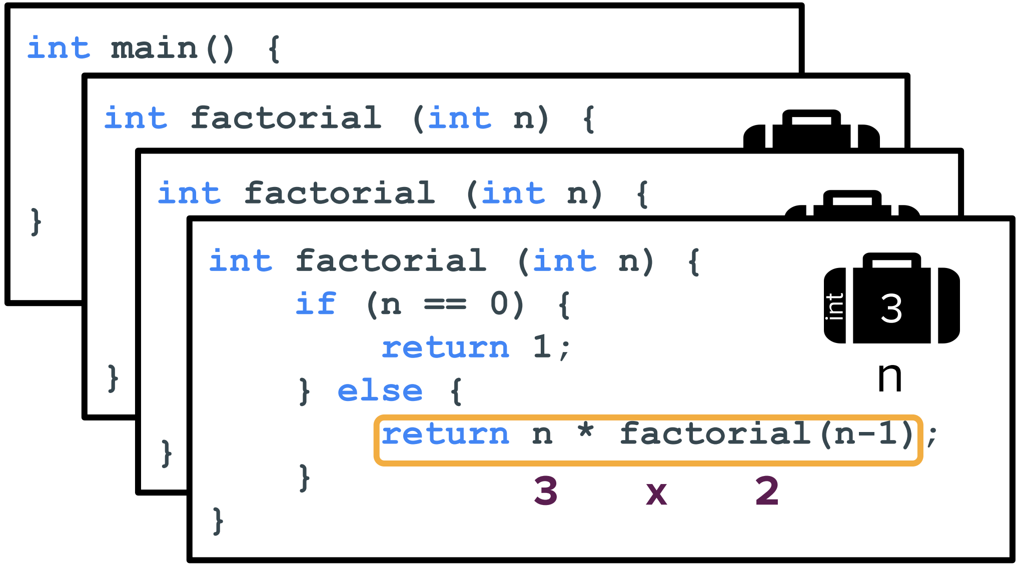
```
int main() {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action



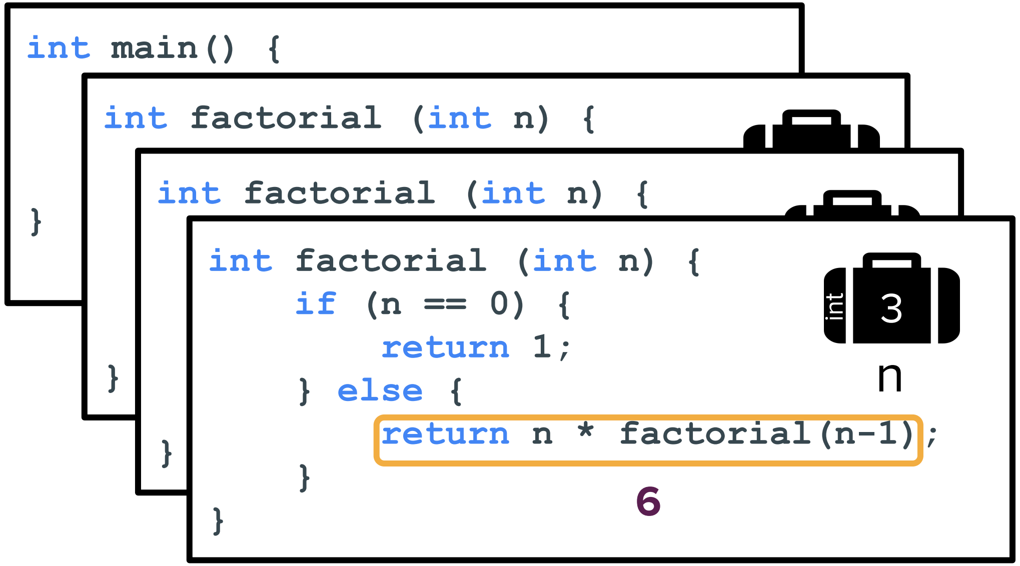
```
int main() {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
                3   x    2
    }
}
```

Recursion in action



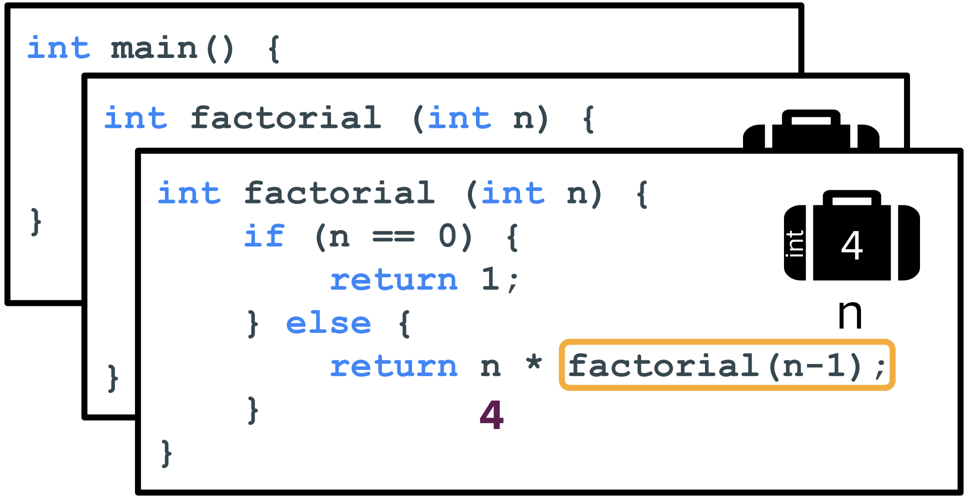
```
int main() {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action

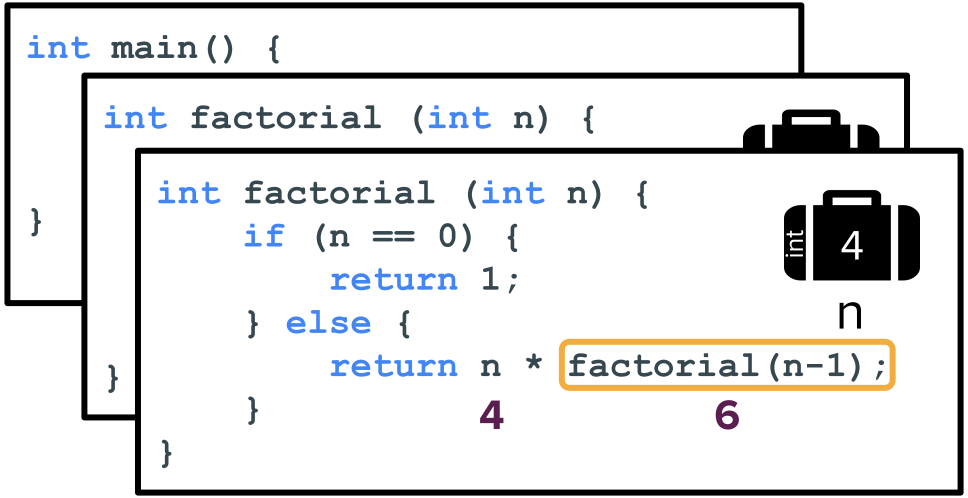


```
int main() {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action

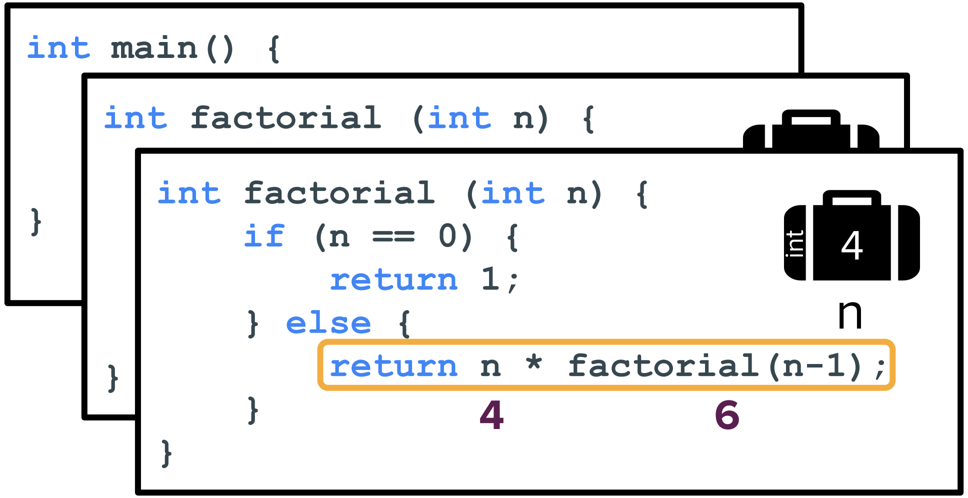


```
int main() {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action

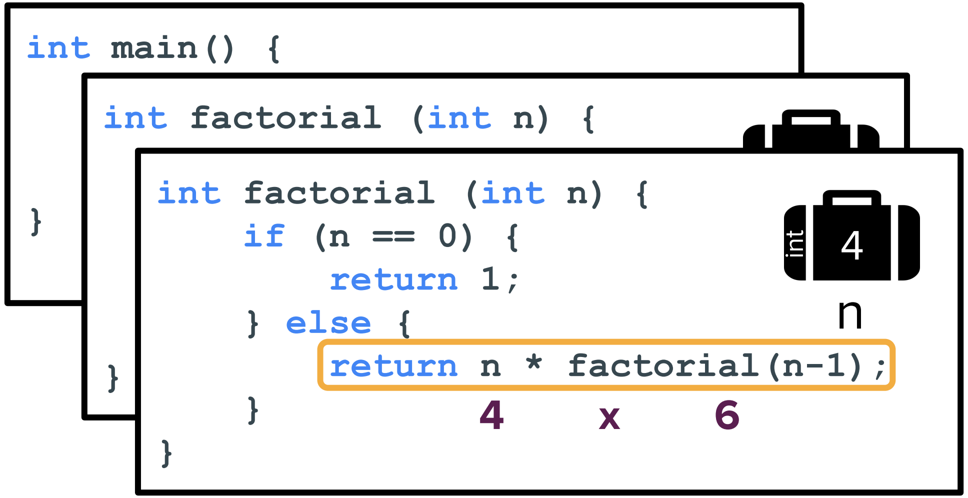


```
int main() {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action

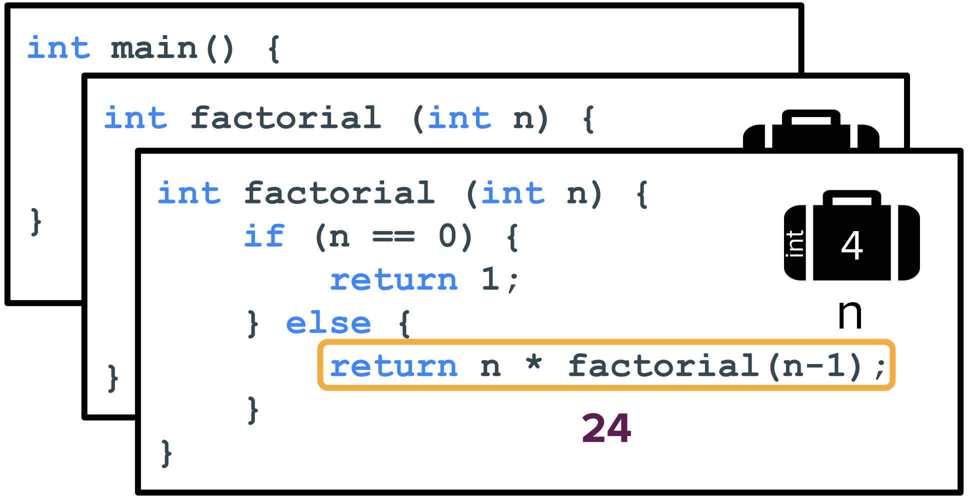


```
int main() {

}
```

```
int factorial (int n) {

}
```

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

Recursion in action



```
int main() {

}
```

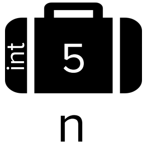
```
int factorial (int n) {

}
```

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

Recursion in action

```
int main() {

}
```

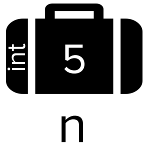
```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }           5
}
```



Recursion in action

```
int main() {

}
```

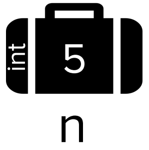
```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }          5         24
}
```



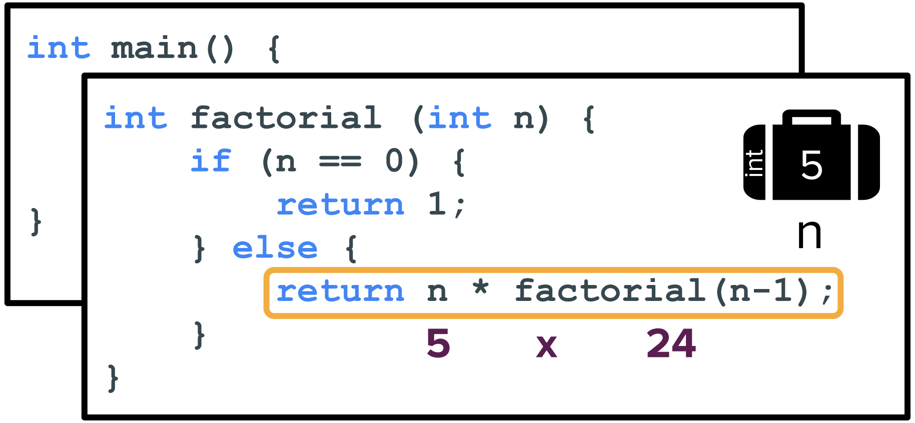
Recursion in action

```
int main() {


}
```

```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }              5          24
}
```



Recursion in action



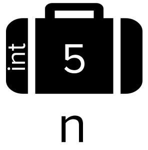
```
int main() {


}
```

```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
                 5   x     24
    }
}
```

Recursion in action

```
int main() {

}
```

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



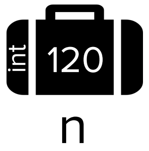
120

Recursion in action

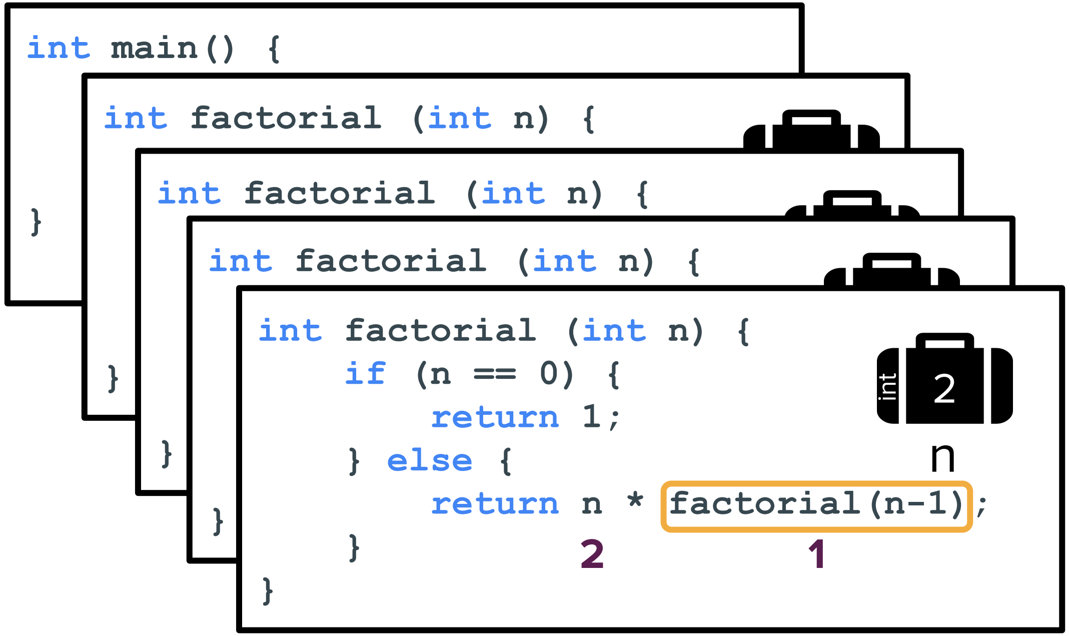
```
int main() {
    int n = factorial(5);
    cout << "5! = " << n << endl;
    return 0;
}
```

Recursion in action

```
int main() {
    int n = factorial(5);
    cout << "5! = " << n << endl;
    return 0;
}
```



Summary of Recursion in action



```
int main() {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

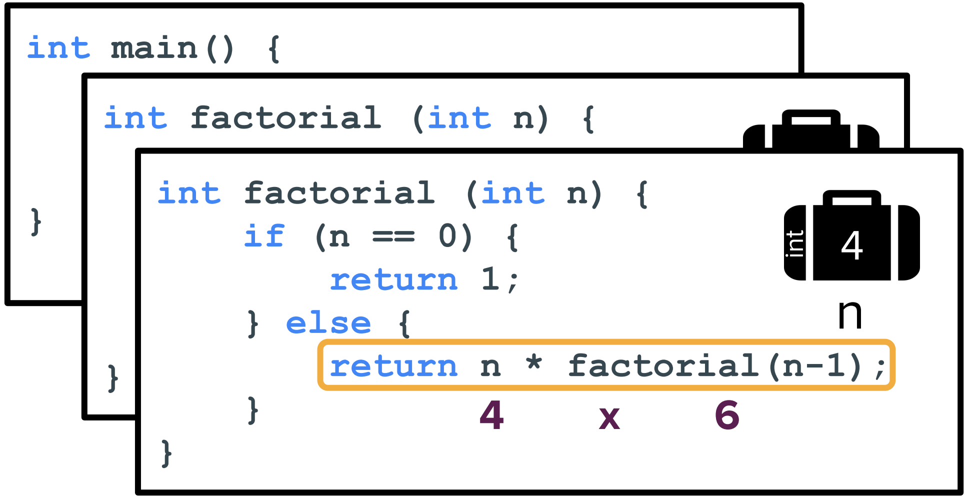
```
int factorial (int n) {

}
```

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

d

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

Recursion in action



```
int main() {
}
```

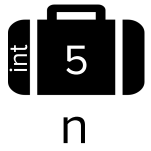
```
int factorial (int n) {
}
```

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

Recursion in action

```
int main() {


}
```

```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
                   5    x    24
    }
}
```



Recursive vs. Iterative

d

```
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;
}
```

Announcements

Announcements

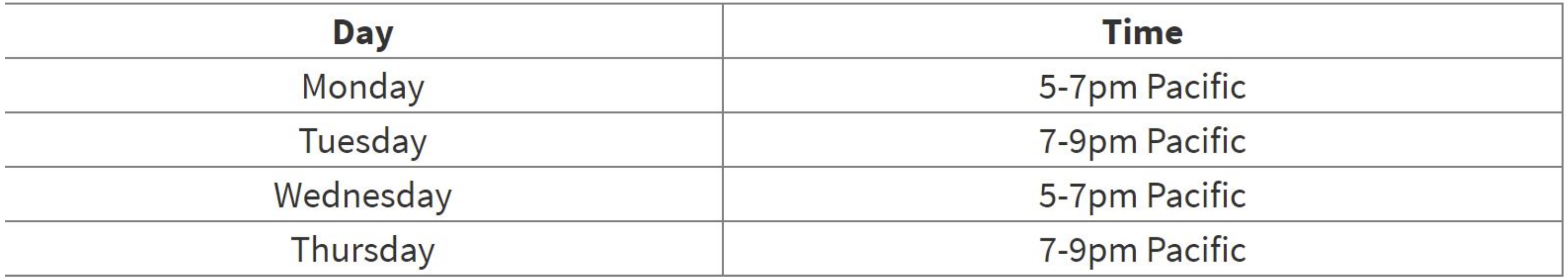
- Assignment 2 is due **Tomorrow**, 7/7 at 11:59pm PT. The grace period expires at the same time on **Friday.** After that, we will **not** be accepting submissions.

- Assignment 3 will be released by the end of the day on Thursday and will be due 8 days later on a Friday.

- The mid-quarter diagnostic will cover through the middle of next week (7/14 will be the last day of content covered).
    - We'll have practice problems ready by this weekend, with more next week.

Announcements II

- Here's the LaIR schedule for the quarter:



Day	Time
Monday	5-7pm Pacific
Tuesday	7-9pm Pacific
Wednesday	5-7pm Pacific
Thursday	7-9pm Pacific

- Recall that we have a special queue in the LaIR for conceptual questions. If you want to review lecture material, LaIR is a great place to get extra practice with concepts.

Reverse string example

How can we reverse a string?

Suppose we want to reverse strings like in the following examples:

“dog” ➜ “god”

“stressed” ➜ “desserts”

“recursion” ➜ “noisrucer”

“level” ➜ “level”

“a” ➜ “a”

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 (like the current value of **`n`** in each **`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)

How can we reverse a string?

- Look for self-similarity: **stressed → desserts**

How can we reverse a string?

- Look for self-similarity: **stressed → desserts**
    - What’s the first step you would take to reverse “stressed”?

How can we reverse a string?

- Look for self-similarity: **stressed → desserts**
    - Take the s and put it at the end of the string.

How can we reverse a string?

- Look for self-similarity: **stressed → desserts**
  - Take the s and put it at the end of the string.
  - Then reverse “tressed”

How can we reverse a string?

- Look for self-similarity: **stressed ➜ desserts**
    - Take the s and put it at the end of the string.
    - Then reverse “tressed”:
        - Take the t and put it at the end of the string.
        - Then reverse “ressed”

How can we reverse a string?

- Look for self-similarity: **stressed ➔ desserts**
    - Take the s and put it at the end of the string.
    - Then reverse “tressed”:
        - Take the t and put it at the end of the string.
        - Then reverse “ressed”:
            - Take the r and put it at the end of the string.
            - Then reverse “essed”

How can we reverse a string?

- Look for self-similarity: **stressed ➔ desserts**
    - Take the s and put it at the end of the string.
    - Then reverse “tressed”:
        - Take the t and put it at the end of the string.
        - Then reverse “ressed”:
            - Take the r and put it at the end of the string.
            - Then reverse “essed”:
                - …
                    - Take the d and put it at the end of the string.
                    - **Base case**: reverse “” ➔ get “”

How can we reverse a string?

- Look for self-similarity: **stressed ➔ desserts**
    - Take the s and put it at the end of the string.
    - Then reverse “tressed”:
        - Take the t and put it at the end of the string.
        - Then reverse “ressed”:
            - Take the r and put it at the end of the string.
            - Then reverse “essed”:
                - …
                    - Take the d and put it at the end of the string.
                    - **Base case**: reverse “” ➔ get “”

How can we express the recursive case?

How can we reverse a string?

- Look for self-similarity: **stressed ➜ desserts**
  - **Take the s and put it at the end of the string.**
  - **Then reverse “tressed”:**
    - Take the t and put it at the end of the string.
    - Then reverse “ressed”:
      - Take the r and put it at the end of the string.
      - Then reverse “essed”:
        - …
          - Take the d and put it at the end of the string.
          - **Base case**: reverse “” ➜ get “”

How can we express the recursive case?

How can we reverse a string?

- Look for self-similarity: **stressed ➔ desserts**
    - **reverse("stressed") = reverse("tressed") + 's'**
        - Take the t and put it at the end of the string.
        - Then reverse "ressed":
            - Take the r and put it at the end of the string.
            - Then reverse "essed":
                - ...
                    - Take the d and put it at the end of the string.
                    - **Base case**: reverse "" ➔ get ""

How can we express the recursive case?

How can we reverse a string?

- Look for self-similarity: **stressed → desserts**
  - reverse("stressed") = reverse("tressed") + 's'
    - **Take the t and put it at the end of the string.**
    - **Then reverse "ressed":**
      - Take the r and put it at the end of the string.
      - Then reverse "essed":
        - …
          - Take the d and put it at the end of the string.
          - **Base case**: reverse "" → get ""

How can we express the recursive case?

How can we reverse a string?

- Look for self-similarity: **stressed → desserts**
  - reverse(“stressed”) = reverse(“tressed”) + ‘s’
    - **reverse(“tressed”) = reverse(“ressed”) + ‘t’**
      - Take the r and put it at the end of the string.
      - Then reverse “essed”:
        - …
          - Take the d and put it at the end of the string.
          - **Base case**: reverse “” → get “”

How can we express the recursive case?

How can we reverse a string?

How can we express the recursive case?

- Look for self-similarity: **stressed → desserts**
  - reverse(“stressed”) = reverse(“tressed”) + ‘s’
    - reverse(“tressed”) = reverse(“ressed”) + ‘t’
      - **Take the r and put it at the end of the string.**
      - **Then reverse “essed”:**
        - …
          - Take the d and put it at the end of the string.
          - **Base case**: reverse “” → get “”

How can we reverse a string?

- Look for self-similarity: **stressed → desserts**
  - reverse(“stressed”) = reverse(“tressed”) + ‘s’
    - reverse(“tressed”) = reverse(“ressed”) + ‘t’
      - **reverse(“ressed”) = reverse(“essed”) + ‘r’**
        - …
          - Take the d and put it at the end of the string.
          - **Base case**: reverse “” → get “”

How can we express the recursive case?

How can we reverse a string?

- Look for self-similarity: **stressed → desserts**
    - reverse(“stressed”) = reverse(“tressed”) + ‘s’
        - reverse(“tressed”) = reverse(“ressed”) + ‘t’
            - reverse(“ressed”) = reverse(“essed”) + ‘r’
                - …
                    - **Base case**: reverse(“”) = “”

How can we reverse a string?

- **Recursive case:** reverse(str) = reverse(str without first letter) + first letter of str
- **Base case**: reverse(“”) = “”

How can we reverse a string?

- **Recursive case:** reverse(str) = reverse(str without first letter) + first letter of str
- **Base case**: reverse(“”) = “”

Depending on how you thought of the problem, you may have also come up with:

- **Recursive case:** reverse(str) = last letter of str + reverse(str without last letter)
- **Base case**: reverse(“”) = “”

Let’s code it!

(live coding)

Summary

Summary

- 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).

Summary

- 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.

Summary

- 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.

Summary

- 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**.

Summary

- 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**.

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.

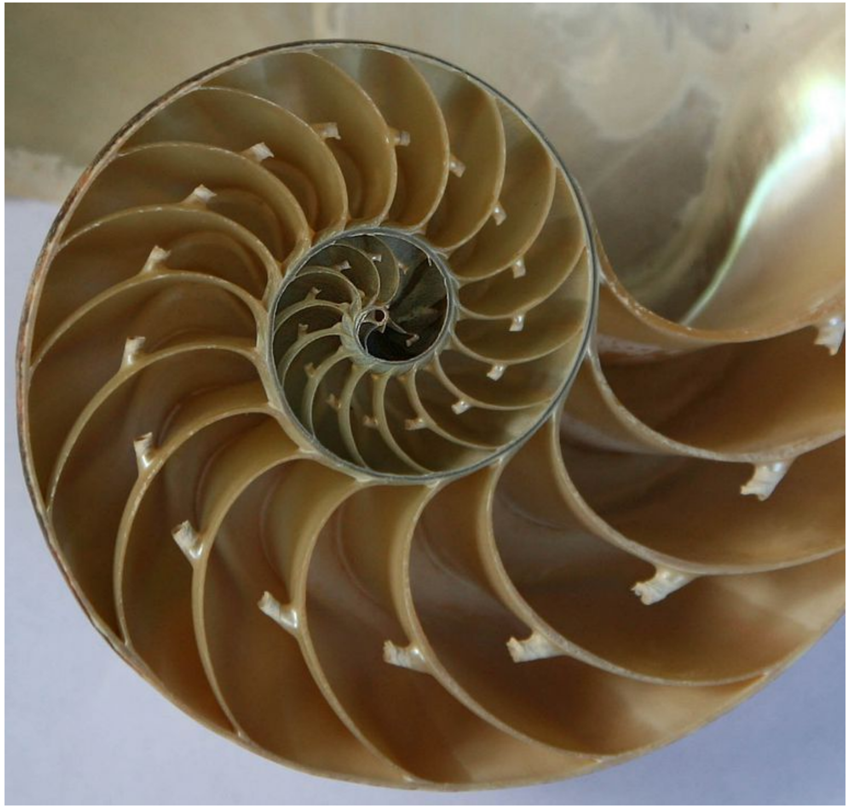
Self-Similarity

- 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

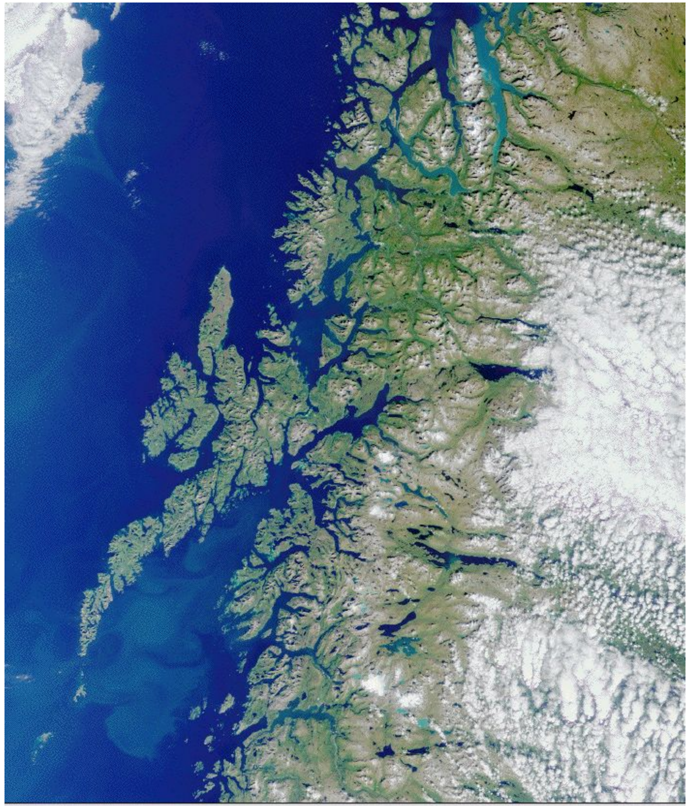
- Solving problems recursively and analyzing recursive phenomena involves identifying **self-similarity**

- 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.

Fractals

Fractals

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

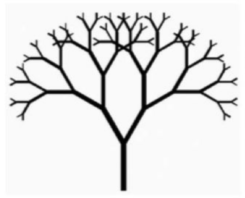
Fractals

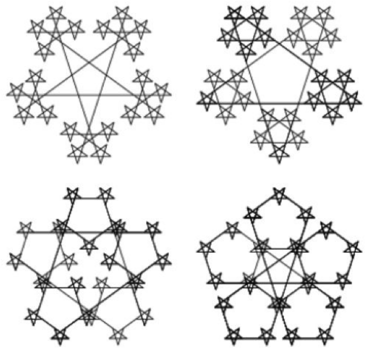
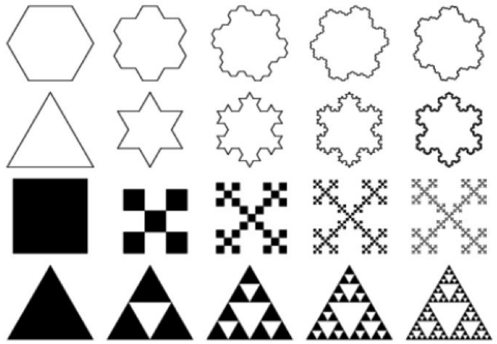
- A **fractal** is any repeated, graphical pattern.

- 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.

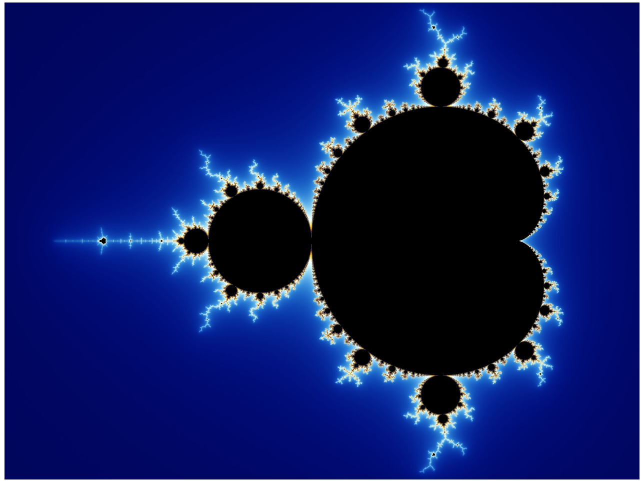


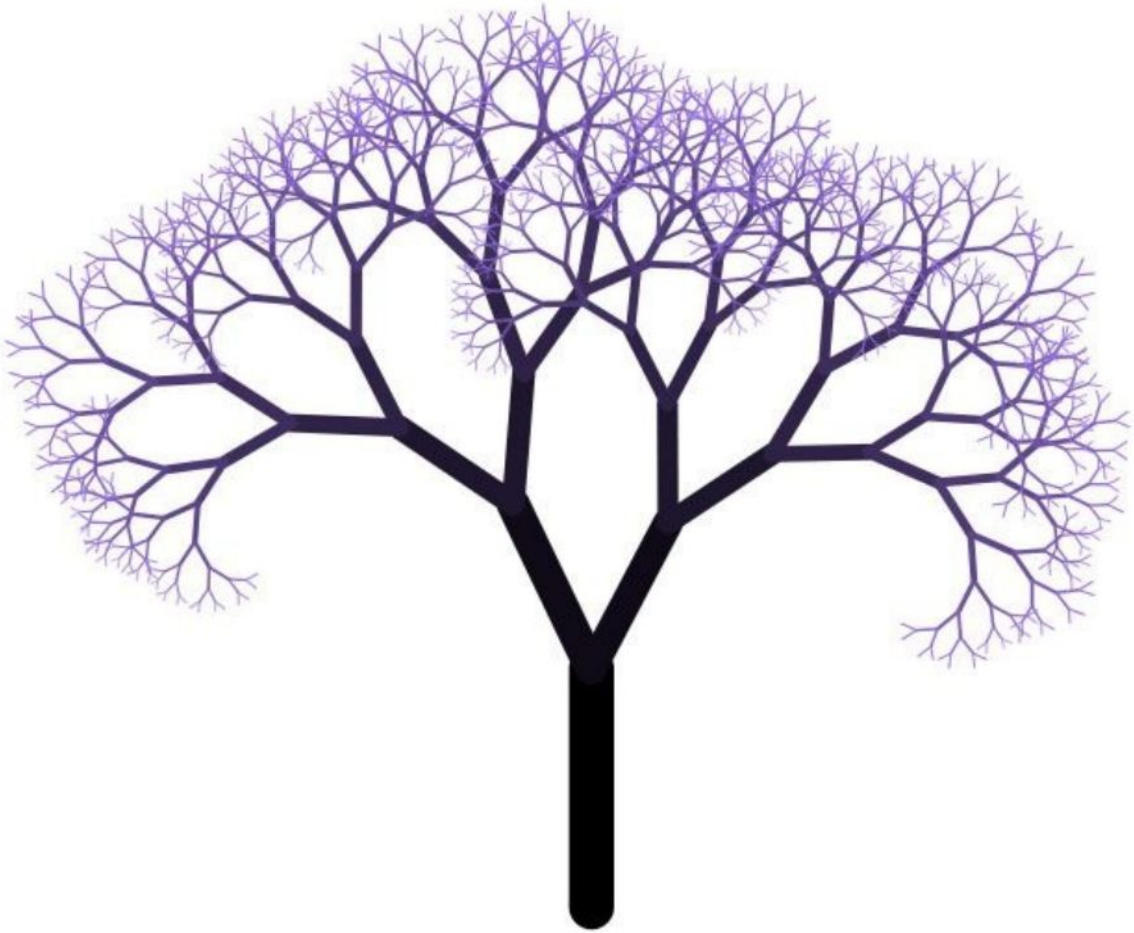
Fractals

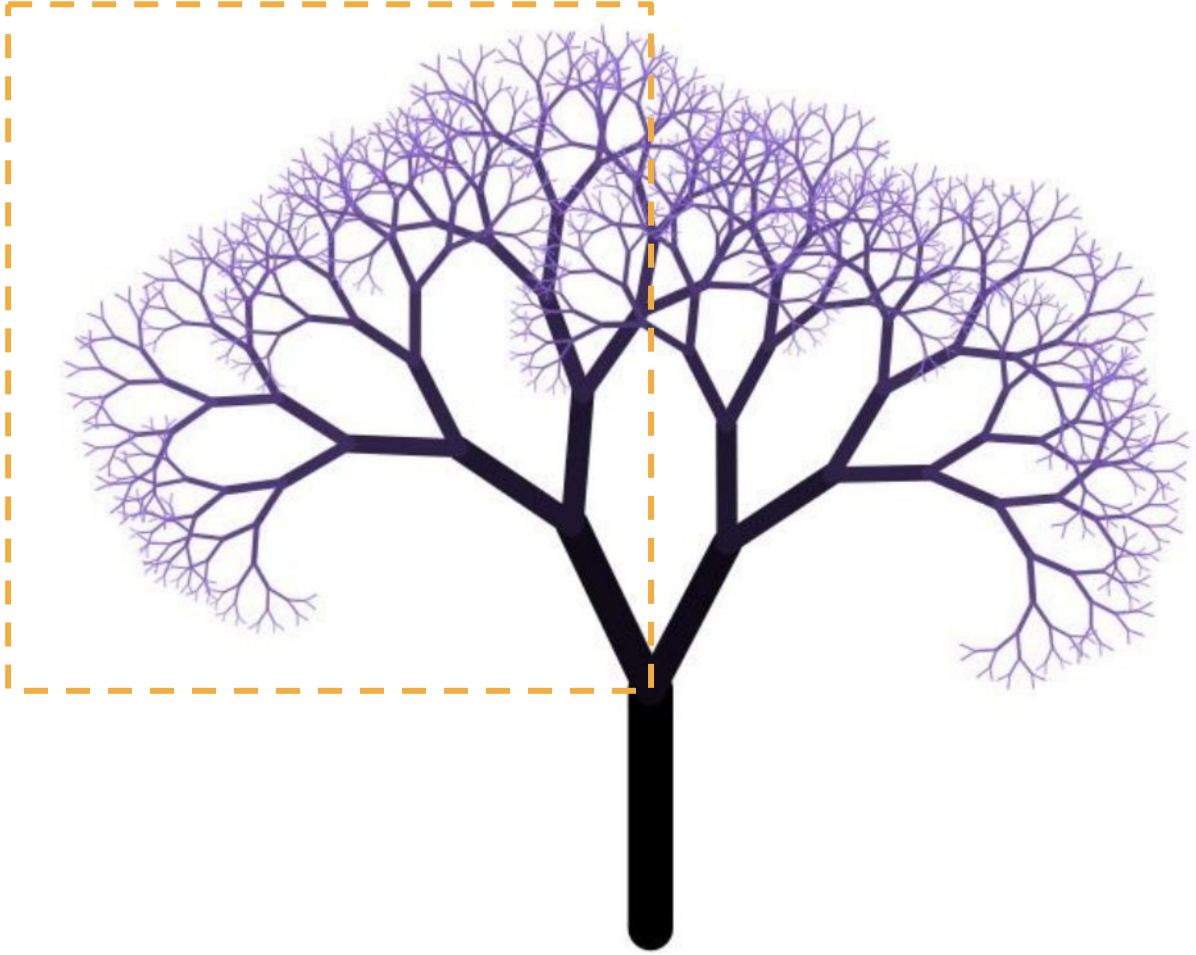
- A **fractal** is any repeated, graphical pattern.

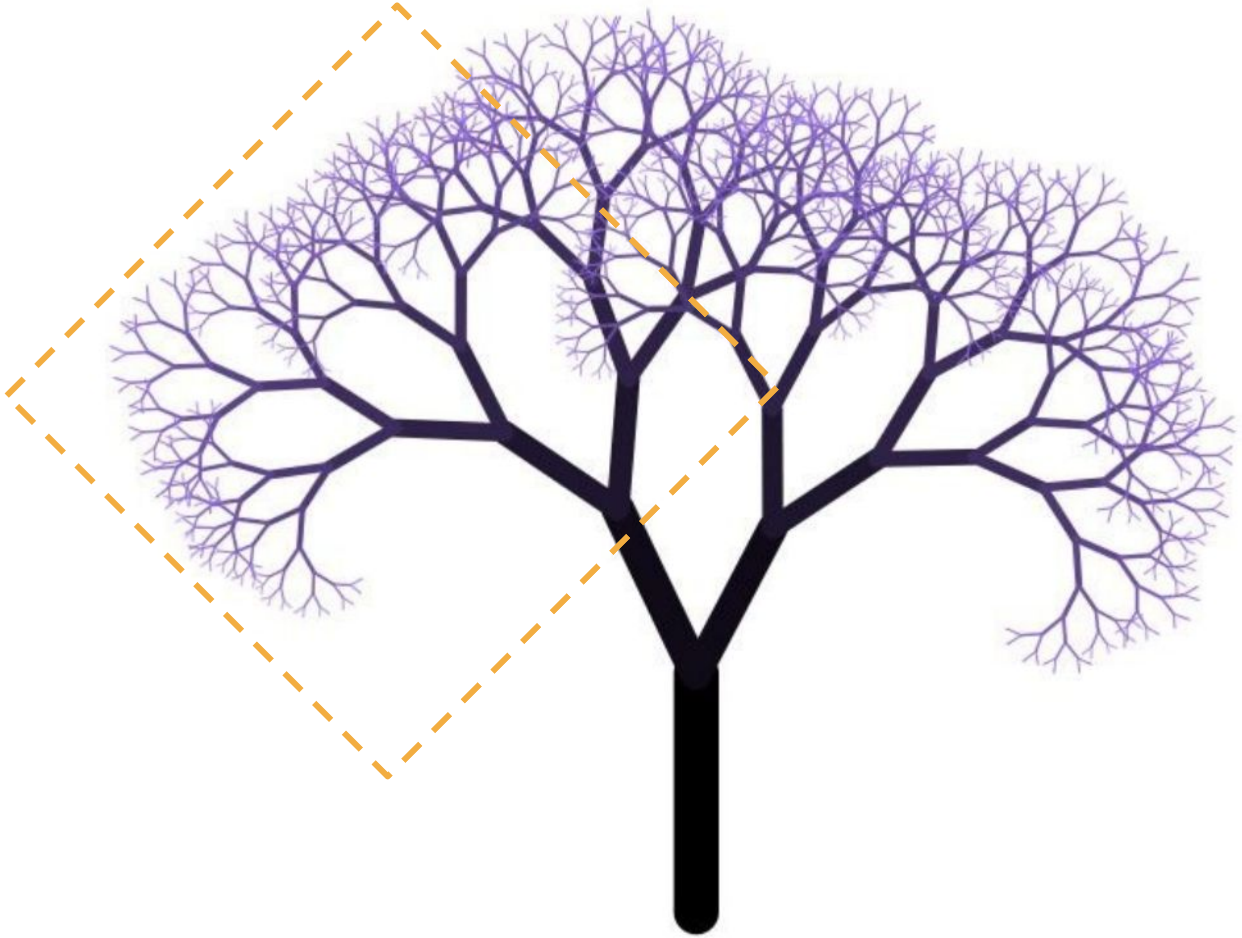
- A fractal is composed of **repeated instances of the same shape or pattern**, arranged in a structured way.

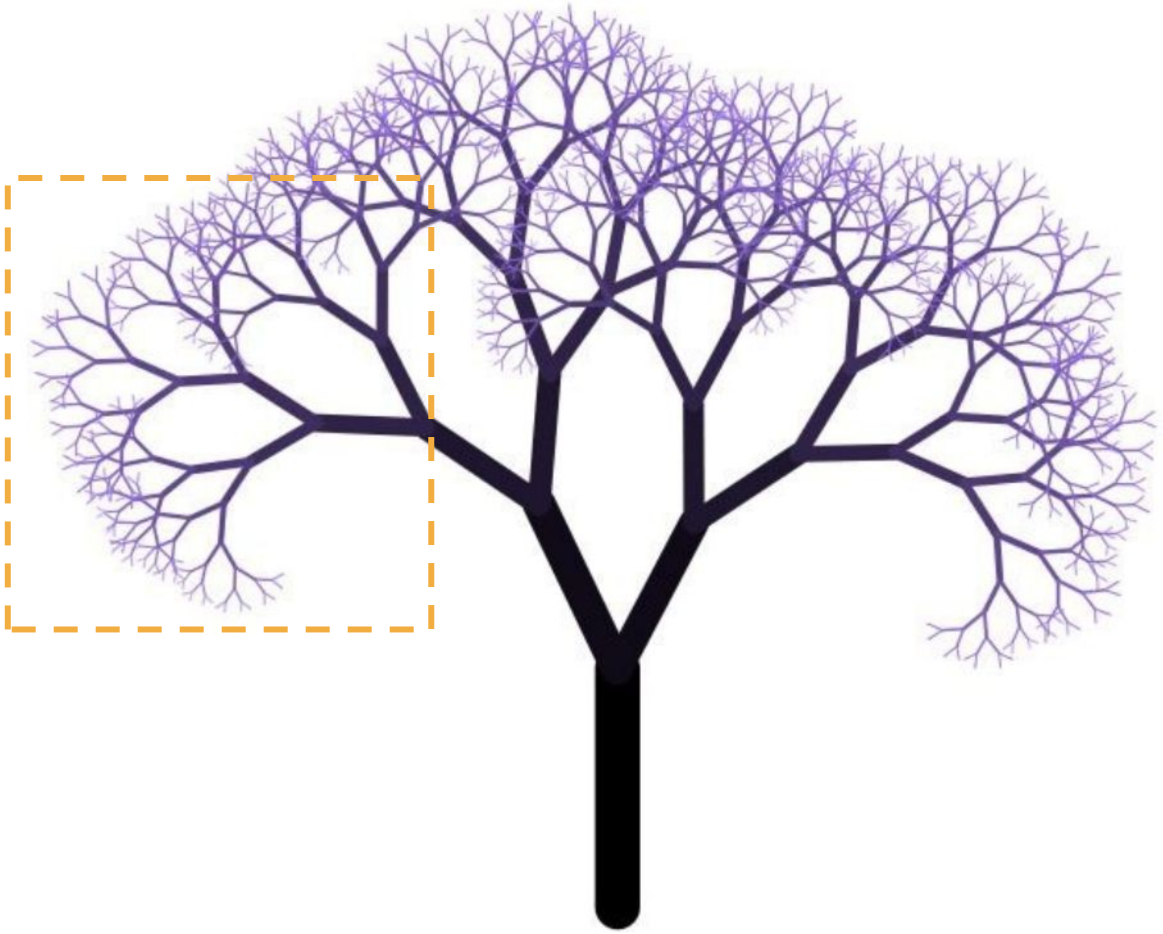


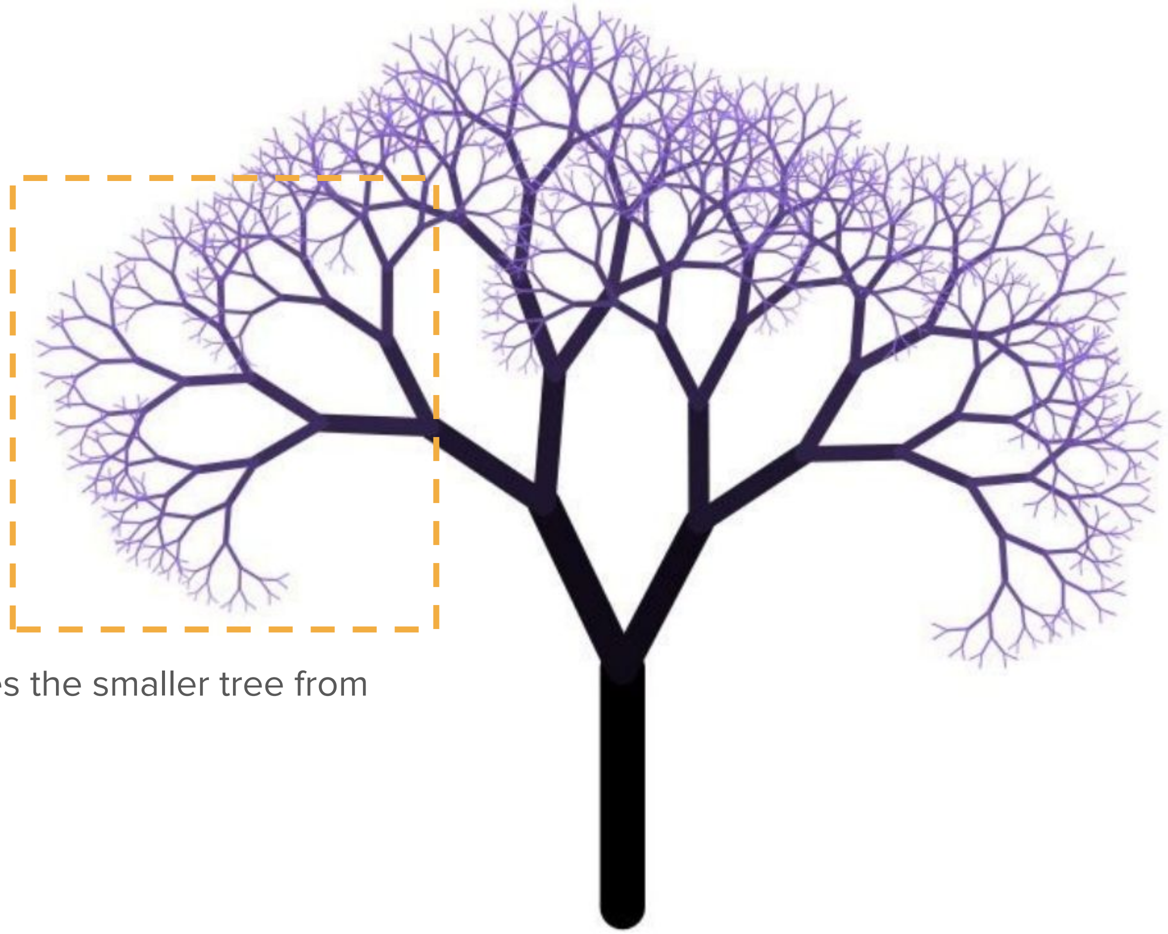
Understanding Fractal Structure



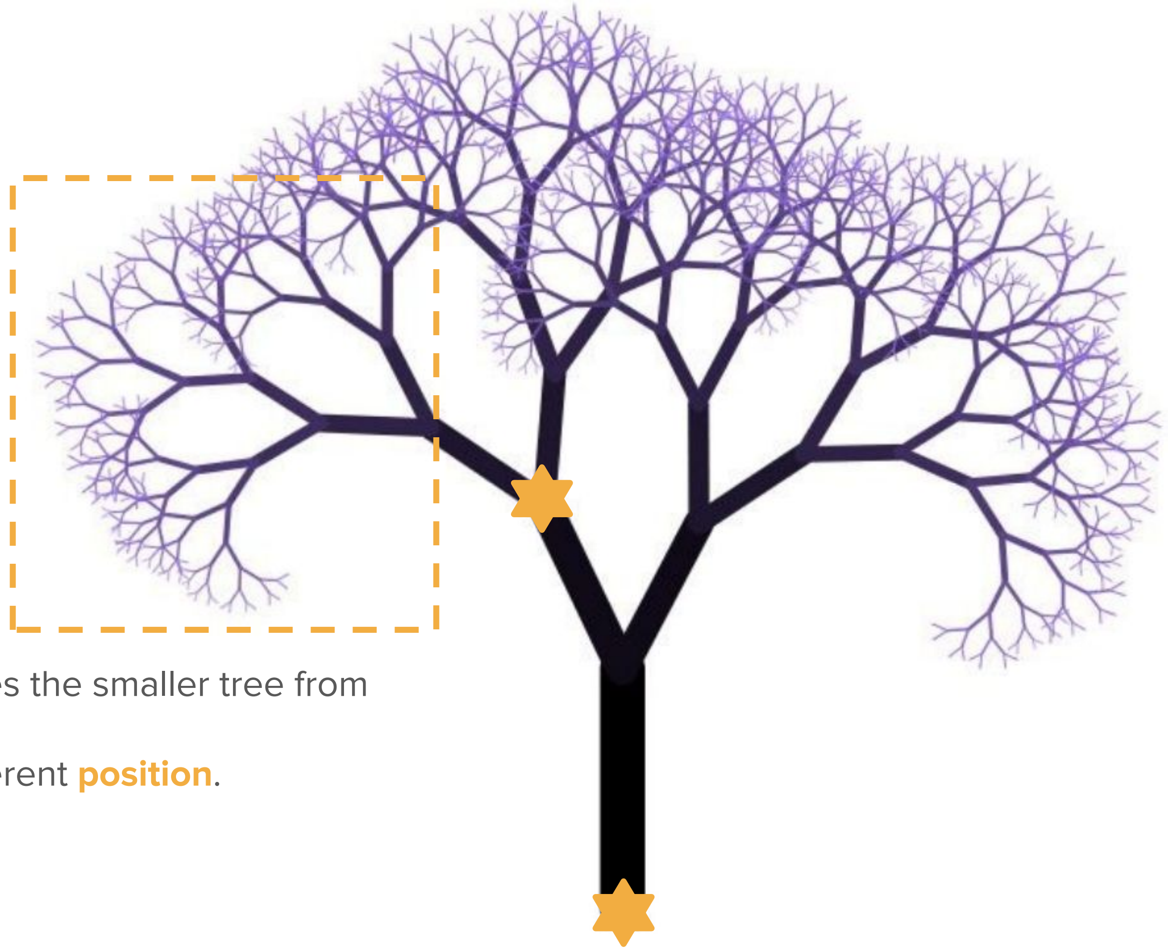






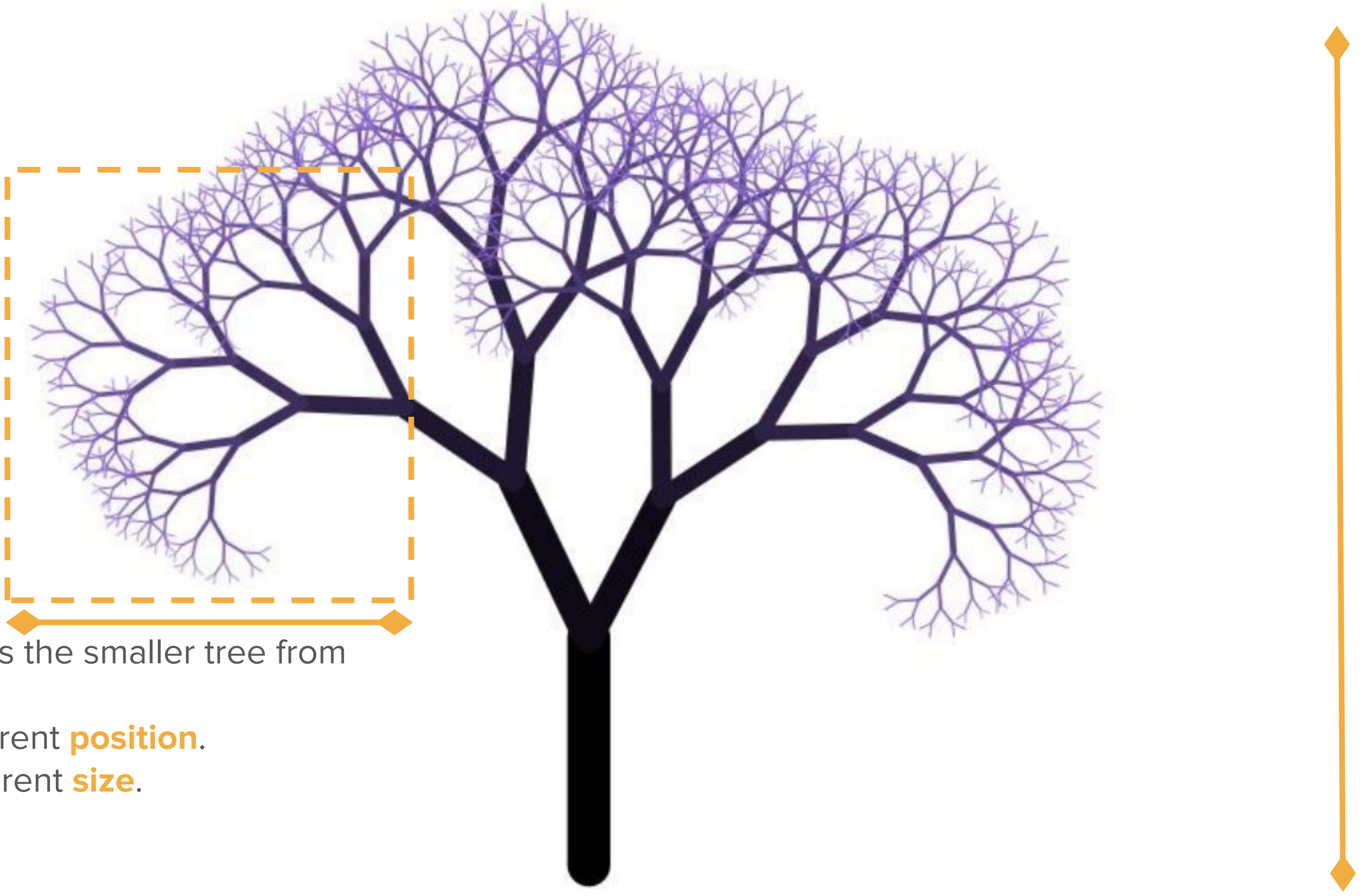


What differentiates the smaller tree from the bigger one?



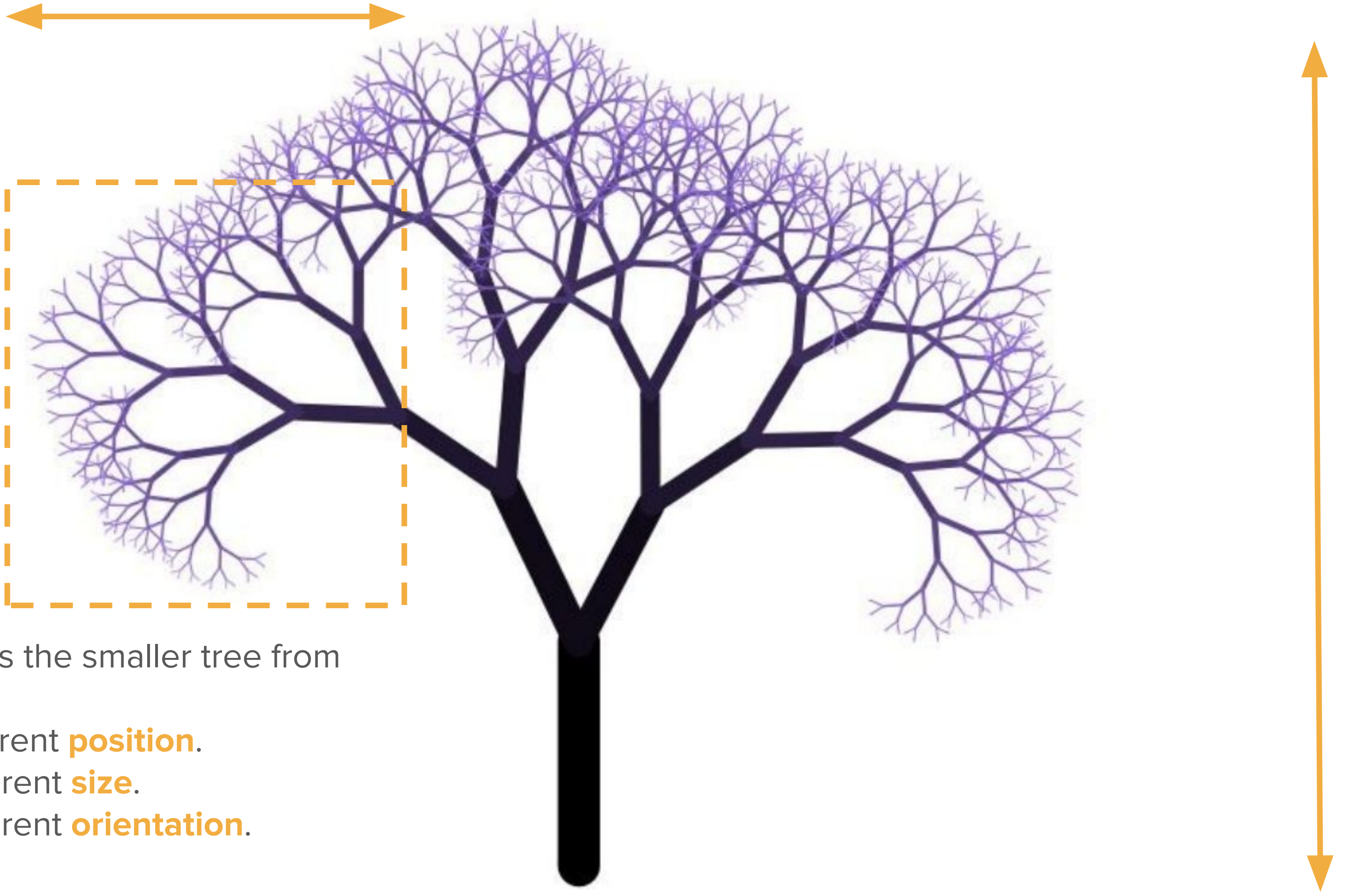
What differentiates the smaller tree from the bigger one?

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



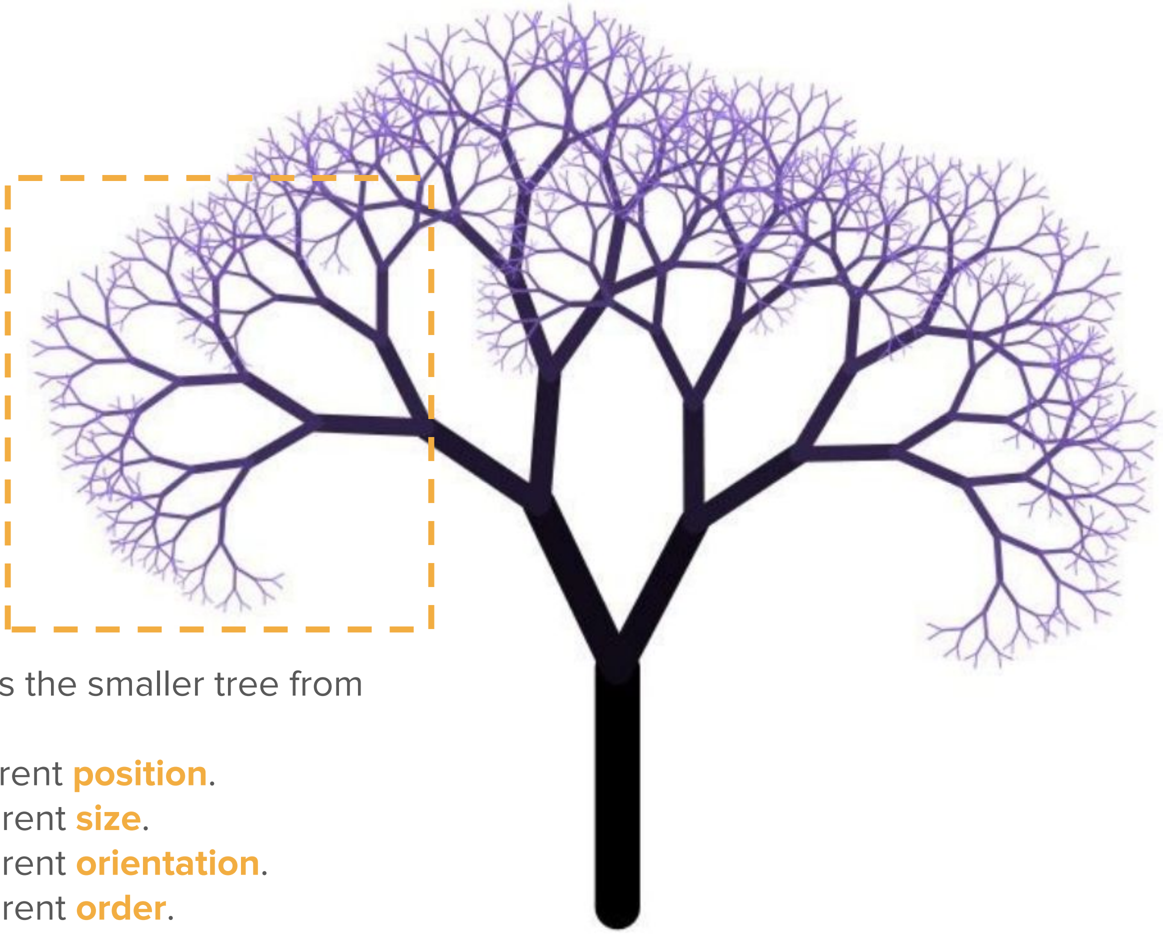
What differentiates the smaller tree from the bigger one?

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



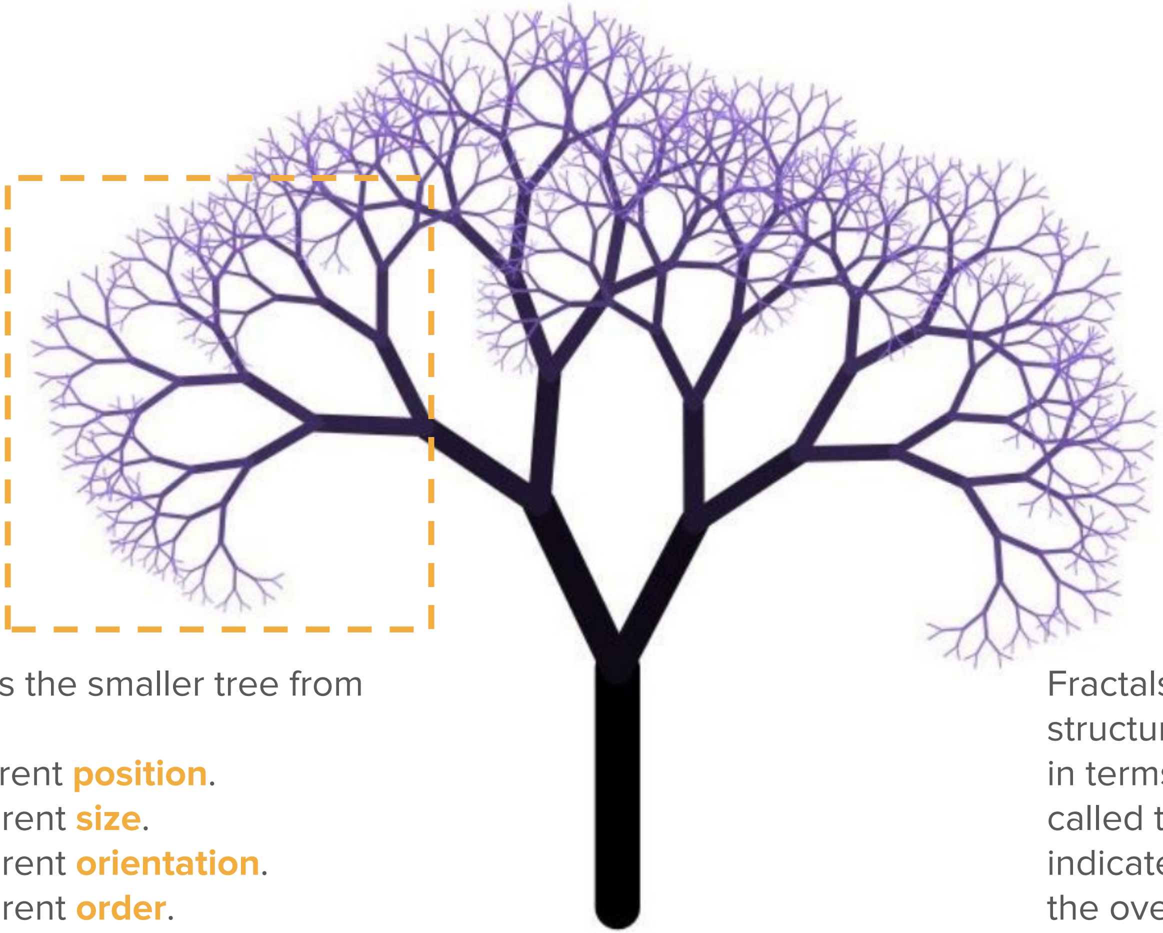
What differentiates the smaller tree from the bigger one?

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?

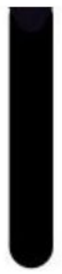
1. It's at a different **position**.
2. It has a different **size**.
3. It has a different **orientation**.
4. It has a different **order**.

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

An order-1 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**.

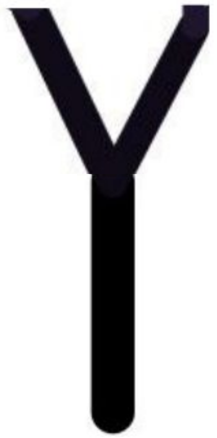


Fractals and self-similar structures are often defined in terms of some parameter called the **order**, which indicates the complexity of the overall structure.

An order-2 tree

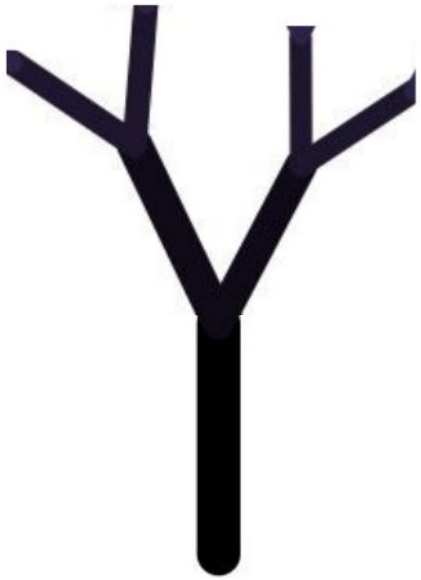
What differentiates the smaller tree from the bigger one?

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.

An order-3 tree

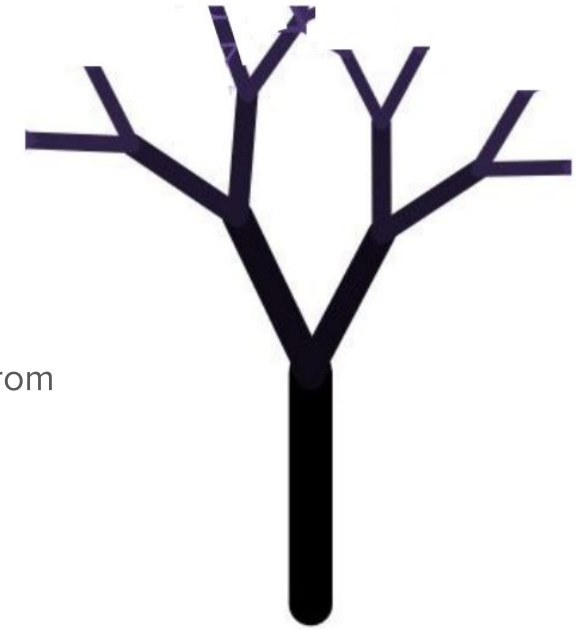


What differentiates the smaller tree from the bigger one?

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.

An order-4 tree

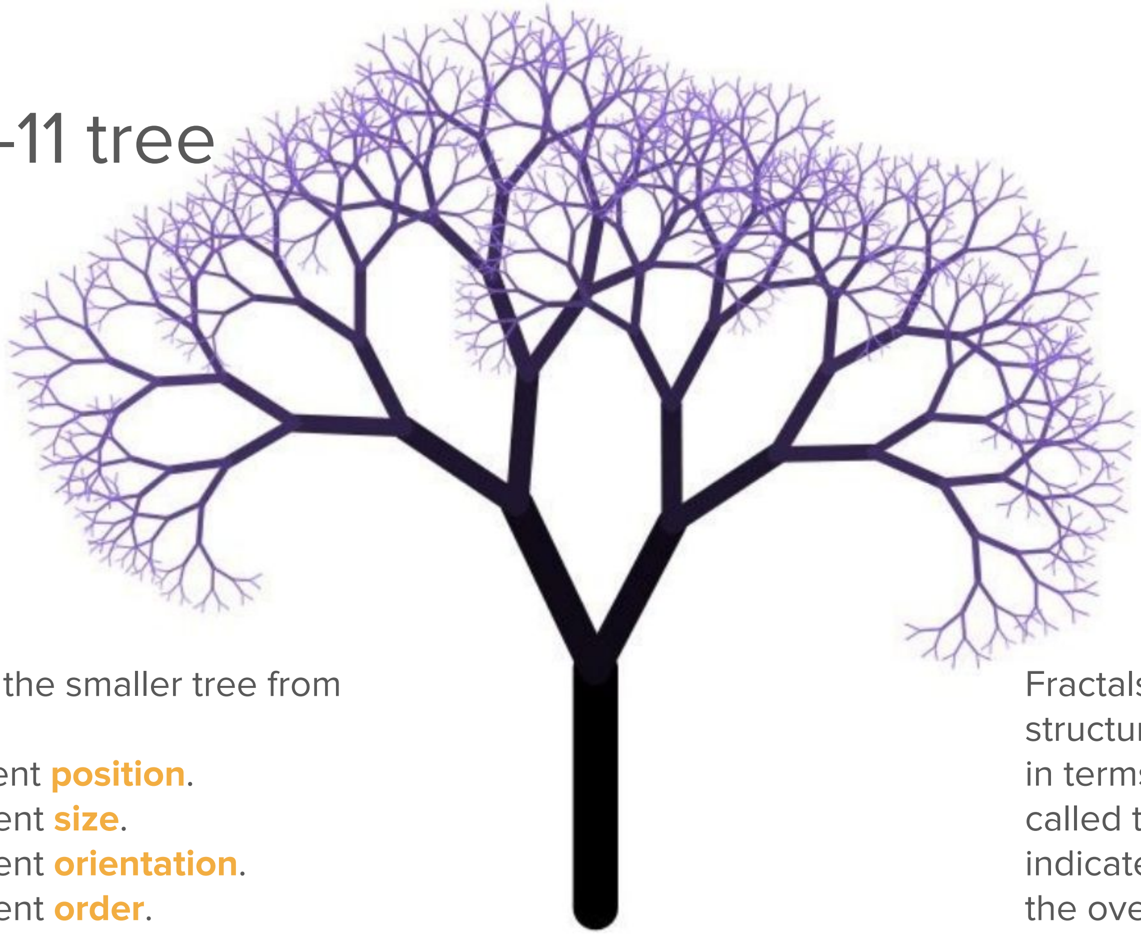


What differentiates the smaller tree from the bigger one?

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.

An order-11 tree

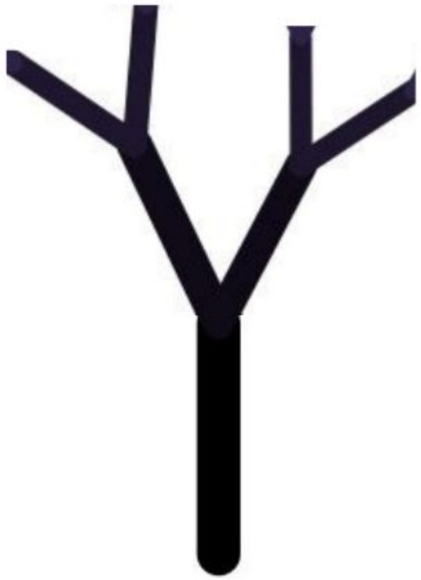


What differentiates the smaller tree from the bigger one?

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.

An order-3 tree



What differentiates the smaller tree from the bigger one?

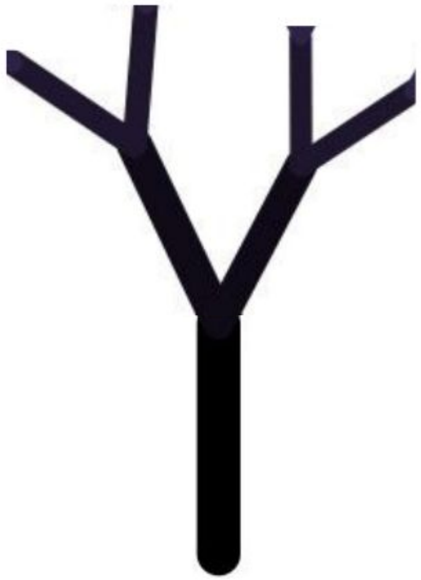
1. It's at a different **position**.
2. It has a different **size**.
3. It has a different **orientation**.
4. It has a different **order**.

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

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?

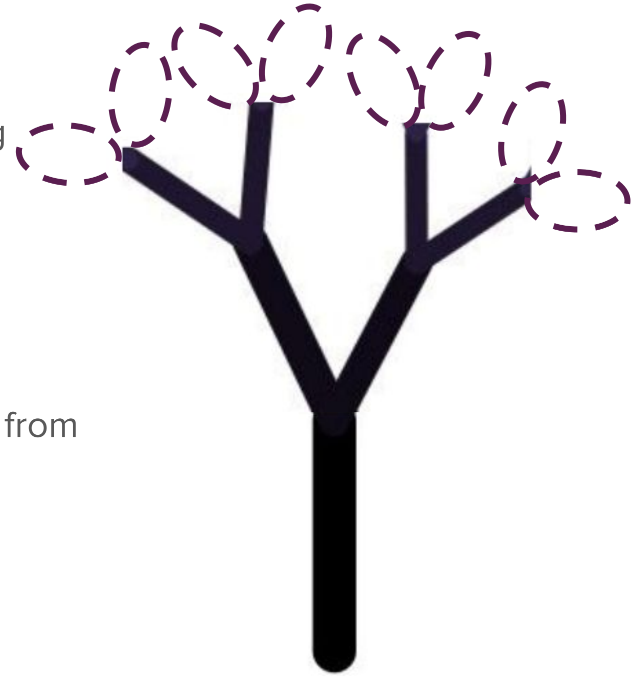
1. It's at a different **position**.
2. It has a different **size**.
3. It has a different **orientation**.
4. It has a different **order**.

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

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.

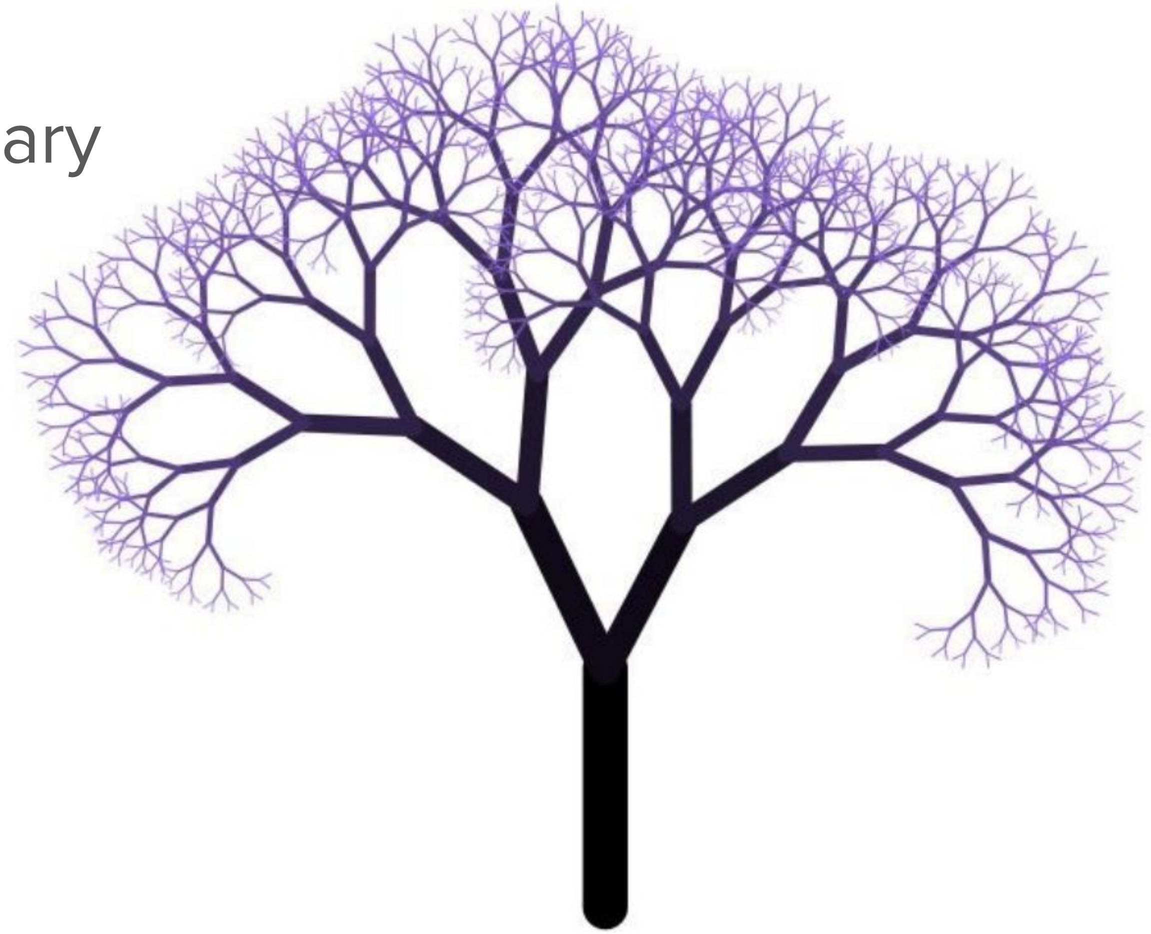


What differentiates the smaller tree from the bigger one?

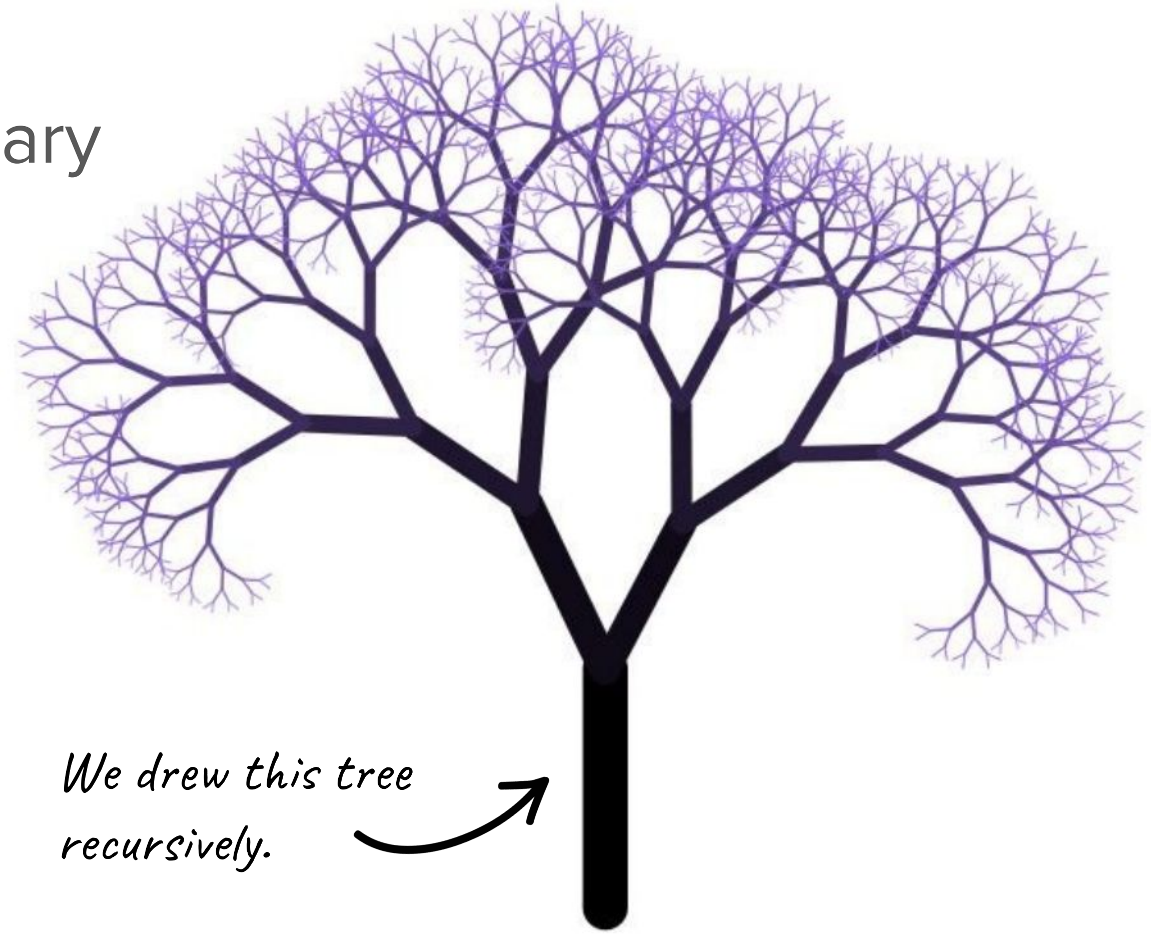
1. It's at a different **position**.
2. It has a different **size**.
3. It has a different **orientation**.
4. It has a different **order**.

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

In Summary

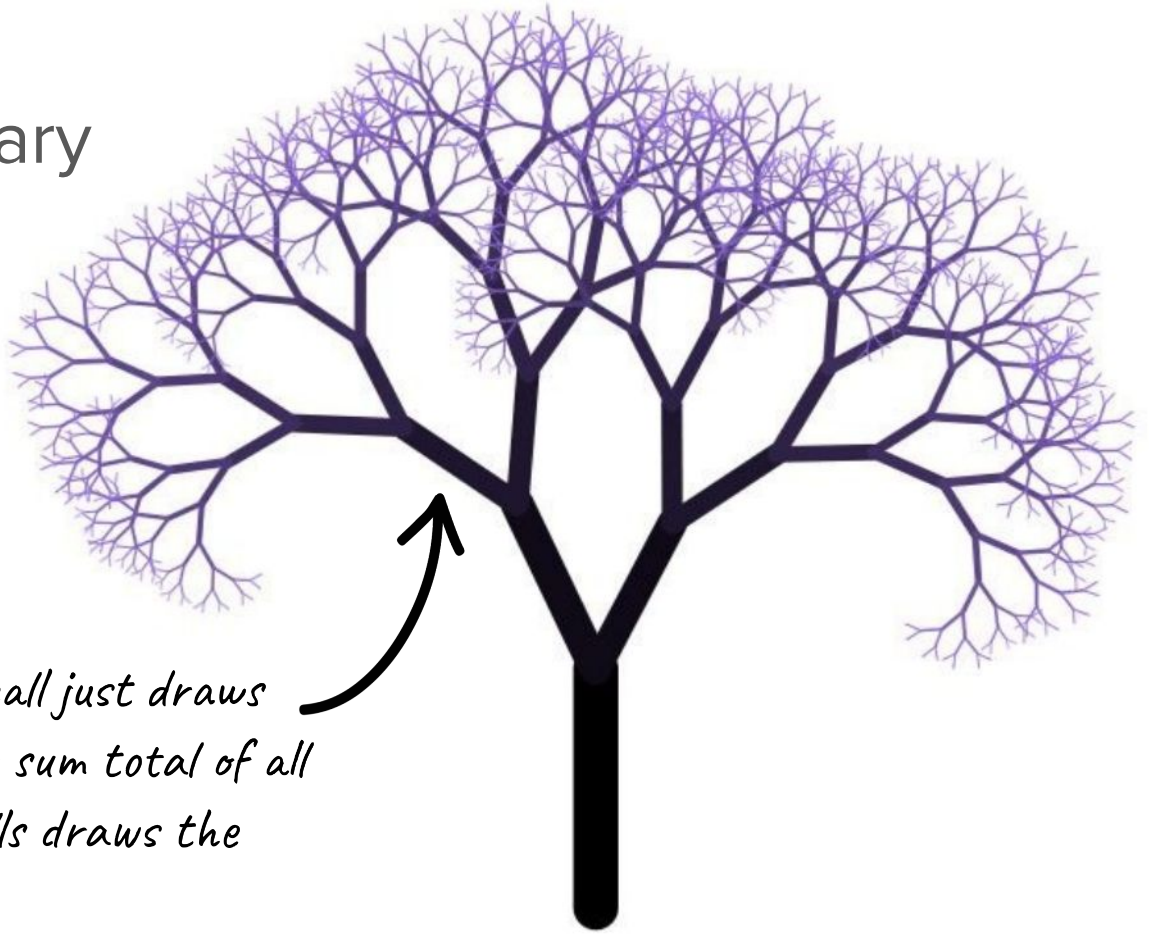


In Summary



We drew this tree recursively.

In Summary

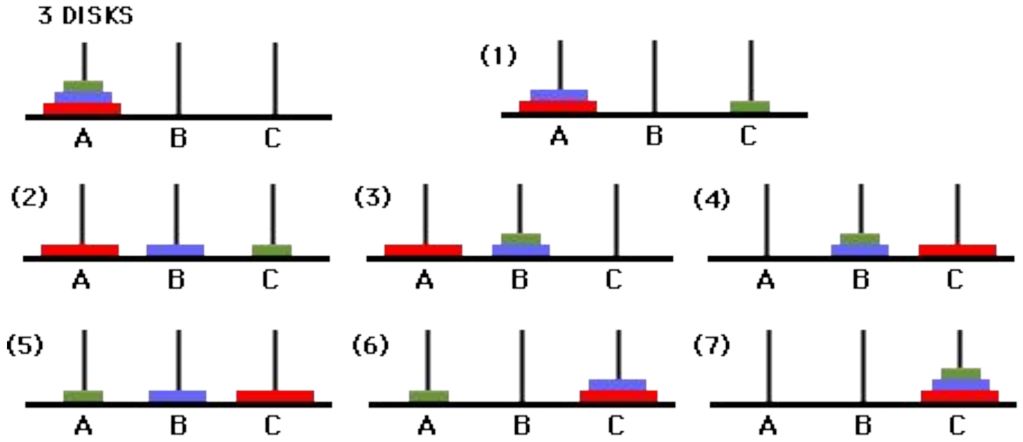


Each recursive call just draws one branch. The sum total of all the recursive calls draws the whole tree.

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

To Do 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!

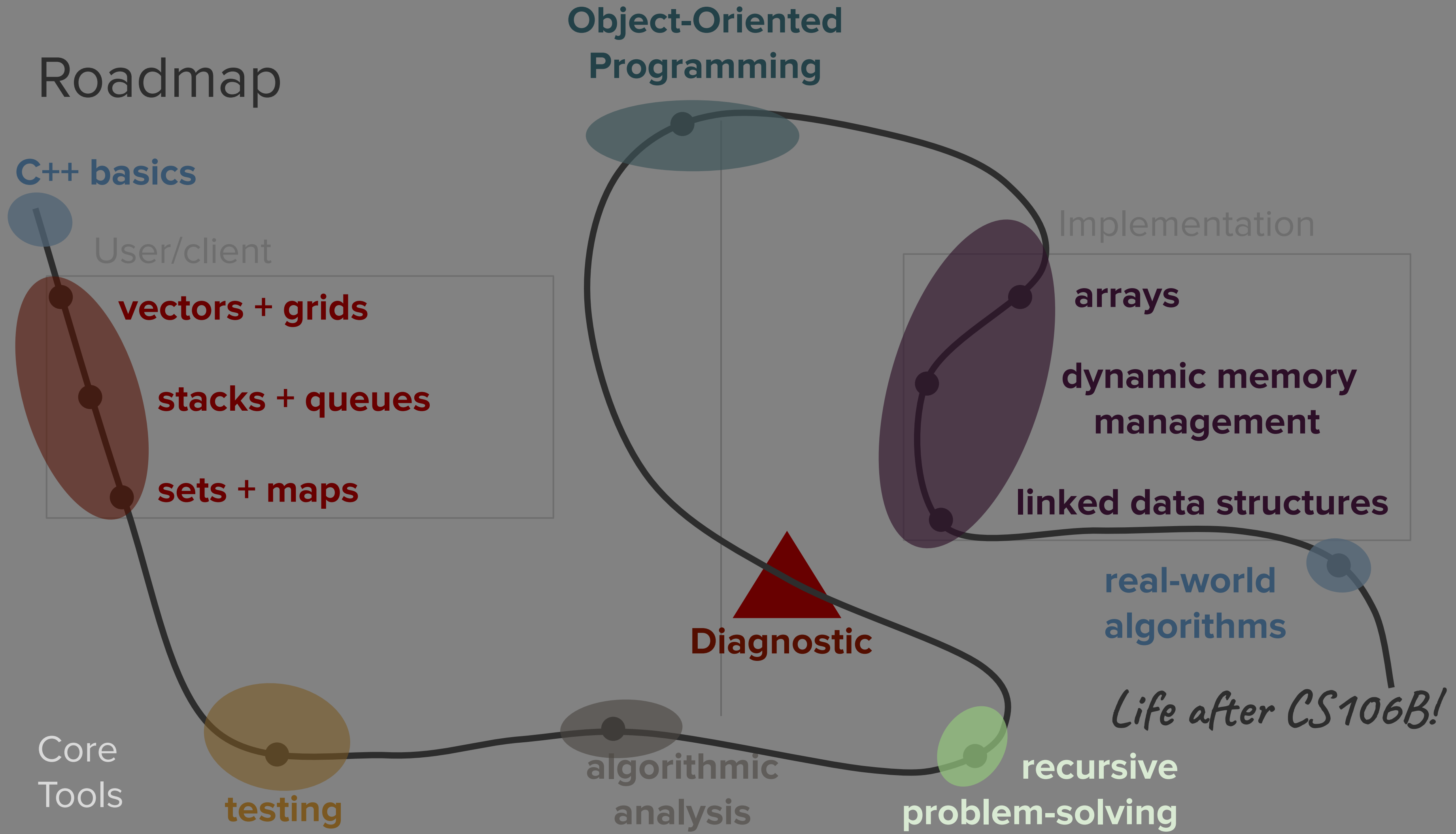
- **Extra challenge** (optional)**:** How would you define this problem recursively?
    - Don’t worry about data structures here. Assume we have a function **`moveDisk(X, Y)`** that will handle moving a disk from the top of post **`X`** to the top of post **`Y`**.

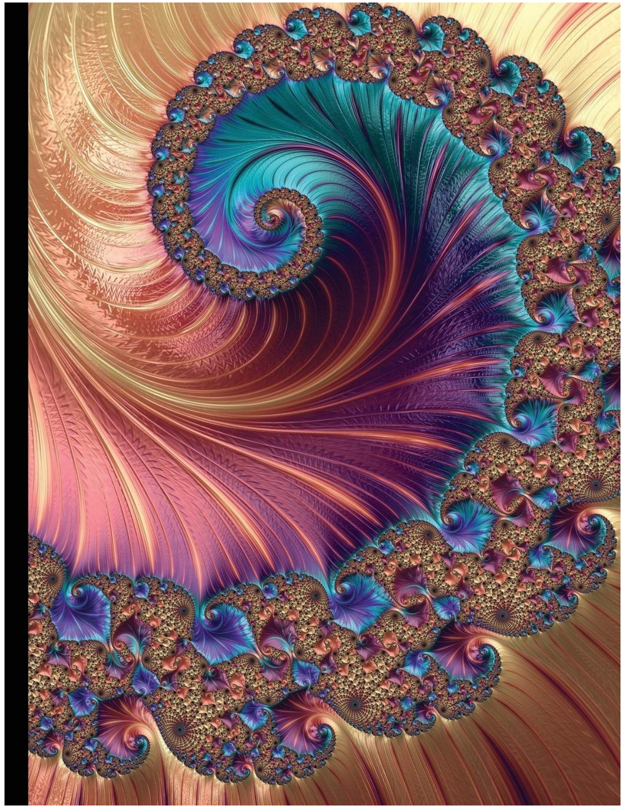
An Awesome Website!

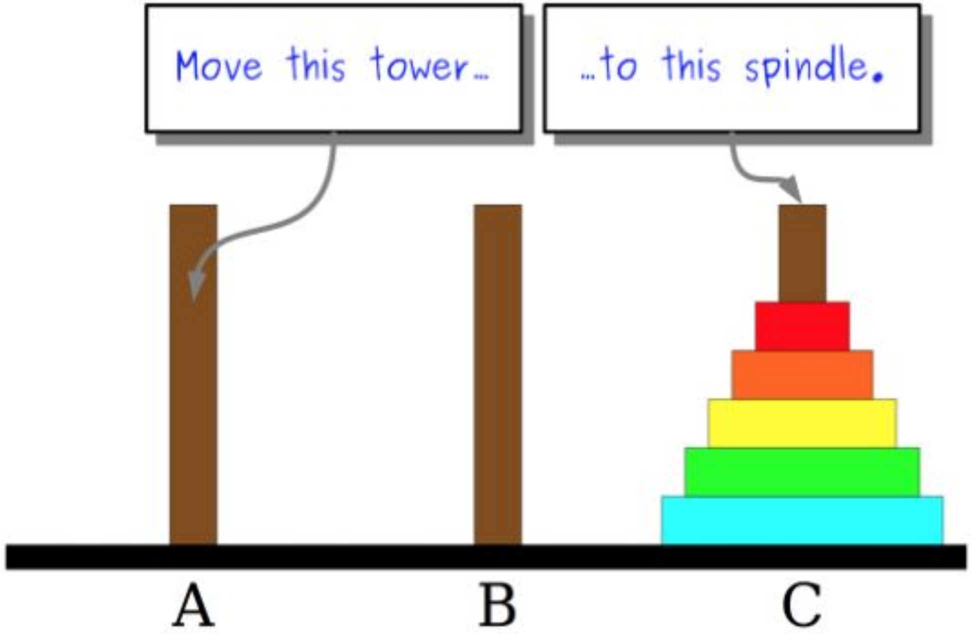
http://recursivedrawing.com/

What’s next?

Roadmap


C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Core Tools
testing
algorithmic analysis
Object-Oriented Programming
Diagnostic
recursive problem-solving
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Life after CS106B!




Move this tower…
…to this spindle.
A
B
C