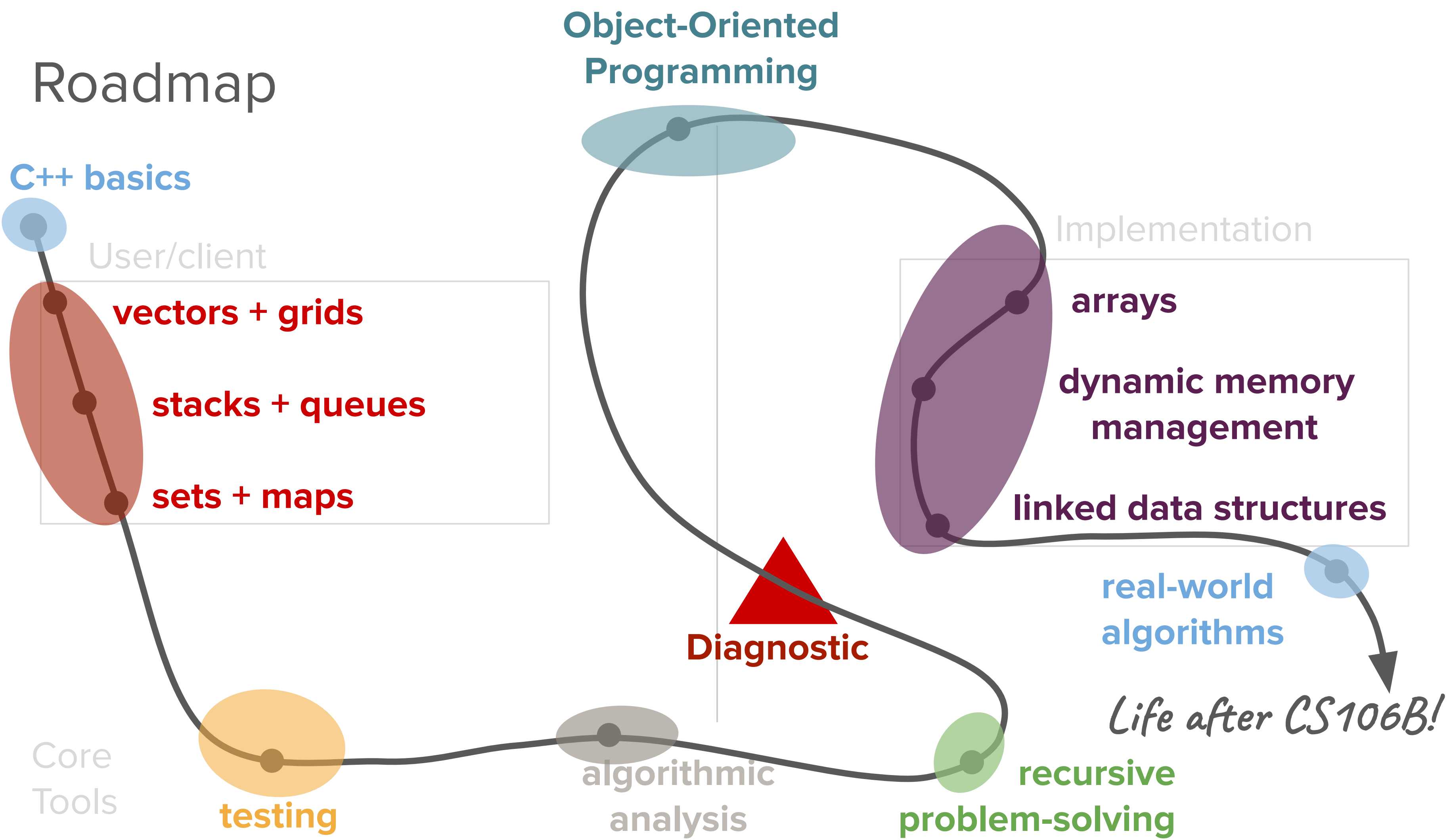
Introduction to Recursion

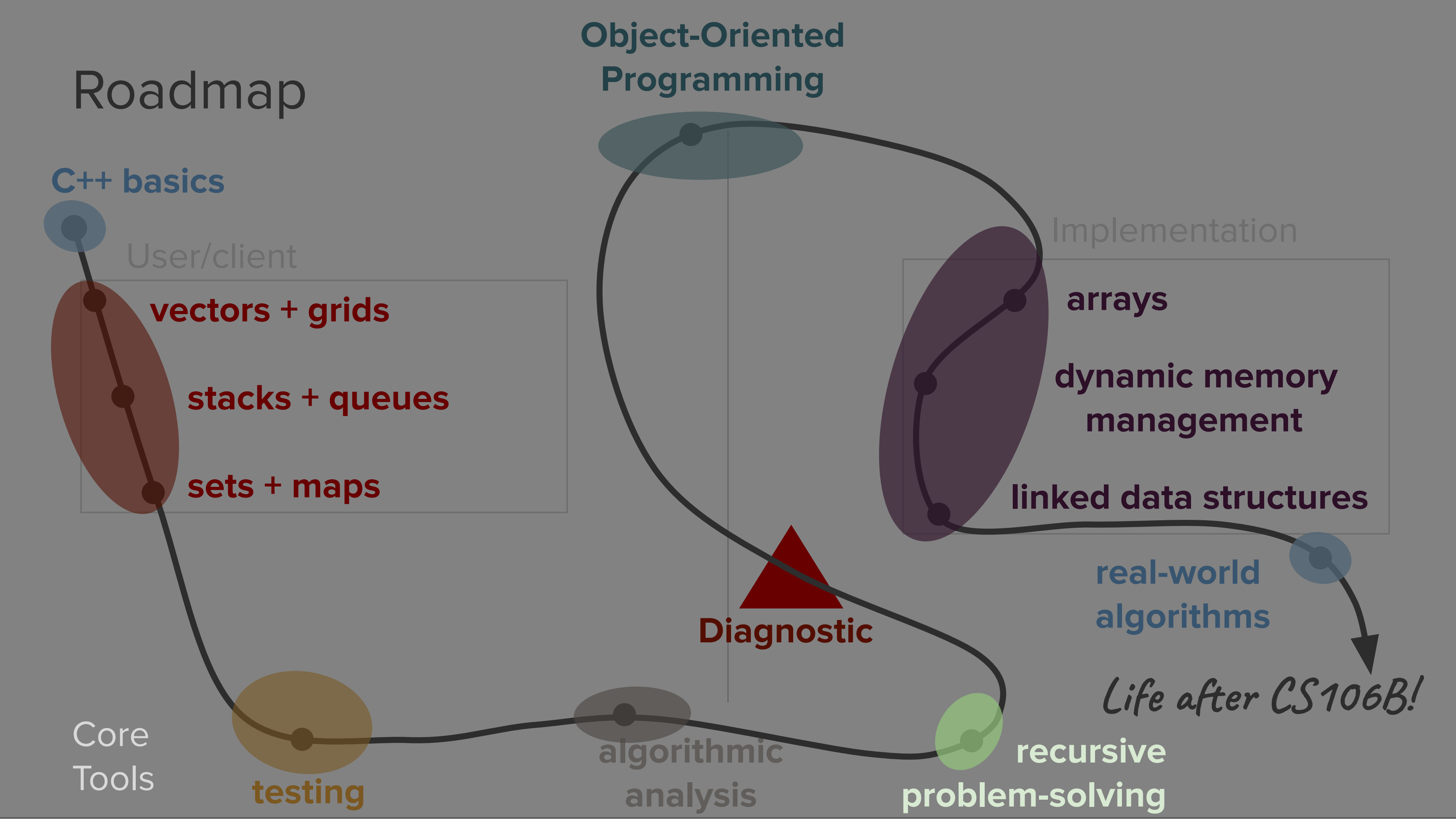
What's been the most challenging part of Assignment 2 for you so far?

(put your answers the chat)

Roadmap



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


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

Today's question

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

Today's topics

1. Review
2. Defining recursion
3. Recursion + Stack Frames (e.g. factorials)
4. Recursive Problem-Solving (e.g. string reversal)

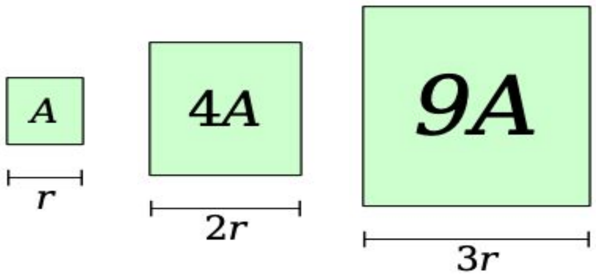
Review

(Big O)

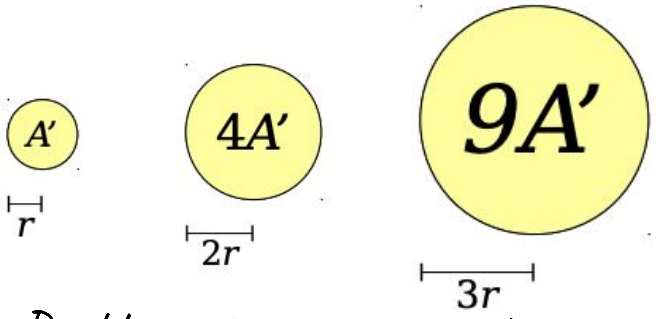
Big-O Notation

- **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(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

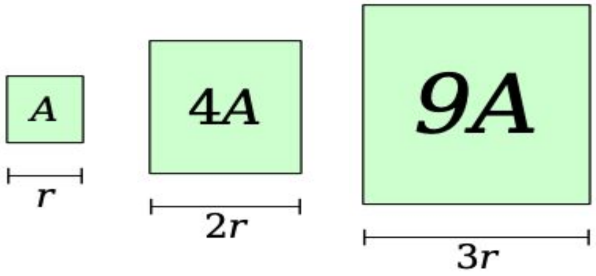


Doubling r increases area 4x
Tripling r increases area 9x

Big-O Notation

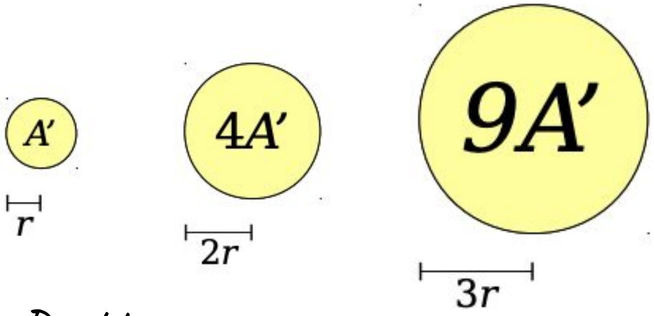
With respect to a given input variable!

- **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

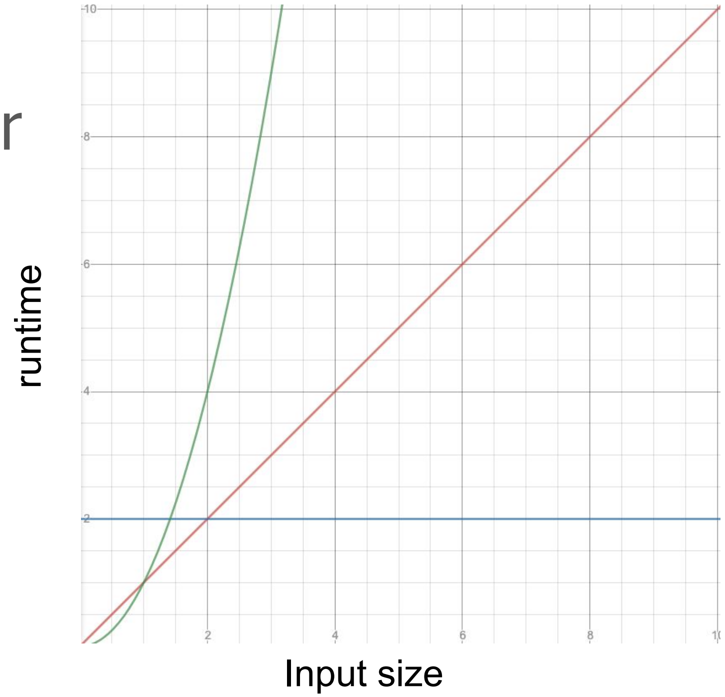


Doubling r increases area 4x

Tripling r increases area 9x

Efficiency Categorizations So Far

- Constant Time – O(1)
    - Super fast, this is the best we can hope for!
    - 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
    - Exhaustive Search for Perfect Numbers
- How do all the ADT operations we've seen so far fall into these categories?



ADT Big-O Matrix

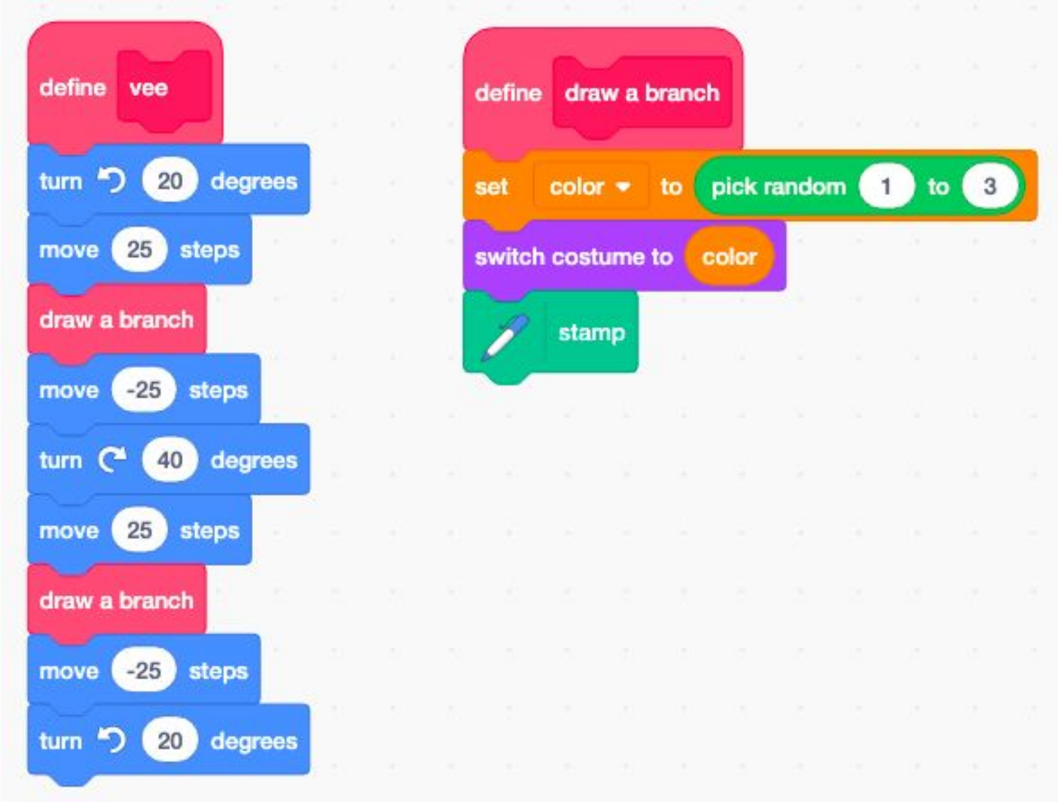
- Vectors
  - `.size() - O(1)`
  - `.add() - O(1)`
  - `v[i] - O(1)`
  - `.insert() - O(n)`
  - `.remove() - O(n)`
  - `.clear() - O(n)`
  - `traversal - O(n)}`
- Grids
  - `.numRows()/.numCols() - O(1)`
  - `g[i][j] - O(1)`
  - `.inBounds() - O(1)`
  - traversal - O(n^2)
- Queues
  - `.size() - O(1)`
  - `.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?

Activity: Vee

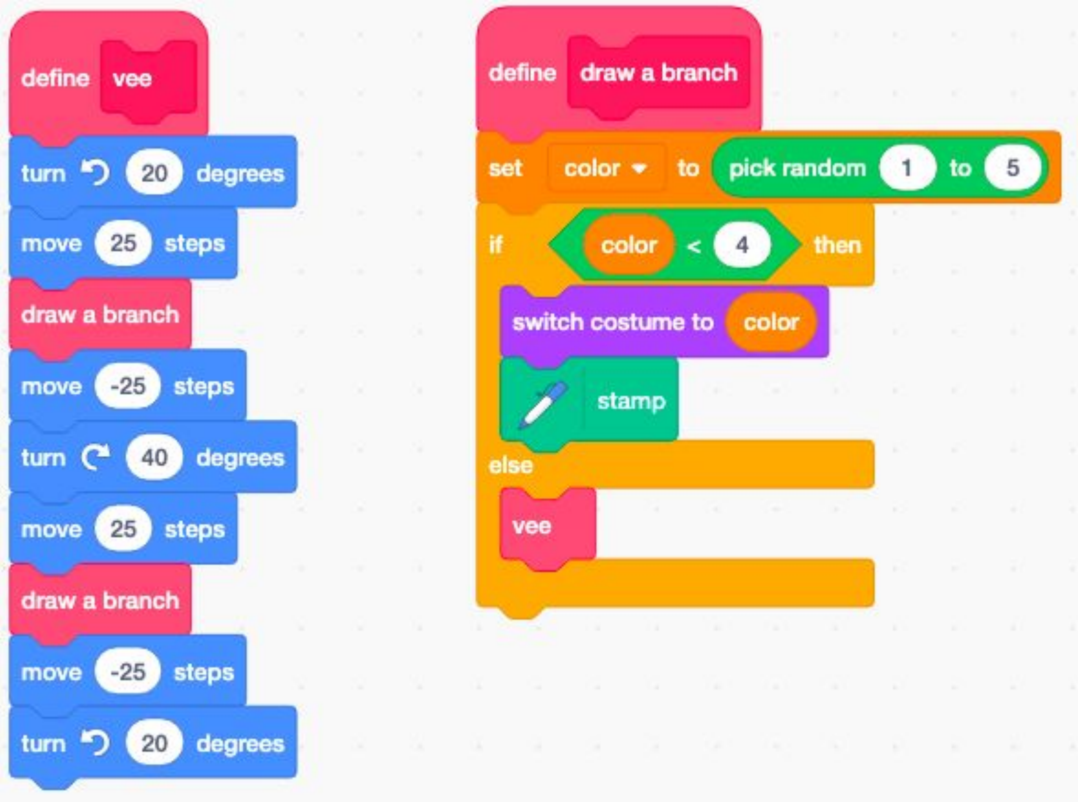
(https://scratch.mit.edu/projects/409796637/)

This code creates a “vee” shape with random colors.

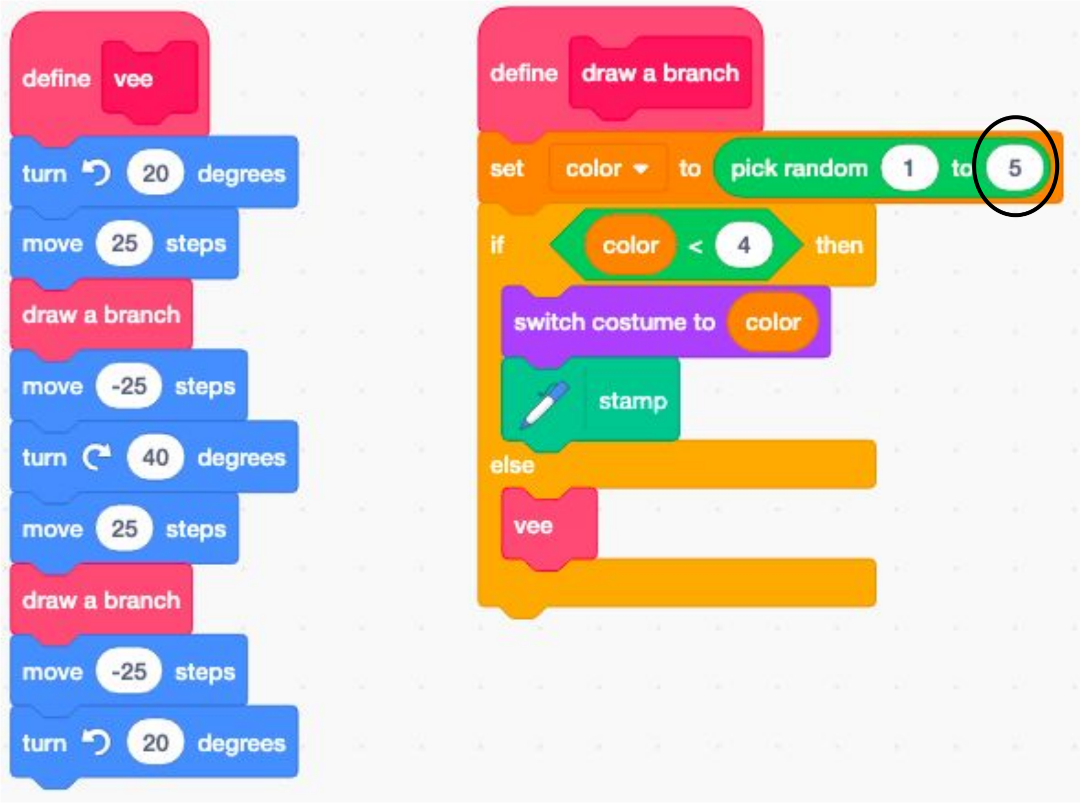


Based on Vee by Dan Garcia

Discuss in breakout rooms: What will this code do?



Discuss in breakout rooms: What will this code do?



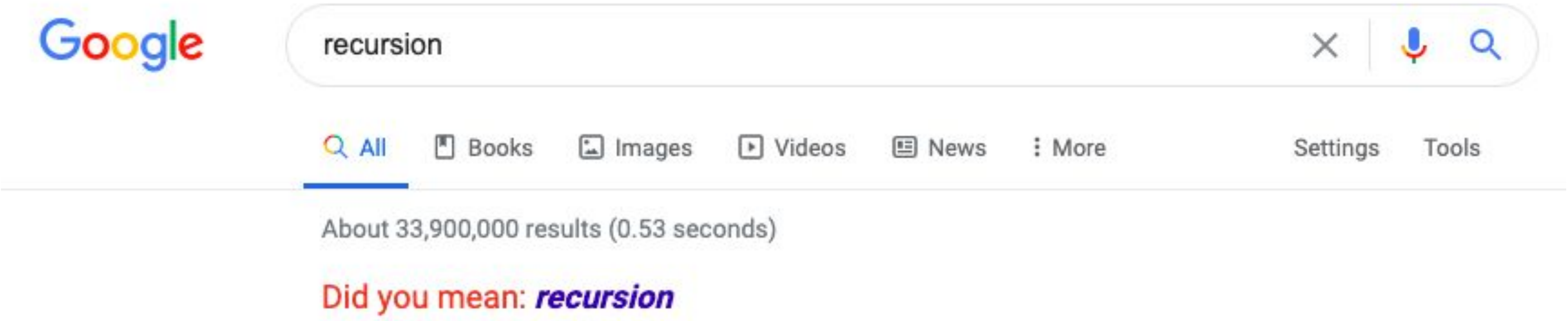
Notice the differences

Demo: Recursive Vee

(https://scratch.mit.edu/projects/409785610/)

What is recursion?

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



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

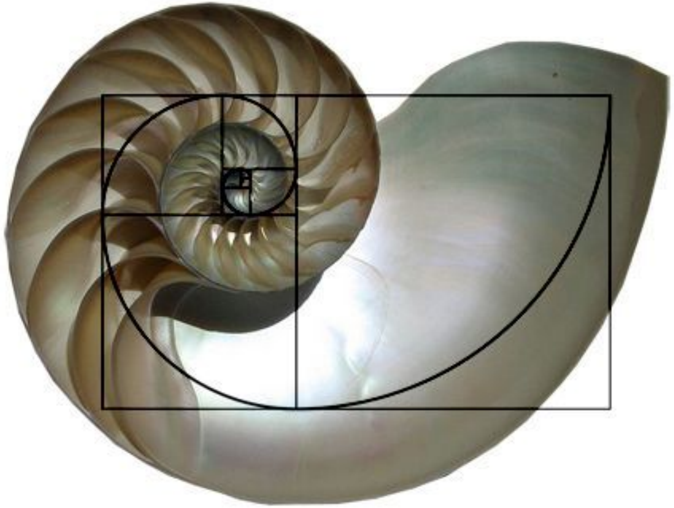
What is recursion?

- 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-COVID] 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?

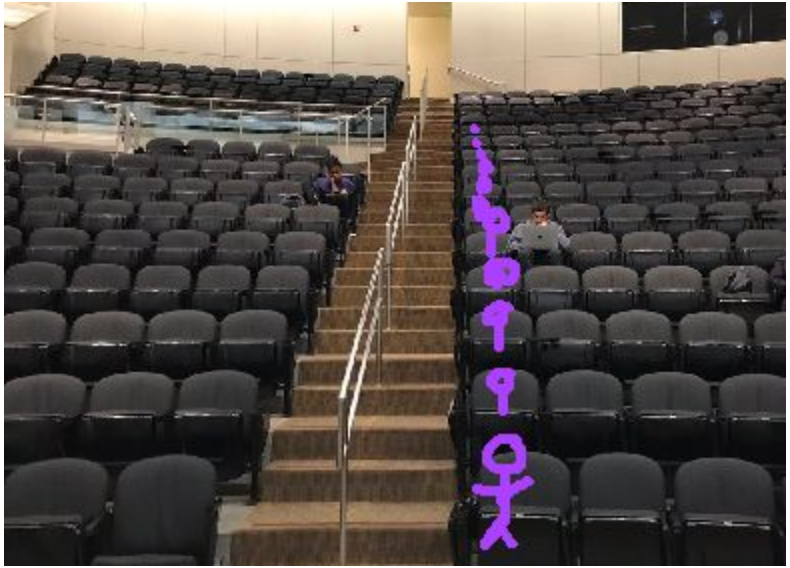
- Let’s suppose I want to find out how many people are at lecture today, but I don’t want to walk around and count each person.

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

We can solve this problem recursively!

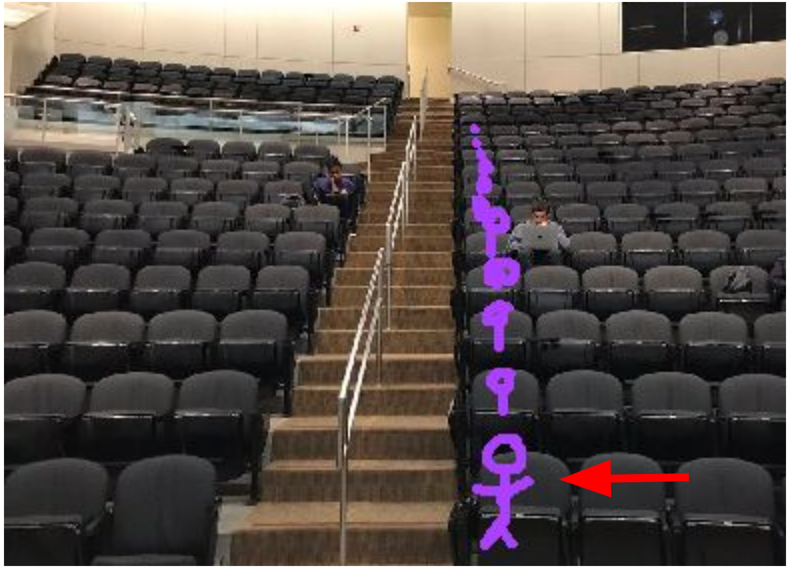
How many students are in the lecture hall?

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



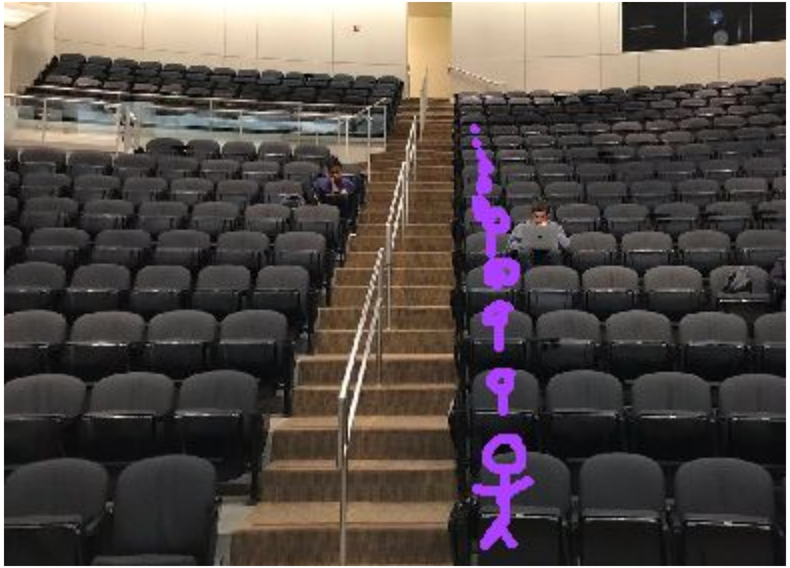
How many students are in the lecture hall?

- 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'?"



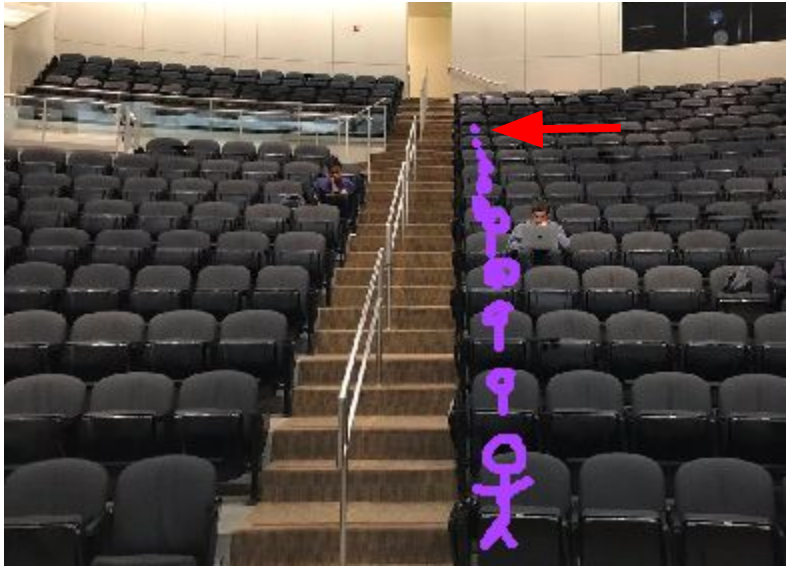
How many students are in the lecture hall?

- 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.



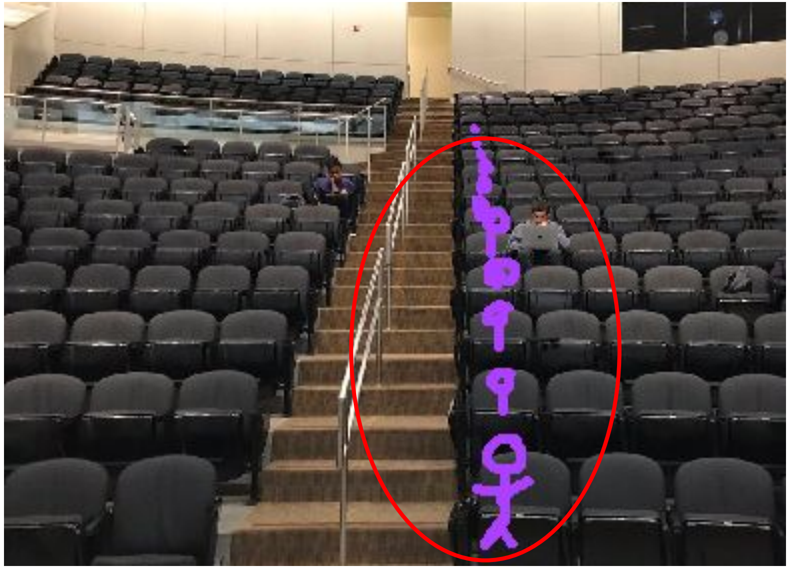
How many students are in the lecture hall?

- 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.



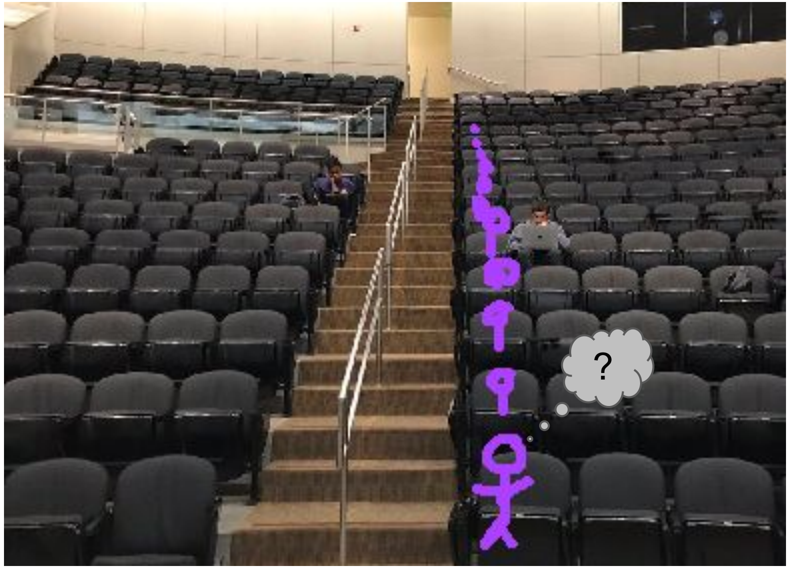
How many students are in the lecture hall?

- 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.



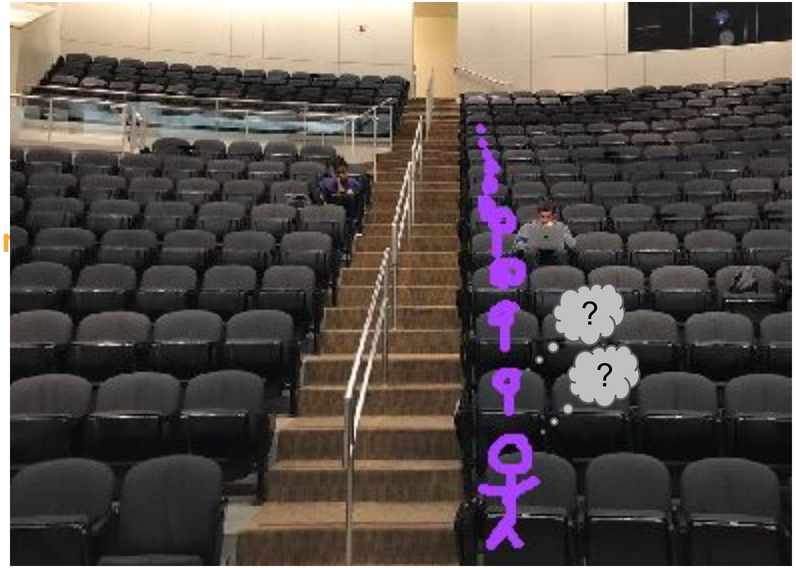
How many students are in the lecture hall?

- 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.



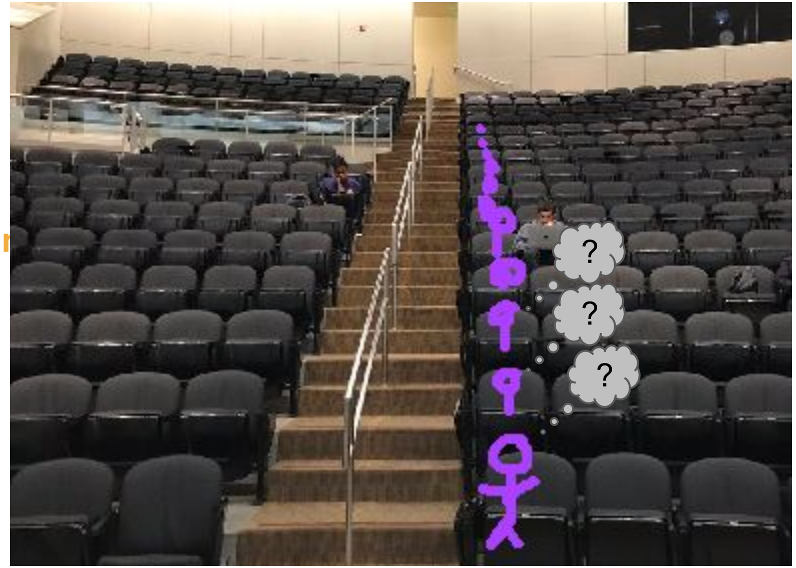
How many students are in the lecture hall?

- 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.



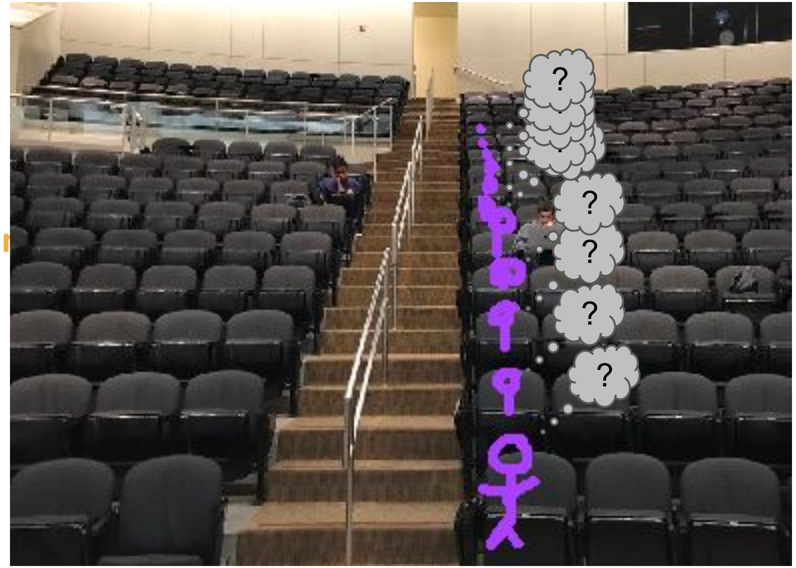
How many students are in the lecture hall?

- 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.

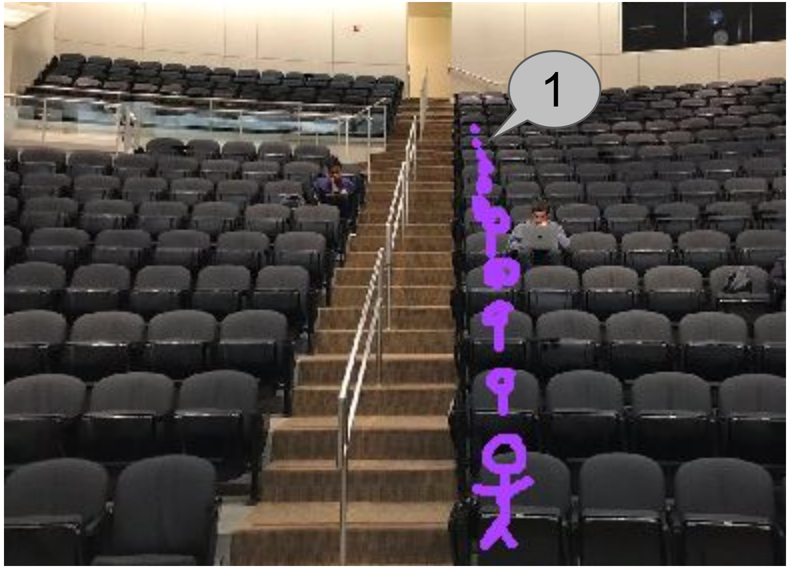


How many students are in the lecture hall?

- 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.

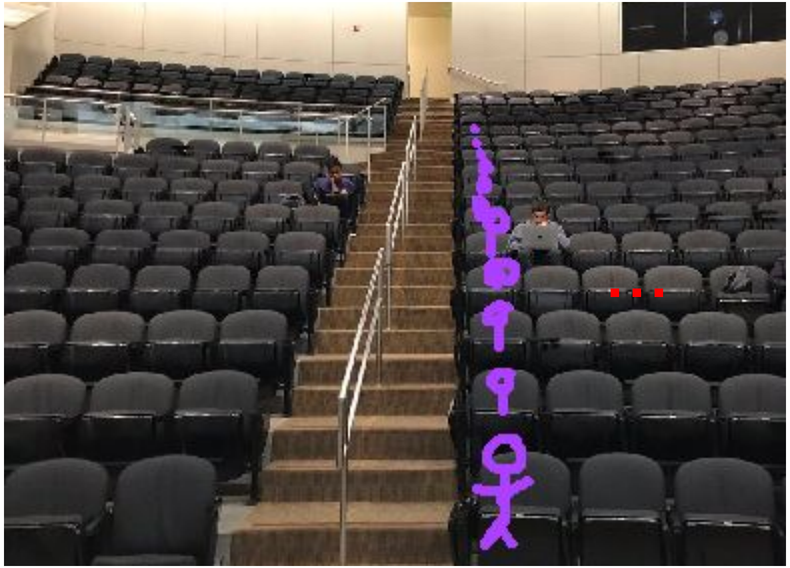


How many students are in the lecture hall?

- 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.

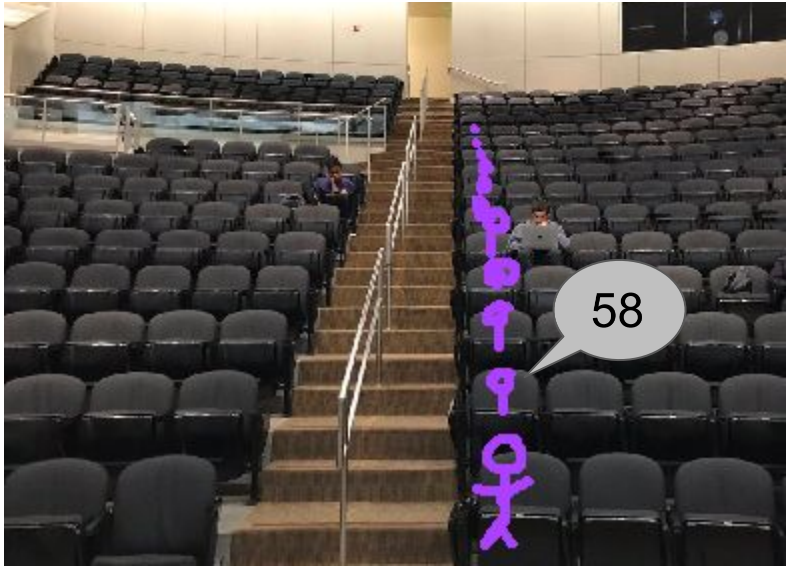


How many students are in the lecture hall?

- 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.

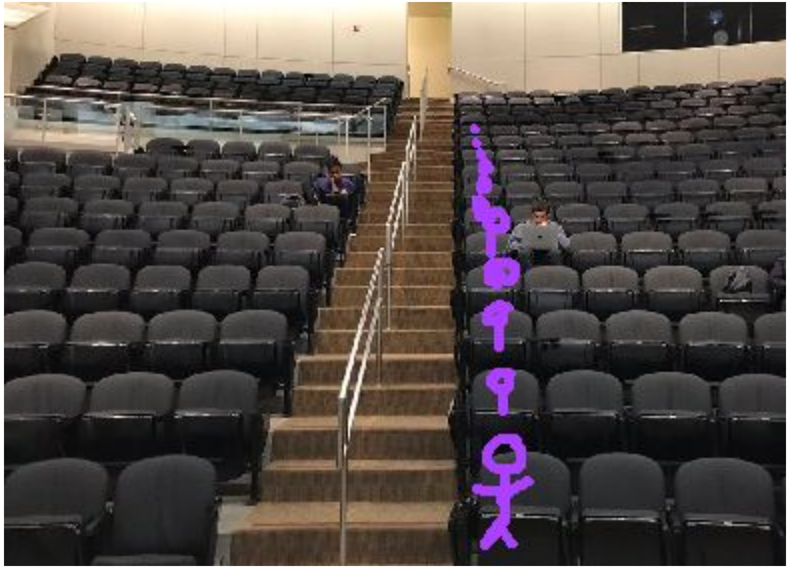


How many students are in the lecture hall?

- 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!

Announcements

Announcements

- Assignment 2 is due Wednesday, 7/8.
- Assignment 3 will be released by the end of the day on Thursday.
- The mid-quarter diagnostic will cover through the end of this week (Thursday will be the last day of content covered).
- Please remember to only ask questions in the chat that are necessary for your immediate understanding!

Factorial example

Factorials

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

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

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 x 4!
4! = 4 x 3!
3! = 3 x 2 x 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.

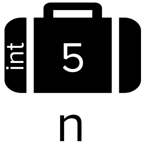
Recursion in action

```
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);
    }
}
```

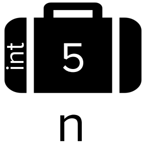


When a function gets called, a new stack frame gets created.

Recursion in action

```
int main() {

}
```

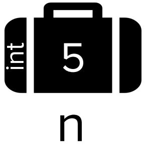
```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```



Recursion in action

```
int main() {



}
```

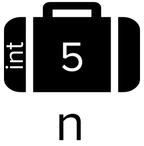
```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```



Recursion in action

```
int main() {

}
```

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

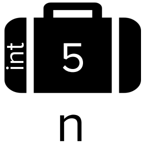


Recursion in action

```
int main() {

}
```

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

5

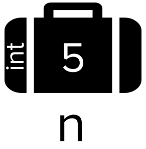


Recursion in action

```
int main() {

}
```

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

5

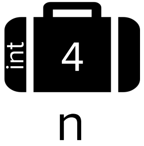


Recursion in action

```
int main() {

}
```

```
int factorial (int n) {

}
```

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



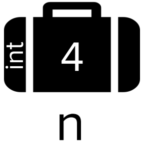
Every time we call **`factorial()`**, we get a new copy of the local variable **n** that's independent of all the previous copies because it exists inside the new frame.

Recursion in action

```
int main() {

}
```

```
int factorial (int n) {

}
```

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

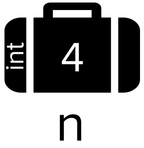


Recursion in action

```
int main() {

}
```

```
int factorial (int n) {

}
```

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

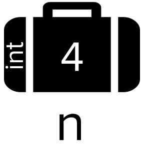


Recursion in action

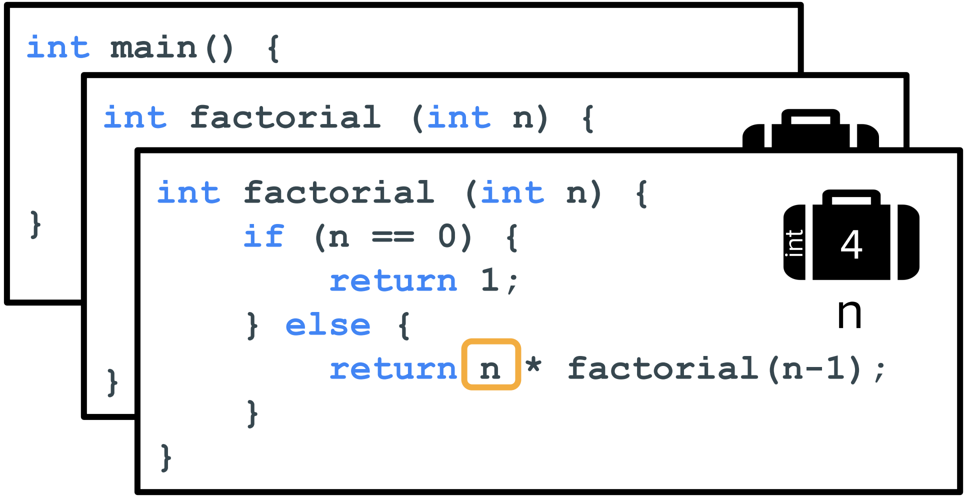
```
int main() {

}
```

```
int factorial (int n) {

}
```

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



Recursion in action

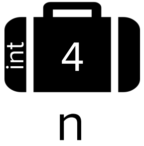


Recursion in action

```
int main() {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
                4
    }
}
```

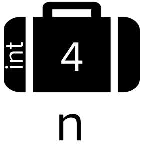


Recursion in action

```
int main() {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }           4
}
```



Recursion in action

```
int main() {

}
```

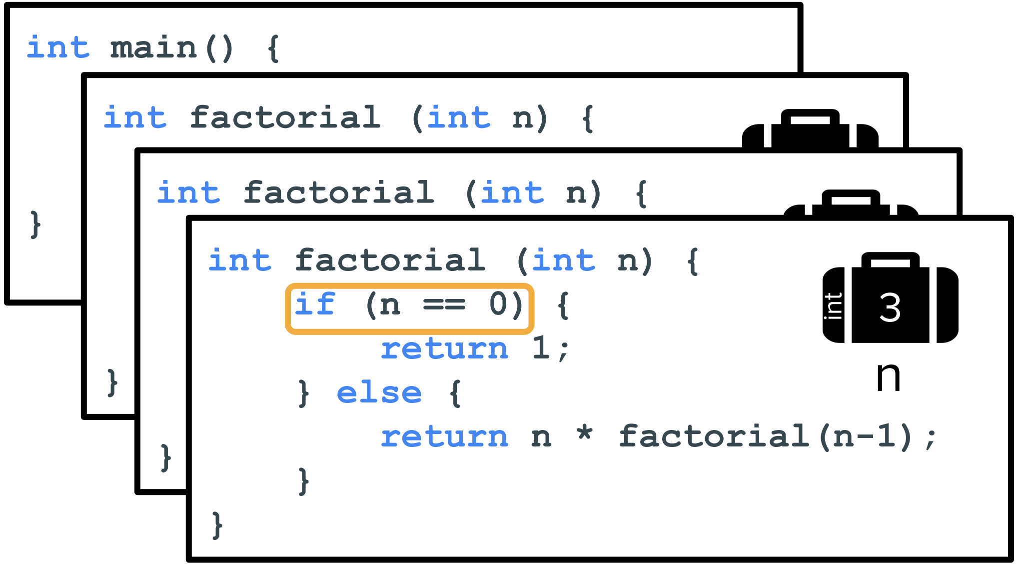
```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

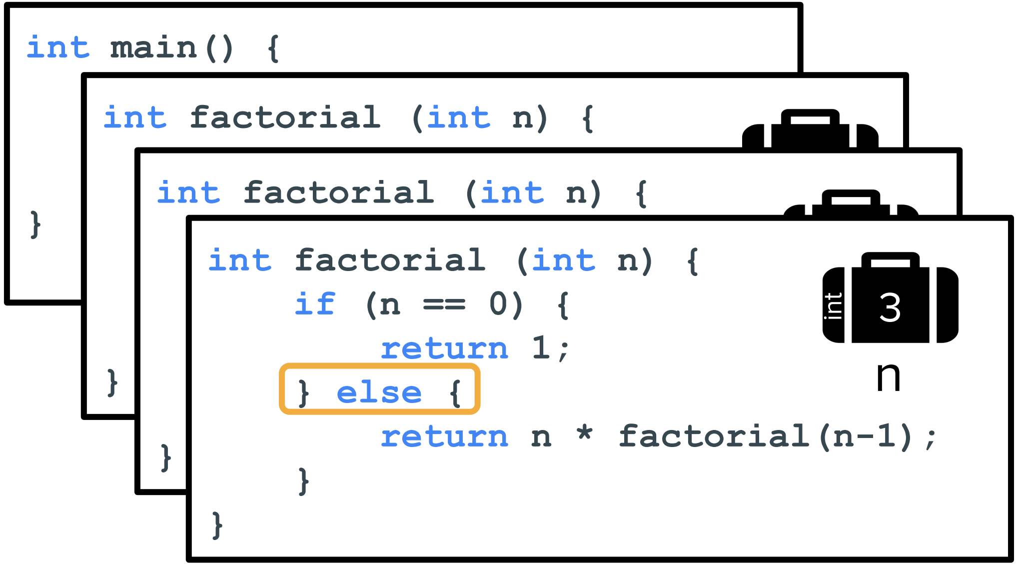
```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```

n

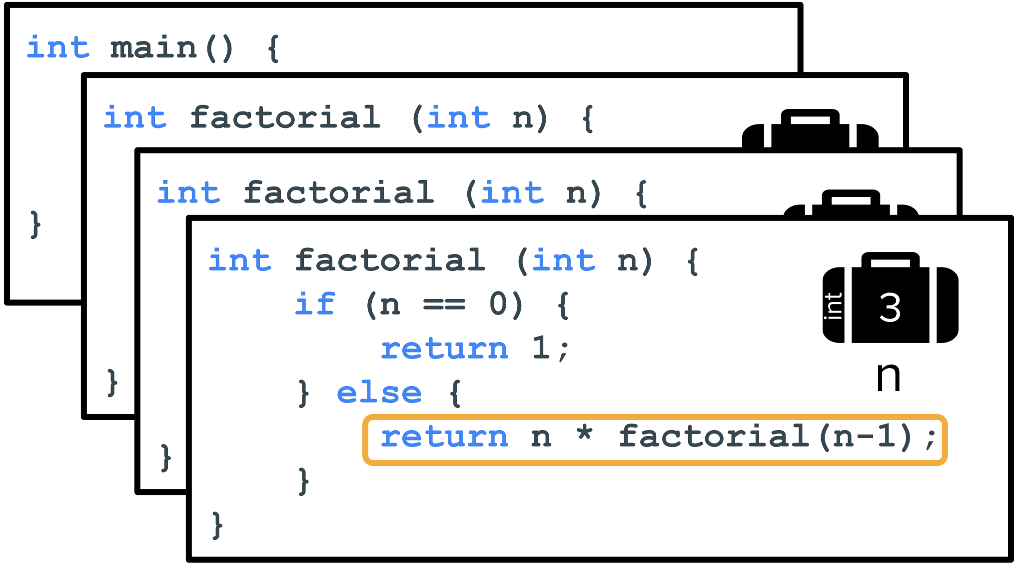
Recursion in action



Recursion in action



Recursion in action



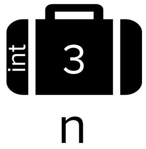
Recursion in action

```
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);
    }
}
```



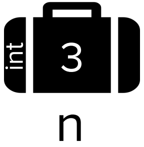
Recursion in action

```
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

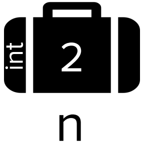
```
int main() {
}
```

```
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

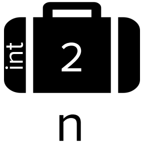
```
int main() {

}
```

```
int factorial (int n) {

}
```

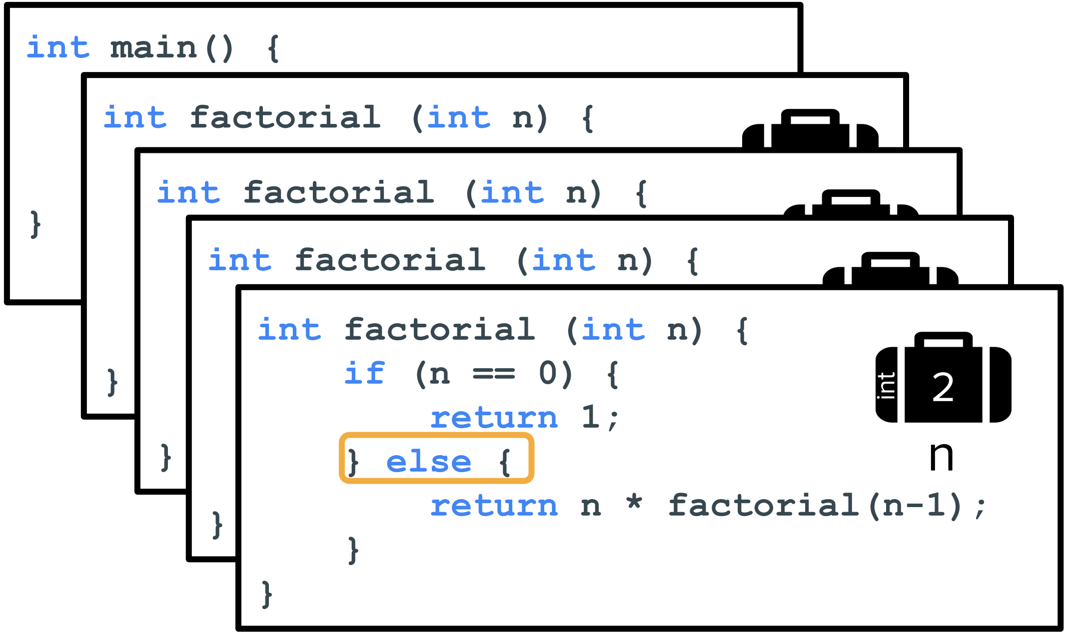
```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

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



Recursion in action



Recursion in action

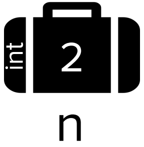
```
int main() {
}
```

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

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

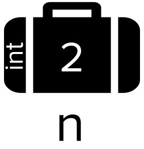
```
int factorial (int n) {
}
```

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



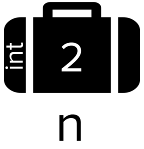
Recursion in action

```
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);
                    }
                }
            }
        }
    }
}
```



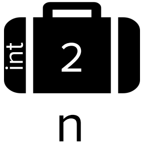
Recursion in action

```
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);
                    }          2
                }
            }
        }
    }
}
```

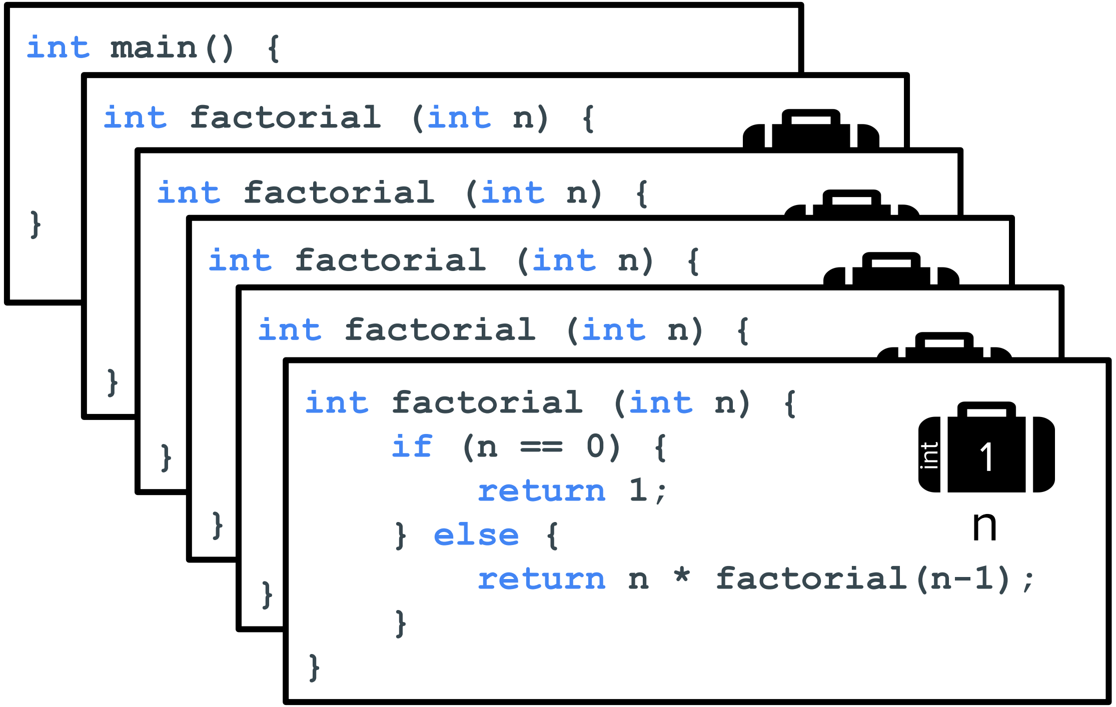


Recursion in action

```
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



Recursion in action

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

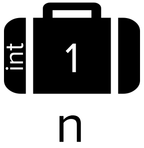
```
int factorial (int n) {
}
```

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

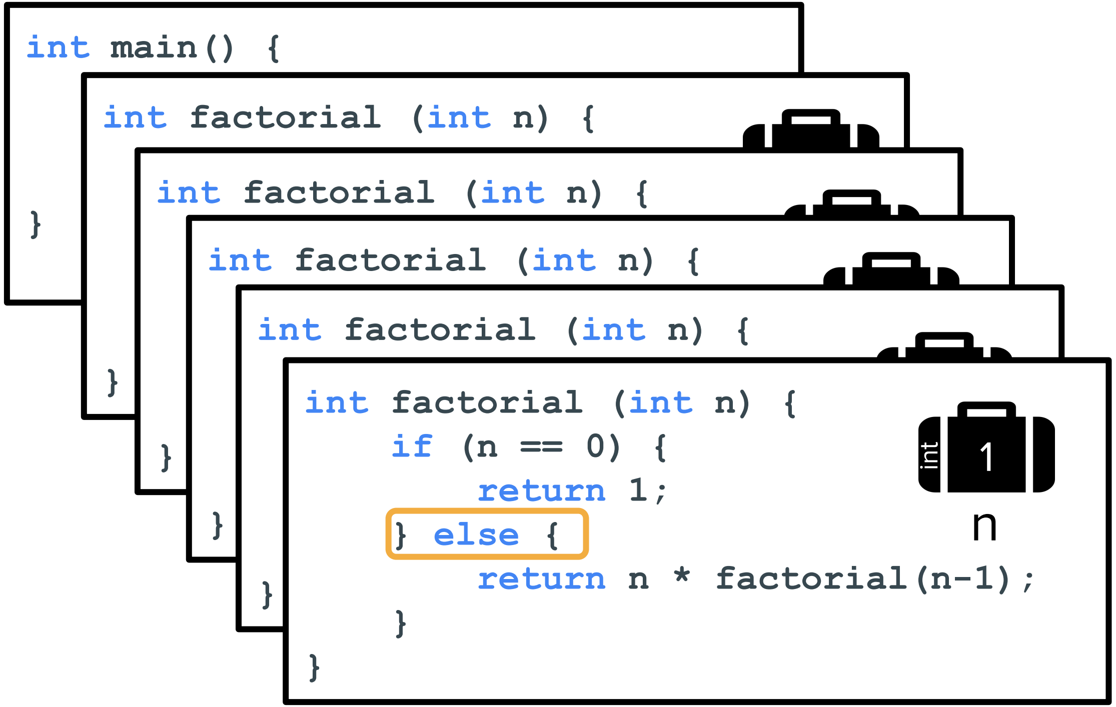
```
int factorial (int n) {
}
```

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

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



Recursion in action



Recursion in action

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

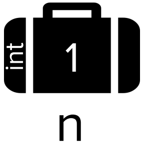
```
int factorial (int n) {
}
```

```
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() {

}
```

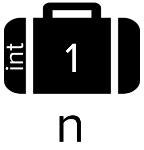
```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

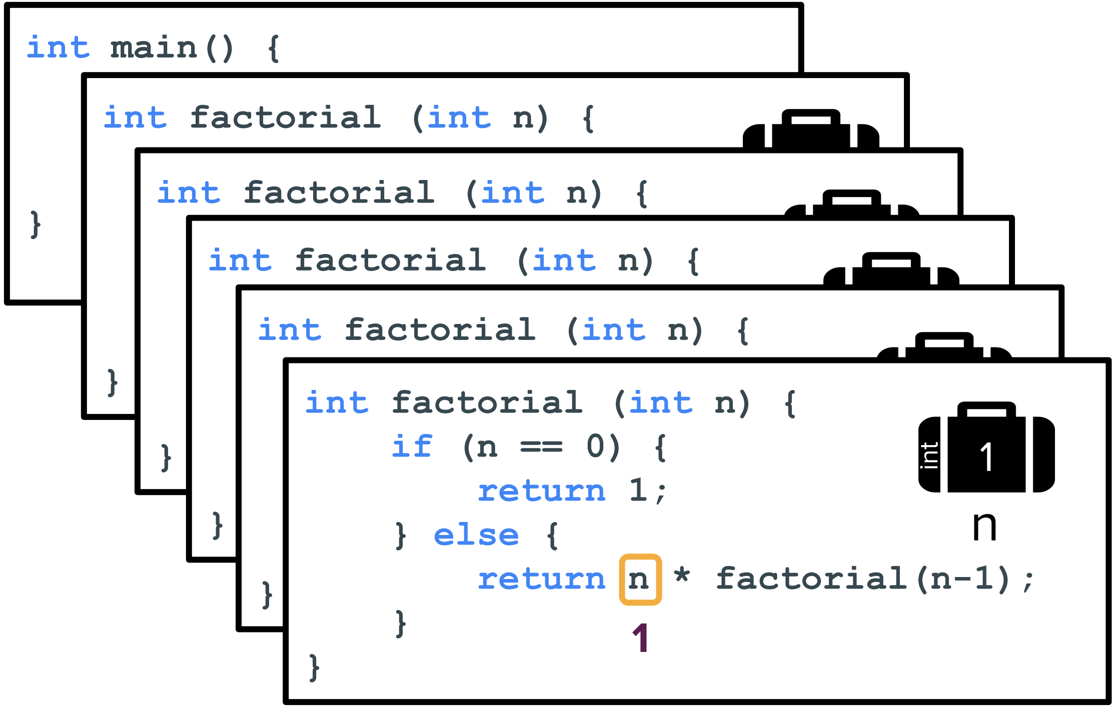
```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

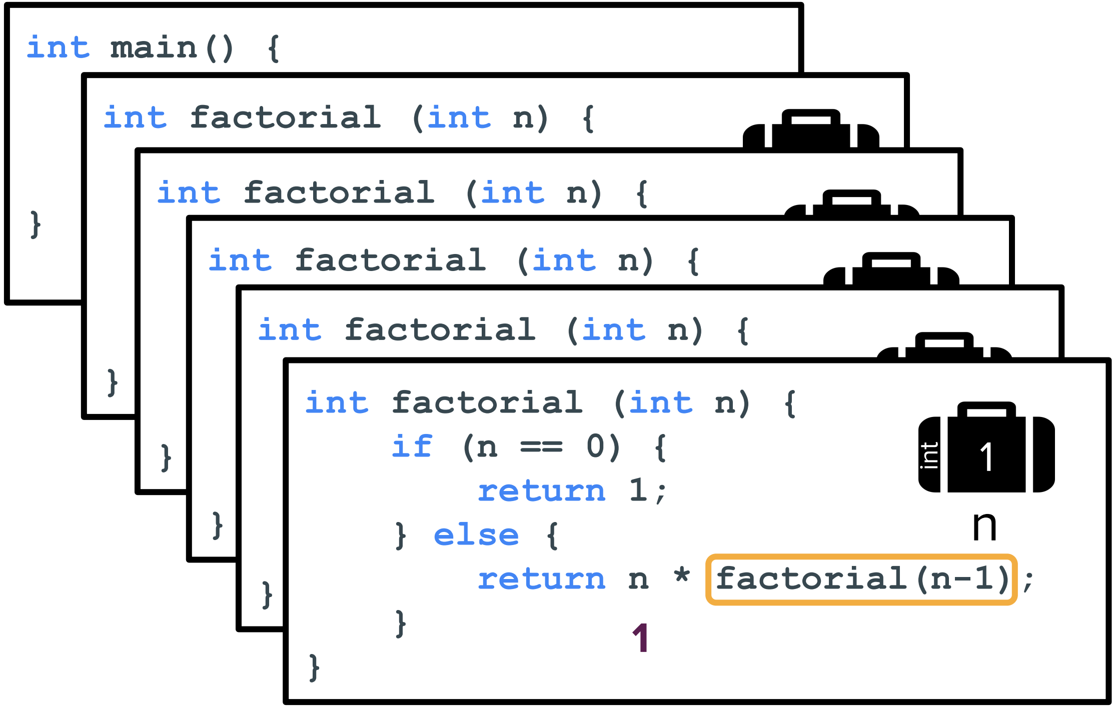
```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```



Recursion in action



Recursion in action



Recursion in action

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

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

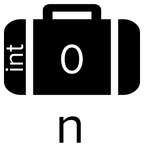
```
int factorial (int n) {
}
```

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

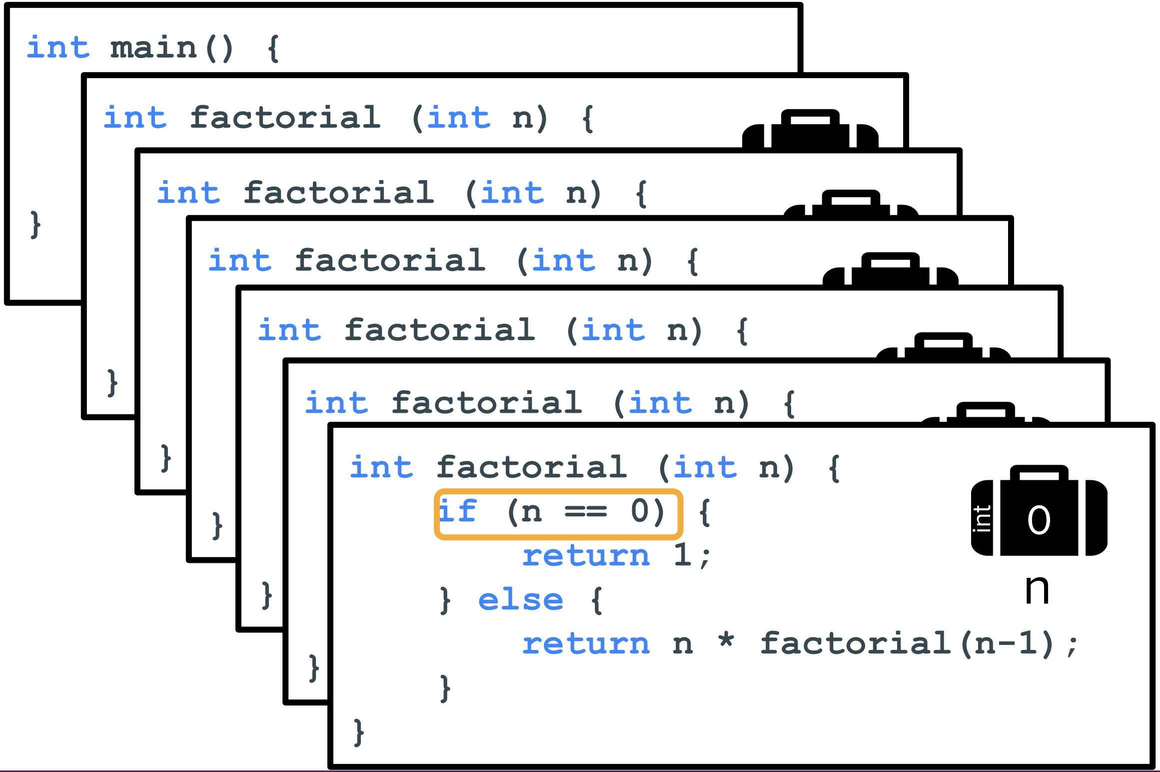
```
int factorial (int n) {
}
```

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

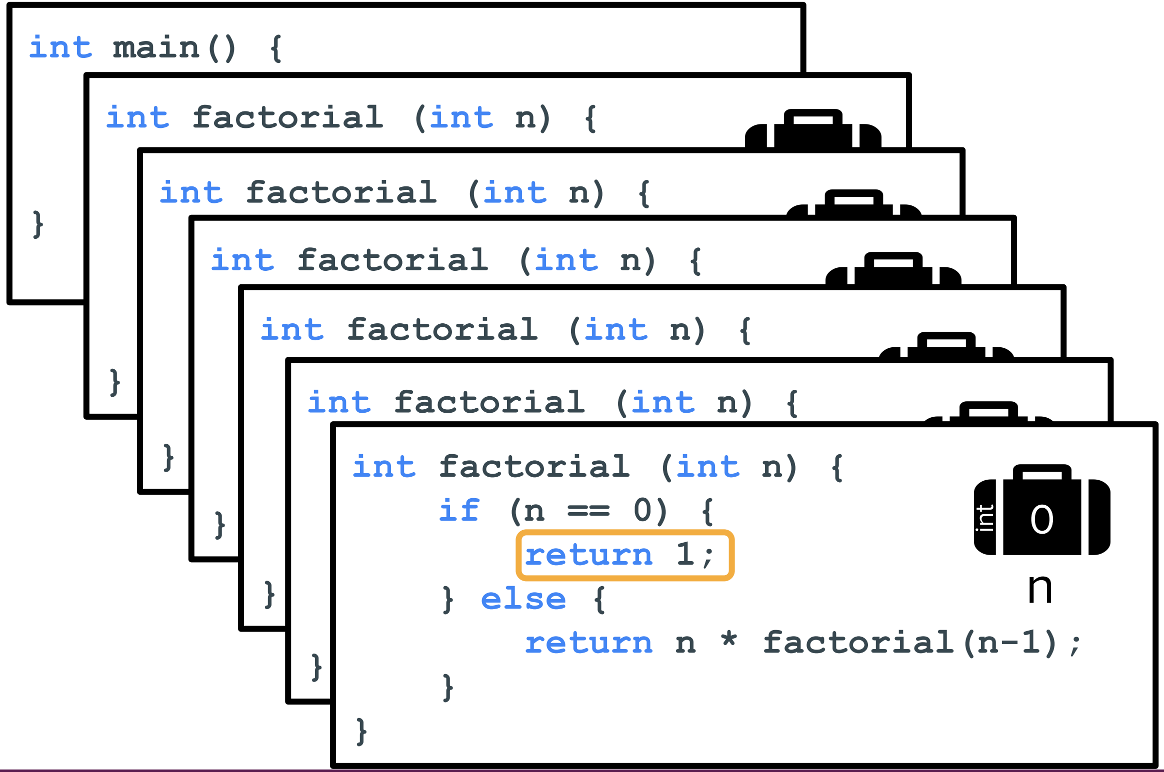
```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```



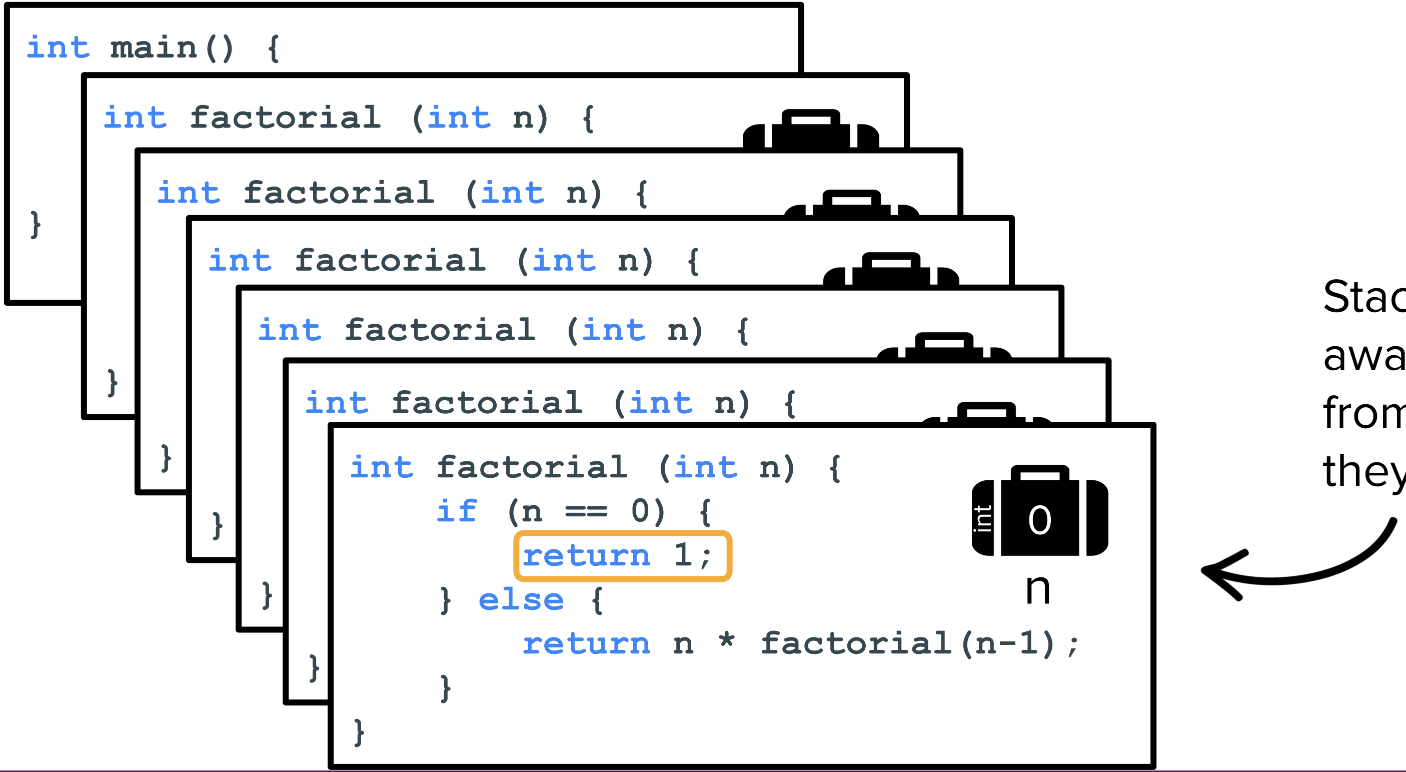
Recursion in action



Recursion in action

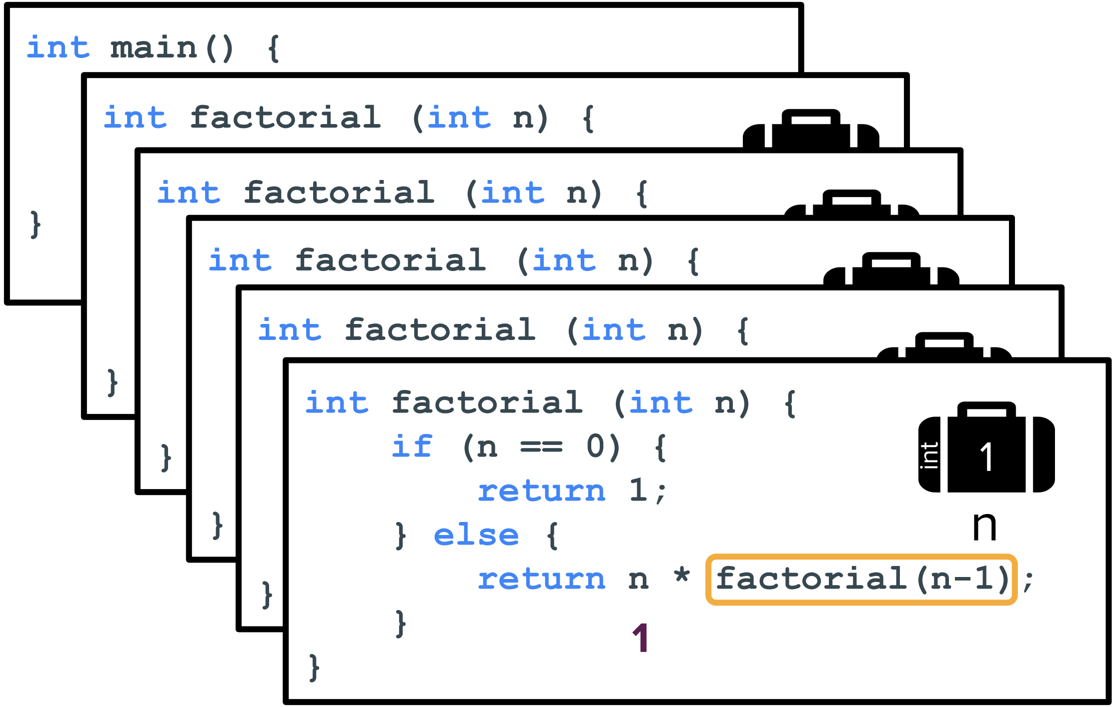


Recursion in action



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

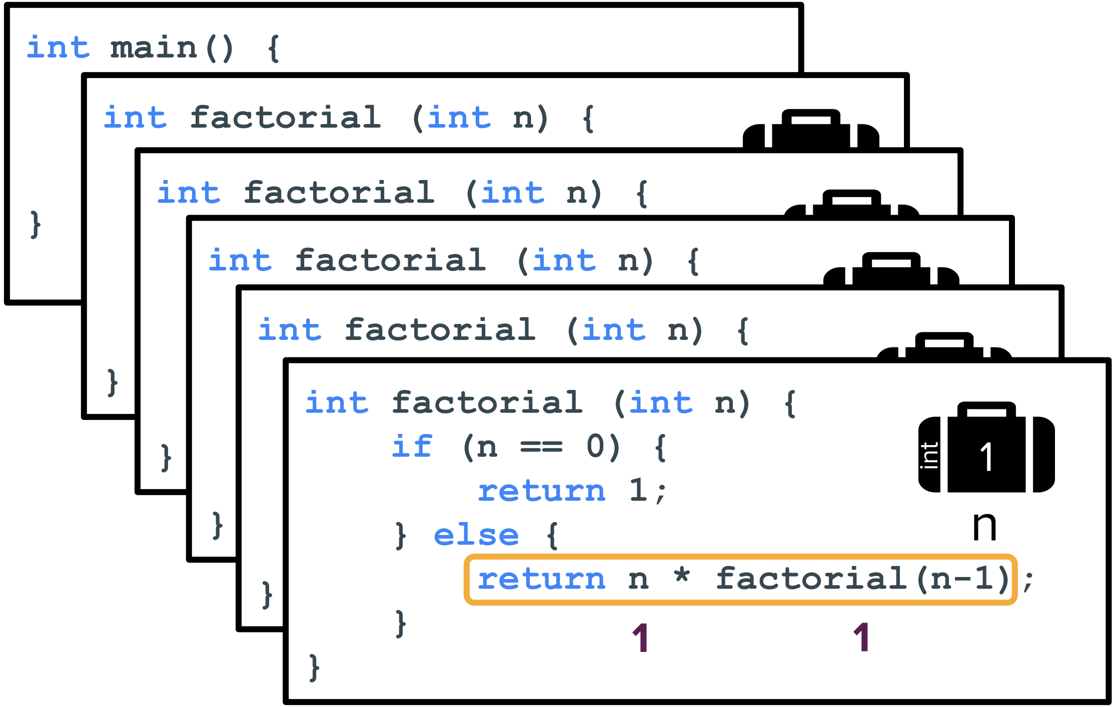
Recursion in action



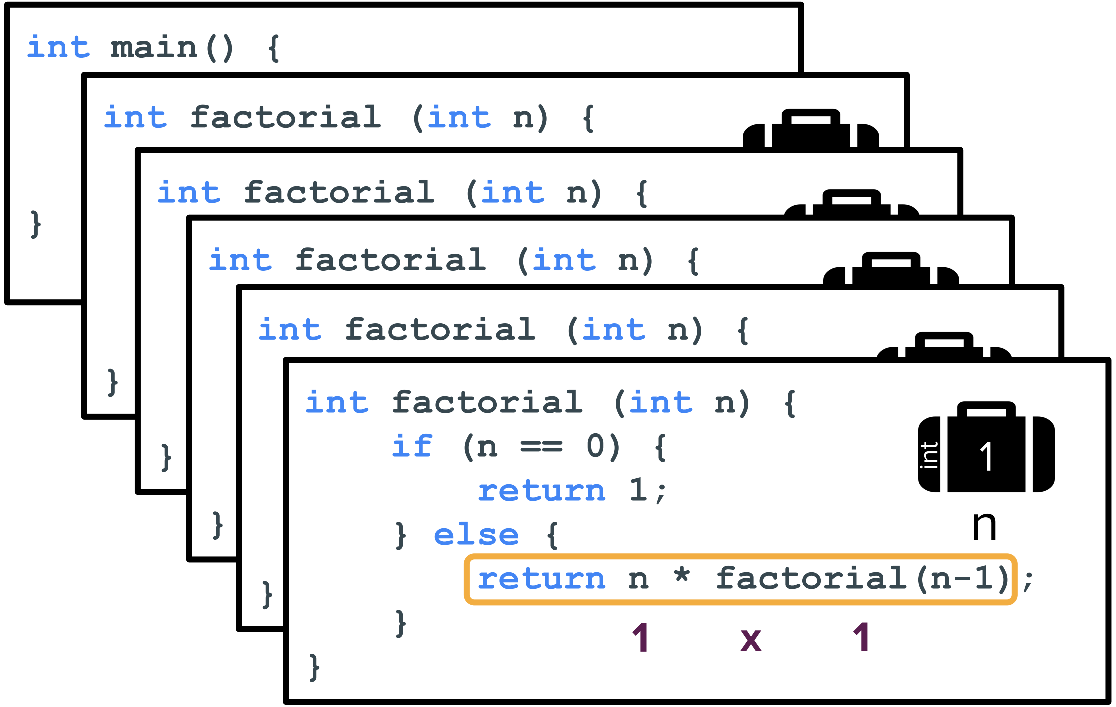
Recursion in action

```
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



Recursion in action



Recursion in action

```
int main() {

}
```

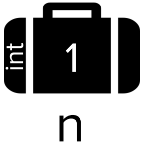
```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

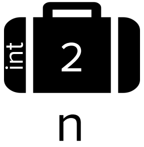
```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```



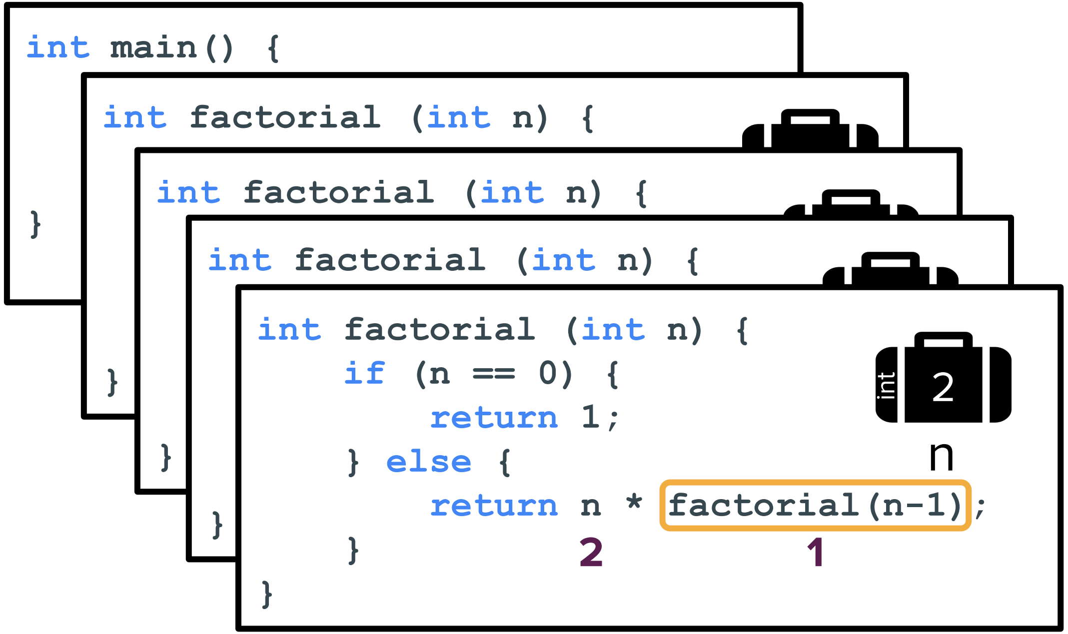
1

Recursion in action

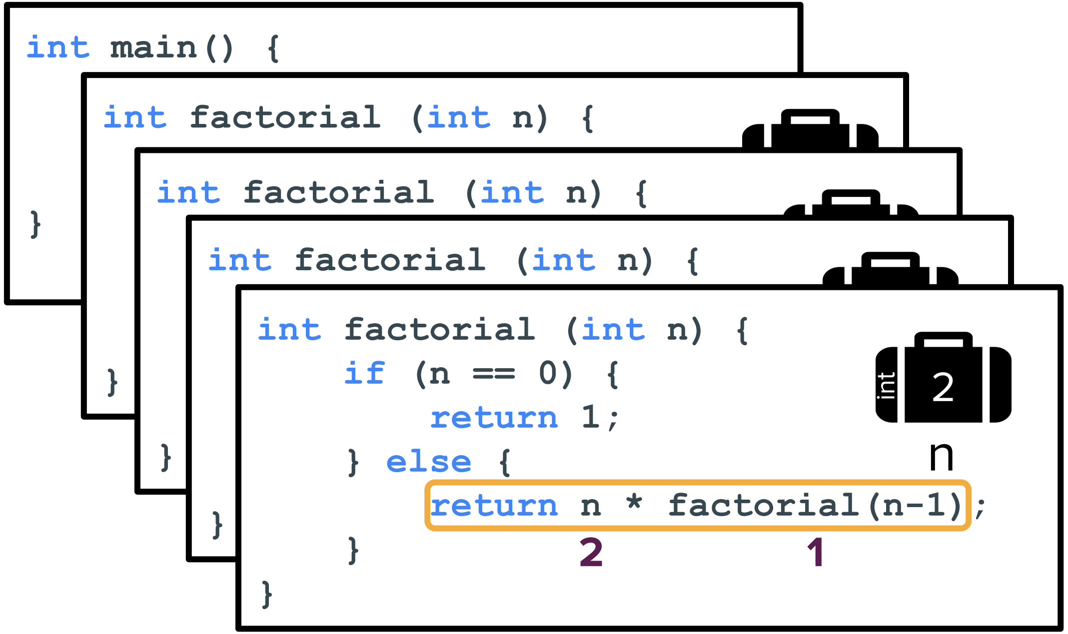
```
int main() {

}
    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
                }
```



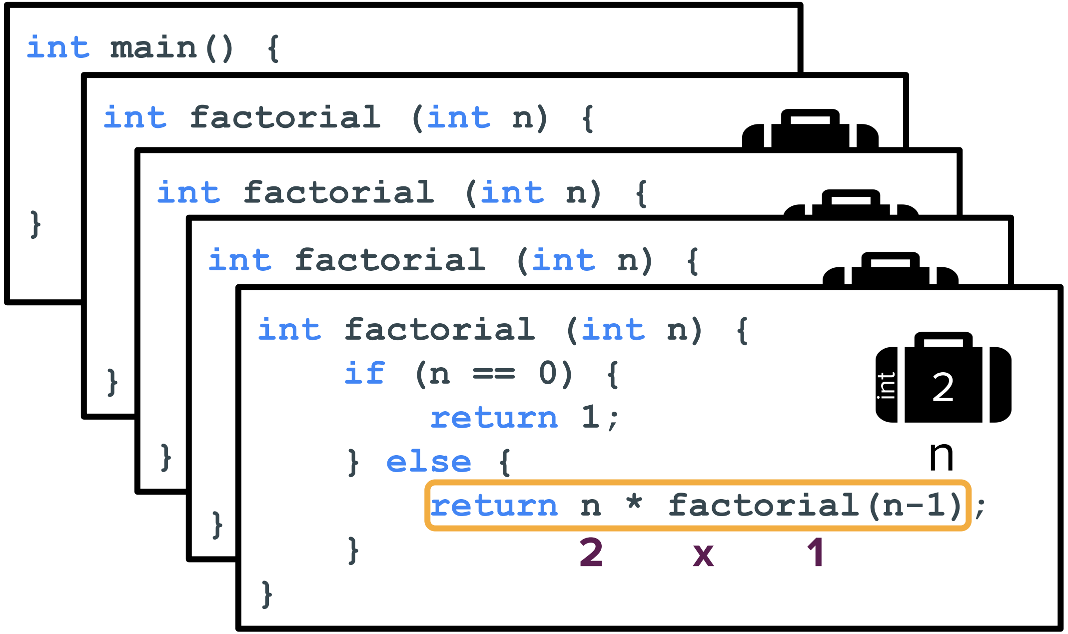
Recursion in action



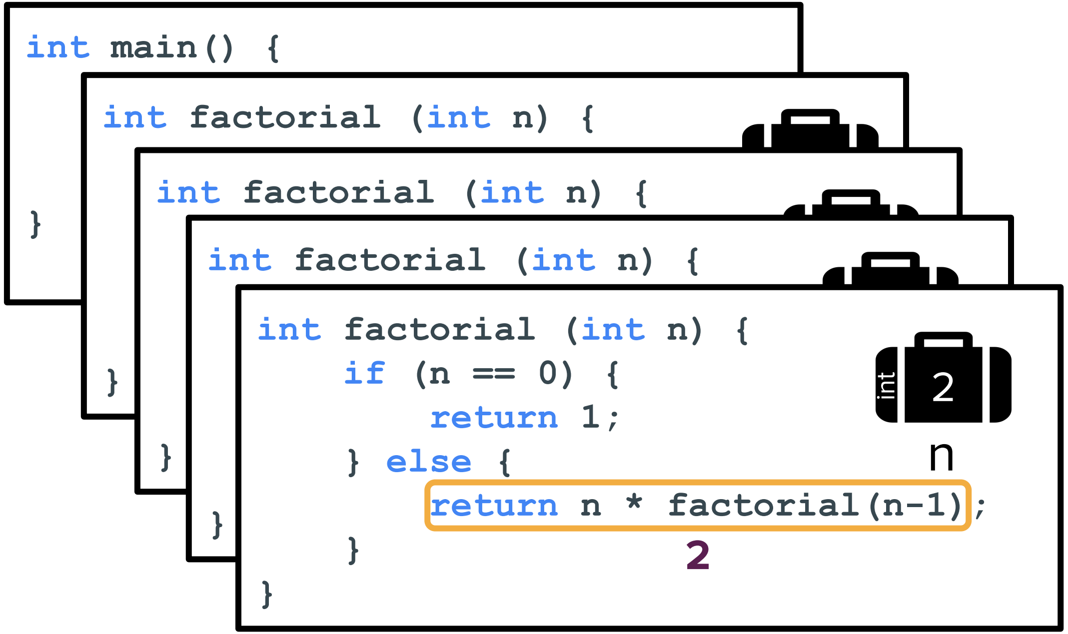
Recursion in action



Recursion in action



Recursion in action



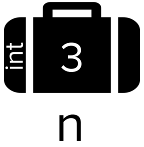
Recursion in action

```
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
}
```



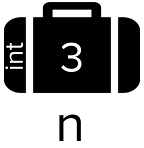
Recursion in action

```
int main() {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {

}
```

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

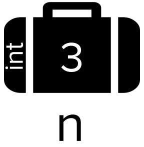


3

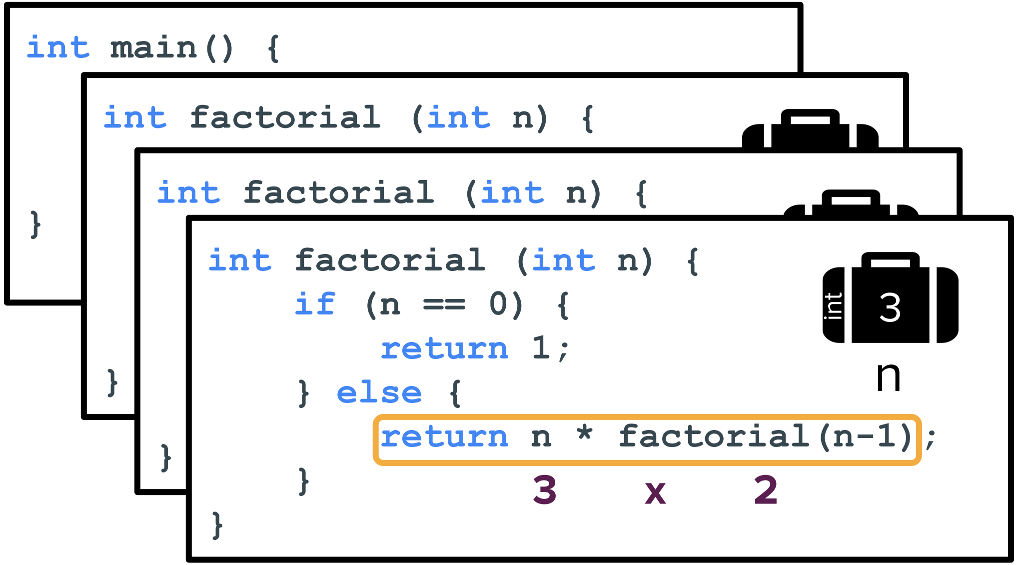
2

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);
                }           3             2
            }
        }
    }
}
```

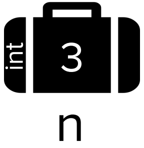


Recursion in action



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);
                }
            }
        }
    }
}
```



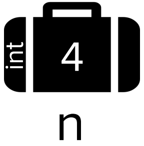
6

Recursion in action

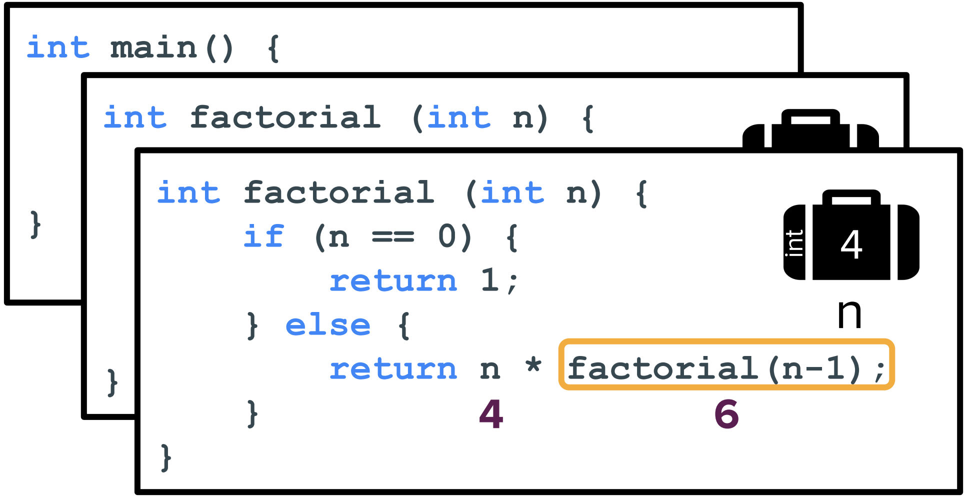
```
int main() {

}
```

```
int factorial (int n) {

}
```

```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }          4
}
```



Recursion in action



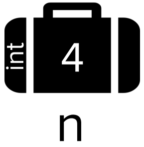
Recursion in action

```
int main() {

}
```

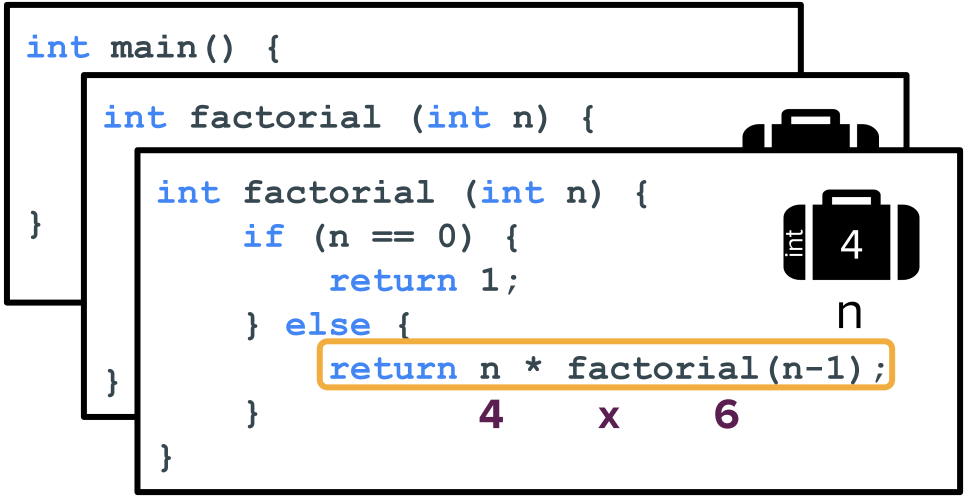
```
int factorial (int n) {

}
```

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

4 6



Recursion in action



Recursion in action

```
int main() {

}
```

```
int factorial (int n) {

}
```

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

n

24

Recursion in action

```
int main() {

}
```

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

n

Recursion in action

```
int main() {

}
```

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

n

5

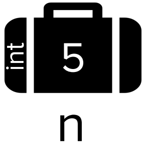
24

Recursion in action

```
int main() {

}
```

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

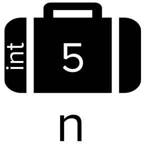
5 24



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);
    }
}
```

n

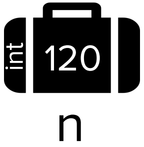
120

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;
}
```



Recursive vs. Iterative

[Qt Creator]

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?

How can we express the recursive case?

- 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?

How can we express the recursive case?

- 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?

How can we express the recursive case?

- 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 reverse a string?

How can we express the recursive case?

- 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 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?

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?

How can we express the recursive case?

- 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 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**.

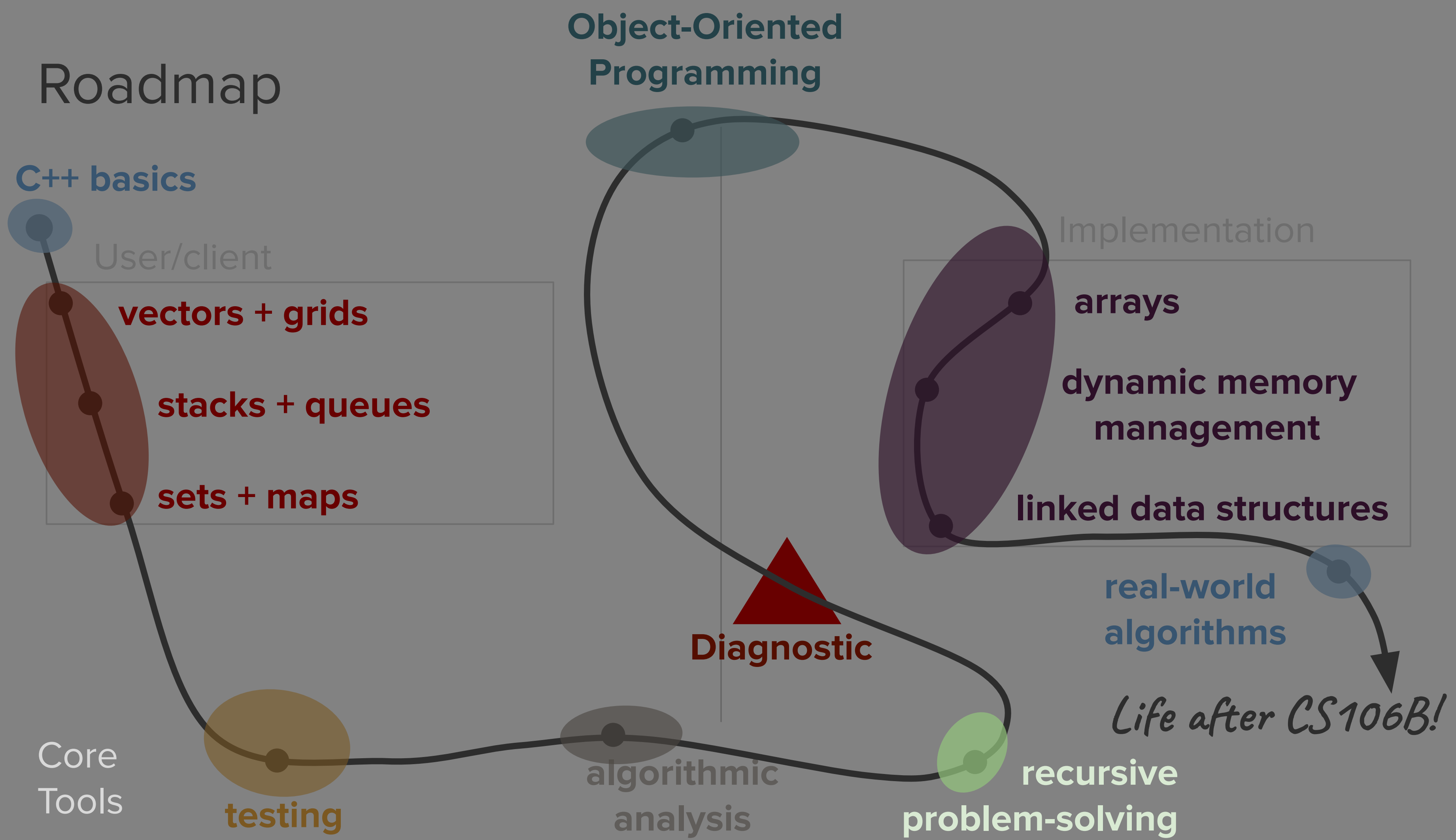
- Recursion has two main parts: the **base case** and the **recursive case**.

- 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**.

What’s next?

Roadmap



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

Fractals