SWEN 601
Software Construction

Recursion & Binary Search

Activity: Getting Started

1. Begin by accepting the GitHub Classroom invitation for today's homework.
    a. *The project **<u>may</u>** already contain some code!*
2. Create a `session` package. This is where you will write your solutions to today's activities.
3. Create a `homework` package. This is where you will implement your solution to the homework.

When you submit your homework, you will include your activities. You may earn up to a 10% bonus on the homework if you have completed all of the activities.

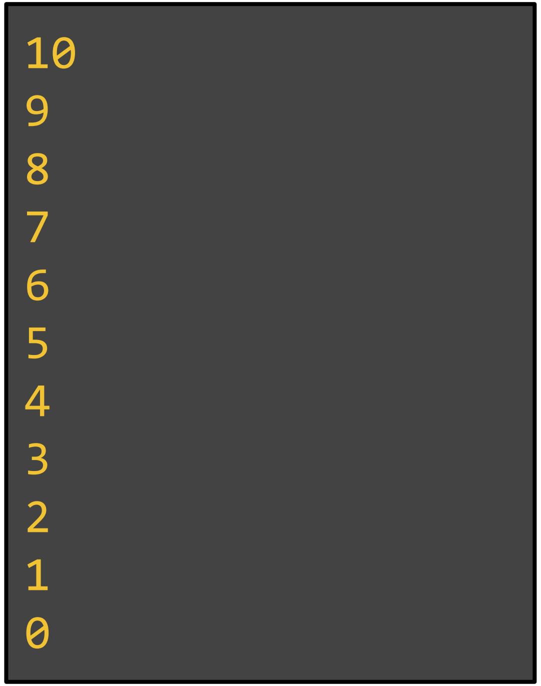
<u>Do not</u> submit code that ***does not compile***. Comment it out if necessary.

Next Two Weeks

WEEK 04	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ			Quiz #10		Quiz #11		
LECTURE			Recursion & Binary Search		Sorts & Complexity		
HOMEWORK	Hwk 10 Due (**11:30pm**)		*Hwk 11 Assigned*		*Hwk 12 Assigned*	Hwk 11 Due (**11:30pm**)	

WEEK 05	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ					Quiz #12		
LECTURE			**No Class (Fall Break)**		Abstract Data Types, Queues, & Stacks		
HOMEWORK	Hwk 12 Due (**11:30pm**)				*Hwk 13 Assigned*		

Activity: Counting Down I

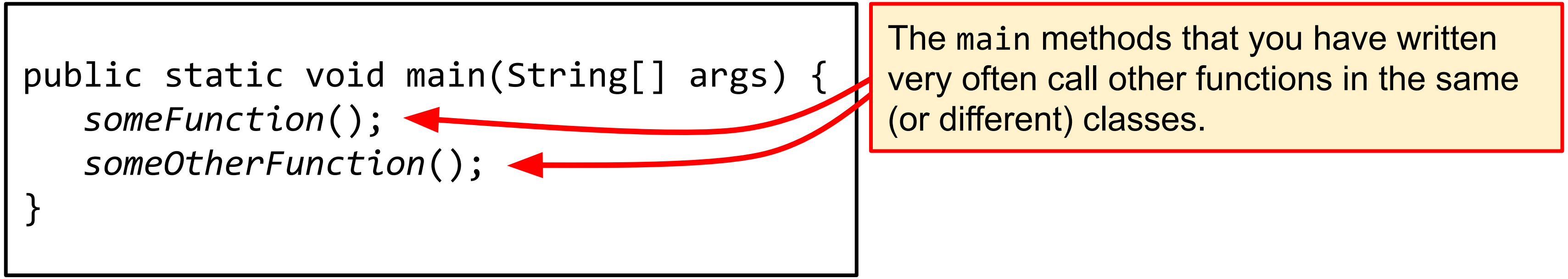


```
10
9
8
7
6
5
4
3
2
1
0
```

- Create a new class named `Recursion`.
- Write a `static` function, `countDown(int n)`, that prints every number from `n` to `0` on a separate line.
  - Use the loop of your choice.
- Write a `main` method and call the function with an argument of `10`.

Recursion

- We’ve already seen many examples of one function calling another function.

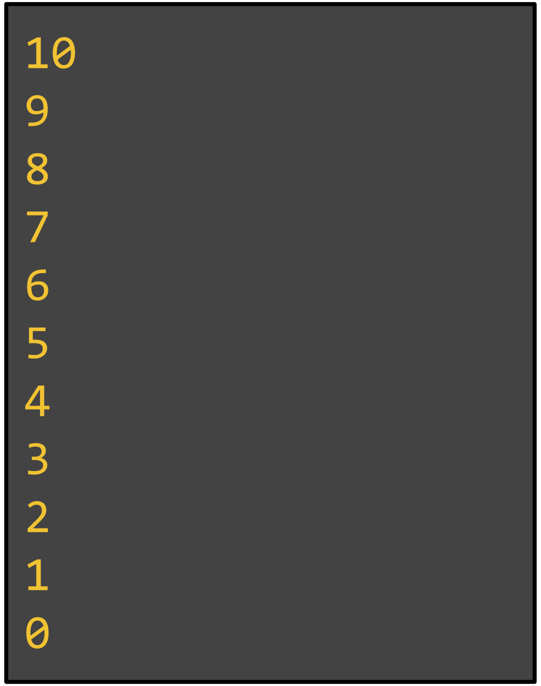


```
public static void main(String[] args) {
    someFunction();
    someOtherFunction();
}
```

The `main` methods that you have written very often call other functions in the same (or different) classes.

- A ***recursive*** function is a function that ***calls itself***.
- Recursive functions are really good at solving large problems by...
    - Doing a small part of the work
    - Calling itself to handle the rest.

Activity: Counting Down II



```
10
9
8
7
6
5
4
3
2
1
0
```

- Write a new `static` function, `countDownRec(int n)`, that prints every number from `n` to `0` on a separate line.
  - ***Do not*** use a loop. Use recursion instead.
- Modify your `main` method to call the new function with an argument of `10`.

Recursion

Q: Remember that a recursive function does one unit of work and then calls itself to handle the rest. So the first question that you need to answer is: what is *one unit of work*?

A: *Print a single number.*

Q: Next, a recursive function ***calls itself*** to handle the rest of the work. In this case, what is *the rest of the work*?

A: *Print the remaining numbers (n-1, n-2, and so on down to 0).*

```
void countDown(int n) {
  System.out.println(n);
  int rest = n - 1;
  countDown(rest);
}
```

Q: How many of you came up with something like this? What is the problem with this solution?

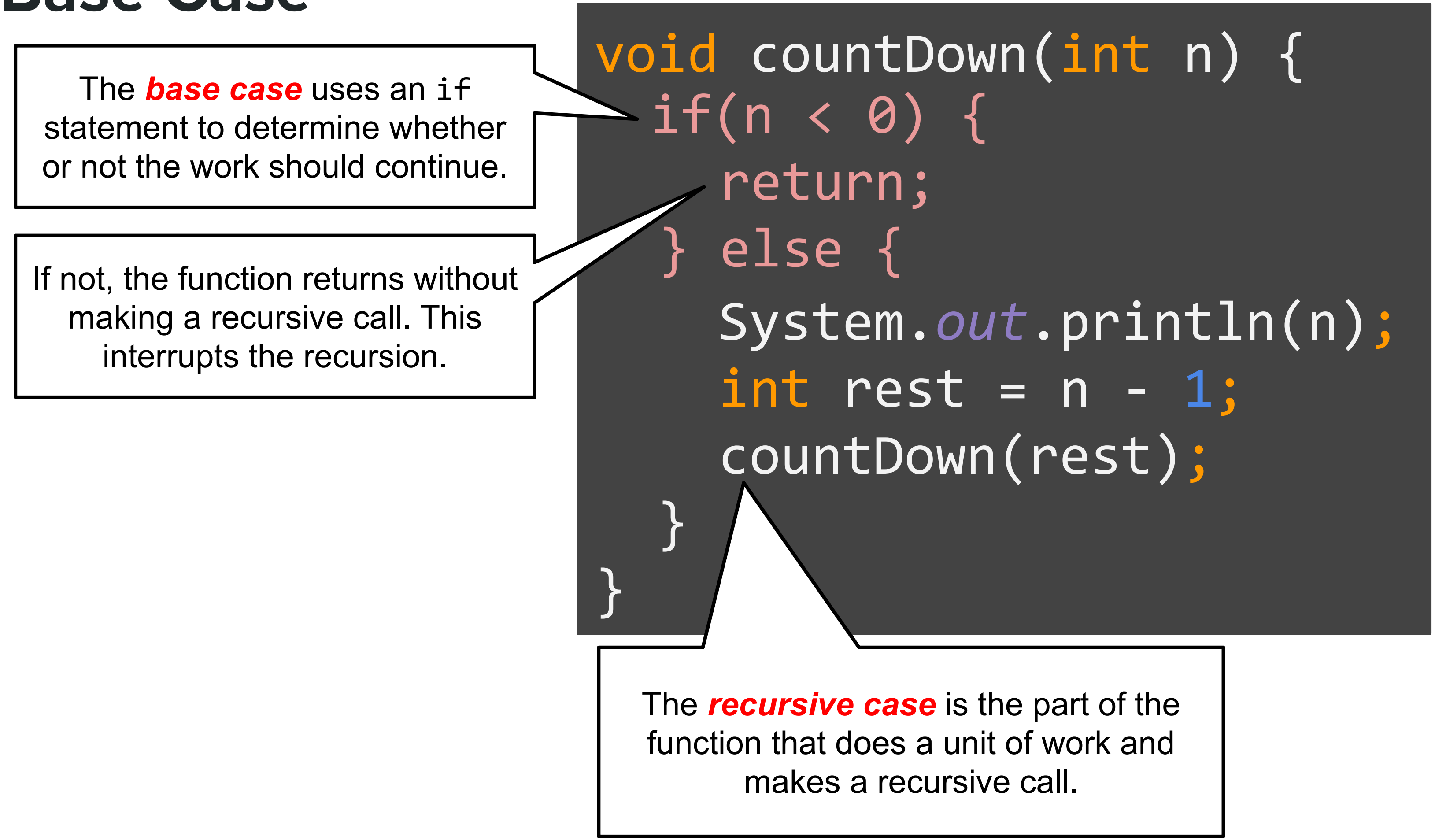
The Base Case

- Q: What is the problem with the function in the previous example?
  - A: It will continue counting down past 0, off into negative infinity.
- A recursive function needs to know when to ***stop*** doing work.
- We need to check some ***condition*** to determine when the work is done.
  - This is is called a ***base case.***
- The ***base case*** is the part of the function that determines when to ***stop*** and ***does not*** make a recursive call.
  - The base case stops recursion from continuing indefinitely.

Q: What should the base case for the countdown function look like?

```
if(n < 0) {
   return;
}
```

The Base Case



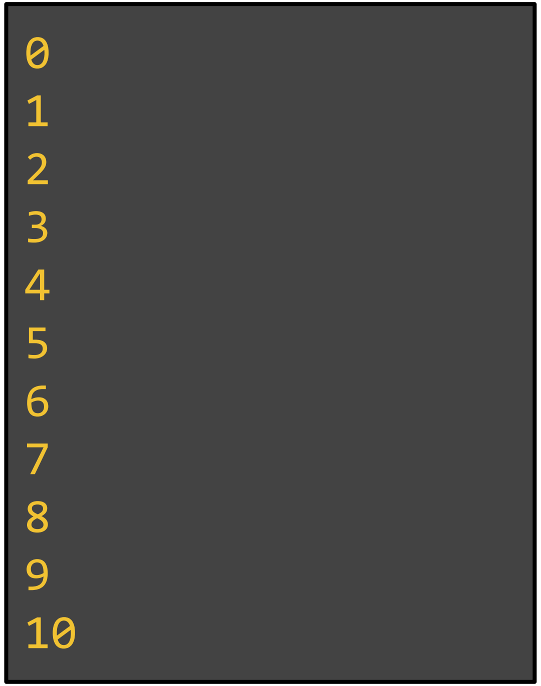
```
void countDown(int n) {
 if(n < 0) {
  return;
 } else {
  System.out.println(n);
  int rest = n - 1;
  countDown(rest);
 }
}
```

The ***base case*** uses an `if` statement to determine whether or not the work should continue.

If not, the function returns without making a recursive call. This interrupts the recursion.

The ***recursive case*** is the part of the function that does a unit of work and makes a recursive call.

Activity: Counting Up

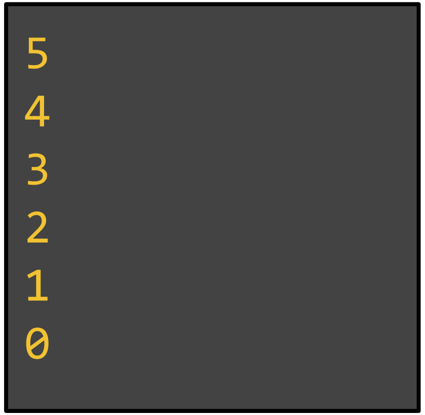


```
0
1
2
3
4
5
6
7
8
9
10
```

- Create a new `static` function, `countUp(int n)`, that counts **up** from 0 to n (rather than down).
    - Counting starts at 0.
    - It should take a single parameter that is the number at which counting stops.
    - As before, each number should print on a separate line.

Recursion and Return Values

- It is very often the case that a recursive function should ***return*** some computed value.
- Each call to the function
  - ***Computes*** its part of some total.
  - ***Combines*** its part with the return value from the recursive call.
- For example: what if we wanted to know the total value of all of the numbers printed by the countdown function?
  - Each call to the function prints the value that is passed in as an argument to its parameter.
  - How would the total of all of the numbers be calculated?
  - Let’s look at it as a mathematical function...

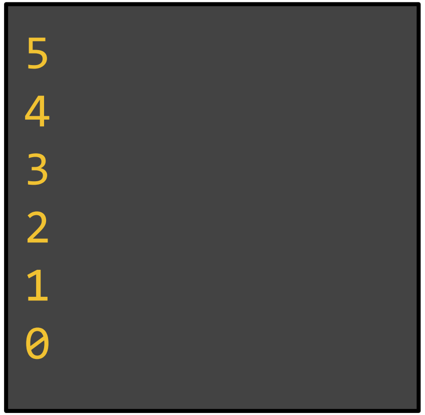


```
5
4
3
2
1
0
```

What is the total value of all of the numbers that are printed?

Recursion and Return Values

- The following is a recursive definition for a function, ***T***, that can be used to compute the total value of all the numbers printed by the count_down function:
    - **$T(0) = 0$**, which is to say the total value of all of the numbers printed is 0.
    - **$T(n) = n + T(n-1)$**, where $n > 0$, which is to say that the total value of all of the numbers printed when the first number is n, is n plus the value returned by $T(n-1)$.
    - **$T(n)$** where n <0 is 0.

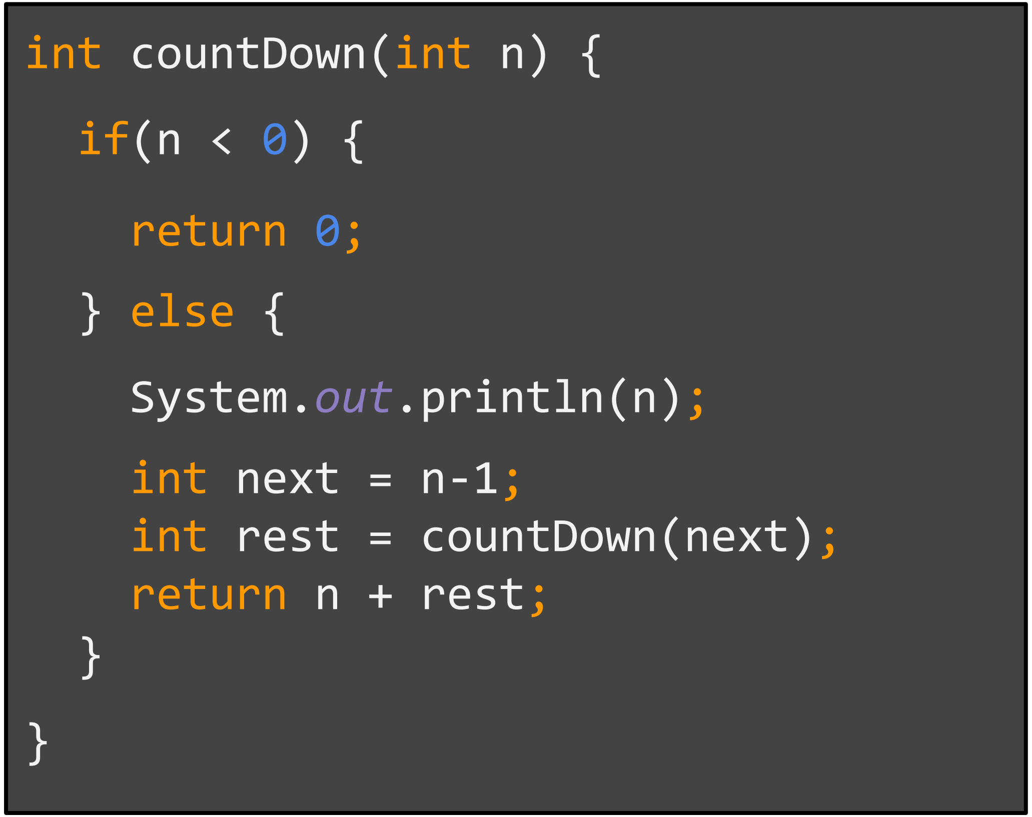


```
5
4
3
2
1
0
```

Using this definition, can we modify the existing `countDownRec` function to return the total of all the numbers printed?

Recursion and Return Values

- The recursive definition of calculating the total of the numbers printed by count_down:
  - $T(0) = 0$
  - $T(n) = n + T(n-1)$
  - $T(n)$ where $n<0$ is 0.



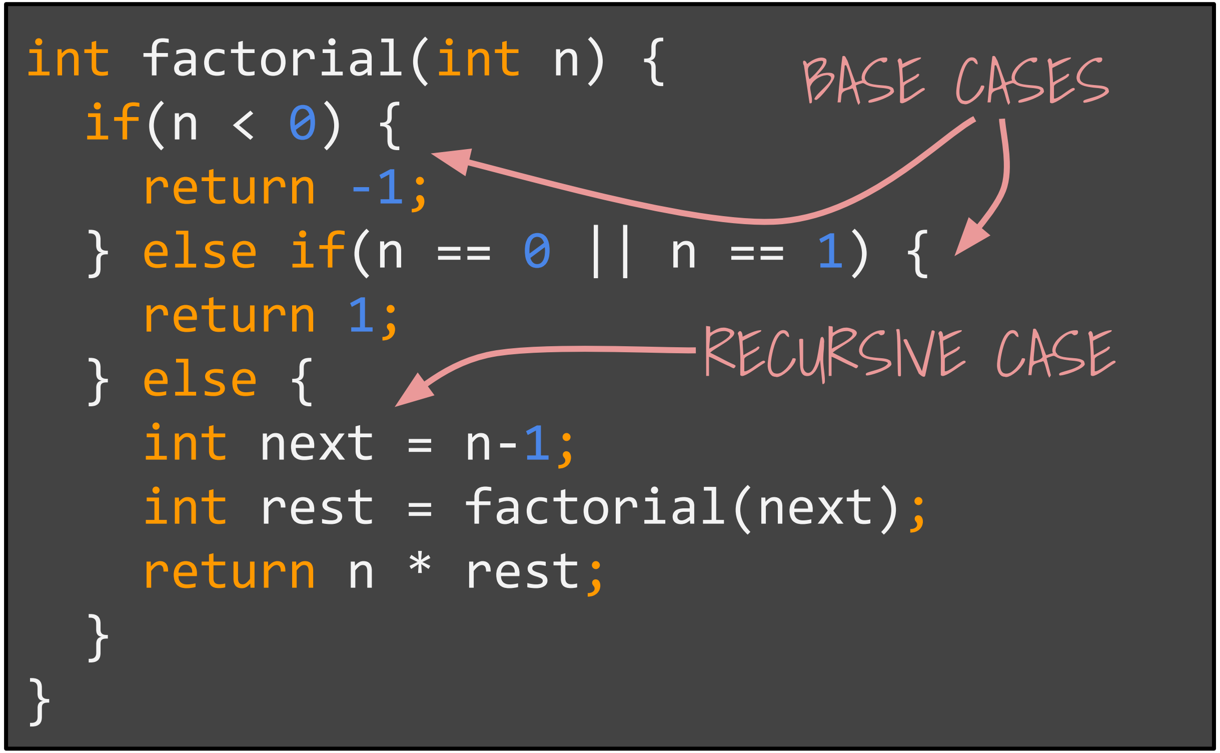
```java
int countDown(int n) {
  if(n < 0) {
    return 0;
  } else {
    System.out.println(n);
    int next = n-1;
    int rest = countDown(next);
    return n + rest;
  }
}
```

Activity: Factorial!

- Write a function that calculates the factorial for any positive value of N.
  - The factorial, N!, of a number N is N * N-1 * N-2 * N-3 * ... * 1.

The recursive definition of factorial is:

- *F(0) = 1*
- *F(1) = 1*
- *F(N) = N * F(N-1)*
- *F(N)* where *N* is less than 0 is undefined.



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

BASE CASES
RECURSIVE CASE

Activity: Fibonacci!

- Write a function that returns the NTH number in the Fibonacci sequence.

The recursive definition of Fibonacci is:

- *F(1) = 0*
- *F(2) = 1*
- *F(N) = F(N-1)+F(N-2)*
- *F(N)* where *N* is less than 0 is undefined.

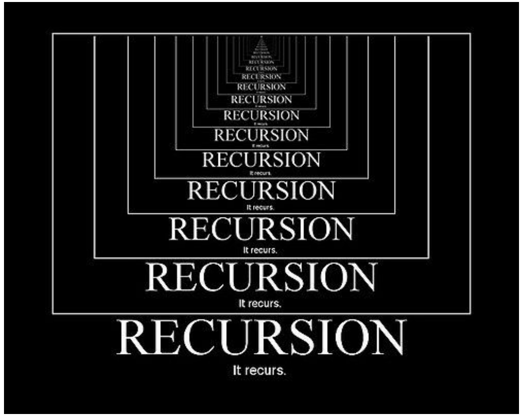
```
int fibonacci(int n) {
  if(n < 0) {
    return -1;
  } else if(n == 1 || n == 2) {
    return n - 1;
  } else {
    int fn_1 = fibonacci(n-1);
    int fn_2 = fibonacci(n-2);
    return fn_1 + fn_2;
  }
}
```

BASE CASES

RECURSIVE CASE

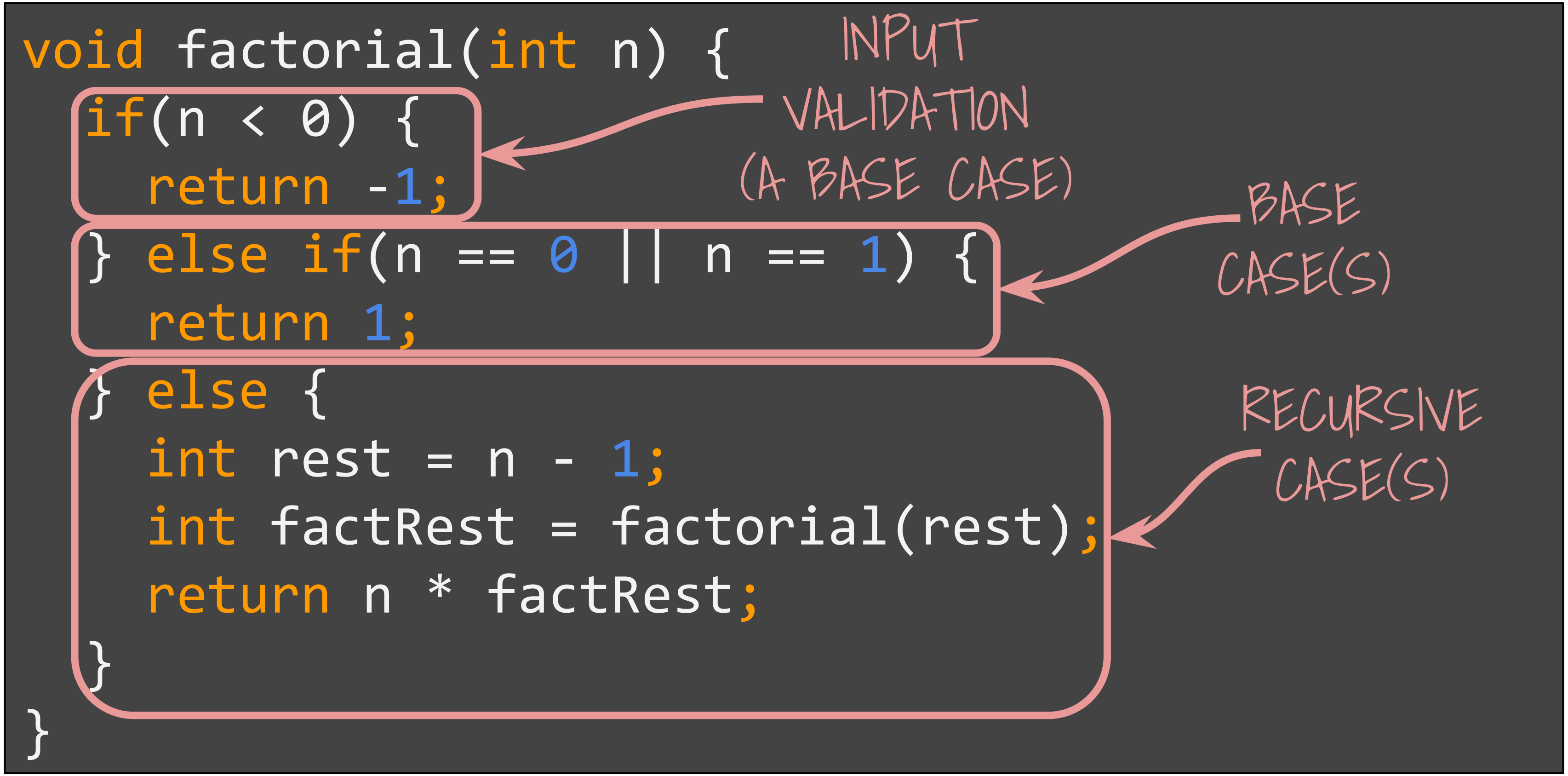
The Recursive Pattern

- By now you should notice that many recursive functions follow a similar ***pattern***:
  - There is often some kind of ***input validation*** e.g. is n >= 0?
  - One or more ***base cases*** (there may be more than one) - finishes up, stops recursion.
  - A ***recursive case*** that does the following:
    - Do one unit of work (e.g. print n)
    - Prepare for the recursive call (e.g. subtract 1 from n)
    - Make a recursive call
- Most of the recursive functions that we write will follow this pattern.
  - The recursive case may be in a different order.



Input validation is really a kind of ***base case***; when does recursion stop?

A Typical Recursive Pattern



```
void factorial(int n) {
  if(n < 0) {
    return -1;
  } else if(n == 0 || n == 1) {
    return 1;
  } else {
    int rest = n - 1;
    int factRest = factorial(rest);
    return n * factRest;
  }
}
```

INPUT VALIDATION (A BASE CASE)

BASE CASE(S)

RECURSIVE CASE(S)

Variations in the Pattern

There are many variations in the pattern. For example, sometimes the recursive call is made ***after*** the unit of work...

```
void countDown(int n) {
  if(n < 0) { # base case
    return;
  } else {    # recursive case
    # complete a unit of work
    System.out.println(n);
    # prepare for recursion
    int next = n - 1;
    # make the recursive call
    countDown(next);
}
```

...and sometimes the recursive call is made ***before*** the unit of work...

```
void countUp(int n) {
  if(n < 0) { # base case
    return;
  } else {    # recursive case
    # prepare for recursion
    int next = n - 1;
    # make the recursive call
    countUp(next);
    # complete a unit of work
    System.out.println(n);
  }
}
```

...and sometimes there may be ***multiple*** base cases and/or ***multiple*** recursive calls.

```
int fib(int n) {
  if(n < 1) { # base case
    return -1;
  } else if(n == 1 || # base case
            n == 2) {
    return n - 1;
  } else { # recursive case
    # prepare and make call
    int fn_1 = fib(n - 1);
    int fn_2 = fib(n - 2);
    # complete a unit of work
    return fn_1 + fn_2;
  }
}
```

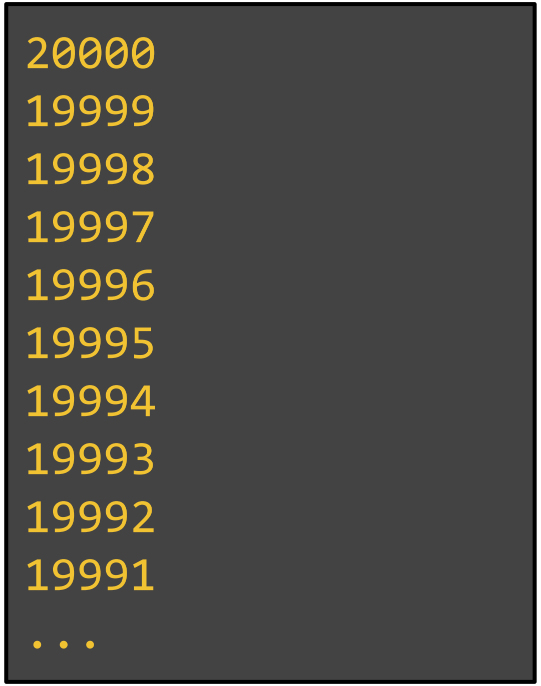
Recursion Depth

- ***Recursion depth*** refers to the number of times a recursive function call is made before hitting the base case.
  - Each recursive call adds a ***stack frame*** to the ***call stack***.
  - Stack Frames are only removed when a function returns.
  - Recursive functions only return when the base case is reached.
- There is a limit to the number of stack frames that can be added to the call stack.
  - Depending on your configuration, the default may be **5000-10000** or so.
  - Exceeding the limit will crash you program.
  - We call this **“blowing up the stack.”**

Because of this limit, we need to be careful when writing and calling recursive functions.

Let’s take a look at how easy it is to ***blow up the stack***.

Activity: Blow Up The Stack!



```
20000
19999
19998
19997
19996
19995
19994
19993
19992
19991
...
```

- Call one of the recursive functions that you have written today with a value large enough to blow up the stack.
  - For example, modify your `main` function to call either your `countUp` or `countDown` function with n=`20000`.

Q: So why does this matter?

Depending on how a recursive function is implemented, the scope of the problem is limited by the size of the stack.

Activity: Searching

- Write a function, `search(int[] array, int target)`, that returns `true` if the target value is in the array, and `false` otherwise.

Intro to Complexity

- The algorithm that you wrote to search almost certainly looked something like this:

```java
boolean search(int[] array, int target) {
  for(int value : array) {
    if(value == target) {
      return true;
    }
  }
  return false;
}
```

- It the length of the array is N, how many comparisons would this algorithm need to make if the target value is ***not*** in the array?
- How would this change as N increases (i.e. the size of the array gets bigger)?

Even though computers are very fast, the amount of time that it takes to perform an operation is greater than 0.

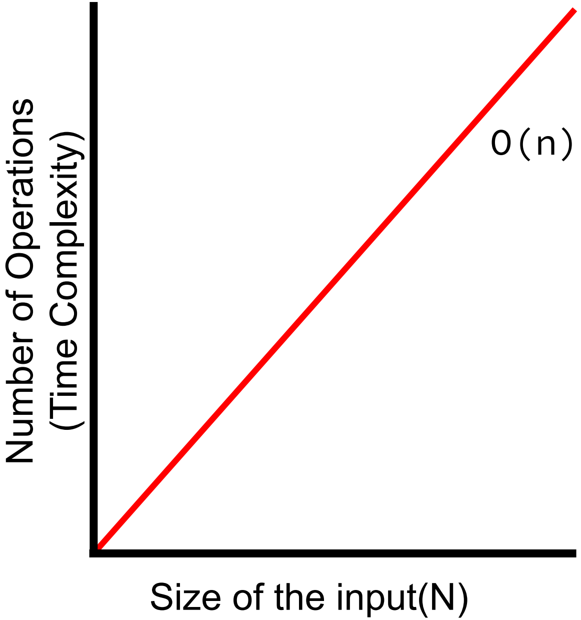
The more operations an algorithm needs to perform, the longer it takes to run.

Analysis of how much longer an algorithm will take to run as the size of its input (N) increases is referred to as ***time complexity***.

We will talk about time complexity a little today, and more in the next session.

Linear Time

- Each time the size of the array being searched increases by 1, the search algorithm needs to perform 1 additional operation in the ***worst case***.
    - In the average case, the number of operations will be N/2.
- This means that the number of operations increases at the same rate that N increases.
- This means that the algorithm runs in ***linear time***, which is to say that the rate at which the time complexity increases relative to the input is linear.
- This is represented in “Big O” notation as O(N).



Q: If the array of values is ***unsorted***, is there any alternative to a linear search?

Q: What if the array is ***sorted***?

Binary Search

- ***Binary Search*** can dramatically outperform a linear search if the data is in sorted order.
- Binary search works like this:
    - Begin with a **start** index of 0 and an **end** index of length-1.
    - Examine the value at the **mid** point (half way between the start and the end).
        - If the **mid** value is the target, return true.
        - If the **mid** value is larger than the target, set **end** to **mid - 1**.
        - If the **mid** value is less than the target, set **start** to **mid + 1**.
    - Repeat until the target is found, or **start > end**.

Binary Search is a ***divide and conquer*** algorithm because it ***divides the input in half*** with each iteration.

Because of this, Binary Search has a time complexity of **O(Nlog$_2$N)**.

Given an array with 1 million values in it, a linear search would need to compare all 1 million values to the target.

Binary search would need to compare about 20.

Binary Search

2	9	10	12	21	36	37	38	39	41	56	57	77	79	83	93	98
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

The first iteration of binary search sets start to `0` and end to `length-1` (in this case `16`).

In each iteration, the value at `mid = (start + end) // 2` is compared to the target value.

If the value does not match the target, the `start` and `end` indexes are updated to `start = mid + 1` if the value is before the target or `end = mid - 1` if the value is after the target.

target = 21							
iteration	0	1	2	3	4	5	6
start	0	0	4	4			
end	16	7	7	4			
mid	8	3	5	4			
value	39	12	36	21			

Activity: Practicing Binary Search

Using the list below, fill out the tables to perform a binary search for the specified target values.

2	9	10	12	21	36	37	38	39	41	56	57	77	79	83	93	98
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

target = 79							
iteration	0	1	2	3	4	5	6
start							
end							
mid							
value							

target = 11							
iteration	0	1	2	3	4	5	6
start							
end							
mid							
value							