Pre-Class Work 10: Recursive Programming

How to Write Recursive Code [Background Reading]

Now that you know how to *read* recursive code, let's talk about how we would go about *writing* recursive code.

Recall that two components of recursive code is the **base case** and the **recursive case**. You may find it easier to think about the **base case** first before thinking about the **recursive case** or, vice versa, think about the **recursive case** before thinking about the **base case**.

This is the beauty of **recursion**! It is largely up to you and how you want to approach the problem.

For this reading, we will be approaching the **base case** first for all of the problems.

1️⃣ Base Case

Let's first consider the **base case**. In the previous reading, we defined the **base case** as the case which ends the **recursion** so it shouldn't make any **recursive** calls. We're going to add onto this definition and say that the **base case** is the solution for the "simplest" case.

Let's revisit the problem which we presented in the previous reading. Given an integer `n`, we want the sum of numbers from 1 all the way to `n` (inclusive). We also made the assumption that `n` is always greater than or equal to 1. What is the simplest case in this problem? In this problem, the simplest case would be if `n` is 1 since all of the numbers between 1 to 1 is just 1!

Therefore, our base case is when `n` is 1 and we should just return 1:

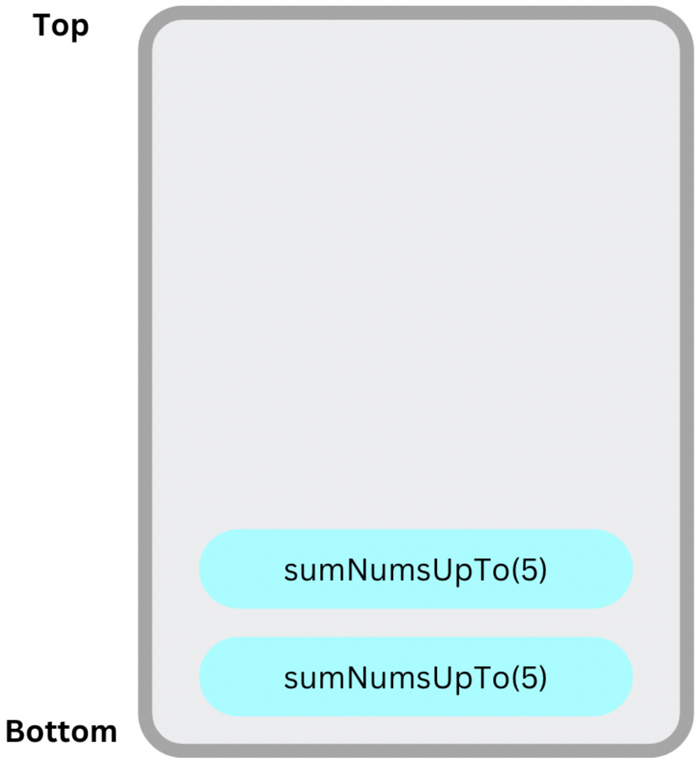
```java
// pre: n >= 1
public static int sumNumsUpTo(int n) {
    if (n == 1) { // Base case
        return 1;
    }
}
```

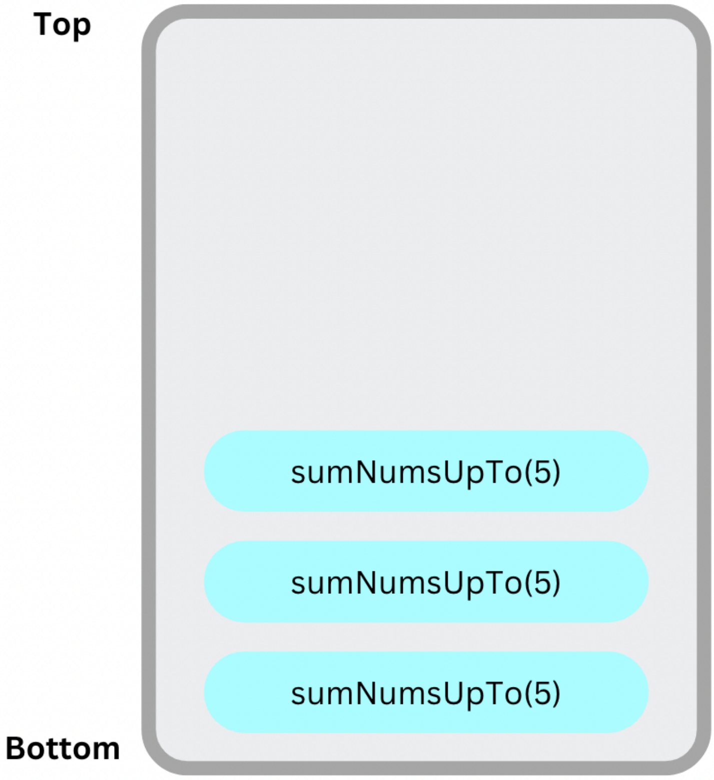
2️⃣ Recursive Case

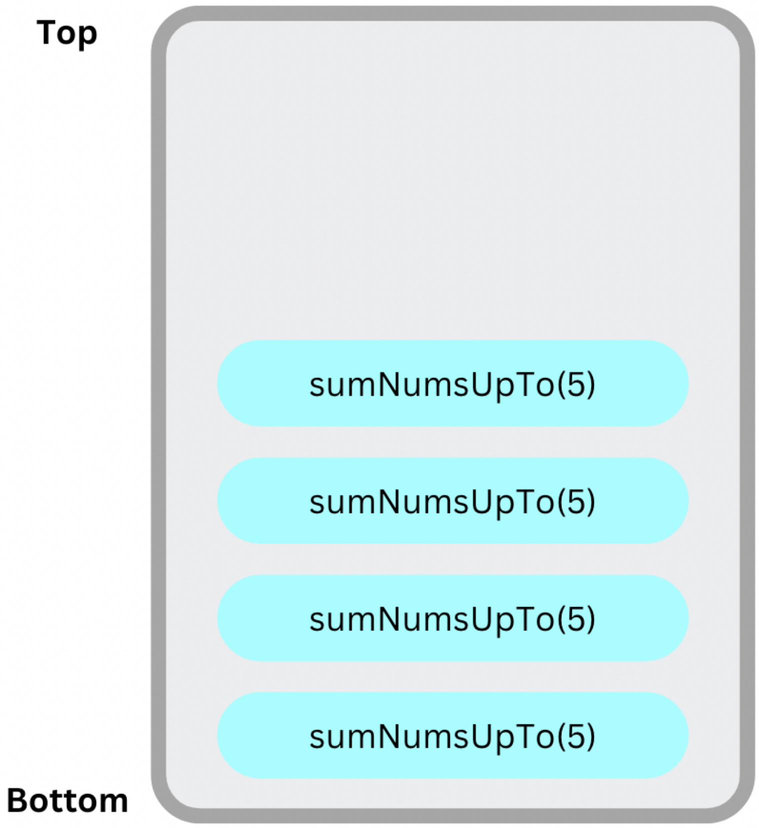
Now, let's look at the **recursive case**. We previously defined the **recursive case** as the one calling the method. One cool fact to realize is that if something is not considered a **base case**, then it is automatically a **recursive case**. Thus, every input `n` that is equal to 2 or greater should go into the recursive case!

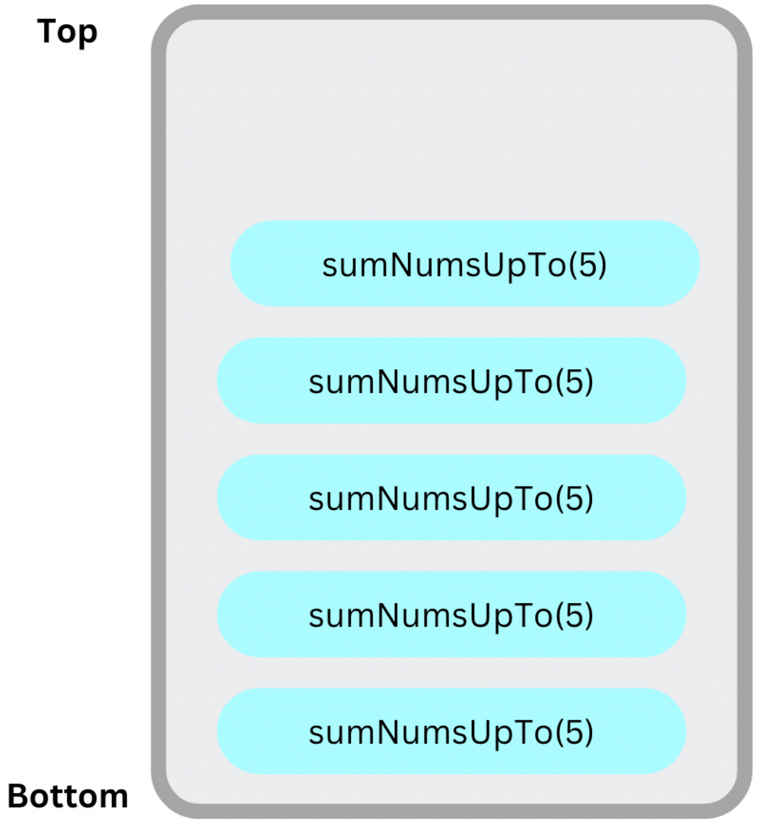
```java
// pre: n >= 1
public static int sumNumsUpTo(int n) {
    if (n == 1) { // Base case
        return 1;
    } else {      // Recursive case
        return sumNumsUpTo(n);
    }
}
```

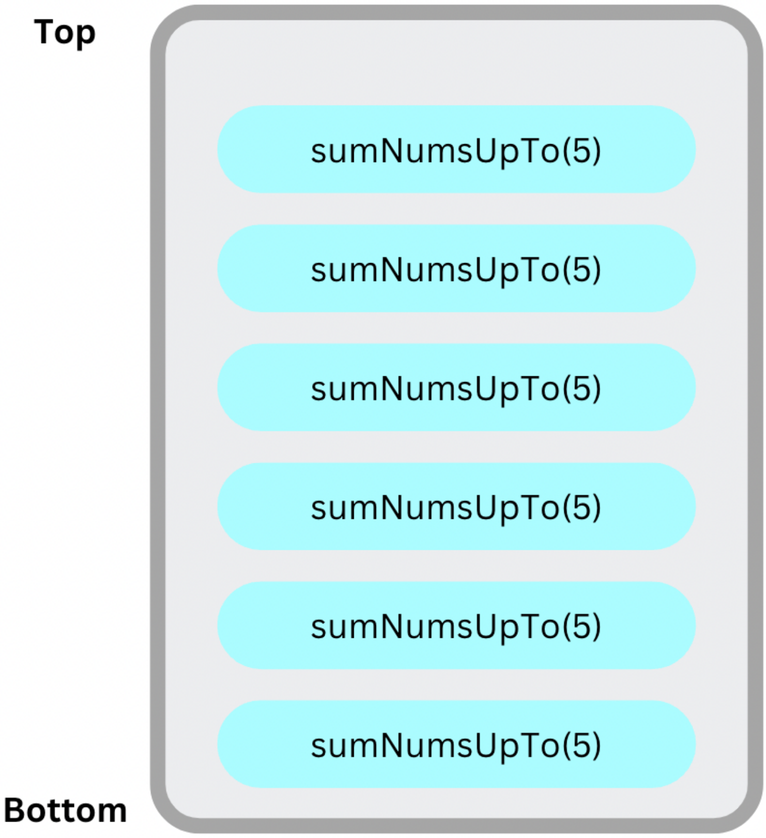
Will this work? If you are violently shaking your head from left to right then you are correct! In the **recursive case**, you are making a **recursive** call but notice how you never modify the parameter. Let's apply our **recursive code** tracing knowledge to see what the **call stack** would look like if we inputted 5:

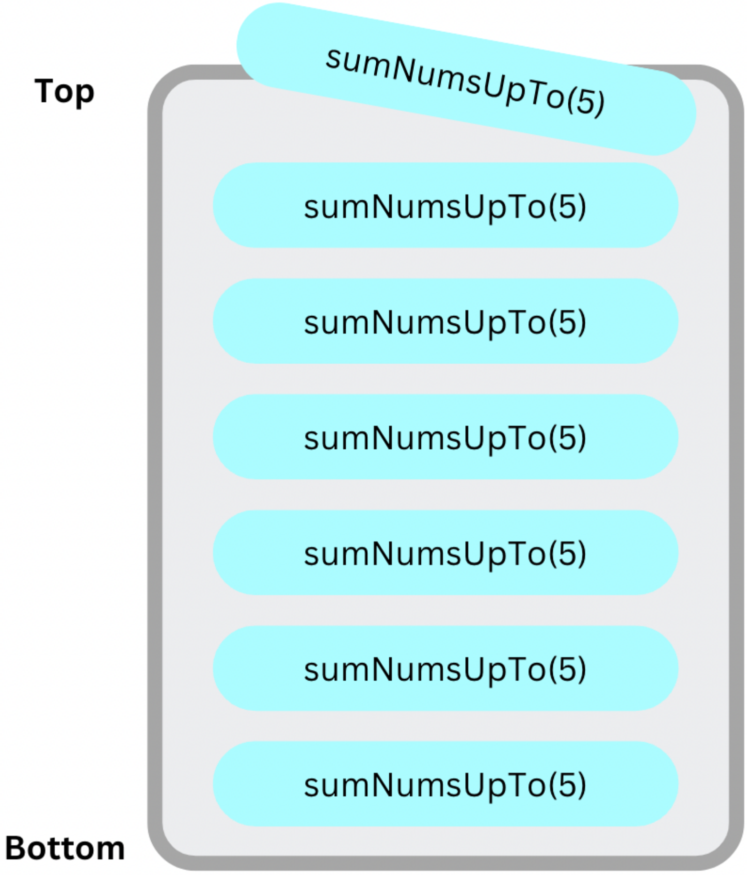

Top
sumNumsUpTo(5)
sumNumsUpTo(5)
Bottom

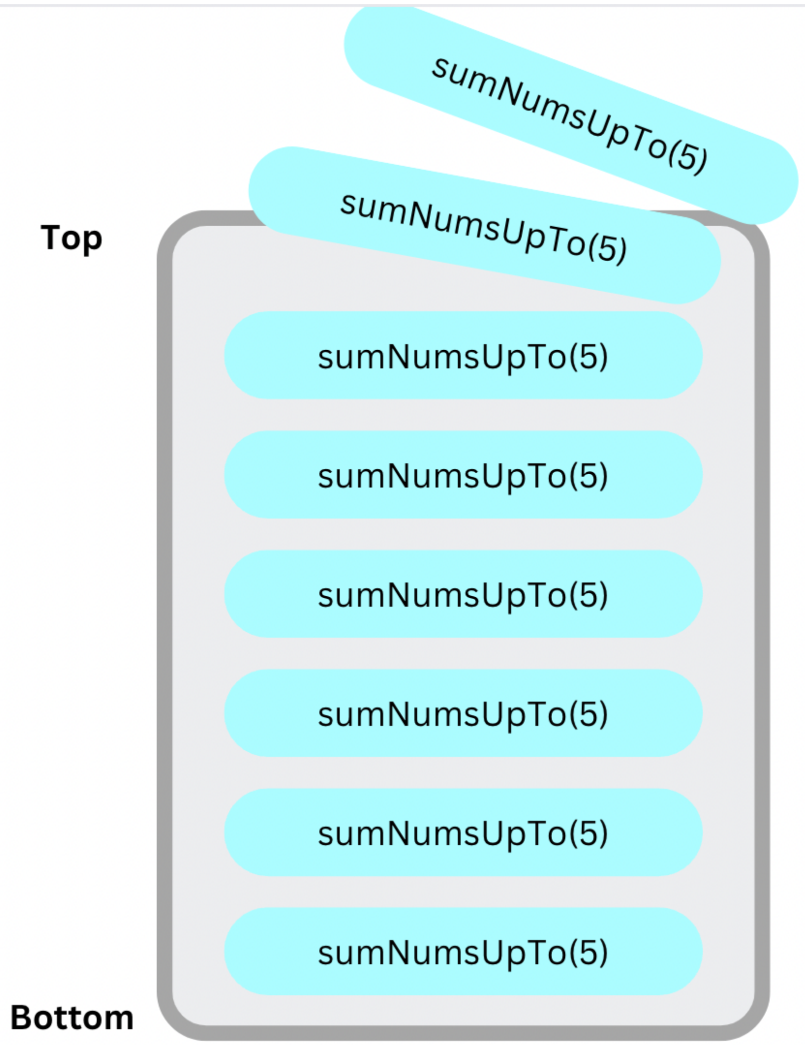

Top
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
Bottom

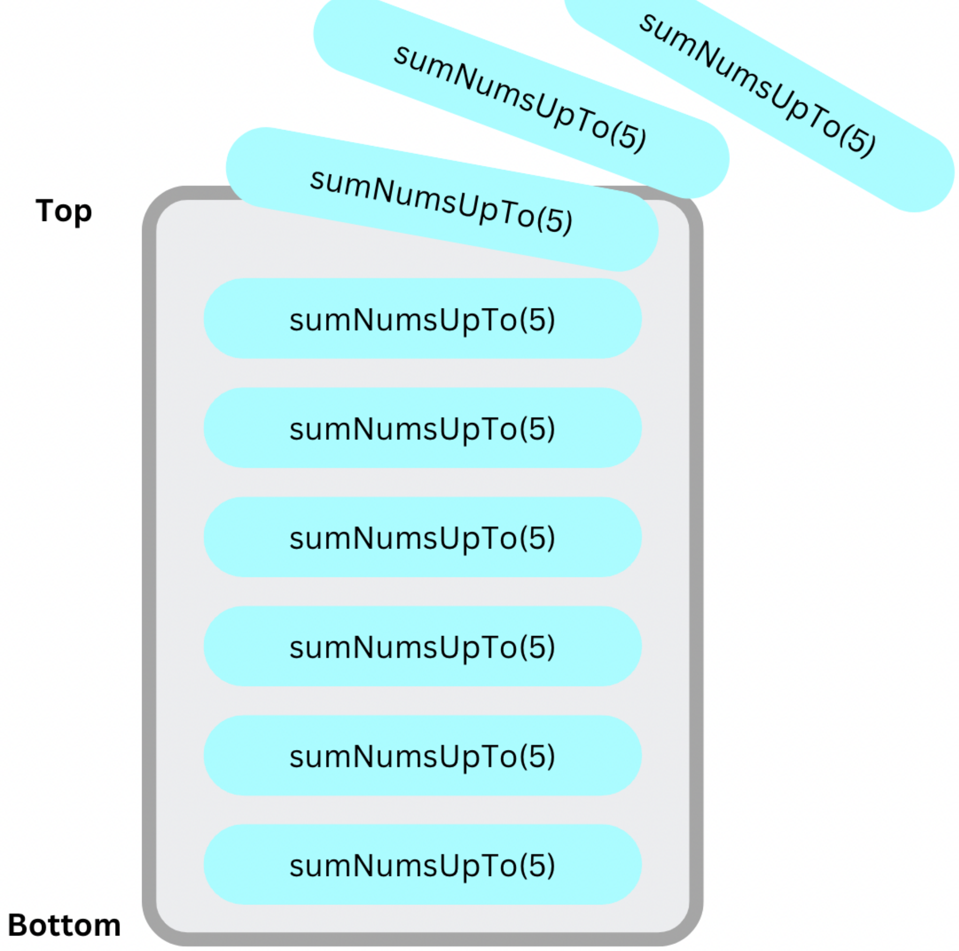

Top
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
Bottom


Top
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
Bottom


Top
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
Bottom


Top
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
Bottom


sumNumsUpTo(5)
sumNumsUpTo(5)
Top
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
Bottom


sumNumsUpTo(5)
sumNumsUpTo(5)
Top
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
sumNumsUpTo(5)
Bottom

It never ends! Our **call stack** is overflowing with methods 😱!

▯ StackOverflowError

In Java, this is known as a `StackOverflowError`. Even well-versed recursive programmers will encounter this error so do not feel bad if you encounter this error!

While it is true that the **recursive case** should be the one calling the method, we're going to add onto this definition and say that the goal of the **recursive case** is to get closer to the **base case**. If we never make progress closer to the **base case**, then we'll just keep recursing forever and eventually hit a `StackOverflowError`!

Remember that the purpose of recursion is to take a large task and break it down into smaller tasks. Since we want to sum all the numbers from 1 to `n` (inclusive), this is our large task. Now, let's think critically on how to properly sub-task the large task. If we are summing all numbers from 1 to `n` (inclusive) as our large task, what is smaller version of this?

What sub-task will help us get closer to the **base case** (i.e. when `n` is 1)? If we are `sumNumsUpTo(5)`, we know that `n` is currently 5. Thus, the numbers summing up to 5 is just 5 plus the numbers summing up to 4 (i.e. `sumNumsUpTo(4)`)! On each recursive call, if we subtract one each time, we will eventually reach our base case which stops the recursion!

```java
// pre: n >= 1
public static int sumNumsUpTo(int n) {
    if (n == 1) { // Base case
        return 1;
    } else {      // Recursive case
        return n + sumNumsUpTo(n - 1);
    }
}
```

The main takeaway you should get from this is that the **base case** stops the recursion and the **recursive case** works towards the **base case**.

If the **recursive case** doesn't properly update the parameter on each **recursive call**, or if it goes in the opposite direction of the **base case** (like if the **recursive case** was `n + sumNumsUpTo(n + 1)`, then you'll get a `StackOverflowError` which means your **recursion** never ended!

▯ Main Points

- With **recursion**, we want to break down a larger task into smaller sub-tasks. Each recursive call we make should progress towards the **base case**, otherwise we may get infinite recursion.

- The **base case** is the condition that stops **recursion.**
  - It's the simplest/smallest case that we know the solution of.
  - When we want to sum up numbers from 1 to n, then our **base case** would be when `n = 1`, since we know that the sum of all numbers from 1 to 1 is `1`.
- The **recursive case** is where the method calls itself with <u>modified</u> parameters so we can make further progress towards the **base case.**
- If we don't properly update the arguments of our **recursive case**, or if we don't eventually end up at the **base case**, we may get a `StackOverflowError`, which happens when our call stack is too deep because of infinite recursion.

Parenthesize [Programming Question]

Write a recursive method called `parenthesize` that takes a String and an integer n as parameters and that returns the string inside n sets of parentheses.

For example, this code:

```
System.out.println(parenthesize("Brett Wortzman", 2));
System.out.println(parenthesize("The University of Washington", 6));
System.out.println(parenthesize("cats", 1));
```

should produce these 3 lines of output:

```
((Brett Wortzman))
((((((The University of Washington))))))
(cats)
```

Your method should throw an `IllegalArgumentException` if passed a negative number.

It could be passed 0, as in: `parenthesize("CSE122, Winter 2022", 0`);

In this case the returned String would have no (i.e., 0) parentheses:

```
CSE122, Winter 2022
```

For this question, we will provide you with the **recursive call**:

```
return "(" + parenthesize(str, n - 1) + ")";
```

Write Chars [Programming Question]

Write a recursive method `writeChars` that accepts an integer parameter `n` and returns `n` characters as follows. The middle character of the output should always be an asterisk ("*"). If you are asked to write out an even number of characters, then there will be two asterisks in the middle ("**").

Before the asterisk(s) you should write out less-than characters ("<"). After the asterisk(s) you should write out greater-than characters (">").

For example, the following calls produce the following output:

- `writeChars(1)` returns `*`
- `writeChars(2)` returns `**`
- `writeChars(3)` returns `<*>`
- `writeChars(4)` returns `<**>`
- `writeChars(5)` returns `<<*>>`
- `writeChars(6)` returns `<<**>>`
- `writeChars(7)` returns `<<<*>>>`
- `writeChars(8)` returns `<<<**>>>`

Your method should throw an `IllegalArgumentException` if passed a value less than 1. Note that the output does not advance to the next line.

For this question, we will provide you with the **base cases:**

```
if (n == 1) {
    return "*";
} else if (n == 2) {
    return "**";
}
```