Pre-Class Work 9: Recursion / Recursive Tracing

Introduction to Recursion [Background Reading]

□ Motivation

Imagine, on a regular morning at UW, your Keurig broke so you decide to get some coffee at the Starbucks Coffee inside Suzzallo ☕. Upon arrival, you notice a long line which extended from the Starbucks cashier all the way to the entrance of Suzzallo. You decide to line up and join the rest of your UW students in the wait, slowly inching forward every couple of minutes.

After half an hour, you begin contemplating whether or not you should even stay in line. On one hand, you need your daily coffee so that you can stay awake in lecture. On the other, you're going to be late for lecture if the line doesn't speed up soon! You want to figure out your position in the line so that you can gauge whether or not it's worth it to stay in line.

Assuming that the line is 0-based index meaning that for a line with n people, the person at the start of the line is in the 0th position and the person at the end of the line is in the $(n-1)^{th}$ position, how would you go about figuring out what position you are in?

Your first instinct might be to step out of line and begin counting from the front of the cashier all the way until you hit your position. For example, if the line looked like the following:

Start of the line ☕

Brett

Miya

Yourself

Elba

Hunter

Kasey

End of the line

You would begin counting from `Brett`, then `Miya`, and then eventually land on `Yourself`! Once you have landed back to `Yourself`, you will have concluded that you are in the 2nd position (i.e. you are third in line).

What you just performed is known as an **iteration**. Essentially what you just did was initialize a

position and continuously increase that number until you landed on yourself!

We can express this using a for-loop:

```
public int findPosition(List<String> line) {
    for (int position = 0; position < line.size(); position++) {
        String name = line.get(position);
        if (name.equals("Yourself")) {
            return position;
        }
    }
    return -1; // We'll return -1 to indicate that you are not in line
}
```

Anytime you are using a for-loop, you are starting at some position (usually the front) and **iterating** through the data. Great! Now that you know what position you are in line, you decide that you can wait a little longer since you are only third in line. However, there is one slight concern.

We Stepped Out Of Line! 😿

Because you stepped out of line to count, the whole line of people behind you moved forward so if you want to get in line, you have to enter through the back of the line again! 😱

How would you have gone about figuring out your position without needing to **iterate**?

One idea you might have would be to ask the person in front of you what position *they* are in. In this case, you would ask `Miya` what position she is in. She would most likely shrug and say "I don't know, let me check" then she'll go ahead and ask the person in front of her what position *they* are in. Now, `Miya` asks `Brett` what position he is in. No one is in front of `Brett` so he knows for sure that he is in the 0th position (i.e. he is the first person in line). Since he knows the answer, he can respond back to `Miya` and tell `Miya` that he is in the 0th position.

Now that `Miya` knows that `Brett` is in the 0th position, she can add 1 to obtain her own position! `Miya` adds 1 and figures out that she is in the 1st position and then lets you know that she is in the 1st position (i.e. she is the second person in line). Since you know `Miya` is in the 1st position, then you can add 1 to her position and conclude that you are in the 2nd position (i.e. you are the third person in line).

Wow, this is great! You figured out your position in line without ever needing to step out of it.

We Just Did Recursion!

In computer science terms, what you just performed is known as **recursion**. Notice how each person in front of you asked each other the same question. When you asked `Miya`, you asked what position

she is in. When `Miya` asked `Brett`, she asked what position he is in. The question being asked is **recurring**!

The reason why the series of questions stopped **recursing** is because we hit our **base case**. In other words, since no one is in front of `Brett`, `Brett` can just answer that he is at the start of the line.

You might be wondering why you can't just ask the person in front of us what position you are in. Well, let's try it out! First, you would ask `Miya` what position `Yourself` is in. She would most likely shrug and say "I don't know, let me check" then she'll go ahead and ask the person in front of her what position is `Yourself` in. Now, `Miya` asks `Brett` what position is `Yourself` in. He would most likely shrug and say "I don't know, let me check".

However, no one is in front of `Brett` so he wouldn't have anyone to ask! He would probably turn back to `Miya` and say I don't know what position `Yourself` is in! Note that `Brett` is fully aware that he is in the 0th position in line but that's not the question that `Miya` asked him so he would never tell `Miya` that information!

Idea of Recursion

The idea of **recursion** is to take a large task and break it up into a series of sub-tasks. In this case, the large task is figuring out what position you are in and we broke it down into a smaller task which was asking people what position *they* are in. Your smaller task can't be asking people what position you are in because that's what you have defined your large task to be already!

Simply put, your large task and smaller task can't be the same task, otherwise that would mean our larger task was never broken up into smaller tasks!

□ Why Recursion?

You might be wondering why we even need to learn **recursion** if **iteration** is equally as powerful. After all, in both the **iterative** and **recursive** approach, we figured out that we were in the 2nd position.

There are many reasons why **recursion** is an important topic to study. Firstly, many **recursive** solutions tend to take breathtakingly less code to write than **iterative** solutions which all programmers love since it means less typing. Secondly, in the coming weeks, we will be learning new data structures which are much easier to solve **recursively**.

Finally, learning **recursion** will provide you with a different perspective that you can apply to solve future problems! □

🔑 Main Points

- **Recursion** is a problem-solving technique that breaks down one big problem into smaller, similar sub-problems.
- **Iteration** involves using loops to repeat a task, while **recursion** involves breaking down a problem into smaller, similar sub-problems and solving each sub-problem (and is often more concise).
- In recursive problem-solving, we identify two components:
    - A **base case** (the simplest case we can solve directly), and
    - A **recursive case** (a case we can break down into smaller sub-problems) -- in this case, we usually solve a similar problem just on a much smaller scale
- To *terminate*, the **recursive** process must eventually reach the **base case** e.g. when there's no one in front of a person to ask about their position in line.
- **Recursion** can give us a more elegant approach to solving some problems as well as a new way of thinking about them!

Anatomy of Recursion [Background Reading]

🔗 Analyzing Recursive Code

So what does **recursion** look like in Java? Consider the following method:

```
// pre: n >= 1
public static int sumNumsUpTo(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum += i;
    }
    return sum;
}
```

In this method, we are given a parameter `n` and we are summing up all integers between `1` to `n` (inclusive). We'll make the assumption that `n` will always be greater than or equal to 1. For example, if `n` is 4, then we would return 10 since 1 + 2 + 3 + 4 = 10. The current method is written **iteratively**. How would we express this **recursively**?

Consider the following method:

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

Immediately, you might notice that `sumNumsUpTo()` calls itself in the `else` case! When you see a method calling itself, this is known as a **recursive call**. Thus, the `else` case is referred to as the **recursive case**, the one doing the actual **recursion** by calling the sub-task. If we take a look at the `if` case, notice how we don't call `sumNumsUpTo()` at all. This `if` case is referred to as the **base case**.

The **base case** of your code will be the one which ends your **recursion** so it should not do any sort of recursion at all! All recursive code must contain these two crucial components: the **base case** and the **recursive case**. It is entirely possible to have multiple **base cases** and **recursive cases** but for now we will only analyze code which has one **base case** and one **recursive case**.

Let's see what happens if `n` is 4!

When you call `sumNumsUpTo(4)`, we will enter the `else` case and return the following:

4 + sumNumsUpTo(3)

When you called `sumNumsUpTo(4)`, you ended up calling `sumNumsUpTo(3)` so what does `sumNumsUpTo(3)` return?

4 + sumNumsUpTo(3)

3 + sumNumsUpTo(2)

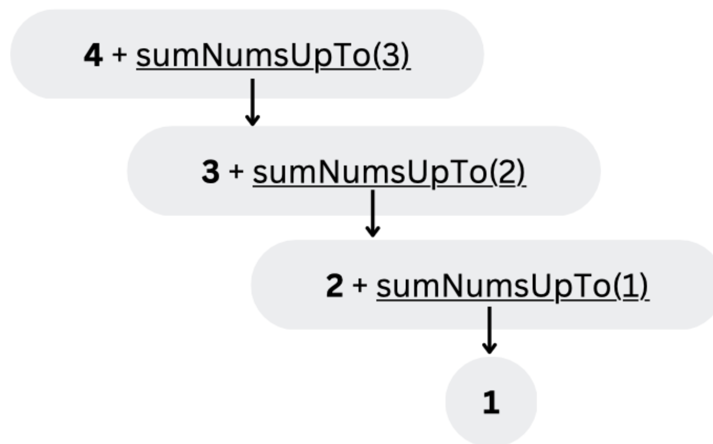
When you called `sumNumsUpTo(3)`, you ended up calling `sumNumsUpTo(2)` so what does `sumNumsUpTo(2)` return?

4 + sumNumsUpTo(3)

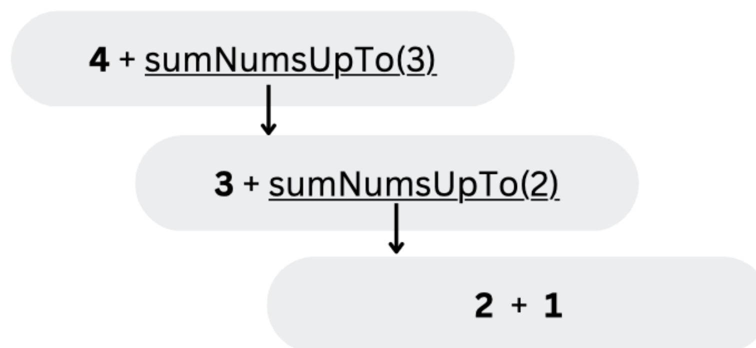
3 + sumNumsUpTo(2)

2 + sumNumsUpTo(1)

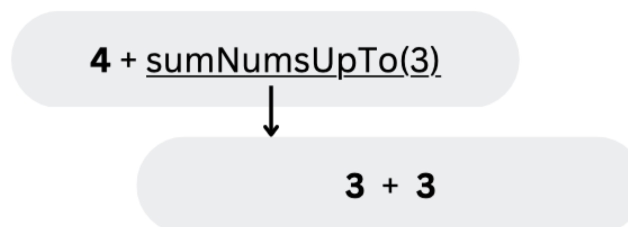
When you called `sumNumsUpTo(3)`, you ended up calling `sumNumsUpTo(1)` so what does `sumNumsUpTo(1)` return?



Nice! When we called `sumNumsUpTo(1)`, we hit a **base case** which stopped the recursion! Here, we see that `sumNumsUpTo(1)` returns 1 so what does `sumNumsUpTo(2)` do now?



`sumNumsUpTo(2)` sees that `sumNumsUpTo(1)` is 1 so it will return 2 + 1! Here, `sumNumsUpTo(2)` returns 3 so what does `sumNumsUpTo(3)` do now?



`sumNumsUpTo(3)` sees that `sumNumsUpTo(2)` is 3 so it will return 3 + 3! Here, `sumNumsUpTo(3)` returns 6 so what does `sumNumsUpTo(4)` do now?

4 + 6

`sumNumsUpTo(4)` sees that `sumNumsUpTo(3)` is 6 so it will return 4 + 6! Thus, when we called `sumNumsUpTo(4)`, the method will return the value 10!

Notice how whenever you made a **recursive call**, you are in a completely separate method. This is known as the **call stack**. Java's underlying implementation uses a `Stack` to keep track of the methods that are being called. You can imagine each method literally being "stacked" on top of the other.

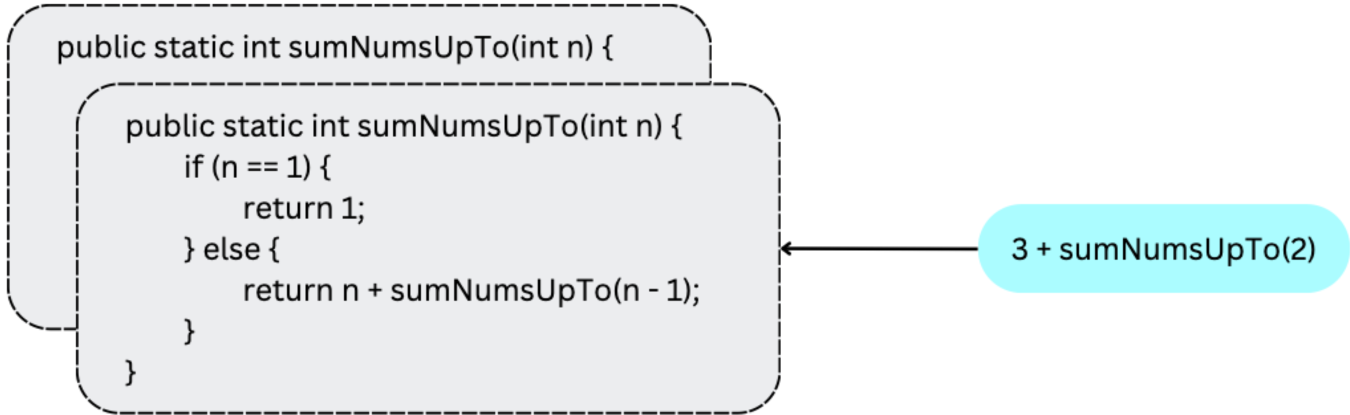
We will define our method as follows:

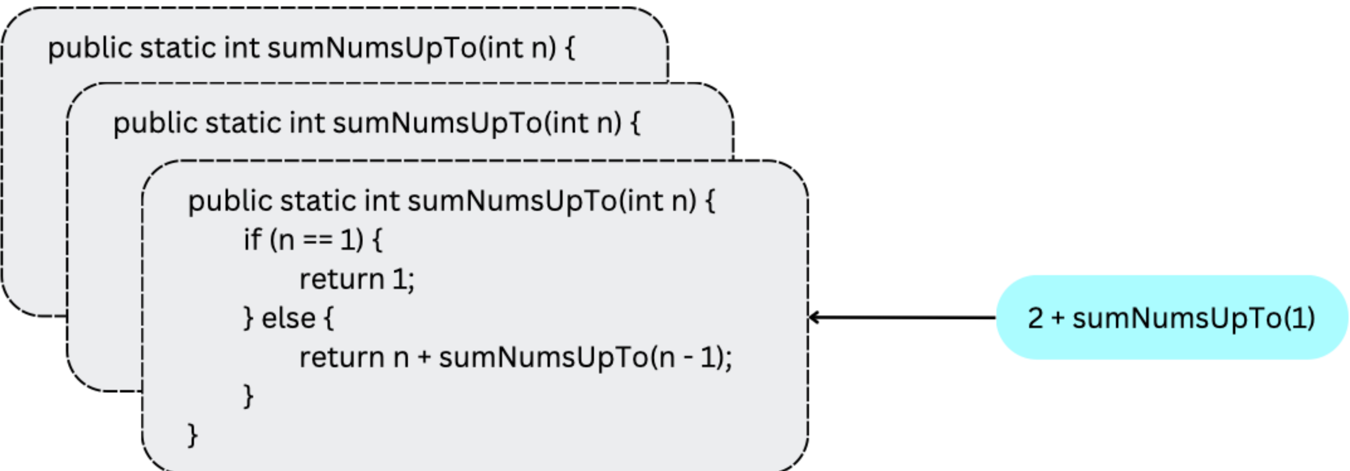
```
public static int sumNumsUpTo(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n + sumsNumsUpTo(n - 1);
    }
}
```

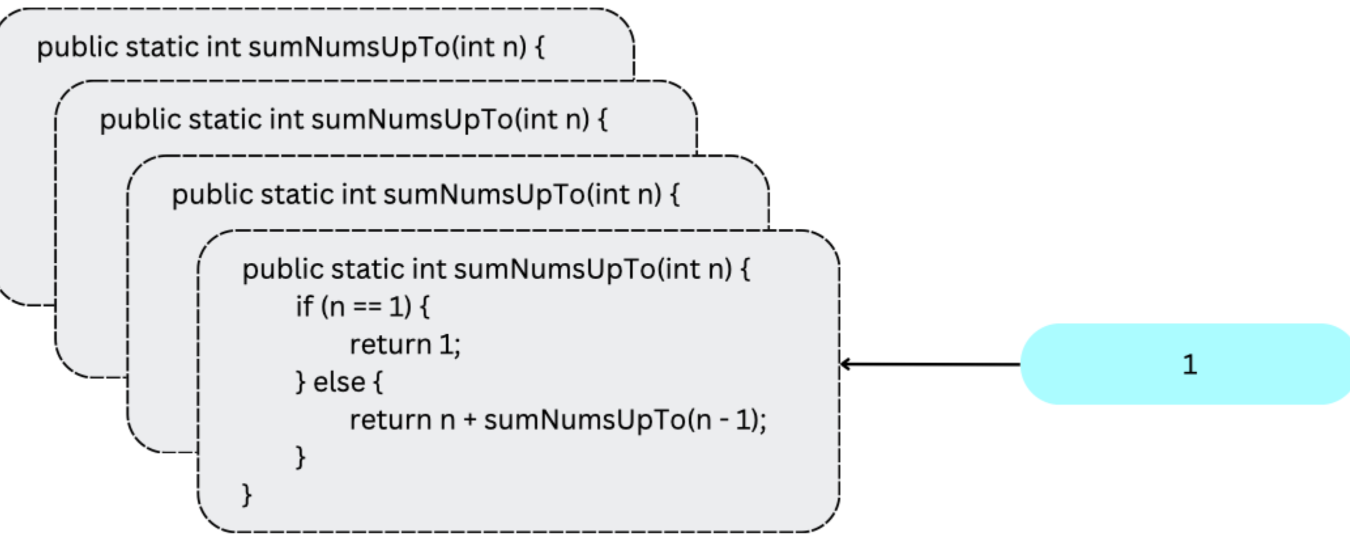
For example, let's set `n` to be 4 again:

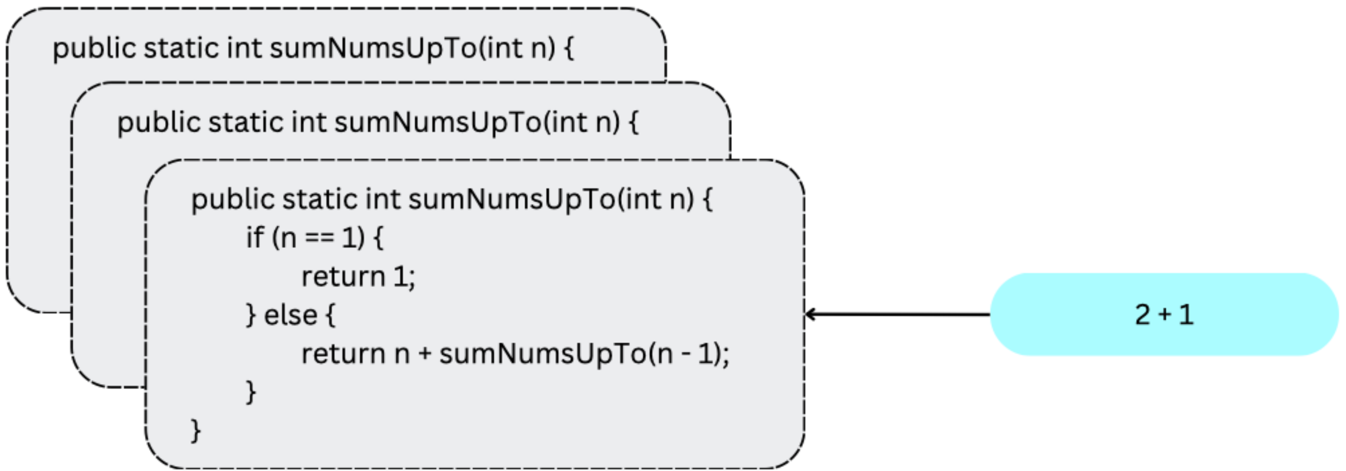
```
public static int sumNumsUpTo(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n + sumNumsUpTo(n - 1);
    }
}
```

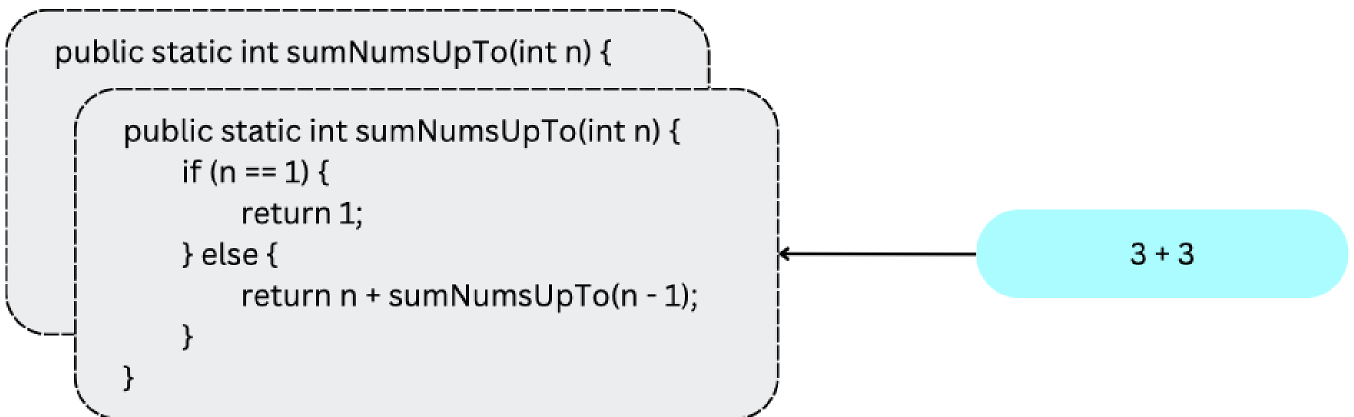
4 + sumNumsUpTo(3)

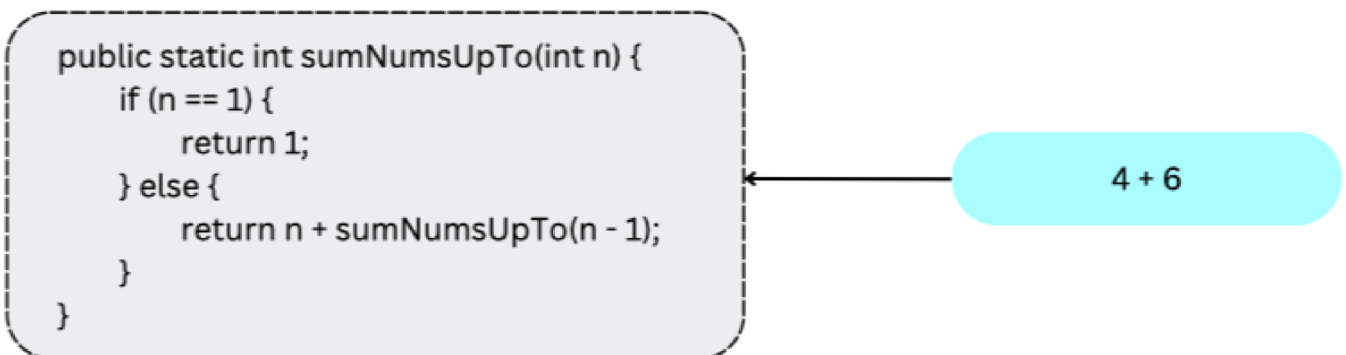

public static int sumNumsUpTo(int n) {
public static int sumNumsUpTo(int n) {
if (n == 1) {
return 1;
} else {
return n + sumNumsUpTo(n - 1);
}
}
3 + sumNumsUpTo(2)


public static int sumNumsUpTo(int n) {
public static int sumNumsUpTo(int n) {
public static int sumNumsUpTo(int n) {
if (n == 1) {
return 1;
} else {
return n + sumNumsUpTo(n - 1);
}
}
2 + sumNumsUpTo(1)


public static int sumNumsUpTo(int n) {
public static int sumNumsUpTo(int n) {
public static int sumNumsUpTo(int n) {
public static int sumNumsUpTo(int n) {
if (n == 1) {
return 1;
} else {
return n + sumNumsUpTo(n - 1);
}
}
1



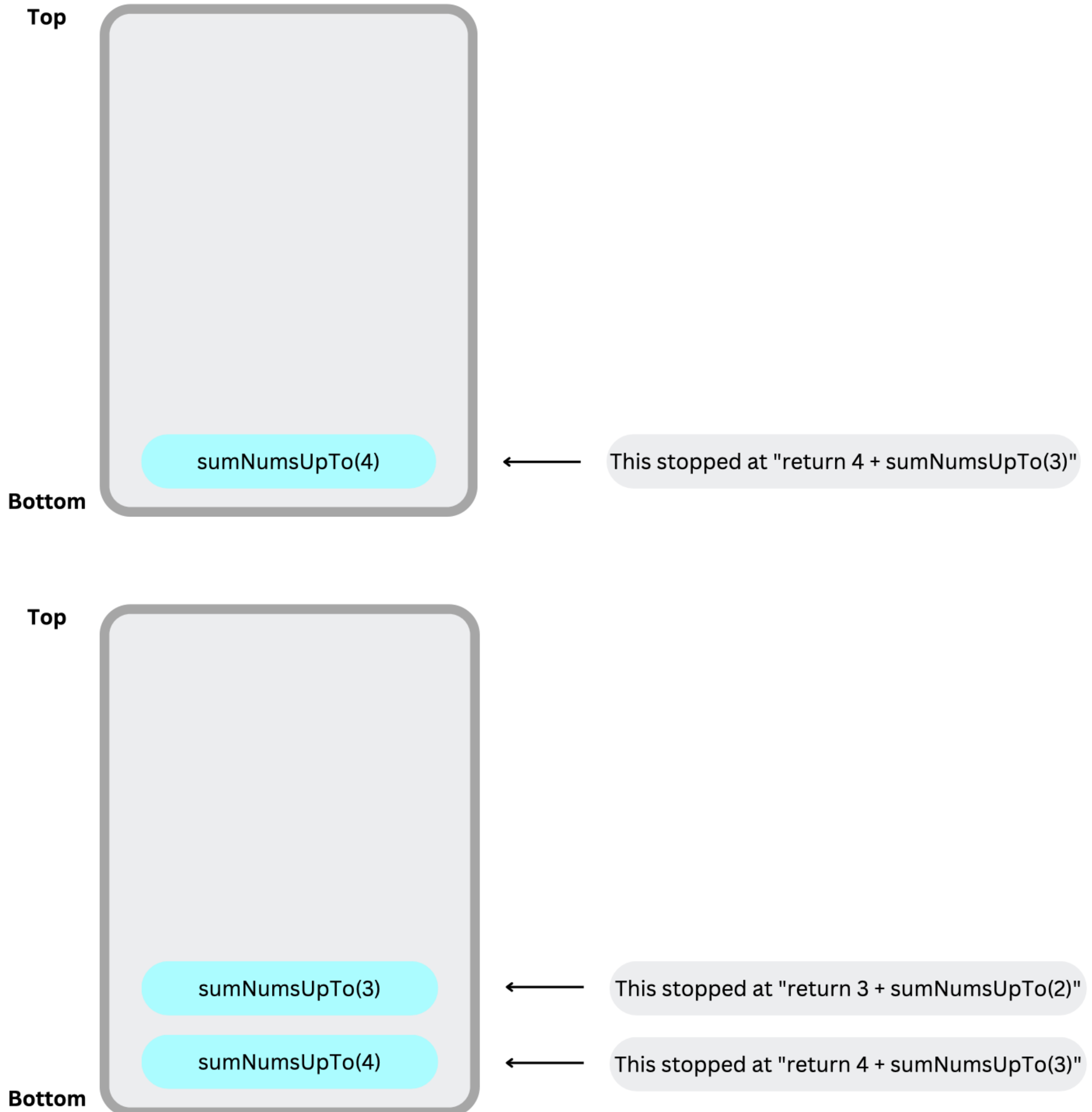


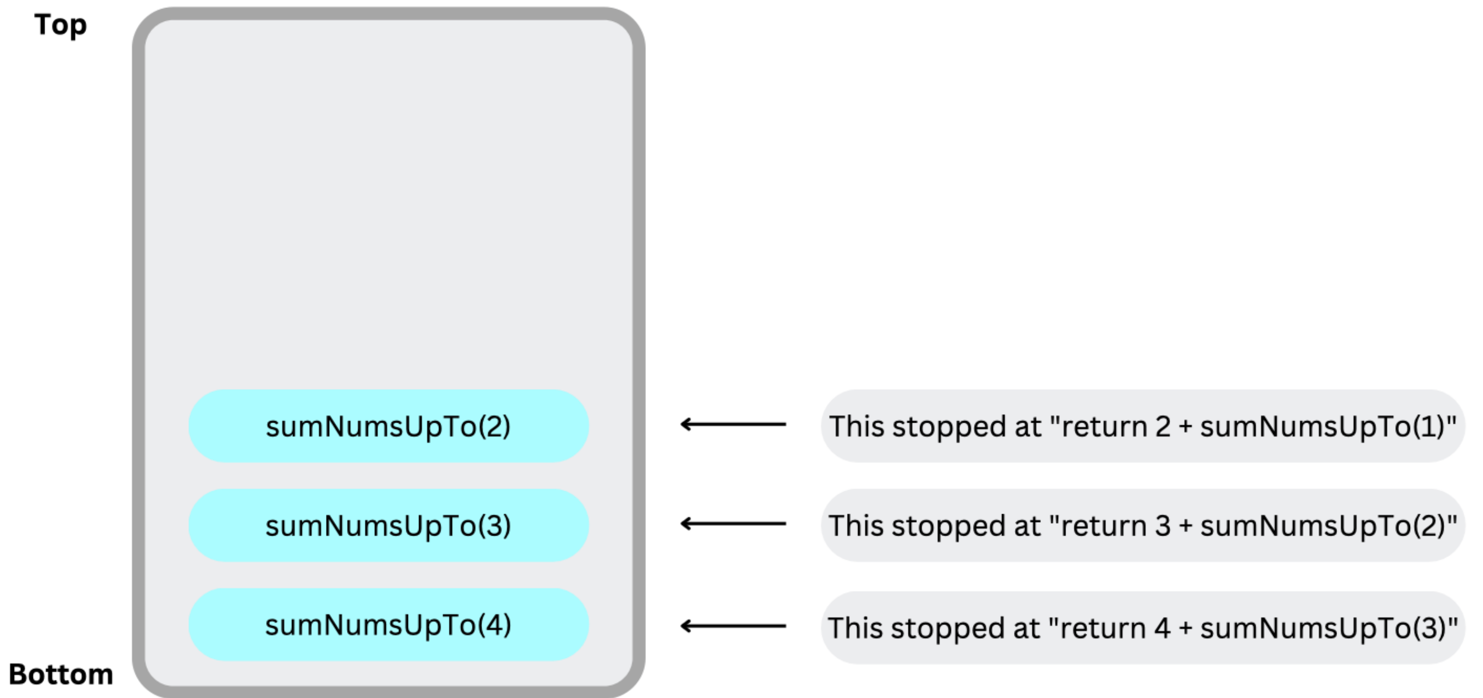


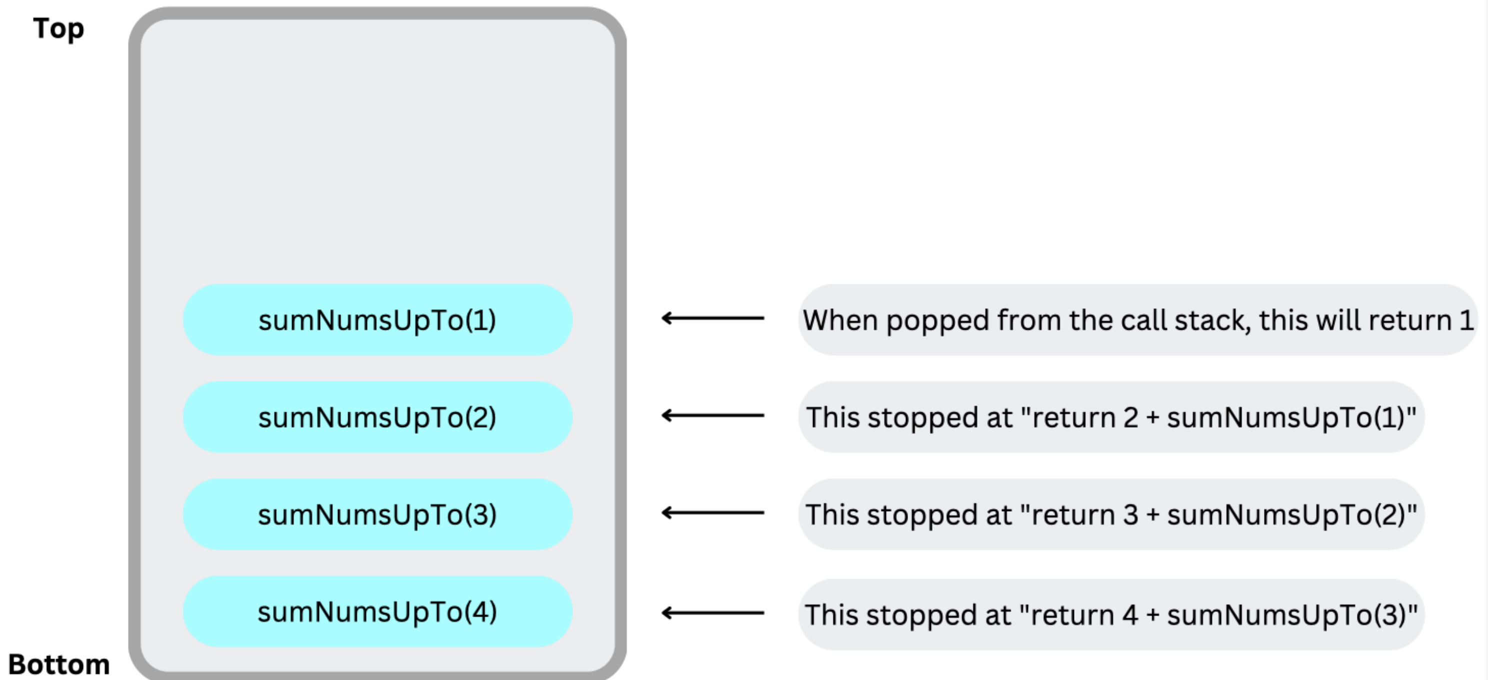
Any time a **recursive call** is made, Java stops all operations on its current method and goes to work on the method that you just called. When you called `sumNumsUpTo(4)`, the moment it sees that you made a call to `sumNumsUpTo(3)`, it will stop all operations it is doing on `sumNumsUpTo(4)` and focus on working through `sumNumUpsTo(3)`.

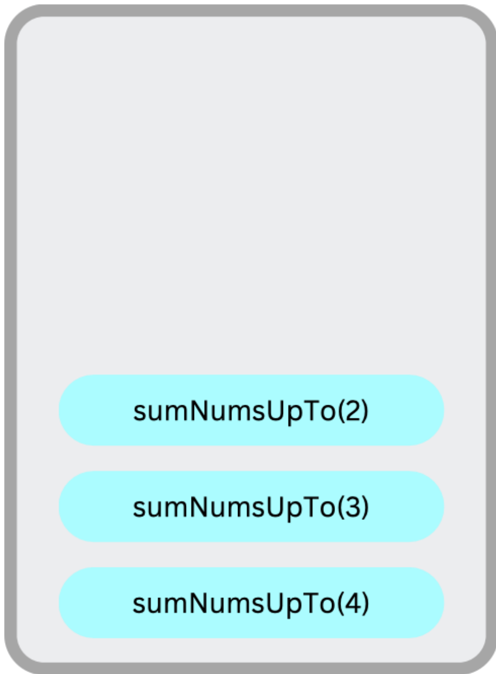
Hence, we see `sumNumsUpTo(3)` literally being "stacked" on top of `sumNumsUpTo(4)`. Methods will continue being stacked on top of one another until we hit the **base case** which is when Java will "pop" the methods off the **call stack**.

Here is an illustration how the Java **call stack** would appear during the **recursion**:

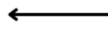



Top
sumNumsUpTo(2)
sumNumsUpTo(3)
sumNumsUpTo(4)
Bottom
This stopped at "return 2 + sumNumsUpTo(1)"
This stopped at "return 3 + sumNumsUpTo(2)"
This stopped at "return 4 + sumNumsUpTo(3)"


Top
sumNumsUpTo(1)
sumNumsUpTo(2)
sumNumsUpTo(3)
sumNumsUpTo(4)
Bottom
When popped from the call stack, this will return 1
This stopped at "return 2 + sumNumsUpTo(1)"
This stopped at "return 3 + sumNumsUpTo(2)"
This stopped at "return 4 + sumNumsUpTo(3)"

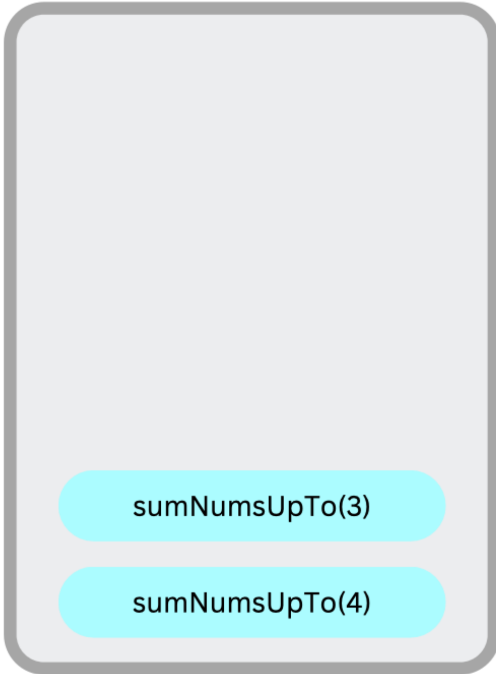


When popped from the call stack, this will return 3

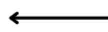


This stopped at "return 3 + sumNumsUpTo(2)"

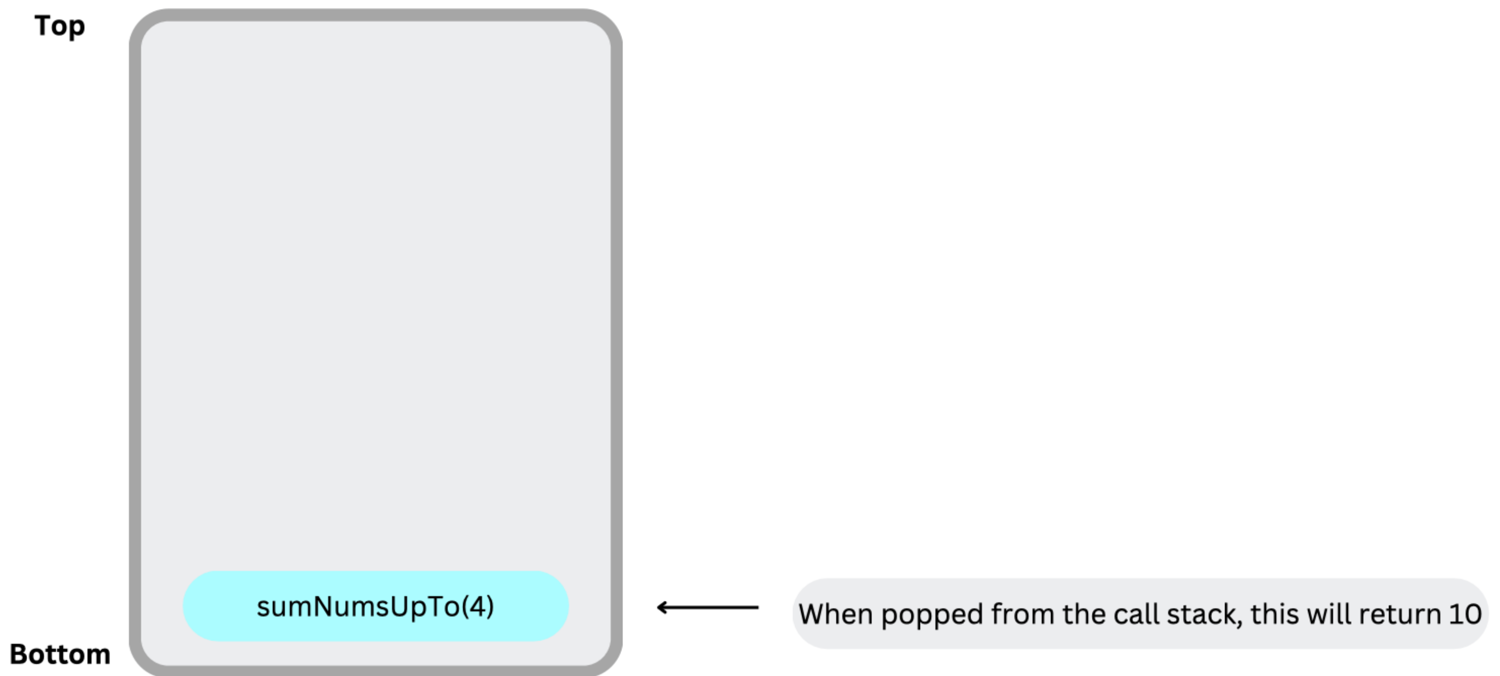
This stopped at "return 4 + sumNumsUpTo(3)"



When popped from the call stack, this will return 6



This stopped at "return 4 + sumNumsUpTo(3)"



󰄬 Main Points

- When you see a method calling itself, this is known as a **recursive call**.
- The **base case** of your code will be the one which ends your **recursion** so it should not do any sort of recursion at all!
- All recursive code must contain a **base case** and a **recursive case**.
- Any time a **recursive call** is made, Java stops all operations on its current method and goes to work on the method that you just called.
- During recursion, the **call stack** gets filled with recursive calls from the bottom to the top, so when we "pop" or return items from the call stack, we will return the most recent recursive call.

Recursive Tracing [Discussion Question]

The following question will give you practice in tracing through recursive code.

Consider the following method:

```
public int recursiveMethod(int x, int y) {
    if (x < y) {
        return x;
    } else {
        return recursiveMethod(x - y, y);
    }
}
```

Question 1

What gets returned from `method(6, 13)`?

No response

Question 2

What gets returned from `method(14, 10)`?

No response

Question 3

What gets returned from `method(37, 10)`?

No response

Question 4

What gets returned from `method(8, 2)`?

No response

Question 5

What gets returned from `method(50, 7)`?

No response