Discussion 03

Recursion

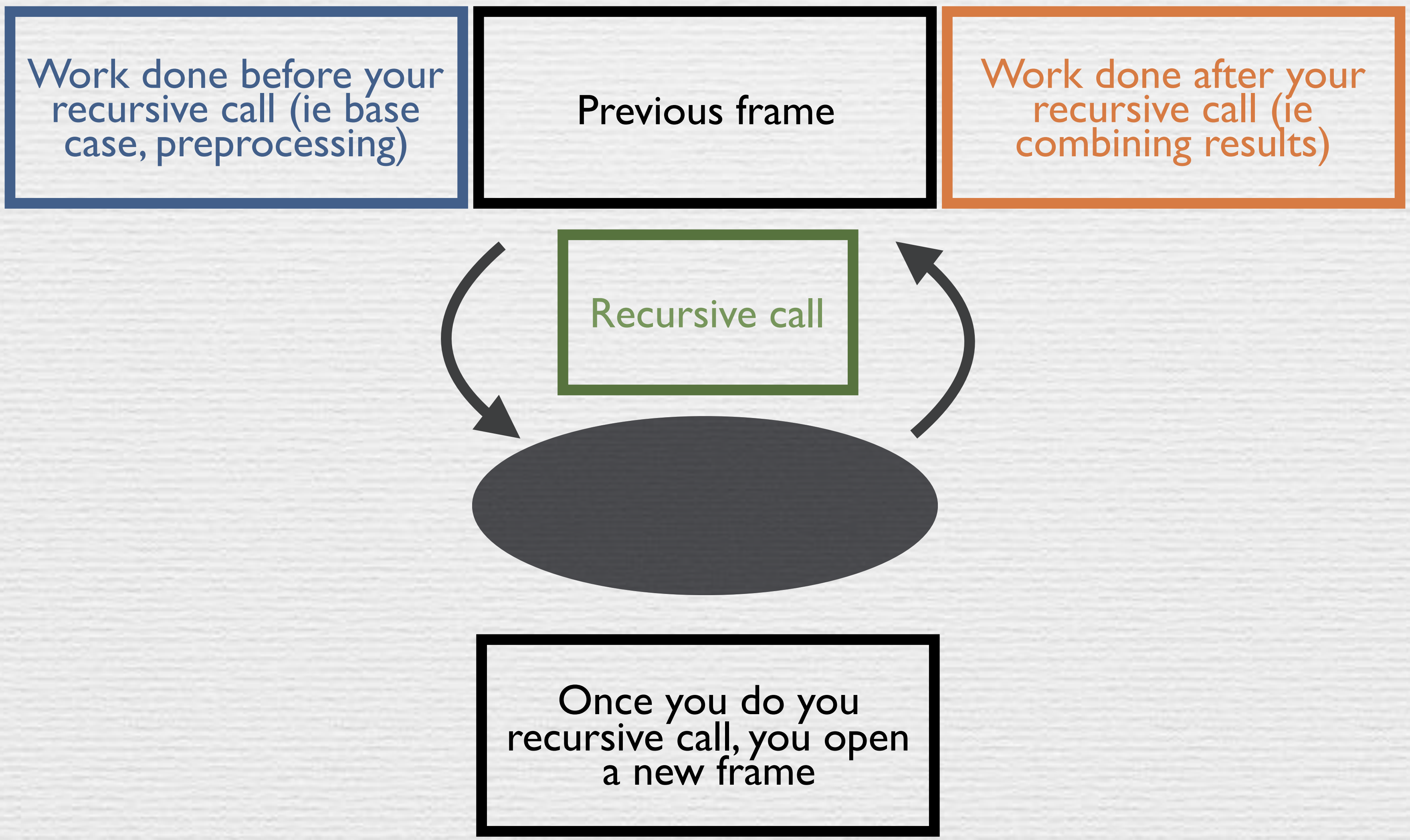
Tree recursion

Recursion Facts

1. **Base case**: What is the simplest problem that you can solve? In other words, is there an input to the problem for which you automatically know what to return?
2. **Make a recursive call**! Assume that you have a working function: how can you use it by breaking down the original problem?
3. **Combine the results**. Now that you have the results of your recursive call, you might need to do some post-processing. This is not always necessary.

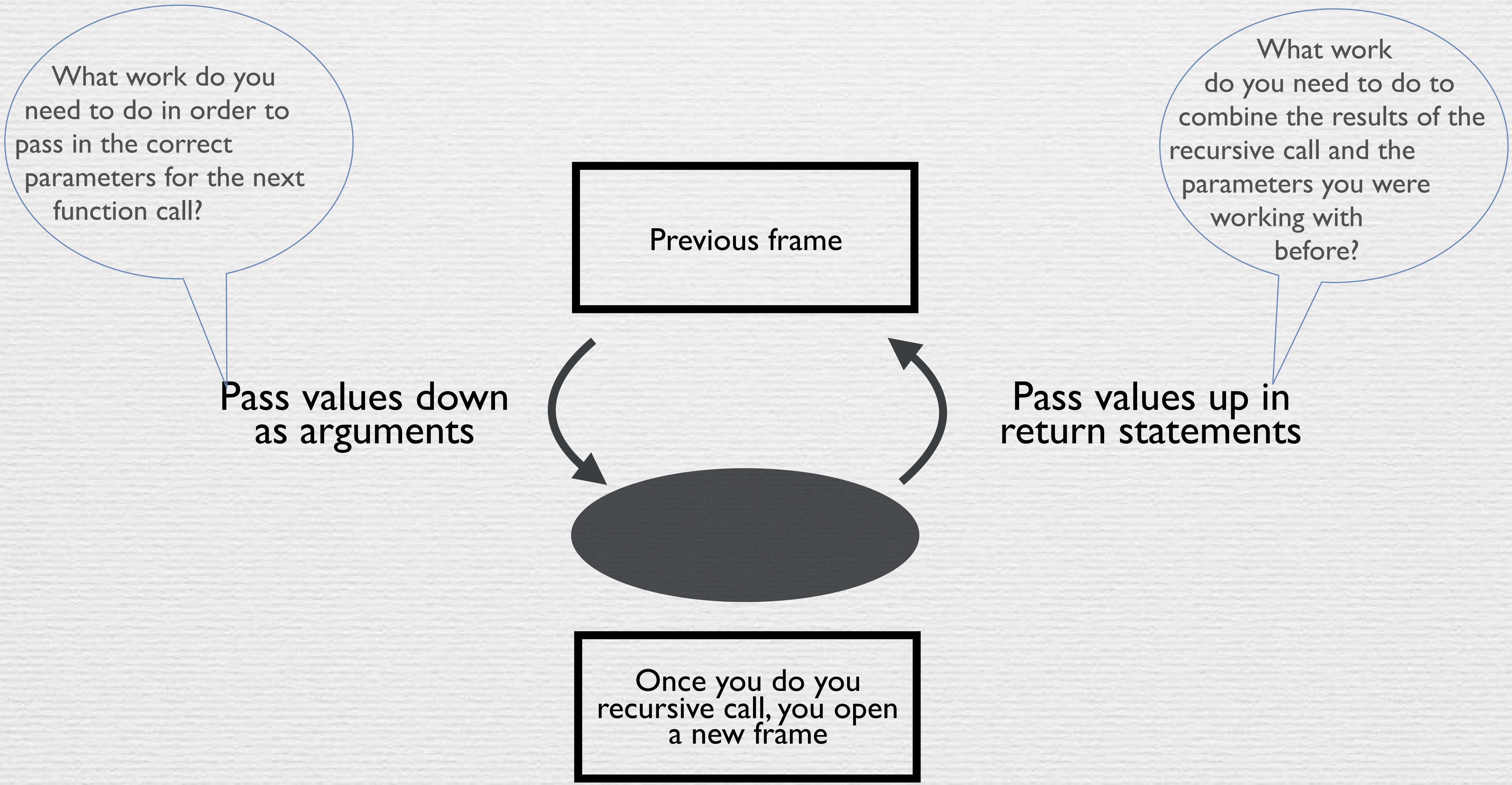
Recursion Visualization

It is helpful to think of each level of recursion as jumping down into a different frame.



How to communicate between frames

Think of your code as a timeline. Assume the function you are writing works correctly.



1.1: countdown

Write a function that counts down from n to 1

```
def countdown(n):
    if n <= 0:
        return
    print(n)
    countdown(n - 1)
```

1.1: countdown

Write a function that counts down from n to 1

```
def countdown(n):
    if n <= 0:
        return
    print(n)
    countdown(n - 1)
```

What are we asked to return? Nothing!
We just want to **print out** numbers

1.1: countdown

Write a function that counts down from n to 1

```
def countdown(n):
    if n <= 0:
        return
    print(n)
    countdown(n - 1)
```

What are we asked to return? Nothing!
We just want to **print out** numbers

Base Case: How do we know we've printed out all of the numbers from n to 1?

1.1: countdown

Write a function that counts down from n to 1

What are we asked to return? Nothing!
We just want to **print out** numbers

```
def countdown(n):
    if n <= 0:
        return
    print(n)
    countdown(n - 1)
```

Base Case: How do we know we've printed out all of the numbers from n to 1?

Print the number you're at right now!

1.1: countdown

Write a function that counts down from n to 1

```
def countdown(n):
    if n <= 0:
        return
    print(n)
    countdown(n - 1)
```

What are we asked to return? Nothing!
We just want to **print out** numbers

Base Case: How do we know we've printed out all of the numbers from n to 1?

Print the number you're at right now!

Assume that countdown works. Since we printed n, now we need to print everything from n - 1 to 1. Do this by recursively calling countdown on n - 1

1.1: countup

Write a function that counts up from 1 to n by only changing one line in countdown

```
def countdown(n):
    if n <= 0:
        return
    countdown(n - 1)
    print(n)
```

1.1: countup

Write a function that counts up from 1 to n by only changing one line in countdown

Base Case: Same as countdown

```
def countdown(n):
    if n <= 0:
        return
    countdown(n - 1)
    print(n)
```

1.1: countup

Write a function that counts up from 1 to n by only changing one line in countdown

```
def countdown(n):
    if n <= 0:
        return
    countdown(n - 1)
    print(n)
```

Base Case: Same as countdown

First we want to jump all the way down to 1, so make the recursive call

1.1: countup

Write a function that counts up from 1 to n by only changing one line in countdown

```
def countdown(n):
    if n <= 0:
        return
    countdown(n - 1)
    print(n)
```

Base Case: Same as countdown

First we want to jump all the way down to 1, so make the recursive call

Now print out the number

Analyzing countdown

```
def countdown(n):
    if n <= 0:
        return
    print(n)
    countdown(n - 1)
```

What happens when we call countdown(3)?

Analyzing countdown

```
def countdown(n):
    if n <= 0:
        return
    print(n)
    countdown(n - 1)
```

Work done before your recursive call (ie base case, preprocessing)

When we first call countdown(3) we have one frame where n is 3

n = 3

Work done after your recursive call (ie combining results)

What happens when we call countdown(3)?

Analyzing countdown

```
def countdown(n):
    if n <= 0:
        return
    print(n)
    countdown(n - 1)
```

Work done before your recursive call (ie base case, preprocessing)

`print(3)`

When we first call countdown(3) we have one frame where n is 3

n = 3

Work done after your recursive call (ie combining results)

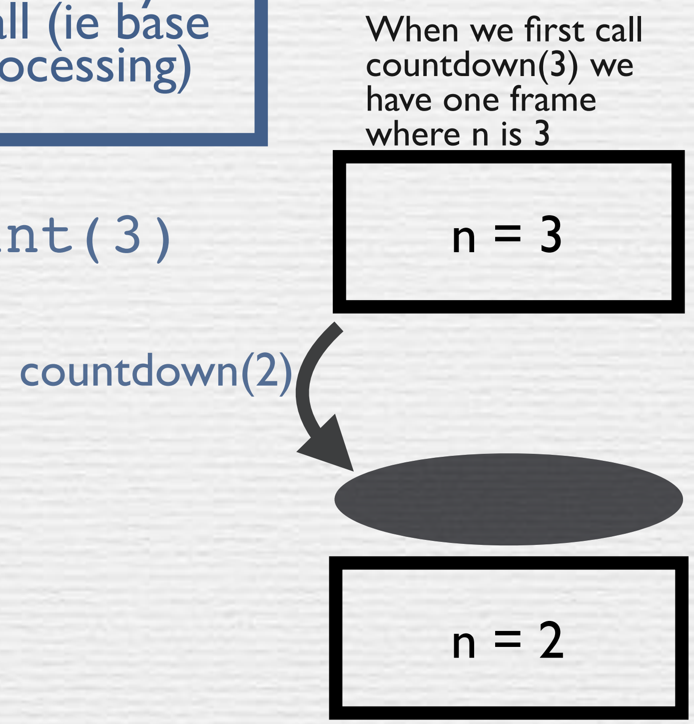
What happens when we call countdown(3)?

Analyzing countdown

```
def countdown(n):
    if n <= 0:
        return
    print(n)
    countdown(n - 1)
```

Work done before your recursive call (ie base case, preprocessing)

`print(3)`



Work done after your recursive call (ie combining results)

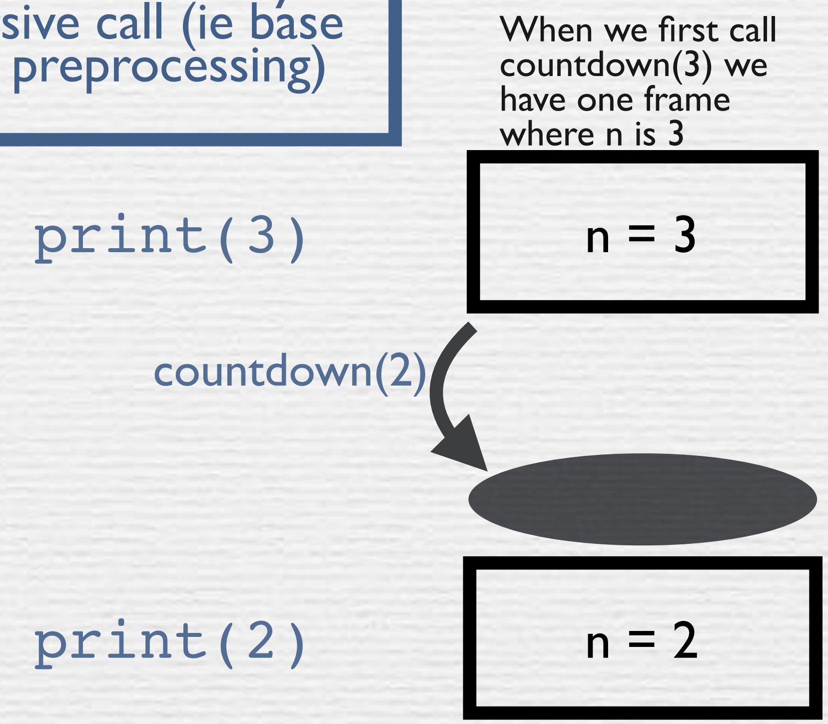
What happens when we call countdown(3)?

Analyzing countdown

```
def countdown(n):
    if n <= 0:
        return
    print(n)
    countdown(n - 1)
```

Work done before your recursive call (ie base case, preprocessing)

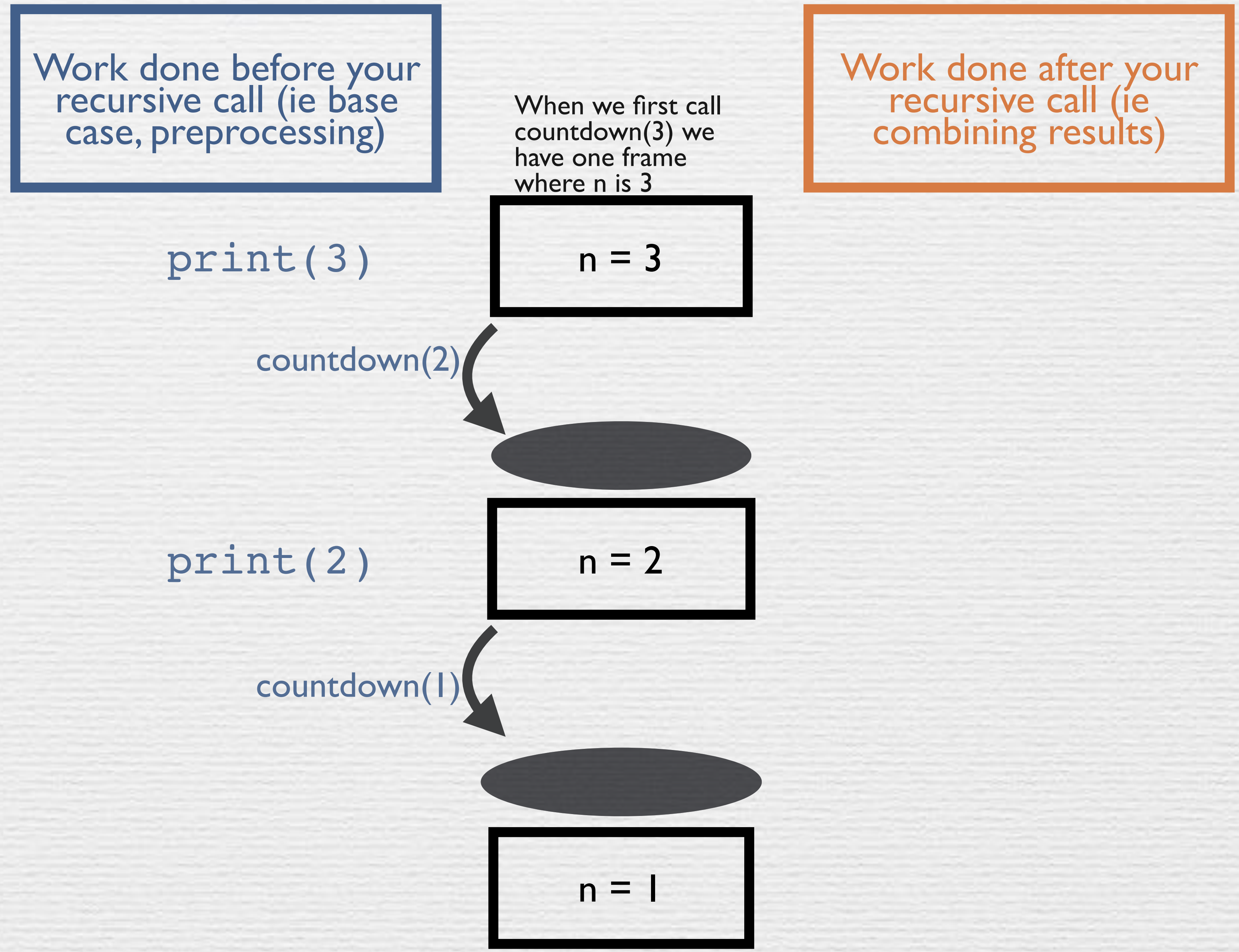
Work done after your recursive call (ie combining results)



What happens when we call countdown(3)?

Analyzing countdown

```
def countdown(n):
    if n <= 0:
        return
    print(n)
    countdown(n - 1)
```



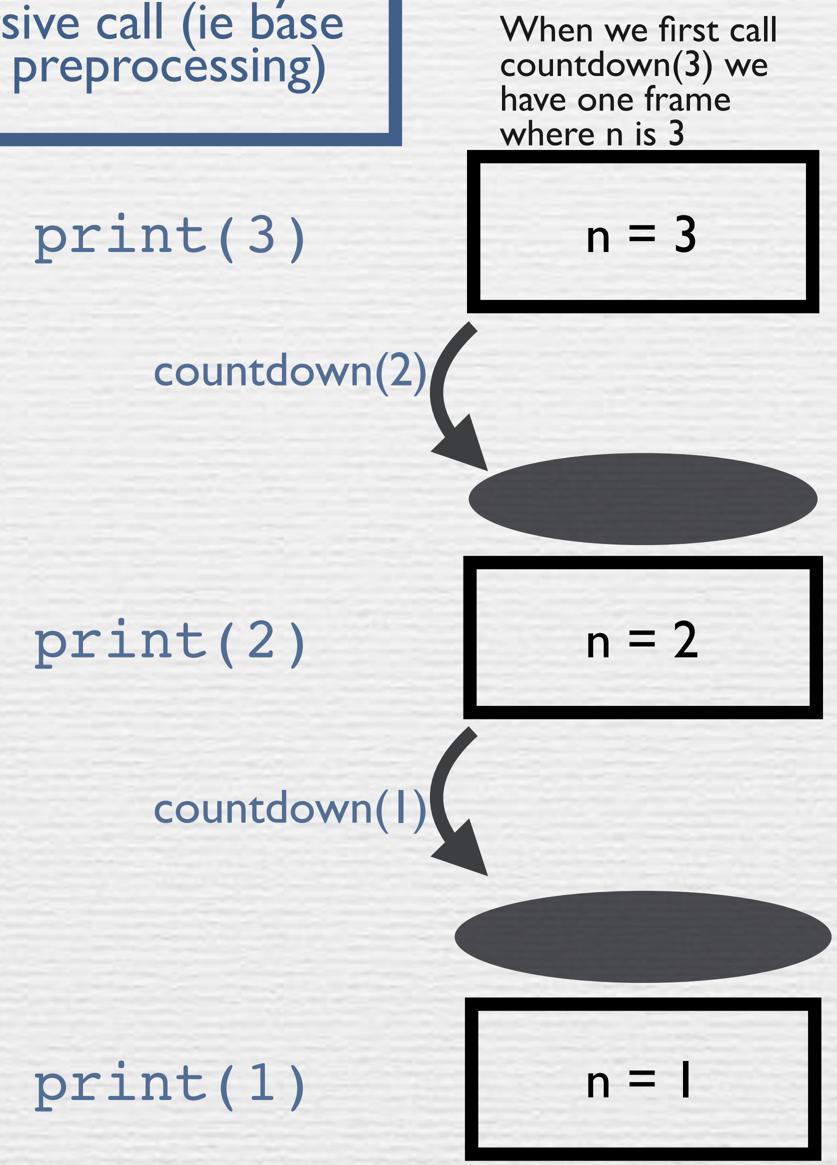
What happens when we call countdown(3)?

Analyzing countdown

```
def countdown(n):
    if n <= 0:
        return
    print(n)
    countdown(n - 1)
```

Work done before your recursive call (ie base case, preprocessing)

Work done after your recursive call (ie combining results)



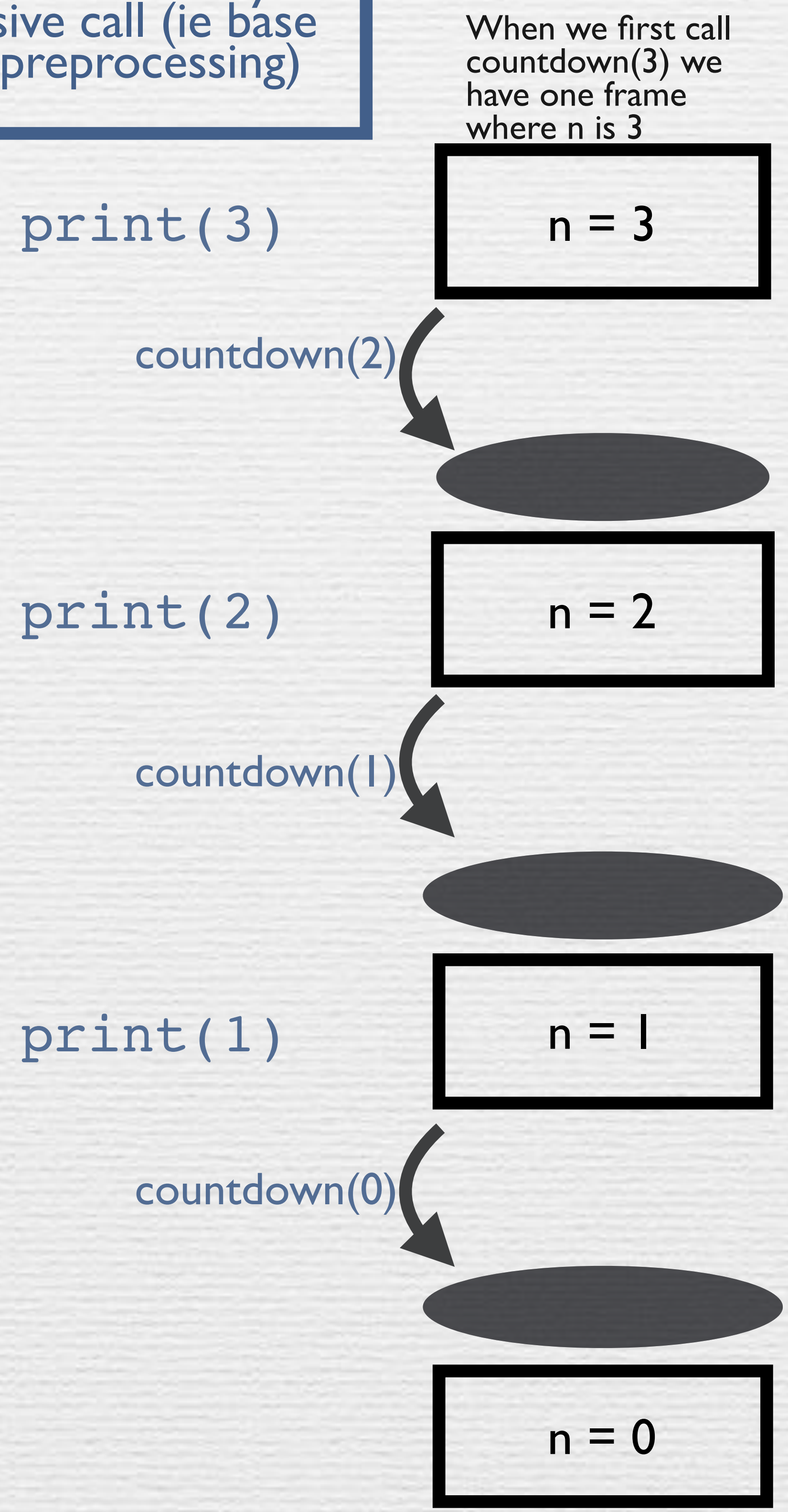
What happens when we call countdown(3)?

Analyzing countdown

```
def countdown(n):
    if n <= 0:
        return
    print(n)
    countdown(n - 1)
```

Work done before your recursive call (ie base case, preprocessing)

Work done after your recursive call (ie combining results)



What happens when we call countdown(3)?

Analyzing countdown

```
def countdown(n):
    if n <= 0:
        return
    print(n)
    countdown(n - 1)
```

What happens when we call countdown(3)?

Work done before your recursive call (ie base case, preprocessing)

When we first call countdown(3) we have one frame where n is 3

Work done after your recursive call (ie combining results)

`print(3)` n = 3

countdown(2)

`print(2)` n = 2

countdown(1)

`print(1)` n = 1

countdown(0)

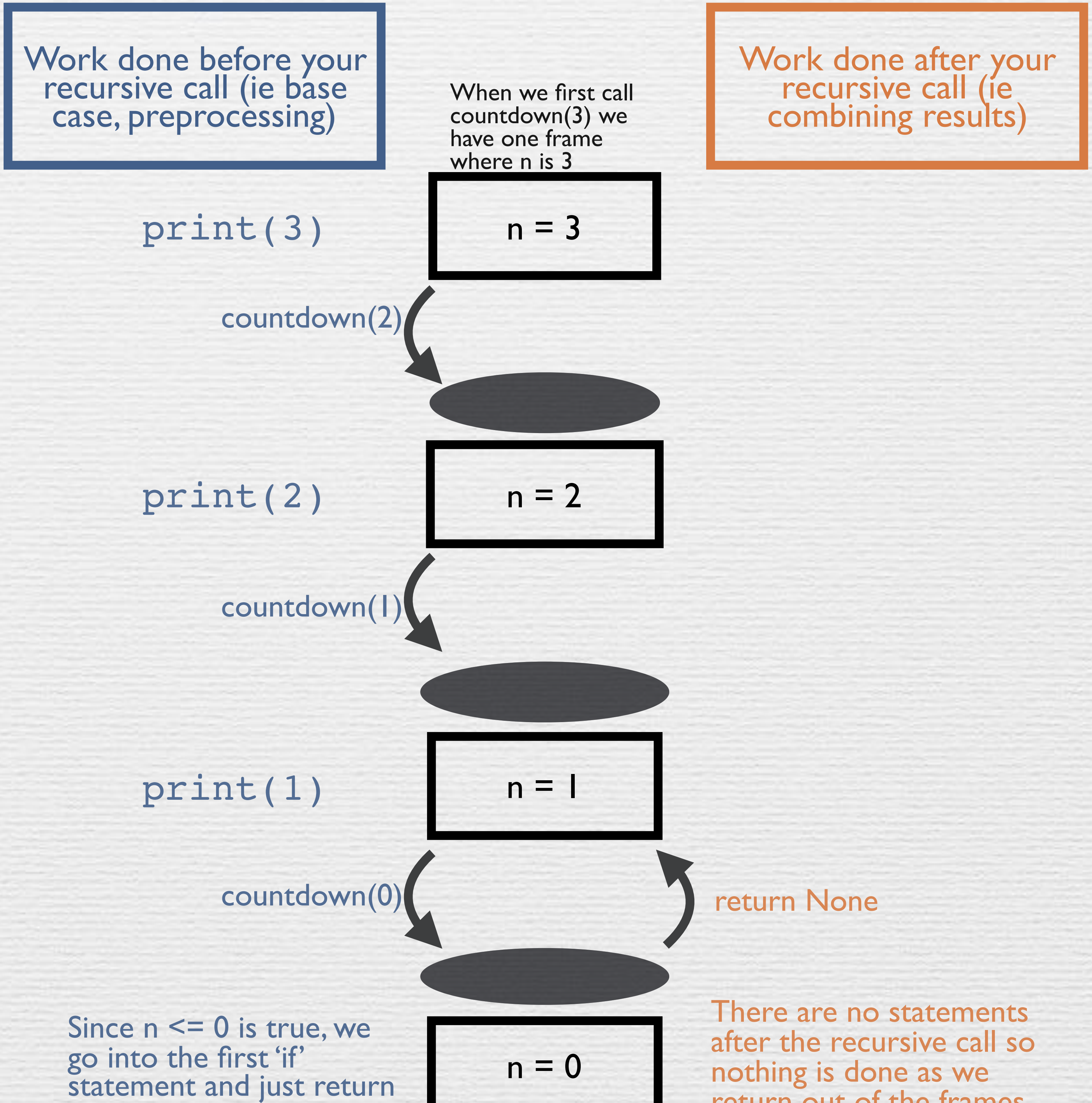
Since n <= 0 is true, we go into the first 'if' statement and just return

n = 0

Analyzing countdown

```
def countdown(n):
    if n <= 0:
        return
    print(n)
    countdown(n - 1)
```

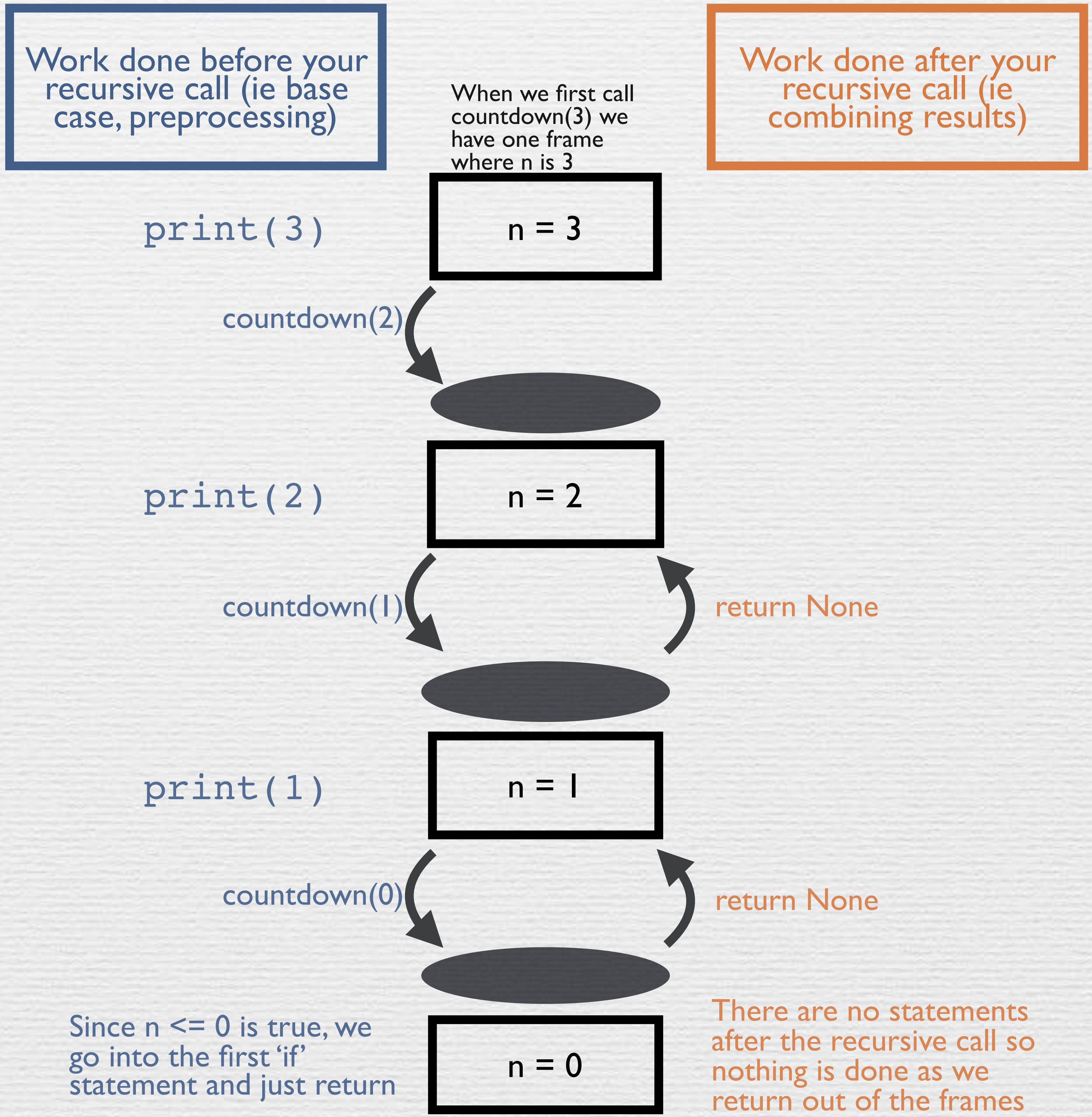
What happens when we call countdown(3)?



Analyzing countdown

```
def countdown(n):
    if n <= 0:
        return
    print(n)
    countdown(n - 1)
```

What happens when we call countdown(3)?



Analyzing countdown

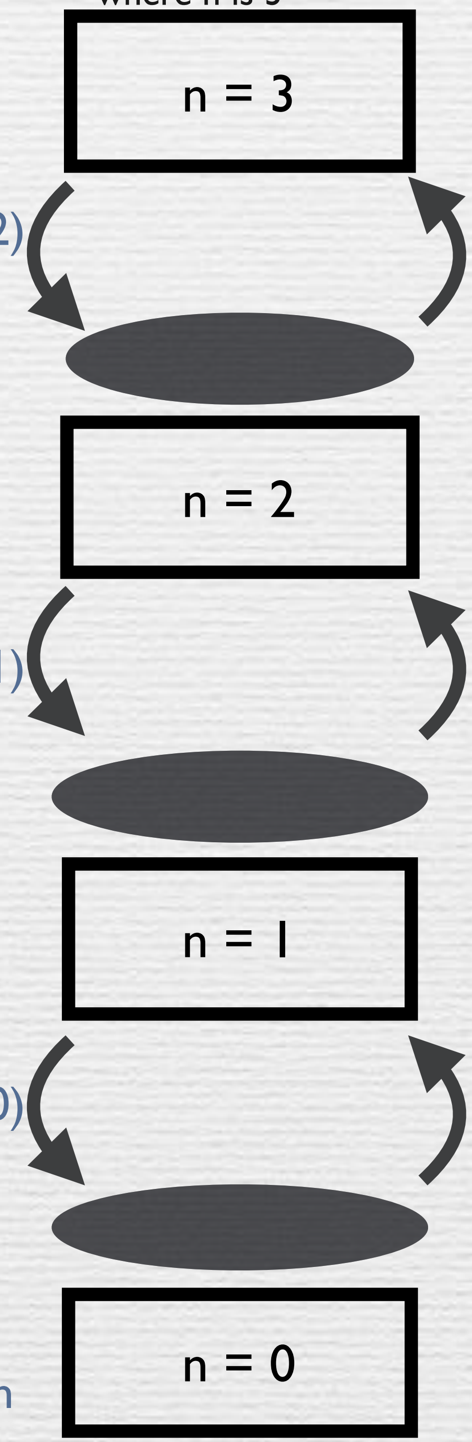
```
def countdown(n):
    if n <= 0:
        return
    print(n)
    countdown(n - 1)
```

What happens when we call countdown(3)?

Work done before your recursive call (ie base case, preprocessing)

Work done after your recursive call (ie combining results)

When we first call countdown(3) we have one frame where n is 3



`print(3)`

countdown(2)

return None

`print(2)`

countdown(1)

return None

`print(1)`

countdown(0)

return None

Since n <= 0 is true, we go into the first 'if' statement and just return

There are no statements after the recursive call so nothing is done as we return out of the frames

Analyzing countup

```
def countup(n):
    if n <= 0:
        return
    countup(n - 1)
    print(n)
```

What happens when we call countup(3)?

Analyzing countup

```
def countup(n):
    if n <= 0:
        return
    countup(n - 1)
    print(n)
```

Work done before your recursive call (ie base case, preprocessing)

When we first call countup(3) we have one frame where n is 3

n = 3

Work done after your recursive call (ie combining results)

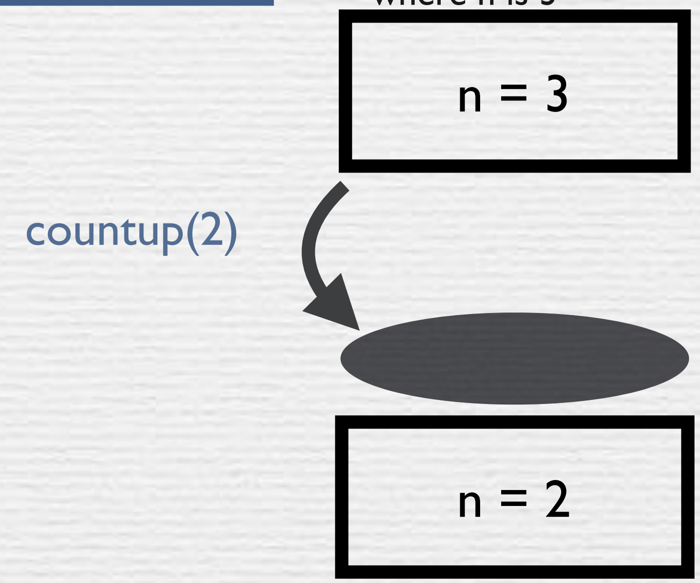
What happens when we call countup(3)?

Analyzing countup

```
def countup(n):
    if n <= 0:
        return
    countup(n - 1)
    print(n)
```

Work done before your recursive call (ie base case, preprocessing)

When we first call countup(3) we have one frame where n is 3



Work done after your recursive call (ie combining results)

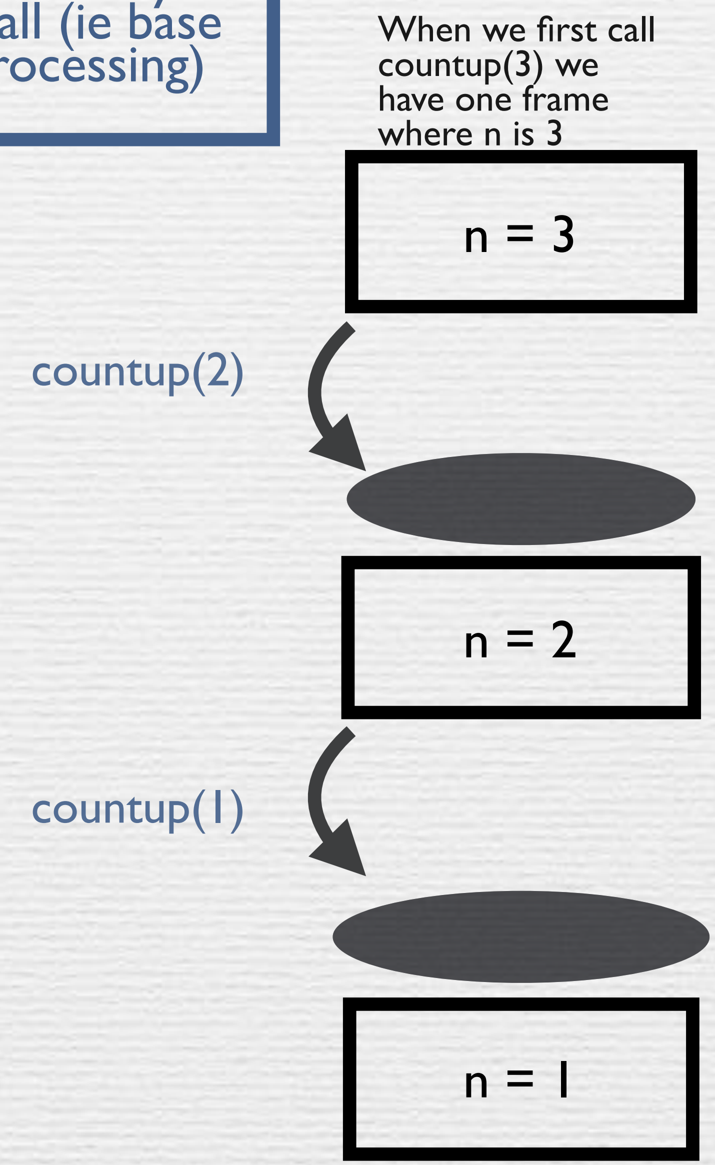
What happens when we call countup(3)?

Analyzing countup

```
def countup(n):
    if n <= 0:
        return
    countup(n - 1)
    print(n)
```

Work done before your recursive call (ie base case, preprocessing)

Work done after your recursive call (ie combining results)

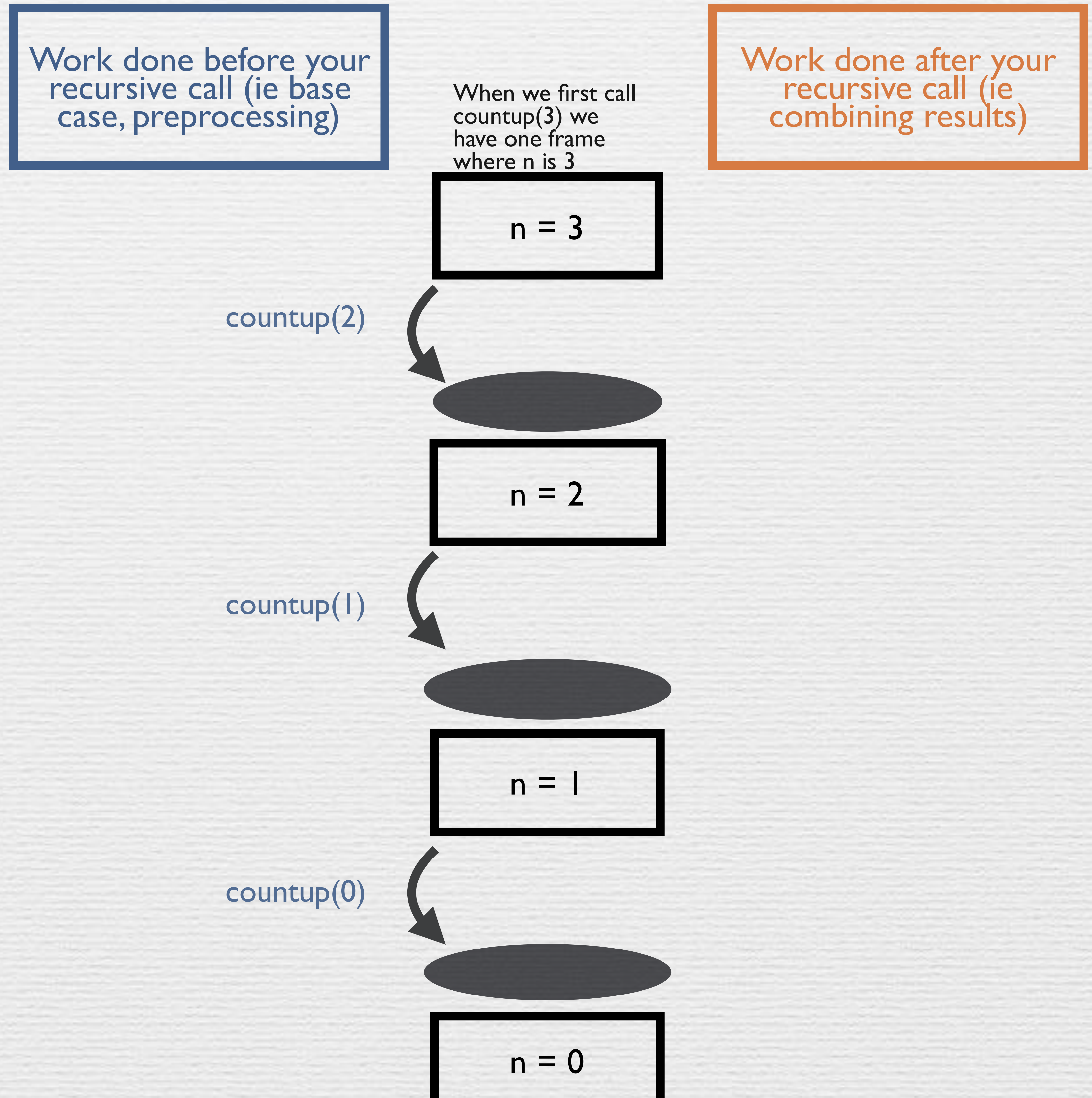


What happens when we call countup(3)?

Analyzing countup

```
def countup(n):
    if n <= 0:
        return
    countup(n - 1)
    print(n)
```

What happens when we call countup(3)?



Analyzing countup

```
def countup(n):
    if n <= 0:
        return
    countup(n - 1)
    print(n)
```

What happens when we call countup(3)?

Work done before your recursive call (ie base case, preprocessing)

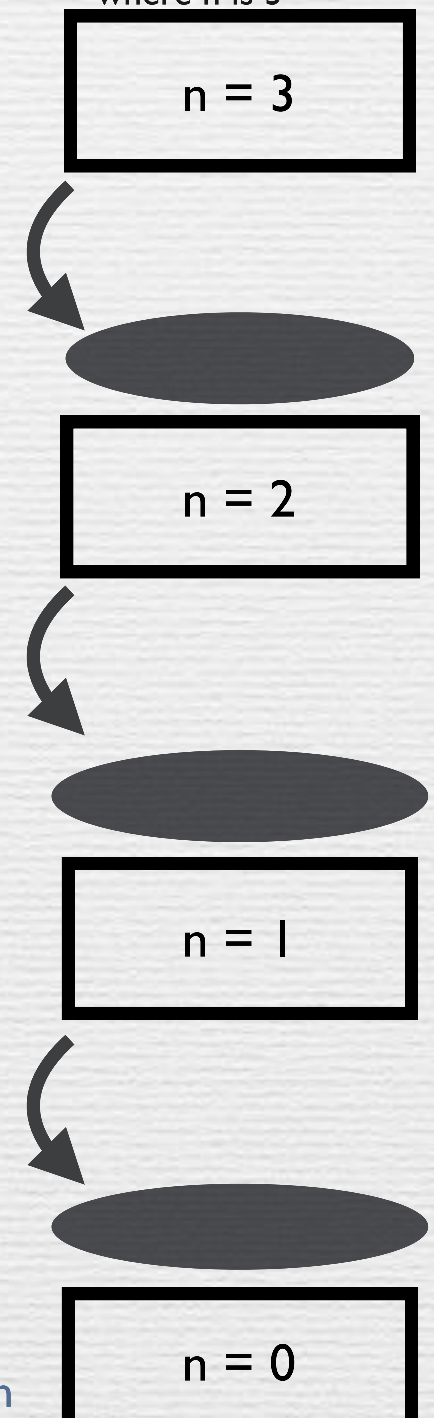
Work done after your recursive call (ie combining results)

When we first call countup(3) we have one frame where n is 3

countup(2)

countup(1)

countup(0)



Since n <= 0 is true, we go into the first 'if' statement and just return

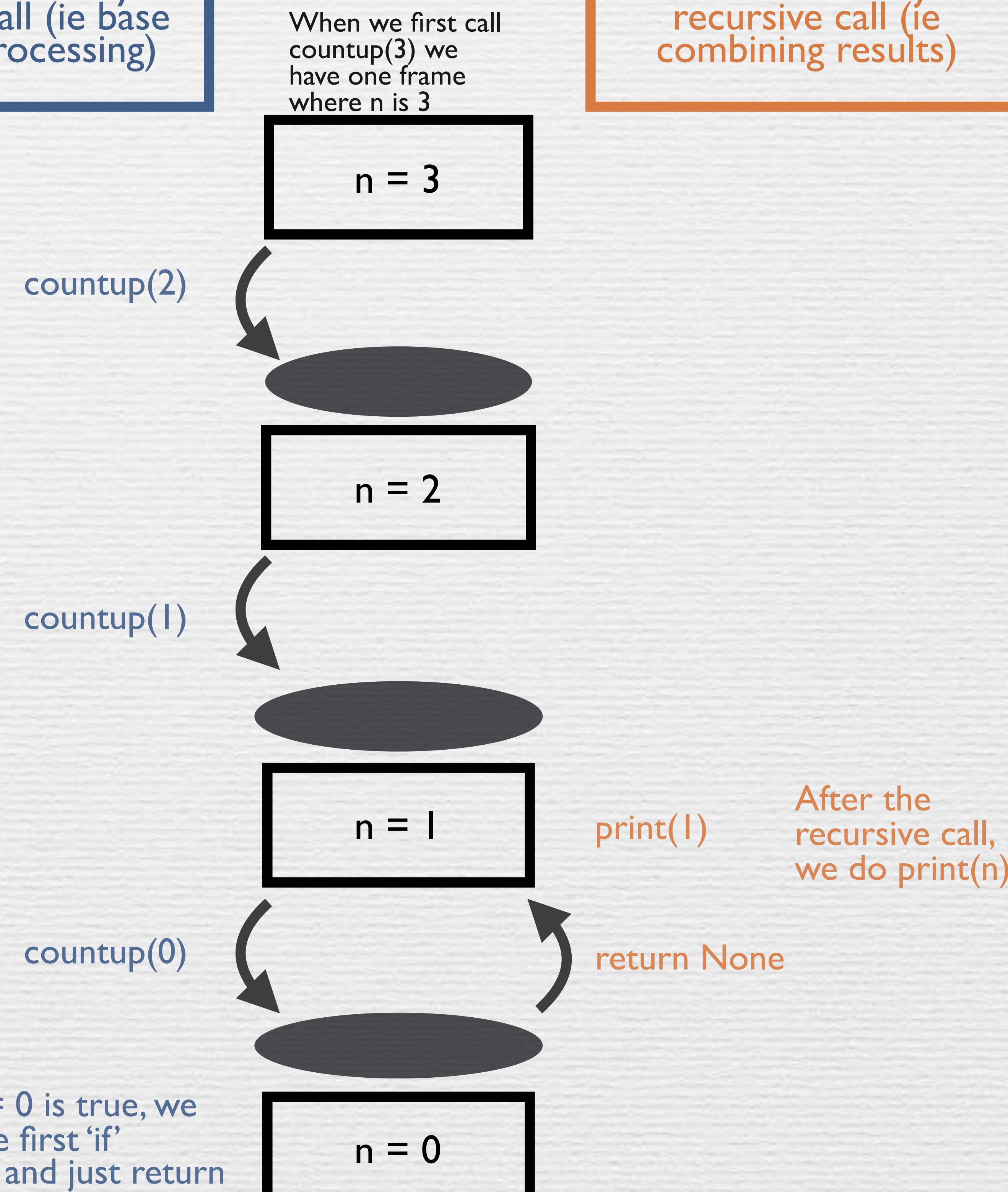
Analyzing countup

```
def countup(n):
    if n <= 0:
        return
    countup(n - 1)
    print(n)
```

What happens when we call countup(3)?

Work done before your recursive call (ie base case, preprocessing)

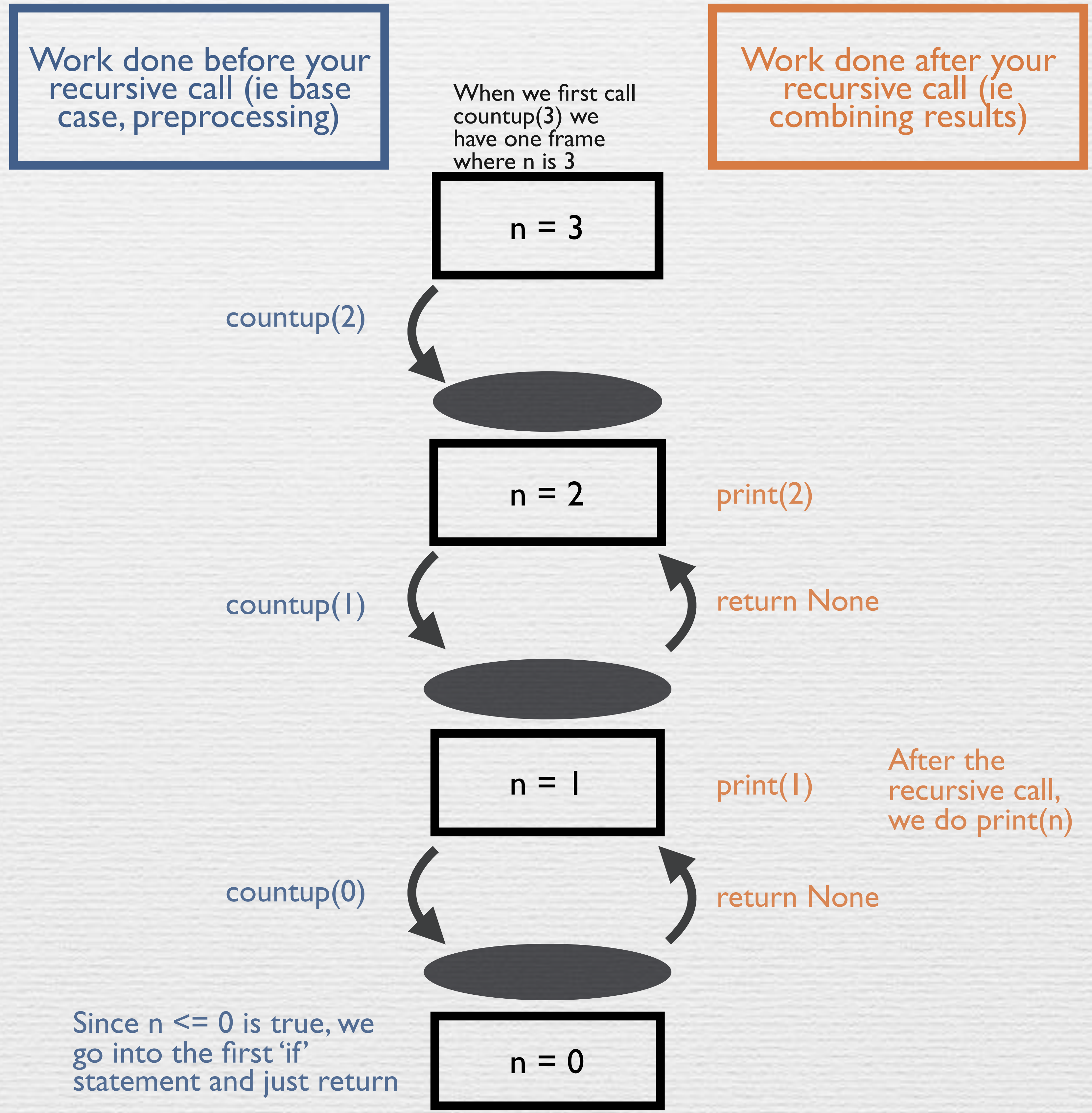
Work done after your recursive call (ie combining results)



Analyzing countup

```
def countup(n):
    if n <= 0:
        return
    countup(n - 1)
    print(n)
```

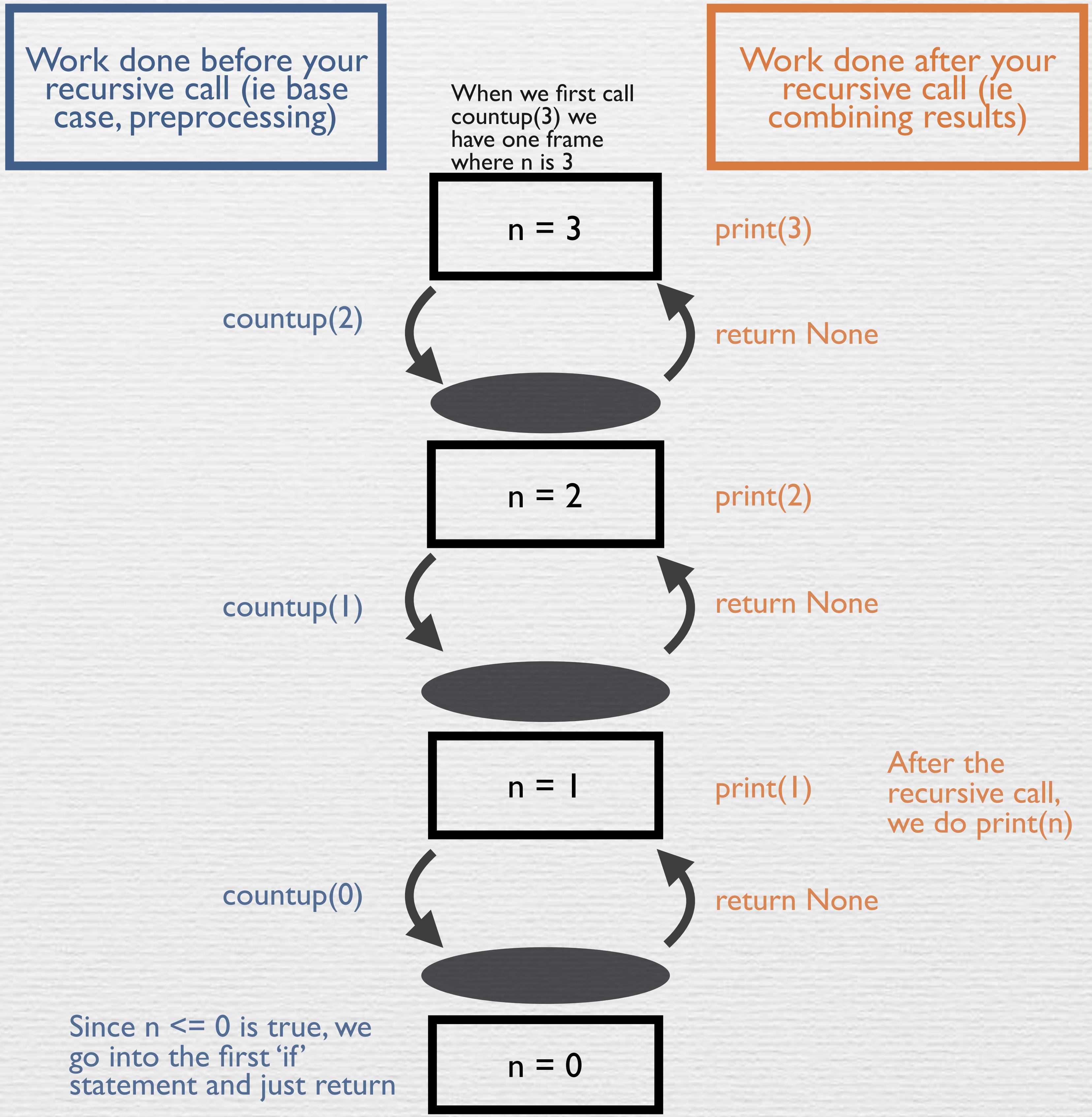
What happens when we call countup(3)?

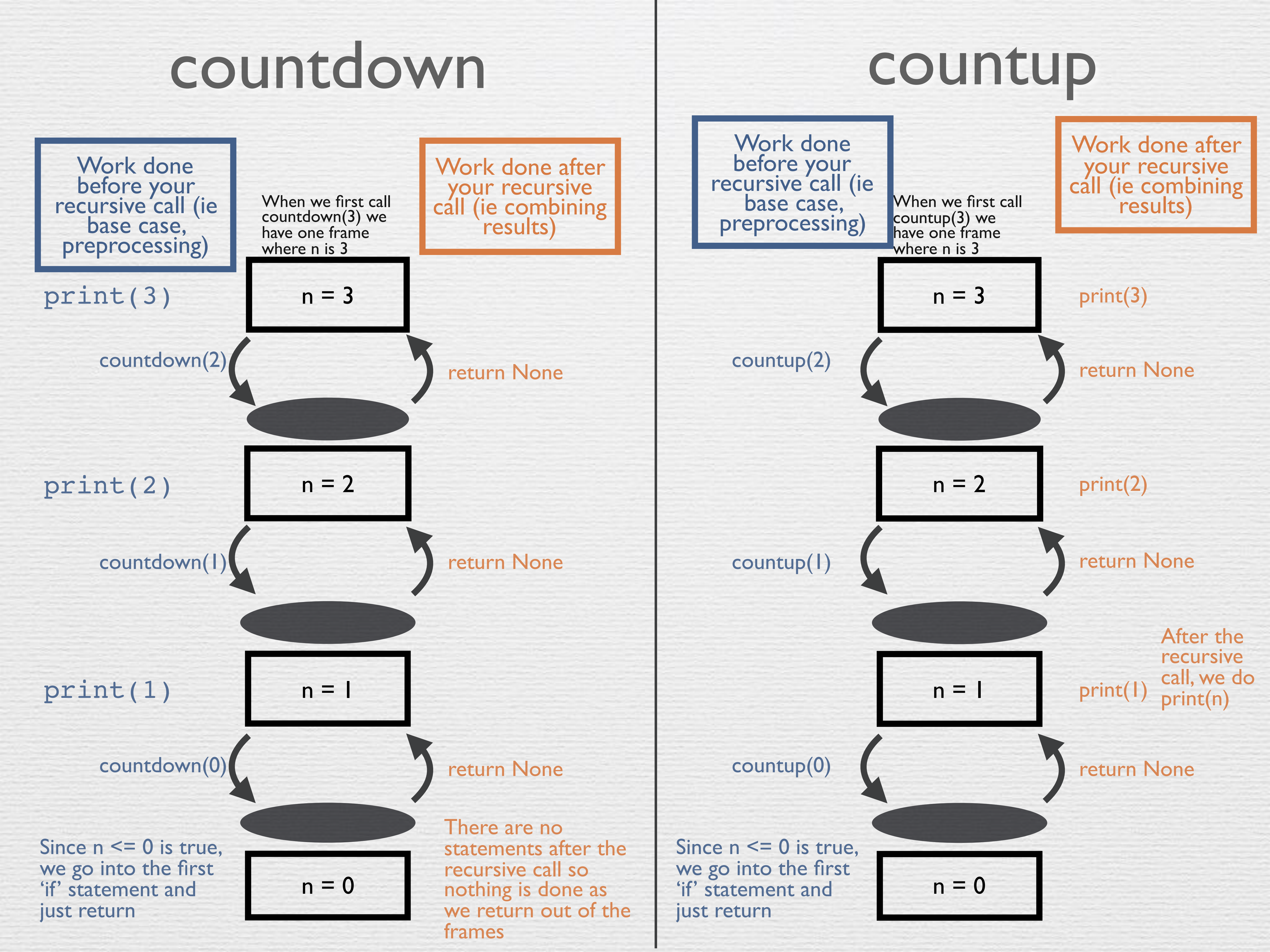


Analyzing countup

```
def countup(n):
    if n <= 0:
        return
    countup(n - 1)
    print(n)
```

What happens when we call countup(3)?




countdown
Work done before your recursive call (ie base case, preprocessing)
When we first call countdown(3) we have one frame where n is 3
Work done after your recursive call (ie combining results)
print(3)
n = 3
countdown(2)
return None
print(2)
n = 2
countdown(1)
return None
print(1)
n = 1
countdown(0)
return None
Since n <= 0 is true, we go into the first 'if' statement and just return
n = 0
There are no statements after the recursive call so nothing is done as we return out of the frames
countup
Work done before your recursive call (ie base case, preprocessing)
When we first call countup(3) we have one frame where n is 3
Work done after your recursive call (ie combining results)
n = 3
print(3)
countup(2)
return None
n = 2
print(2)
countup(1)
return None
After the recursive call, we do print(n)
n = 1
print(1)
countup(0)
return None
Since n <= 0 is true, we go into the first 'if' statement and just return
n = 0

Tree Recursion #1

I want to go up a flight of stairs that has n steps. I can either take 1 or 2 steps each time. How many different ways can I go up this flight of stairs? Write a function **count_stair_ways** that solves this problem for me. Assume n is positive.

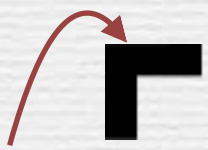
I want to go up a flight of stairs that has n steps. I can either take **1 or 2** steps each time. How many different ways can I go up this flight of stairs? Write a function **`count_stair_ways`** that solves this problem for me. Assume n is positive.

I want to go up a flight of stairs that has n steps. I can either take **1 or 2** steps each time. How many different ways can I go up this flight of stairs? Write a function **`count_stair_ways`** that solves this problem for me. Assume n is positive.

Step 1: Identify your base case
What is the simplest form of the problem? For how many steps do you immediately know what the answer is?

I want to go up a flight of stairs that has n steps. I can either take **1 or 2** steps each time. How many different ways can I go up this flight of stairs? Write a function **`count_stair_ways`** that solves this problem for me. Assume n is positive.

Step 1: Identify your base case
What is the simplest form of the problem? For how many steps do you immediately know what the answer is?

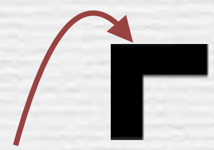


1 way

I want to go up a flight of stairs that has n steps. I can either take **1 or 2** steps each time. How many different ways can I go up this flight of stairs? Write a function **`count_stair_ways`** that solves this problem for me. Assume n is positive.

Step 1: Identify your base case

What is the simplest form of the problem? For how many steps do you immediately know what the answer is?

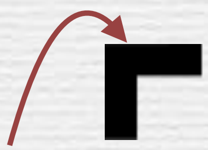


1 way

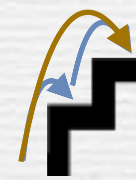
1 + 1 = **2 ways**

I want to go up a flight of stairs that has n steps. I can either take **1 or 2** steps each time. How many different ways can I go up this flight of stairs? Write a function **`count_stair_ways`** that solves this problem for me. Assume n is positive.

Step 1: Identify your base case
What is the simplest form of the problem? For how many steps do you immediately know what the answer is?



1 way



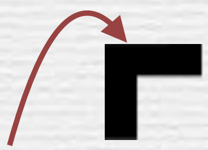
1 + 1 = **2 ways**

Step 2: How do you simplify your problem? Is there a way we can work from n steps to the base case?

I want to go up a flight of stairs that has n steps. I can either take **1 or 2** steps each time. How many different ways can I go up this flight of stairs? Write a function **`count_stair_ways`** that solves this problem for me. Assume n is positive.

Step 1: Identify your base case

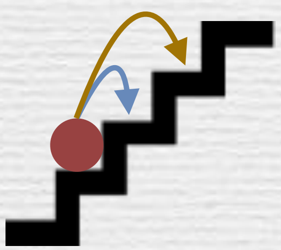
What is the simplest form of the problem? For how many steps do you immediately know what the answer is?



1 way

1 + 1 = **2 ways**

Step 2: How do you simplify your problem? Is there a way we can work from n steps to the base case?

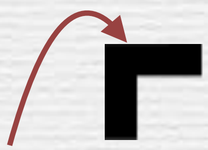


If I am the red dot, I can either move up 1 step or 2 steps, to get closer to the top of the n steps

I want to go up a flight of stairs that has n steps. I can either take **1 or 2** steps each time. How many different ways can I go up this flight of stairs? Write a function **`count_stair_ways`** that solves this problem for me. Assume n is positive.

Step 1: Identify your base case

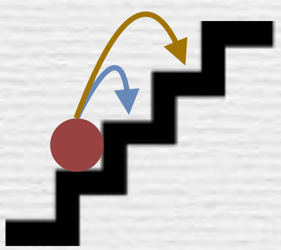
What is the simplest form of the problem? For how many steps do you immediately know what the answer is?



1 way

1 + 1 = 2 ways

Step 2: How do you simplify your problem? Is there a way we can work from n steps to the base case?

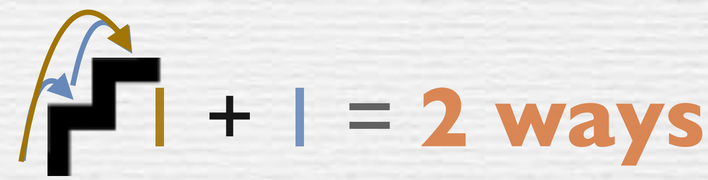


If I am the red dot, I can either move up 1 step or 2 steps, to get closer to the top of the n steps

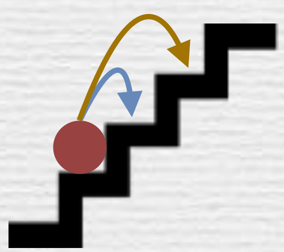
I want to go up a flight of stairs that has n steps. I can either take **1 or 2** steps each time. How many different ways can I go up this flight of stairs? Write a function **`count_stair_ways`** that solves this problem for me. Assume n is positive.

Step 1: Identify your base case

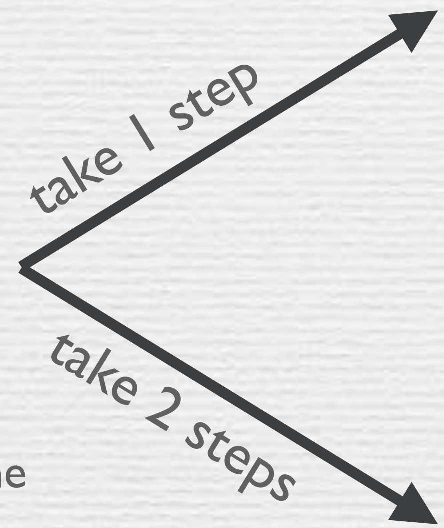
What is the simplest form of the problem? For how many steps do you immediately know what the answer is?



Step 2: How do you simplify your problem? Is there a way we can work from n steps to the base case?



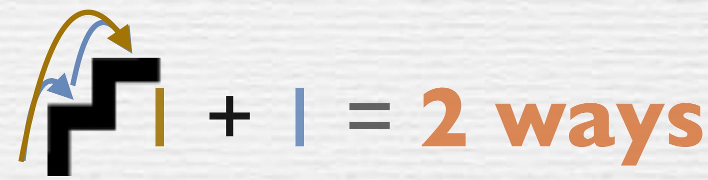
If I am the red dot, I can either move up 1 step or 2 steps, to get closer to the top of the n steps



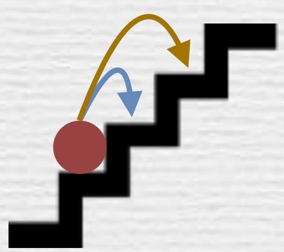
I want to go up a flight of stairs that has n steps. I can either take **1 or 2** steps each time. How many different ways can I go up this flight of stairs? Write a function **`count_stair_ways`** that solves this problem for me. Assume n is positive.

Step 1: Identify your base case

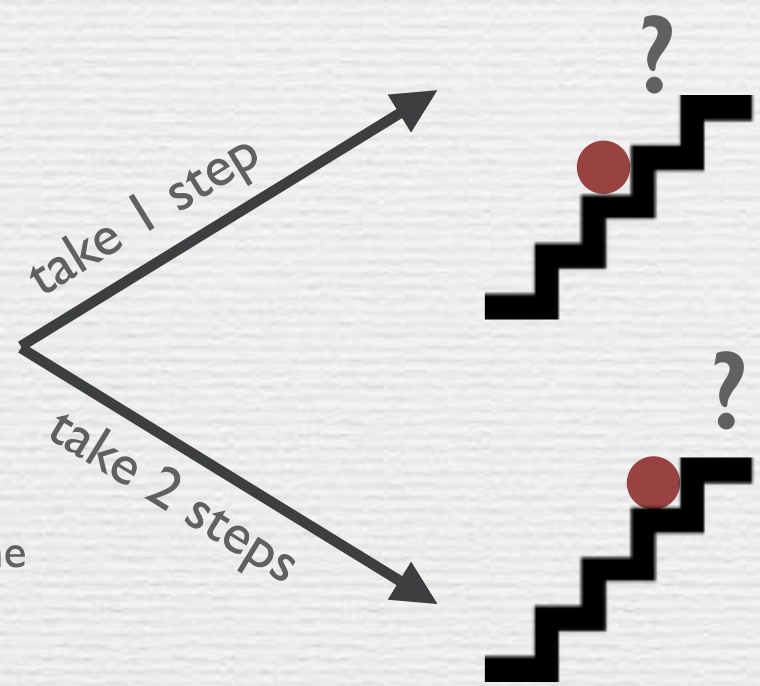
What is the simplest form of the problem? For how many steps do you immediately know what the answer is?



Step 2: How do you simplify your problem? Is there a way we can work from n steps to the base case?



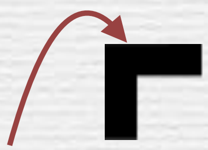
If I am the red dot, I can either move up 1 step or 2 steps, to get closer to the top of the n steps



Now in how many ways can I do the rest of the steps? (Assume that you have a count_stairs_ways function that works)

I want to go up a flight of stairs that has n steps. I can either take **1 or 2** steps each time. How many different ways can I go up this flight of stairs? Write a function `count_stair_ways` that solves this problem for me. Assume n is positive.

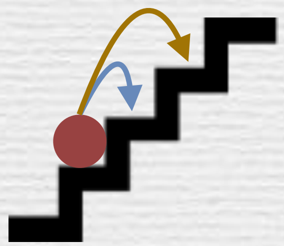
Step 1: Identify your base case
What is the simplest form of the problem? For how many steps do you immediately know what the answer is?



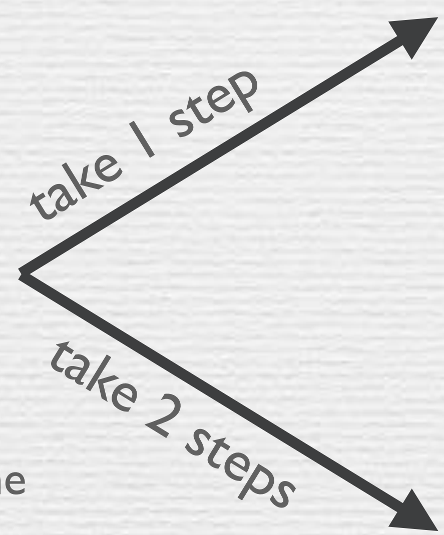
1 way

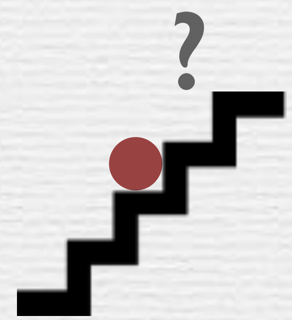
1 + 1 = **2 ways**

Step 2: How do you simplify your problem? Is there a way we can work from n steps to the base case?



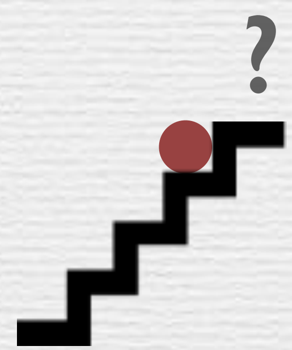
If I am the red dot, I can either move up 1 step or 2 steps, to get closer to the top of the n steps





count_stair_ways(n-1)

Now in how many ways can I do the rest of the steps? (Assume that you have a count_stairs_ways function that works)

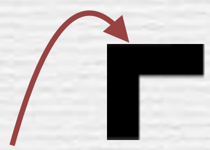


count_stair_ways(n-2)

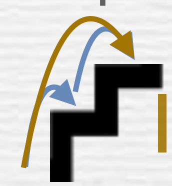
I want to go up a flight of stairs that has n steps. I can either take **1 or 2** steps each time. How many different ways can I go up this flight of stairs? Write a function **`count_stair_ways`** that solves this problem for me. Assume n is positive.

Step 1: Identify your base case

What is the simplest form of the problem? For how many steps do you immediately know what the answer is?

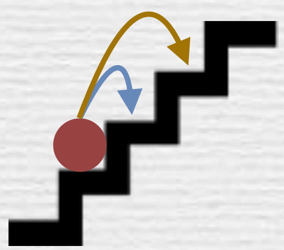


1 way

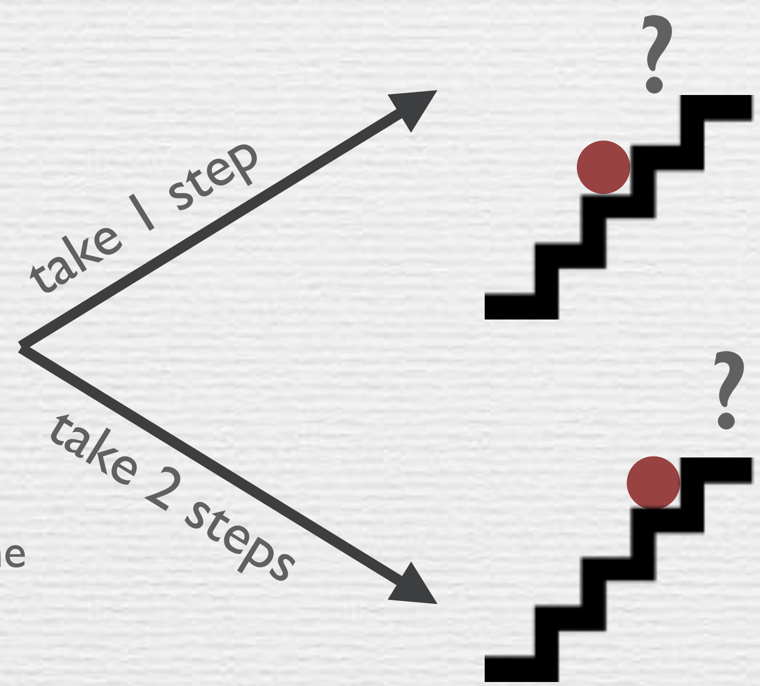


1 + 1 = **2 ways**

Step 2: How do you simplify your problem? Is there a way we can work from n steps to the base case?



If I am the red dot, I can either move up 1 step or 2 steps, to get closer to the top of the n steps



count_stair_ways(n-1)

Now in how many ways can I do the rest of the steps? (Assume that you have a count_stairs_ways function that works)

count_stair_ways(n-2)

Now I know the following facts:

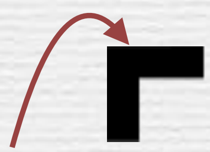
1. I can take either 1 OR 2 steps from my current step
2. Once I take a step(s), I can recursively call my function to determine how many different ways there are for me to continue

Now the million dollar question is: How do I combine these results?

I want to go up a flight of stairs that has n steps. I can either take **1 or 2** steps each time. How many different ways can I go up this flight of stairs? Write a function **`count_stair_ways`** that solves this problem for me. Assume n is positive.

Step 1: Identify your base case

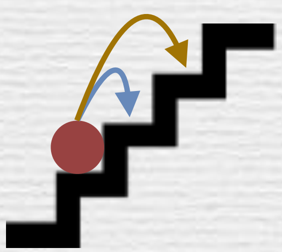
What is the simplest form of the problem? For how many steps do you immediately know what the answer is?



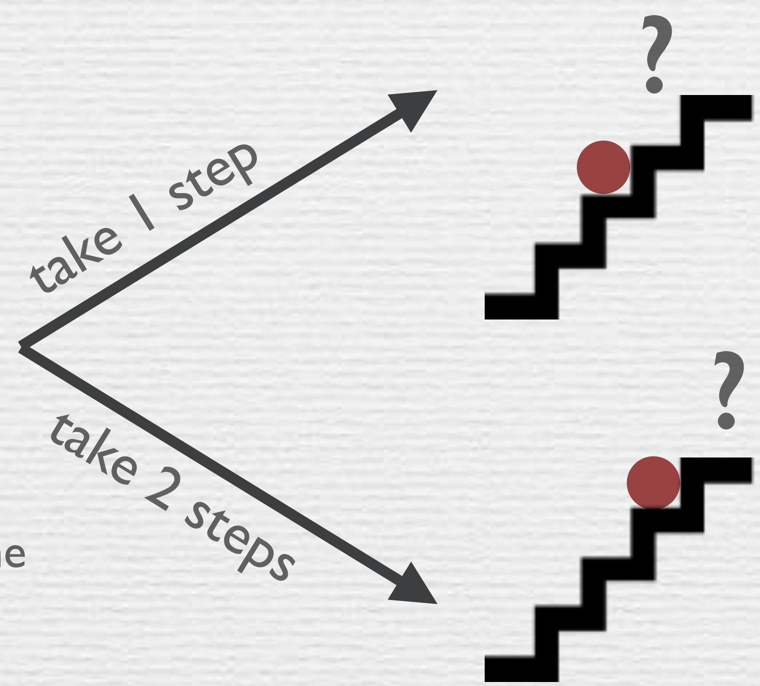
1 way

1 + 1 = **2 ways**

Step 2: How do you simplify your problem? Is there a way we can work from n steps to the base case?



If I am the red dot, I can either move up 1 step or 2 steps, to get closer to the top of the n steps



count_stair_ways(n-1)

Now in how many ways can I do the rest of the steps? (Assume that you have a count_stairs_ways function that works)

count_stair_ways(n-2)

Now I know the following facts:

1. I can take either 1 OR 2 steps from my current step
2. Once I take a step(s), I can recursively call my function to determine how many different ways there are for me to continue

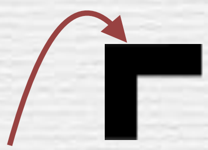
Now the million dollar question is: How do I combine these results?

Step 3: Figure out how your two recursive calls are related. How should you combine their results to figure out what count_stair_ways(n) does?

I want to go up a flight of stairs that has n steps. I can either take **1 or 2** steps each time. How many different ways can I go up this flight of stairs? Write a function **`count_stair_ways`** that solves this problem for me. Assume n is positive.

Step 1: Identify your base case

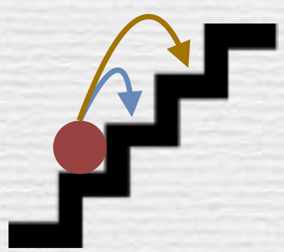
What is the simplest form of the problem? For how many steps do you immediately know what the answer is?



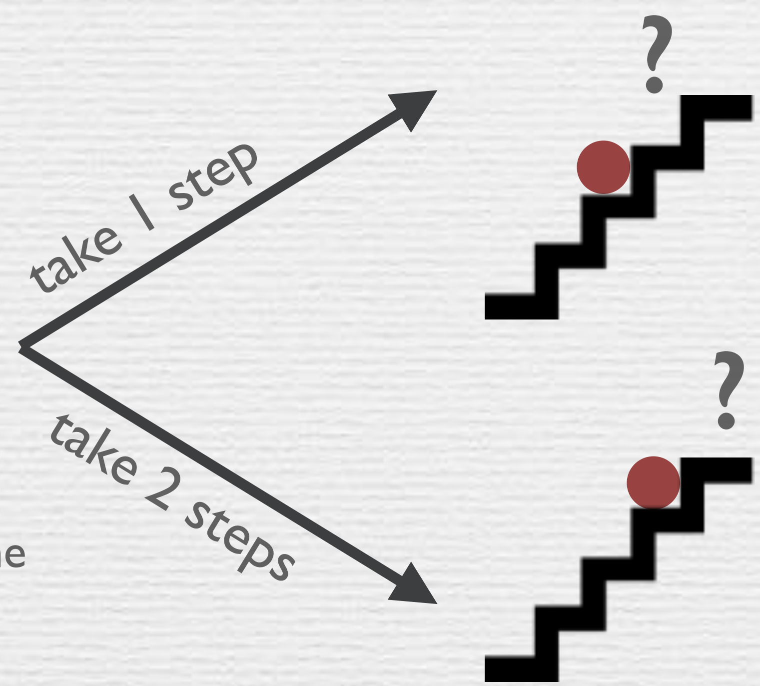
1 way

1 + 1 = 2 ways

Step 2: How do you simplify your problem? Is there a way we can work from n steps to the base case?



If I am the red dot, I can either move up 1 step or 2 steps, to get closer to the top of the n steps



count_stair_ways(n-1)

Now in how many ways can I do the rest of the steps? (Assume that you have a count_stairs_ways function that works)

count_stair_ways(n-2)

Now I know the following facts:

1. I can take either 1 OR 2 steps from my current step
2. Once I take a step(s), I can recursively call my function to determine how many different ways there are for me to continue

Now the million dollar question is: How do I combine these results?

Step 3: Figure out how your two recursive calls are related. How should you combine their results to figure out what count_stair_ways(n) does?

Add them! If I take 1 step, then the total number of remaining combination of steps is count_stair_ways(n-1). If I take 2 steps, then the remaining combination of steps is count_stair_ways(n-2). Since those encompass all of the options I could possibly have from the current step, adding them will ensure I counted all the possible combination of steps from the current step to the top. So we get **count_stair_ways(n-1) + count_stair_ways(n-2).**

Putting it all together

Put the orange text from the previous slide into code:

```
def count_stair_ways(n):
    if n <= 2:
        return n
    return count_stair_ways(n-1) + count_stair_ways(n-2)
```

Tree Recursion #2

Consider an insect in an M by N grid. The insect starts at the bottom left corner, (0, 0), and wants to end up at the top right corner (M - 1, N - 1). The insect can only move up and right. Write a function paths that counts the total number of different paths the insect can take from the start to the goal.

Consider an insect in an M by N grid. The insect starts at the bottom left corner, (0, 0), and wants to end up at the top right corner (M - 1, N - 1). The insect can only move up and right. Write a function paths that counts the total number of different paths the insect can take from the start to the goal.

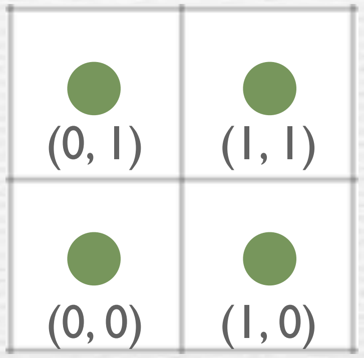
Consider an insect in an M by N grid. The insect starts at the bottom left corner, (0, 0), and wants to end up at the top right corner (M - 1, N - 1). The insect can only move up and right. Write a function paths that counts the total number of different paths the insect can take from the start to the goal.

Step 0: Understand what is asked. Let's draw a picture!

Consider an insect in an M by N grid. The insect starts at the bottom left corner, (0, 0), and wants to end up at the top right corner (M - 1, N - 1). The insect can only move up and right. Write a function paths that counts the total number of different paths the insect can take from the start to the goal.

Step 0: Understand what is asked. Let's draw a picture!

Ex: M = 2, N = 2



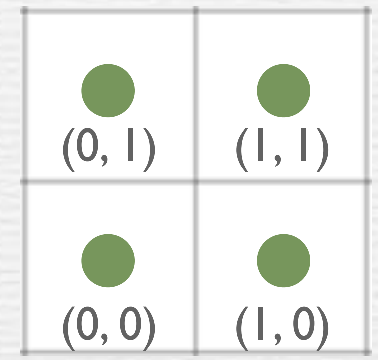
Note: the points are in the **MIDDLE** of each square

Consider an insect in an M by N grid. The insect starts at the bottom left corner, (0, 0), and wants to end up at the top right corner (M - 1, N - 1). The insect can only move up and right. Write a function paths that counts the total number of different paths the insect can take from the start to the goal.

Step 0: Understand what is asked. Let's draw a picture!

Note: The problem is ask the different paths from the bottom left to the top right, if we can only move up and right. This is the same s the number of different paths from the top right to the bottom left, if we can only move down and left (just reverse the direction of each of the paths). We are going to be working with the second case, because the input to paths is M and N, making it easier to start the path at position (M - 1, N - 1).

Ex: M = 2, N = 2



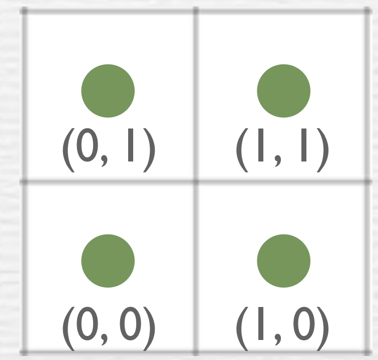
Note: the points are in the **MIDDLE** of each square

Consider an insect in an M by N grid. The insect starts at the bottom left corner, (0, 0), and wants to end up at the top right corner (M - 1, N - 1). The insect can only move up and right. Write a function paths that counts the total number of different paths the insect can take from the start to the goal.

Step 0: Understand what is asked. Let's draw a picture!

Note: The problem is ask the different paths from the bottom left to the top right, if we can only move up and right. This is the same s the number of different paths from the top right to the bottom left, if we can only move down and left (just reverse the direction of each of the paths). We are going to be working with the second case, because the input to paths is M and N, making it easier to start the path at position (M - 1, N - 1).

Ex: M = 2, N = 2



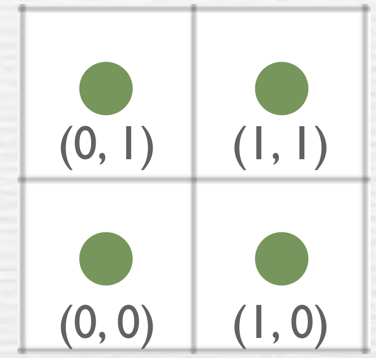
Note: the points are in the **MIDDLE** of each square

Step 1: Identify your base case. For what grids do you know for sure how many paths there are? Are there certain grids that "force" a path?

Consider an insect in an M by N grid. The insect starts at the bottom left corner, (0, 0), and wants to end up at the top right corner (M - 1, N - 1). The insect can only move up and right. Write a function paths that counts the total number of different paths the insect can take from the start to the goal.

Step 0: Understand what is asked. Let's draw a picture!

Ex: M = 2, N = 2

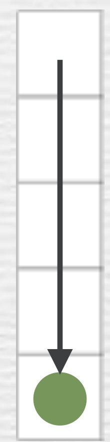


Note: the points are in the **MIDDLE** of each square

Note: The problem is ask the different paths from the bottom left to the top right, if we can only move up and right. This is the same s the number of different paths from the top right to the bottom left, if we can only move down and left (just reverse the direction of each of the paths). We are going to be working with the second case, because the input to paths is M and N, making it easier to start the path at position (M - 1, N - 1).

Step 1: Identify your base case. For what grids do you know for sure how many paths there are? Are there certain grids that "force" a path?

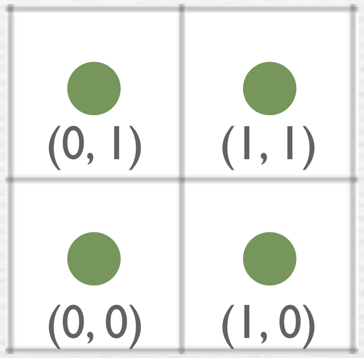
Can only move down



Consider an insect in an M by N grid. The insect starts at the bottom left corner, (0, 0), and wants to end up at the top right corner (M - 1, N - 1). The insect can only move up and right. Write a function paths that counts the total number of different paths the insect can take from the start to the goal.

Step 0: Understand what is asked. Let's draw a picture!

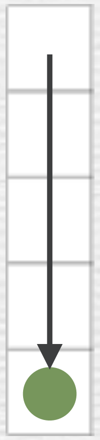
Ex: M = 2, N = 2



Note: the points are in the **MIDDLE** of each square

Note: The problem is ask the different paths from the bottom left to the top right, if we can only move up and right. This is the same s the number of different paths from the top right to the bottom left, if we can only move down and left (just reverse the direction of each of the paths). We are going to be working with the second case, because the input to paths is M and N, making it easier to start the path at position (M - 1, N - 1).

Step 1: Identify your base case. For what grids do you know for sure how many paths there are? Are there certain grids that "force" a path?



Can only move down

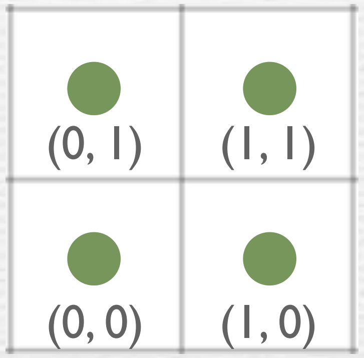
Can only move left

Consider an insect in an M by N grid. The insect starts at the bottom left corner, (0, 0), and wants to end up at the top right corner (M - 1, N - 1). The insect can only move up and right. Write a function paths that counts the total number of different paths the insect can take from the start to the goal.

Step 0: Understand what is asked. Let's draw a picture!

Note: The problem is ask the different paths from the bottom left to the top right, if we can only move up and right. This is the same s the number of different paths from the top right to the bottom left, if we can only move down and left (just reverse the direction of each of the paths). We are going to be working with the second case, because the input to paths is M and N, making it easier to start the path at position (M - 1, N - 1).

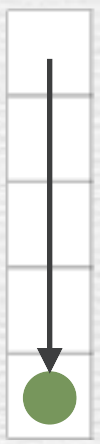
Ex: M = 2, N = 2



Note: the points are in the **MIDDLE** of each square

Step 1: Identify your base case. For what grids do you know for sure how many paths there are? Are there certain grids that "force" a path?

Can only move down



Can only move left

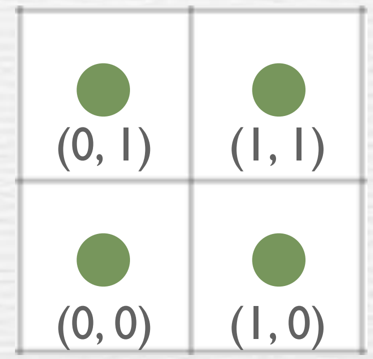
So if N is 1 OR M is 1, we know there is only 1 path

Consider an insect in an M by N grid. The insect starts at the bottom left corner, (0, 0), and wants to end up at the top right corner (M - 1, N - 1). The insect can only move up and right. Write a function paths that counts the total number of different paths the insect can take from the start to the goal.

Step 0: Understand what is asked. Let's draw a picture!

Note: The problem is ask the different paths from the bottom left to the top right, if we can only move up and right. This is the same s the number of different paths from the top right to the bottom left, if we can only move down and left (just reverse the direction of each of the paths). We are going to be working with the second case, because the input to paths is M and N, making it easier to start the path at position (M - 1, N - 1).

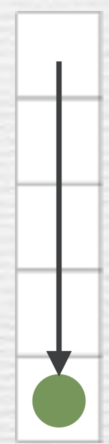
Ex: M = 2, N = 2



Note: the points are in the **MIDDLE** of each square

Step 1: Identify your base case. For what grids do you know for sure how many paths there are? Are there certain grids that "force" a path?

Can only move down



Can only move left

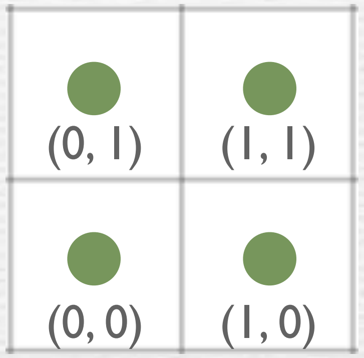
So if N is 1 OR M is 1, we know there is only 1 path

Step 2: Break down your problem into recursive calls. How you can you simplify the original problem?

Consider an insect in an M by N grid. The insect starts at the bottom left corner, (0, 0), and wants to end up at the top right corner (M - 1, N - 1). The insect can only move up and right. Write a function paths that counts the total number of different paths the insect can take from the start to the goal.

Step 0: Understand what is asked. Let's draw a picture!

Ex: M = 2, N = 2

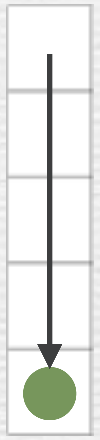


Note: the points are in the **MIDDLE** of each square

Note: The problem is ask the different paths from the bottom left to the top right, if we can only move up and right. This is the same s the number of different paths from the top right to the bottom left, if we can only move down and left (just reverse the direction of each of the paths). We are going to be working with the second case, because the input to paths is M and N, making it easier to start the path at position (M - 1, N - 1).

Step 1: Identify your base case. For what grids do you know for sure how many paths there are? Are there certain grids that "force" a path?

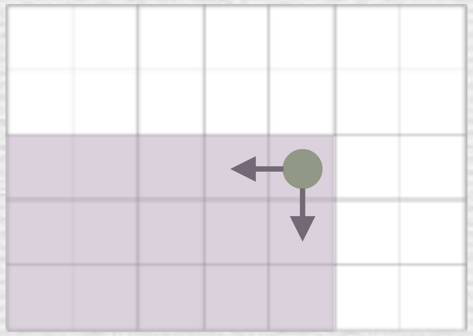
Can only move down



Can only move left

So if N is 1 OR M is 1, we know there is only 1 path

Step 2: Break down your problem into recursive calls. How you can you simplify the original problem?

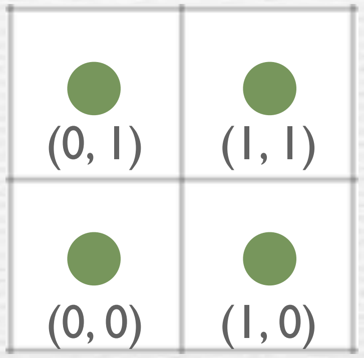


Consider an insect in an M by N grid. The insect starts at the bottom left corner, (0, 0), and wants to end up at the top right corner (M - 1, N - 1). The insect can only move up and right. Write a function paths that counts the total number of different paths the insect can take from the start to the goal.

Step 0: Understand what is asked. Let's draw a picture!

Note: The problem is ask the different paths from the bottom left to the top right, if we can only move up and right. This is the same s the number of different paths from the top right to the bottom left, if we can only move down and left (just reverse the direction of each of the paths). We are going to be working with the second case, because the input to paths is M and N, making it easier to start the path at position (M - 1, N - 1).

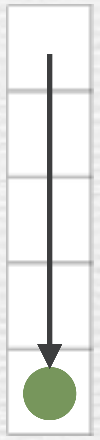
Ex: M = 2, N = 2



Note: the points are in the **MIDDLE** of each square

Step 1: Identify your base case. For what grids do you know for sure how many paths there are? Are there certain grids that "force" a path?

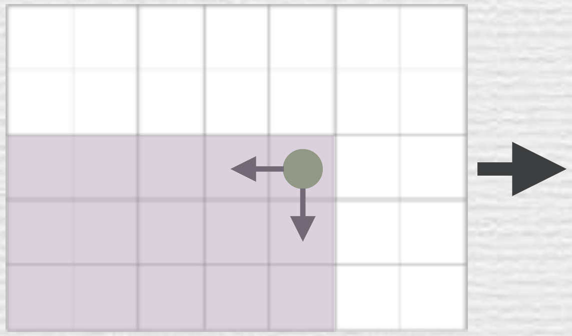
Can only move down



Can only move left

So if N is 1 OR M is 1, we know there is only 1 path

Step 2: Break down your problem into recursive calls. How you can you simplify the original problem?

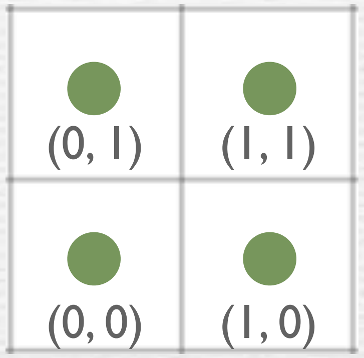


Consider an insect in an M by N grid. The insect starts at the bottom left corner, (0, 0), and wants to end up at the top right corner (M - 1, N - 1). The insect can only move up and right. Write a function paths that counts the total number of different paths the insect can take from the start to the goal.

Step 0: Understand what is asked. Let’s draw a picture!

Note: The problem is ask the different paths from the bottom left to the top right, if we can only move up and right. This is the same s the number of different paths from the top right to the bottom left, if we can only move down and left (just reverse the direction of each of the paths). We are going to be working with the second case, because the input to paths is M and N, making it easier to start the path at position (M - 1, N - 1).

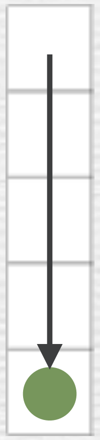
Ex: M = 2, N = 2



Note: the points are in the **MIDDLE** of each square

Step 1: Identify your base case. For what grids do you know for sure how many paths there are? Are there certain grids that “force” a path?

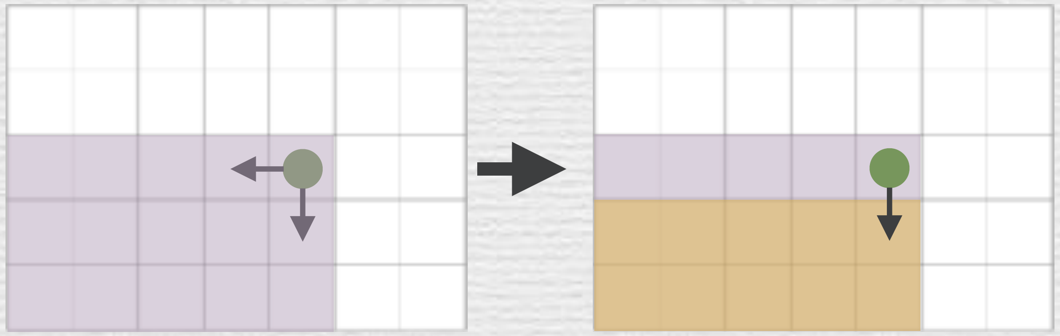
Can only move down



Can only move left

So if N is 1 OR M is 1, we know there is only 1 path

Step 2: Break down your problem into recursive calls. How you can you simplify the original problem?

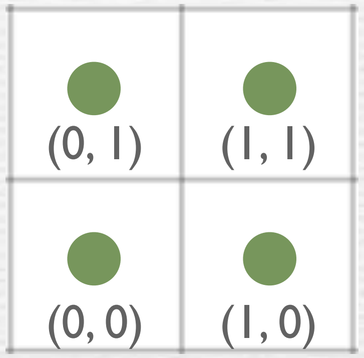


Consider an insect in an M by N grid. The insect starts at the bottom left corner, (0, 0), and wants to end up at the top right corner (M - 1, N - 1). The insect can only move up and right. Write a function paths that counts the total number of different paths the insect can take from the start to the goal.

Step 0: Understand what is asked. Let's draw a picture!

Note: The problem is ask the different paths from the bottom left to the top right, if we can only move up and right. This is the same s the number of different paths from the top right to the bottom left, if we can only move down and left (just reverse the direction of each of the paths). We are going to be working with the second case, because the input to paths is M and N, making it easier to start the path at position (M - 1, N - 1).

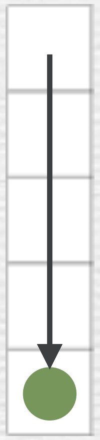
Ex: M = 2, N = 2



Note: the points are in the **MIDDLE** of each square

Step 1: Identify your base case. For what grids do you know for sure how many paths there are? Are there certain grids that "force" a path?

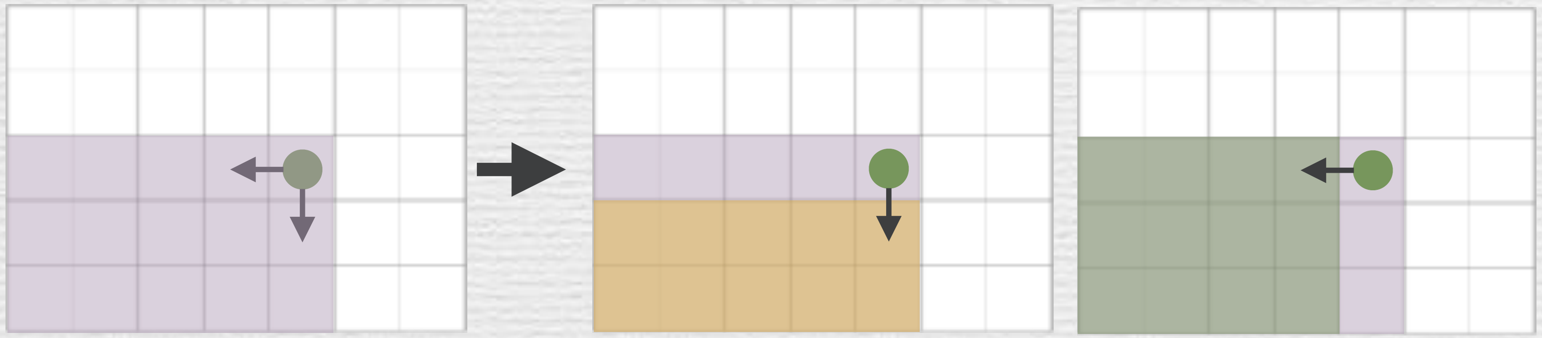
Can only move down



Can only move left

So if N is 1 OR M is 1, we know there is only 1 path

Step 2: Break down your problem into recursive calls. How you can you simplify the original problem?

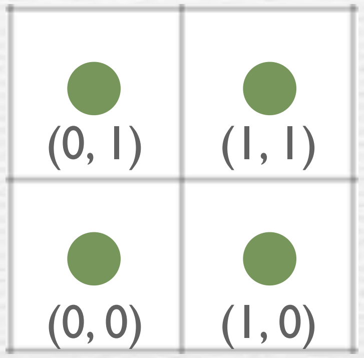


Consider an insect in an M by N grid. The insect starts at the bottom left corner, (0, 0), and wants to end up at the top right corner (M - 1, N - 1). The insect can only move up and right. Write a function paths that counts the total number of different paths the insect can take from the start to the goal.

Step 0: Understand what is asked. Let's draw a picture!

Note: The problem is ask the different paths from the bottom left to the top right, if we can only move up and right. This is the same s the number of different paths from the top right to the bottom left, if we can only move down and left (just reverse the direction of each of the paths). We are going to be working with the second case, because the input to paths is M and N, making it easier to start the path at position (M - 1, N - 1).

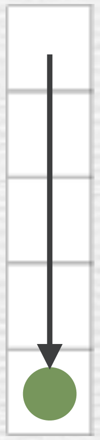
Ex: M = 2, N = 2



Note: the points are in the **MIDDLE** of each square

Step 1: Identify your base case. For what grids do you know for sure how many paths there are? Are there certain grids that "force" a path?

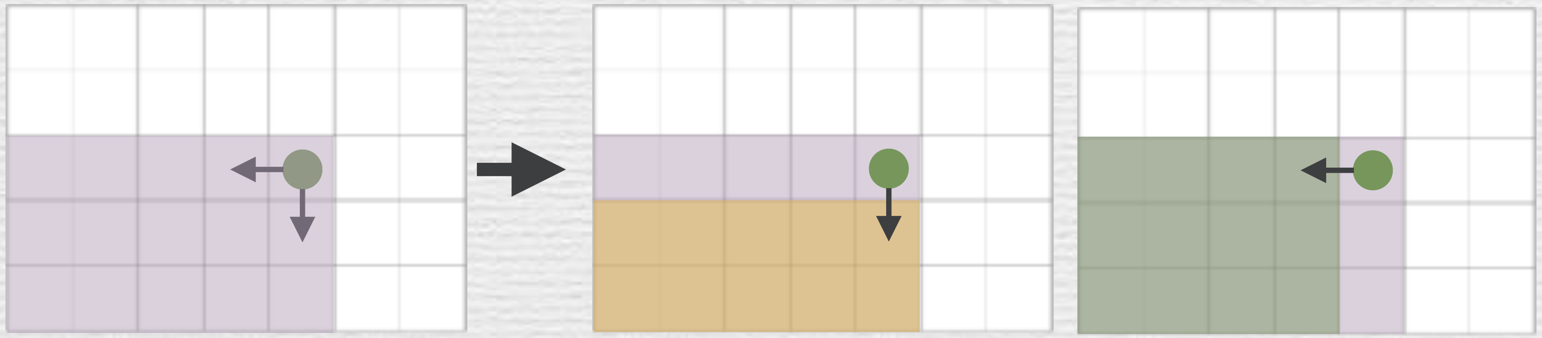
Can only move down



Can only move left

So if N is 1 OR M is 1, we know there is only 1 path

Step 2: Break down your problem into recursive calls. How you can you simplify the original problem?



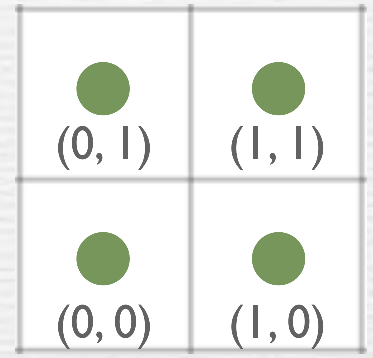
At each point you can only go left or down.

So we have **paths(N, M - 1)** and **paths(N-1, M)**

Consider an insect in an M by N grid. The insect starts at the bottom left corner, (0, 0), and wants to end up at the top right corner (M - 1, N - 1). The insect can only move up and right. Write a function paths that counts the total number of different paths the insect can take from the start to the goal.

Step 0: Understand what is asked. Let's draw a picture!

Ex: M = 2, N = 2

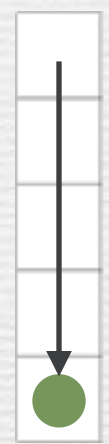


Note: the points are in the **MIDDLE** of each square

Note: The problem is ask the different paths from the bottom left to the top right, if we can only move up and right. This is the same s the number of different paths from the top right to the bottom left, if we can only move down and left (just reverse the direction of each of the paths). We are going to be working with the second case, because the input to paths is M and N, making it easier to start the path at position (M - 1, N - 1).

Step 1: Identify your base case. For what grids do you know for sure how many paths there are? Are there certain grids that "force" a path?

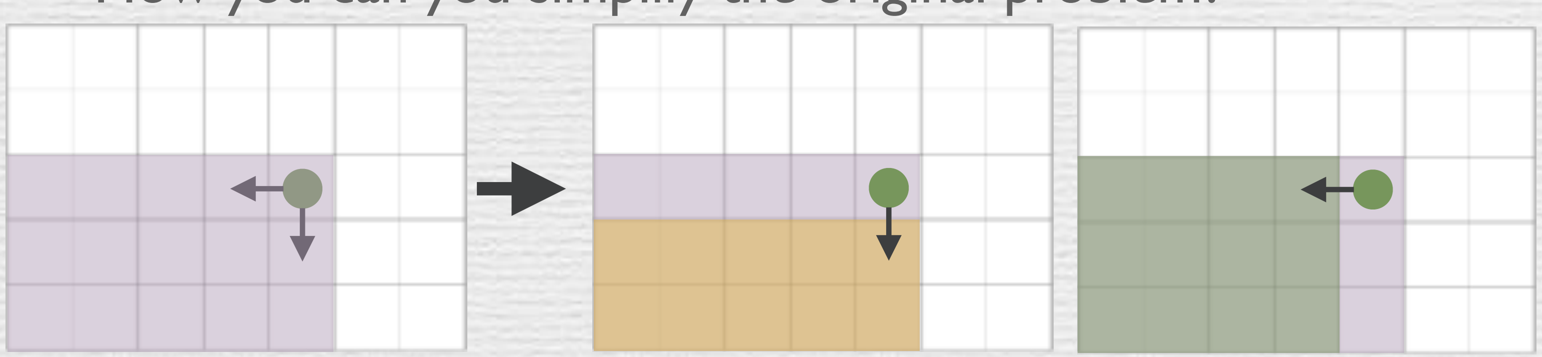
Can only move down



Can only move left

So if N is 1 OR M is 1, we know there is only 1 path

Step 2: Break down your problem into recursive calls. How you can you simplify the original problem?



At each point you can only go left or down.
So we have **paths(N, M - 1)** and **paths(N-1, M)**

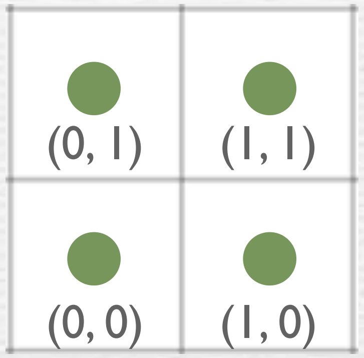
Step 3: Combine the results.
This is almost identical to count_stair_ways. I can get to (0, 0) by going one left (and seeing how many paths there are in the smaller grid) or going down (and seeing how many path there are in that smaller grid). The total number of paths is the sum of those two results.

Consider an insect in an M by N grid. The insect starts at the bottom left corner, (0, 0), and wants to end up at the top right corner (M - 1, N - 1). The insect can only move up and right. Write a function paths that counts the total number of different paths the insect can take from the start to the goal.

Step 0: Understand what is asked. Let's draw a picture!

Note: The problem is ask the different paths from the bottom left to the top right, if we can only move up and right. This is the same s the number of different paths from the top right to the bottom left, if we can only move down and left (just reverse the direction of each of the paths). We are going to be working with the second case, because the input to paths is M and N, making it easier to start the path at position (M - 1, N - 1).

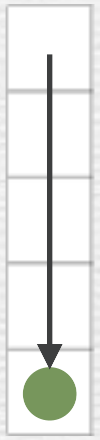
Ex: M = 2, N = 2



Note: the points are in the **MIDDLE** of each square

Step 1: Identify your base case. For what grids do you know for sure how many paths there are? Are there certain grids that "force" a path?

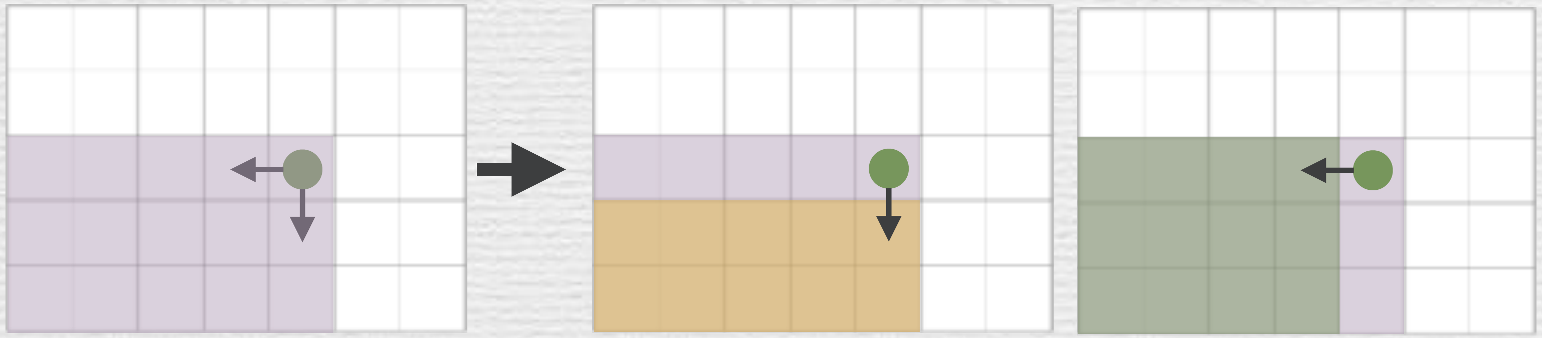
Can only move down



Can only move left

So if N is 1 OR M is 1, we know there is only 1 path

Step 2: Break down your problem into recursive calls. How you can you simplify the original problem?



At each point you can only go left or down.
So we have **paths(N, M - 1)** and **paths(N-1, M)**

Step 3: Combine the results.
This is almost identical to count_stair_ways. I can get to (0, 0) by going one left (and seeing how many paths there are in the smaller grid) or going down (and seeing how many path there are in that smaller grid). The total number of paths is the sum of those two results.

So we get **paths(N, M-1) + paths(N-1, M)**

Putting it all together

Put the orange text from the previous slide into code:

```
def paths(m, n):
    if m == 1 or n == 1:
        return 1
    return paths(m - 1, n) + paths(n, m - 1)
```

Tree Recursion #3

The TAs want to print handouts for their students. However, for some unfathomable reason, both printers are broken; the first printer only prints multiples of n1, and the second printer only prints multiples of n2. Help the TAs figure out whether or not it is possible to print an exact number of handouts!

The TAs want to print handouts for their students. However, for some unfathomable reason, both printers are broken; the first printer only prints multiples of n1, and the second printer only prints multiples of n2. Help the TAs figure out whether or not it is possible to print an exact number of handouts!

The TAs want to print handouts for their students. However, for some unfathomable reason, both printers are broken; the first printer only prints multiples of n1, and the second printer only prints multiples of n2. Help the TAs figure out whether or not it is possible to print an exact number of handouts!

Step 1: Identify the base case. How do you know that you can make total copies? How do you know if you definitely cannot make total copies? What's the *simplest input*?

The TAs want to print handouts for their students. However, for some unfathomable reason, both printers are broken; the first printer only prints multiples of n1, and the second printer only prints multiples of n2. Help the TAs figure out whether or not it is possible to print an exact number of handouts!

Step 1: Identify the base case. How do you know that you can make total copies? How do you know if you definitely cannot make total copies? What's the *simplest input*?

Hint: what if total is n1 or n2?

The TAs want to print handouts for their students. However, for some unfathomable reason, both printers are broken; the first printer only prints multiples of n1, and the second printer only prints multiples of n2. Help the TAs figure out whether or not it is possible to print an exact number of handouts!

Step 1: Identify the base case. How do you know that you can make total copies? How do you know if you definitely cannot make total copies? What's the *simplest input*?

Hint: what if total is n1 or n2?

If the **total is n1 or n2**, I just use 1 multiple of n1 or n2 to attain my goal. So in this case I should return **True**.

The TAs want to print handouts for their students. However, for some unfathomable reason, both printers are broken; the first printer only prints multiples of n1, and the second printer only prints multiples of n2. Help the TAs figure out whether or not it is possible to print an exact number of handouts!

Step 1: Identify the base case. How do you know that you can make total copies? How do you know if you definitely cannot make total copies? What's the *simplest input*?

Hint: what if total is n1 or n2?

If the **total is n1 or n2**, I just use 1 multiple of n1 or n2 to attain my goal. So in this case I should return **True**.

Hint: when are you 100% sure that you cannot use n1 or n2 to make total?

The TAs want to print handouts for their students. However, for some unfathomable reason, both printers are broken; the first printer only prints multiples of n1, and the second printer only prints multiples of n2. Help the TAs figure out whether or not it is possible to print an exact number of handouts!

Step 1: Identify the base case. How do you know that you can make total copies? How do you know if you definitely cannot make total copies? What's the *simplest input*?

Hint: what if total is n1 or n2?

If the **total is n1 or n2**, I just use 1 multiple of n1 or n2 to attain my goal. So in this case I should return **True**.

Hint: when are you 100% sure that you cannot use n1 or n2 to make total?

If the **total is smaller n1 and n2**, there is no way I can use n1 or n2 to print total copies. In this case, I should return **False**.

The TAs want to print handouts for their students. However, for some unfathomable reason, both printers are broken; the first printer only prints multiples of n1, and the second printer only prints multiples of n2. Help the TAs figure out whether or not it is possible to print an exact number of handouts!

Step 1: Identify the base case. How do you know that you can make total copies? How do you know if you definitely cannot make total copies? What's the *simplest input*?

Hint: what if total is n1 or n2?

If the **total is n1 or n2**, I just use 1 multiple of n1 or n2 to attain my goal. So in this case I should return **True**.

Hint: when are you 100% sure that you cannot use n1 or n2 to make total?

If the **total is smaller n1 and n2**, there is no way I can use n1 or n2 to print total copies. In this case, I should return **False**.

Step 2: How should we simplify the problem? How can I make total get closer

The TAs want to print handouts for their students. However, for some unfathomable reason, both printers are broken; the first printer only prints multiples of n1, and the second printer only prints multiples of n2. Help the TAs figure out whether or not it is possible to print an exact number of handouts!

Step 1: Identify the base case. How do you know that you can make total copies? How do you know if you definitely cannot make total copies? What's the *simplest input*?

Hint: what if total is n1 or n2?

If the **total is n1 or n2**, I just use 1 multiple of n1 or n2 to attain my goal. So in this case I should return **True**.

Hint: when are you 100% sure that you cannot use n1 or n2 to make total?

If the **total is smaller n1 and n2**, there is no way I can use n1 or n2 to print total copies. In this case, I should return **False**.

Step 2: How should we simplify the problem? How can I make total get closer

We start from total and try to get to our base case. At each step, we can print n1 copies or n2 copies. Again, we are faced with a recursive case similar to what we have seen.

The TAs want to print handouts for their students. However, for some unfathomable reason, both printers are broken; the first printer only prints multiples of n1, and the second printer only prints multiples of n2. Help the TAs figure out whether or not it is possible to print an exact number of handouts!

Step 1: Identify the base case. How do you know that you can make total copies? How do you know if you definitely cannot make total copies? What's the *simplest input*?

Hint: what if total is n1 or n2?

If the **total is n1 or n2**, I just use 1 multiple of n1 or n2 to attain my goal. So in this case I should return **True**.

Hint: when are you 100% sure that you cannot use n1 or n2 to make total?

If the **total is smaller n1 and n2**, there is no way I can use n1 or n2 to print total copies. In this case, I should return **False**.

Step 2: How should we simplify the problem? How can I make total get closer

We start from total and try to get to our base case. At each step, we can print n1 copies or n2 copies. Again, we are faced with a recursive case similar to what we have seen.

We either see if we can get total to be n1 or n2 by decrementing it by n1 (**has_sum(total - n1, n1 , n2)**) or we see if we can get to n1 or n2 by decrementing by n2 (**has_sum(total - n2, n1, n2)**)

The TAs want to print handouts for their students. However, for some unfathomable reason, both printers are broken; the first printer only prints multiples of n1, and the second printer only prints multiples of n2. Help the TAs figure out whether or not it is possible to print an exact number of handouts!

Step 1: Identify the base case. How do you know that you can make total copies? How do you know if you definitely cannot make total copies? What's the *simplest input*?

Hint: what if total is n1 or n2?

If the **total is n1 or n2**, I just use 1 multiple of n1 or n2 to attain my goal. So in this case I should return **True**.

Hint: when are you 100% sure that you cannot use n1 or n2 to make total?

If the **total is smaller n1 and n2**, there is no way I can use n1 or n2 to print total copies. In this case, I should return **False**.

Step 2: How should we simplify the problem? How can I make total get closer

We start from total and try to get to our base case. At each step, we can print n1 copies or n2 copies. Again, we are faced with a recursive case similar to what we have seen.

We either see if we can get total to be n1 or n2 by decrementing it by n1 (**has_sum(total - n1, n1 , n2)**) or we see if we can get to n1 or n2 by decrementing by n2 (**has_sum(total - n2, n1, n2)**)

Step 3: Combine the results.
We are happy if either of our two recursive calls returns true. So we return true if **has_sum(total - n1, n1 , n2) or has_sum(total - n2, n1 , n2)** returns True.

Putting it all together

Put the orange text from the previous slide into code:

```
def has_sum(total, n1, n2):
   if total == n1 or total == n2:
      return True
   elif total < n1 and total < n2:
      return False
   return has_sum(total - n1, n1, n2) or has_sum(total - n2, n1, n2)
```

Tree Recursion #4

The next day, the printers break down even more! Each time they are used, Printer A prints a random x copies $50 \leq x \leq 60$, and Printer B prints a random y copies $130 \leq y \leq 140$. The TAs also relax their expectations: they are satisfied as long as they get at least lower, but no more than upper, copies printed. (More than upper copies is unacceptable because it wastes too much paper.)

The next day, the printers break down even more! Each time they are used, Printer A prints a random x copies $50 \leq x \leq 60$, and Printer B prints a random y copies $130 \leq y \leq 140$. The TAs also relax their expectations: they are satisfied as long as they get at least lower, but no more than upper, copies printed. (More than upper copies is unacceptable because it wastes too much paper.)

The next day, the printers break down even more! Each time they are used, Printer A prints a random x copies $50 \le x \le 60$, and Printer B prints a random y copies $130 \le y \le 140$. The TAs also relax their expectations: they are satisfied as long as they get at least lower, but no more than upper, copies printed. (More than upper copies is unacceptable because it wastes too much paper.)

We need to keep track of the smallest number of copies our printers could print and the largest number of copies we could end up printing. Since we need to compare these two extremes to lower and upper, we should write a **helper** function (call it **sum_helper**), that can keep track of the **possible_min** and **possible_max**. We can call sum_helper from inside the sum_range definition with both possible_min and possible_max set to **0**. We can always print 0 copies by not using any printer. Then inside sum_range we will use recursion to see if we can find some combination of printers that forces us to be in the range from lower and upper.

The next day, the printers break down even more! Each time they are used, Printer A prints a random x copies $50 \le x \le 60$, and Printer B prints a random y copies $130 \le y \le 140$. The TAs also relax their expectations: they are satisfied as long as they get at least lower, but no more than upper, copies printed. (More than upper copies is unacceptable because it wastes too much paper.)

We need to keep track of the smallest number of copies our printers could print and the largest number of copies we could end up printing. Since we need to compare these two extremes to lower and upper, we should write a **helper** function (call it **sum_helper**), that can keep track of the **possible_min** and **possible_max**. We can call sum_helper from inside the sum_range definition with both possible_min and possible_max set to **0**. We can always print 0 copies by not using any printer. Then inside sum_range we will use recursion to see if we can find some combination of printers that forces us to be in the range from lower and upper.

Step 1: Base case(s)

How do we know we are done? Remember that possible_min and possible_max bound the number of copies we could possibly print. By using a combination of printer A and printer B, we know for sure the number of copies we print is between possible_min and possible_max. How can we be sure that the printers will print more than lower and less than upper? How are **possible_min**, **possible_max** and **lower**, **upper** related?

How do we know that we for sure failed in staying between lower and upper. While possible_min can keep getting bigger to reach lower, when **possible_max is larger than upper** it's game over.

The next day, the printers break down even more! Each time they are used, Printer A prints a random x copies $50 \leq x \leq 60$, and Printer B prints a random y copies $130 \leq y \leq 140$. The TAs also relax their expectations: they are satisfied as long as they get at least lower, but no more than upper, copies printed. (More than upper copies is unacceptable because it wastes too much paper.)

We need to keep track of the smallest number of copies our printers could print and the largest number of copies we could end up printing. Since we need to compare these two extremes to lower and upper, we should write a **helper** function (call it **sum_helper**), that can keep track of the **possible_min** and **possible_max**. We can call sum_helper from inside the sum_range definition with both possible_min and possible_max set to **0**. We can always print 0 copies by not using any printer. Then inside sum_range we will use recursion to see if we can find some combination of printers that forces us to be in the range from lower and upper.

Step 1: Base case(s)

How do we know we are done? Remember that possible_min and possible_max bound the number of copies we could possibly print. By using a combination of printer A and printer B, we know for sure the number of copies we print is between possible_min and possible_max. How can we be sure that the printers will print more than lower and less than upper? How are **possible_min**, **possible_max** and **lower**, **upper** related?
How do we know that we for sure failed in staying between lower and upper. While possible_min can keep getting bigger to reach lower, when **possible_max is larger than upper** it's game over.

Step 2: Recursive call

We're going to be calling our helper function recursively. At each step we can **increment the possible_min by 50 or by 130**, depending on which printer we use. Similarly we can **increment possible_max by 60 or 140.**

The next day, the printers break down even more! Each time they are used, Printer A prints a random x copies 50 ≤ x ≤ 60, and Printer B prints a random y copies 130 ≤ y ≤ 140. The TAs also relax their expectations: they are satisfied as long as they get at least lower, but no more than upper, copies printed. (More than upper copies is unacceptable because it wastes too much paper.)

We need to keep track of the smallest number of copies our printers could print and the largest number of copies we could end up printing. Since we need to compare these two extremes to lower and upper, we should write a **helper** function (call it **sum_helper**), that can keep track of the **possible_min** and **possible_max**. We can call sum_helper from inside the sum_range definition with both possible_min and possible_max set to **0**. We can always print 0 copies by not using any printer. Then inside sum_range we will use recursion to see if we can find some combination of printers that forces us to be in the range from lower and upper.

Step 1: Base case(s)

How do we know we are done? Remember that possible_min and possible_max bound the number of copies we could possibly print. By using a combination of printer A and printer B, we know for sure the number of copies we print is between possible_min and possible_max. How can we be sure that the printers will print more than lower and less than upper? How are **possible_min**, **possible_max** and **lower**, **upper** related?
How do we know that we for sure failed in staying between lower and upper. While possible_min can keep getting bigger to reach lower, when **possible_max is larger than upper** it's game over.

Step 2: Recursive call

We're going to be calling our helper function recursively. At each step we can **increment the possible_min by 50 or by 130**, depending on which printer we use. Similarly we can **increment possible_max by 60 or 140.**

Step 3: Combining the results

Assume that sum_range works. What should we do to the results of our recursive call? We're happy if either using the first printer **OR** the second printer forces us to be inside [lower, upper].

Putting it all together

Put the orange text from the previous slide into code:

```
def sum_range(lower, upper)
  def sum_range(possible_min, possible_max):
     if lower <= possible_min and possible_max <= upper:
        return True
     if upper < possible_min:
        return False
     return sum_range(possible_min + 50, possible_max + 60) or
         sum_range(possible_min + 130, possible_max + 140)
  return sum_range(0, 0)
```