

# Disc 04

Orders of Growth  
Data Abstraction

# Orders of Growth Tips

Carefully examine the provided functions. Try running through them with different kinds of input.

Ask yourself: how many iterations will this function go through based on the size of the input?

For recursion it helps to draw out trees instead of actually trying to follow through what happens in each function call

# Common orders of Growth

Be able to recall an example of each of the common orders of growth. That way when you are faced with a new function you can compare it to what you already know

**$O(1)$**

```
def hello():  
    print("hi")
```

**$O(2^n)$**

```
def fib(x):  
    if x == 0 or x == 1:  
        return x  
    return fib(x - 1) + fib(x - 2)
```

**$O(n)$**

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

**$O(n^2)$**

```
def maze(x, y):  
    for i in range(x):  
        for j in range(y):  
            print(i, j)
```

**$O(\log n)$**

```
def halve(n):  
    if n == 0:  
        return 1  
    else:  
        halve(n//2)
```

# factorial

What is the order of growth of factorial?

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n - 1)
```

# factorial

What is the order of growth of factorial?

```
def factorial(n):
```

```
    if n == 0:
```

```
        return 1
```

```
    return n * factorial(n - 1)
```

Walk through an example

# factorial

What is the order of growth of factorial?

```
def factorial(n):
```

```
    if n == 0:
```

```
        return 1
```

```
    return n * factorial(n - 1)
```

Walk through an example

factorial(4)

# factorial

What is the order of growth of factorial?

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n - 1)
```

Walk through an example

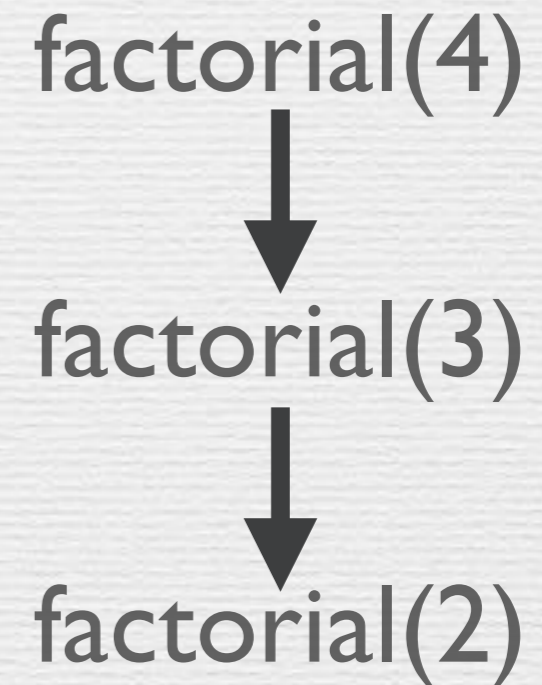
factorial(4)  
↓  
factorial(3)

# factorial

What is the order of growth of factorial?

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n - 1)
```

Walk through an example



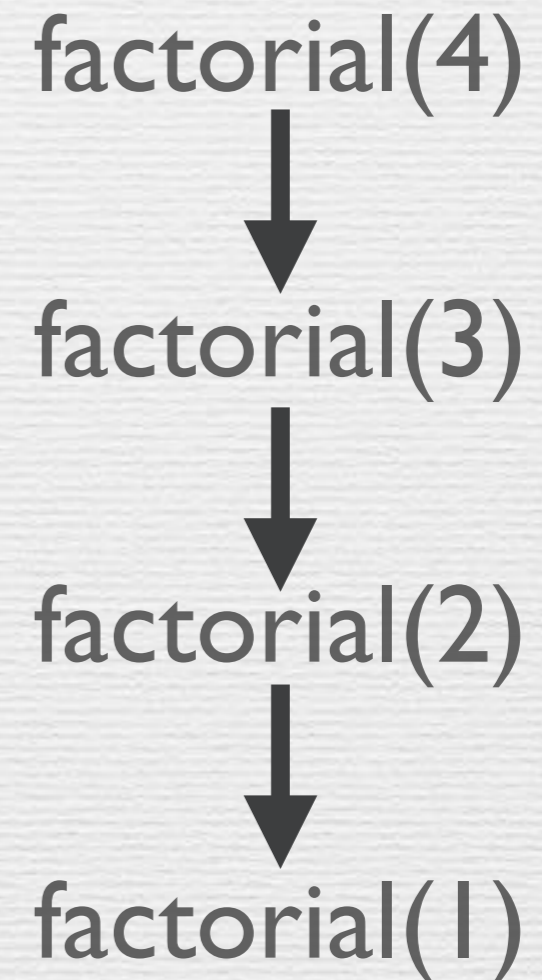


# factorial

What is the order of growth of factorial?

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n - 1)
```

Walk through an example



# factorial

What is the order of growth of factorial?

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n - 1)
```

Walk through an example

factorial(4)



factorial(3)



factorial(2)



factorial(1)



factorial(0)

# factorial

What is the order of growth of factorial?

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n - 1)
```

Notice that we have  $4 + 1$  function calls.  
So the total amount of calls is one more  
than the size of the input.

Walk through an example

factorial(4)



factorial(3)



factorial(2)



factorial(1)



factorial(0)

# factorial

What is the order of growth of factorial?

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n - 1)
```

Notice that we have  $4 + 1$  function calls.  
So the total amount of calls is one more  
than the size of the input.

In each function call we do a constant  
amount of work (multiply two number).

Walk through an example

factorial(4)



factorial(3)



factorial(2)



factorial(1)



factorial(0)

# factorial

What is the order of growth of factorial?

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n - 1)
```

Notice that we have  $4 + 1$  function calls.  
So the total amount of calls is one more  
than the size of the input.

In each function call we do a constant  
amount of work (multiply two number).

Therefore the order of growth is **linear**

Walk through an example

factorial(4)



factorial(3)



factorial(2)



factorial(1)



factorial(0)

# 1.2 #5

```
def bar(n):  
    if n % 2 == 1:  
        return n + 1  
    return n
```

```
def foo(n):  
    if n < 1:  
        return 2  
    if n % 2 == 0:  
        return foo(n - 1) \  
            + foo(n - 2)  
    return 1 + foo(n - 2)
```

What is the order of growth of  $\text{foo}(\text{bar}(n))$ ?

# 1.2 #5

```
def bar(n):  
    if n % 2 == 1:  
        return n + 1  
    return n
```

```
def foo(n):  
    if n < 1:  
        return 2  
    if n % 2 == 0:  
        return foo(n - 1) \  
            + foo(n - 2)  
    return 1 + foo(n - 2)
```

What is the order of growth of  $\text{foo}(\text{bar}(n))$ ?

This problem is a bit tricky!  
Let's draw out the function calls

# 1.2 #5

```
def bar(n):  
    if n % 2 == 1:  
        return n + 1  
    return n
```

```
def foo(n):  
    if n < 1:  
        return 2  
    if n % 2 == 0:  
        return foo(n - 1) \  
            + foo(n - 2)  
    return 1 + foo(n - 2)
```

What is the order of growth of  $\text{foo}(\text{bar}(n))$ ?

This problem is a bit tricky!

Lets draw out the function calls

Before we even go into foo, it is important to know that bar will always return an even number. So our first input to foo will be even.

first we call foo on an  
even number

foo(even)



# 1.2 #5

```
def bar(n):  
    if n % 2 == 1:  
        return n + 1  
    return n
```

```
def foo(n):  
    if n < 1:  
        return 2  
    if n % 2 == 0:  
        return foo(n - 1) \  
            + foo(n - 2)  
    return 1 + foo(n - 2)
```

What is the order of growth of  $\text{foo}(\text{bar}(n))$ ?

This problem is a bit tricky!

Lets draw out the function calls

Before we even go into foo, it is important to know that bar will always return an even number. So our first input to foo will be even.

$\text{foo}(\text{even})$

first we call foo on an even number  
since n is even, we go into the second if clause:  
 $\text{foo}(n - 1) + \text{foo}(n - 2)$ .

# 1.2 #5

```
def bar(n):  
    if n % 2 == 1:  
        return n + 1  
    return n
```

```
def foo(n):  
    if n < 1:  
        return 2  
    if n % 2 == 0:  
        return foo(n - 1) \  
            + foo(n - 2)  
    return 1 + foo(n - 2)
```

What is the order of growth of  $\text{foo}(\text{bar}(n))$ ?

This problem is a bit tricky!

Lets draw out the function calls

Before we even go into foo, it is important to know that bar will always return an even number. So our first input to foo will be even.

Now since n is even,  
subtracting 1 makes  
the value odd

foo(odd)

foo(even)

first we call foo on an  
even number  
since n is even, we go  
into the second if clause:  
 $\text{foo}(n - 1) + \text{foo}(n - 2)$ .

# 1.2 #5

```
def bar(n):  
    if n % 2 == 1:  
        return n + 1  
    return n
```

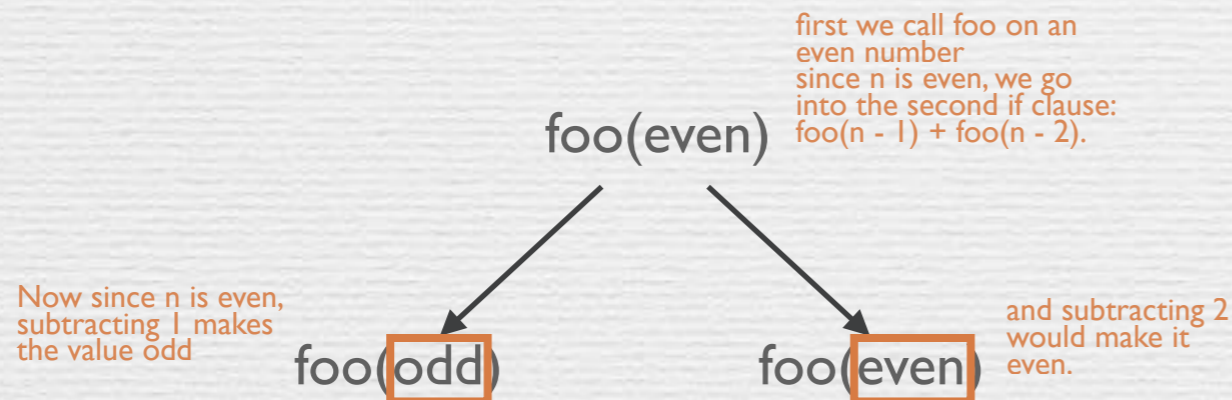
```
def foo(n):  
    if n < 1:  
        return 2  
    if n % 2 == 0:  
        return foo(n - 1) \  
            + foo(n - 2)  
    return 1 + foo(n - 2)
```

What is the order of growth of  $\text{foo}(\text{bar}(n))$ ?

This problem is a bit tricky!

Lets draw out the function calls

Before we even go into foo, it is important to know that bar will always return an even number. So our first input to foo will be even.



# 1.2 #5

```
def bar(n):  
    if n % 2 == 1:  
        return n + 1  
    return n
```

```
def foo(n):  
    if n < 1:  
        return 2  
    if n % 2 == 0:  
        return foo(n - 1) \\  
            + foo(n - 2)  
    return 1 + foo(n - 2)
```

## What is the order of growth of $\text{foo}(\text{bar}(n))$ ?

This problem is a bit tricky!

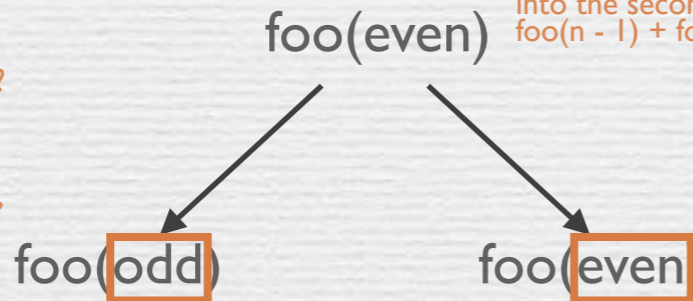
Lets draw out the function calls

Before we even go into foo, it is important to know that bar will always return an even number. So our first input to foo will be even.

! Look at the last if statement !  
Once n becomes odd, we will fall into this case for each recursive call. This looks familiar!! Look back at the examples page, what is the order of growth of foo(odd)?

first we call foo on an even number  
since n is even, we go into the second if clause:  
 $\text{foo}(n - 1) + \text{foo}(n - 2)$ .

Now since n is even, subtracting 1 makes the value odd



and subtracting 2 would make it even.

# 1.2 #5

## What is the order of growth of $\text{foo}(\text{bar}(n))$ ?

```
def bar(n):  
    if n % 2 == 1:  
        return n + 1  
    return n
```

```
def foo(n):  
    if n < 1:  
        return 2  
    if n % 2 == 0:  
        return foo(n - 1) \\  
            + foo(n - 2)  
    return 1 + foo(n - 2)
```

This problem is a bit tricky!

Lets draw out the function calls

Before we even go into foo, it is important to know that bar will always return an even number. So our first input to foo will be even.

! Look at the last if statement !  
Once n becomes odd, we will fall into this case for each recursive call. This looks familiar!! Look back at the examples page, what is the order of growth of foo(odd)?

first we call foo on an even number  
since n is even, we go into the second if clause:  
 $\text{foo}(n - 1) + \text{foo}(n - 2)$ .

Now since n is even, subtracting 1 makes the value odd

and subtracting 2 would make it even.

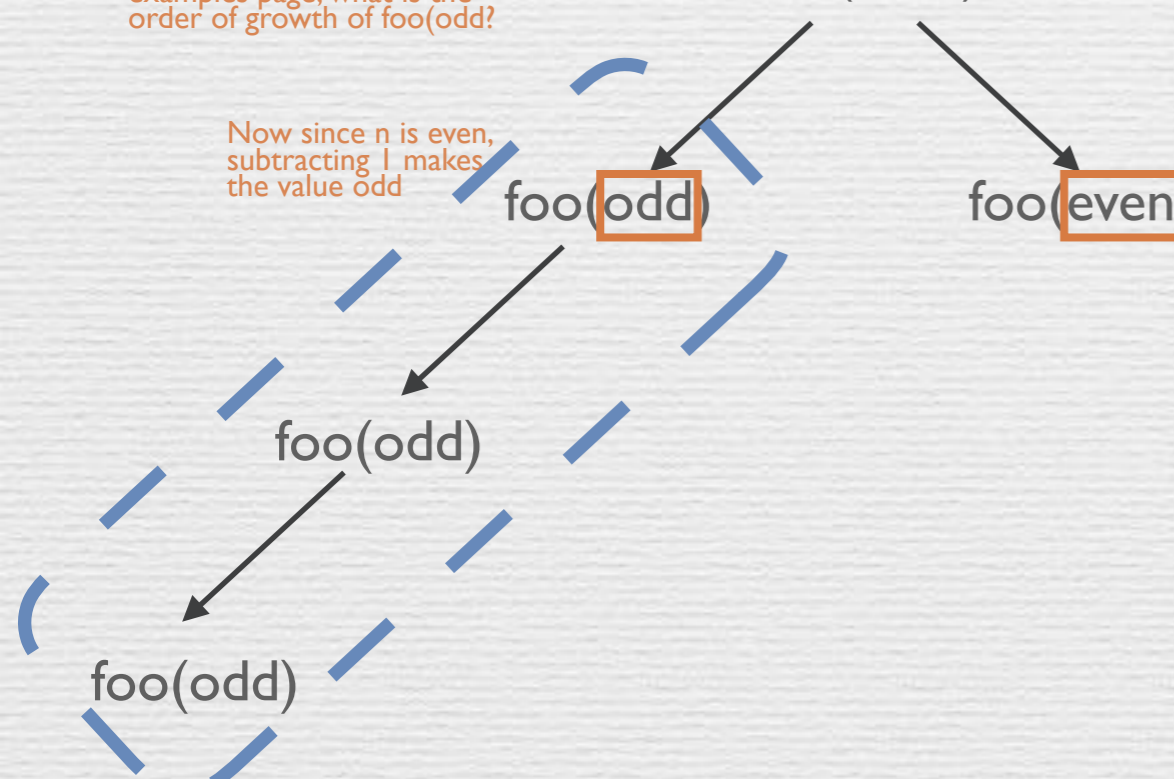
foo(even)

foo(odd)

foo(even)

foo(odd)

foo(odd)



# 1.2 #5

## What is the order of growth of $\text{foo}(\text{bar}(n))$ ?

```
def bar(n):
    if n % 2 == 1:
        return n + 1
    return n
```

```
def foo(n):
    if n < 1:
        return 2
    if n % 2 == 0:
        return foo(n - 1) \
               + foo(n - 2)
    return 1 + foo(n - 2)
```

This problem is a bit tricky!

Lets draw out the function calls

Before we even go into foo, it is important to know that bar will always return an even number. So our first input to foo will be even.

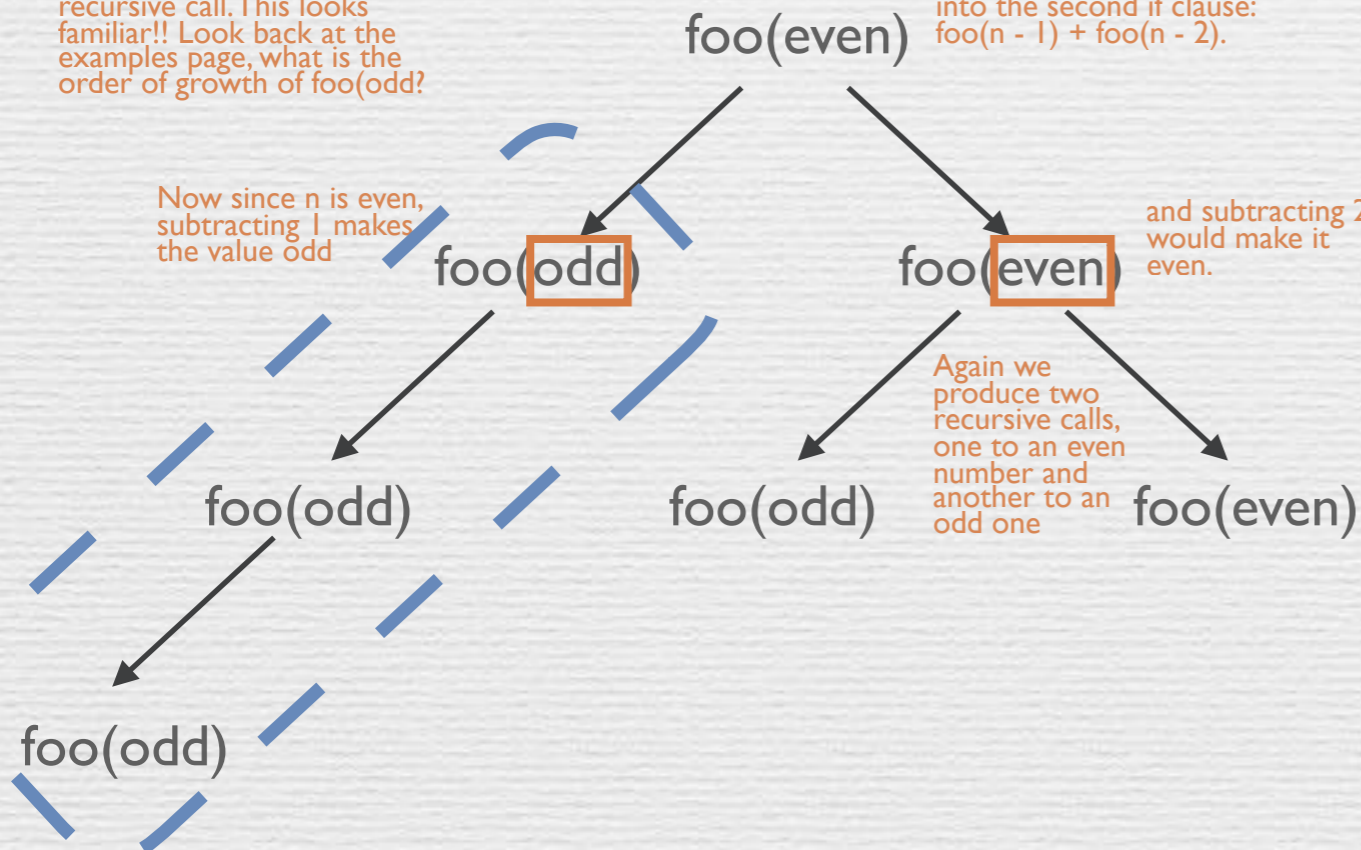
! Look at the last if statement!  
Once n becomes odd, we will fall into this case for each recursive call. This looks familiar!! Look back at the examples page, what is the order of growth of foo(odd)?

first we call foo on an even number since n is even, we go into the second if clause:  $\text{foo}(n - 1) + \text{foo}(n - 2)$ .

Now since n is even, subtracting 1 makes the value odd

and subtracting 2 would make it even.

Again we produce two recursive calls, one to an even number and another to an odd one



# 1.2 #5

## What is the order of growth of $\text{foo}(\text{bar}(n))$ ?

```
def bar(n):  
    if n % 2 == 1:  
        return n + 1  
    return n
```

```
def foo(n):  
    if n < 1:  
        return 2  
    if n % 2 == 0:  
        return foo(n - 1) \  
            + foo(n - 2)  
    return 1 + foo(n - 2)
```

This problem is a bit tricky!

Lets draw out the function calls

Before we even go into foo, it is important to know that bar will always return an even number. So our first input to foo will be even.

! Look at the last if statement !  
Once n becomes odd, we will fall into this case for each recursive call. This looks familiar!! Look back at the examples page, what is the order of growth of foo(odd)?

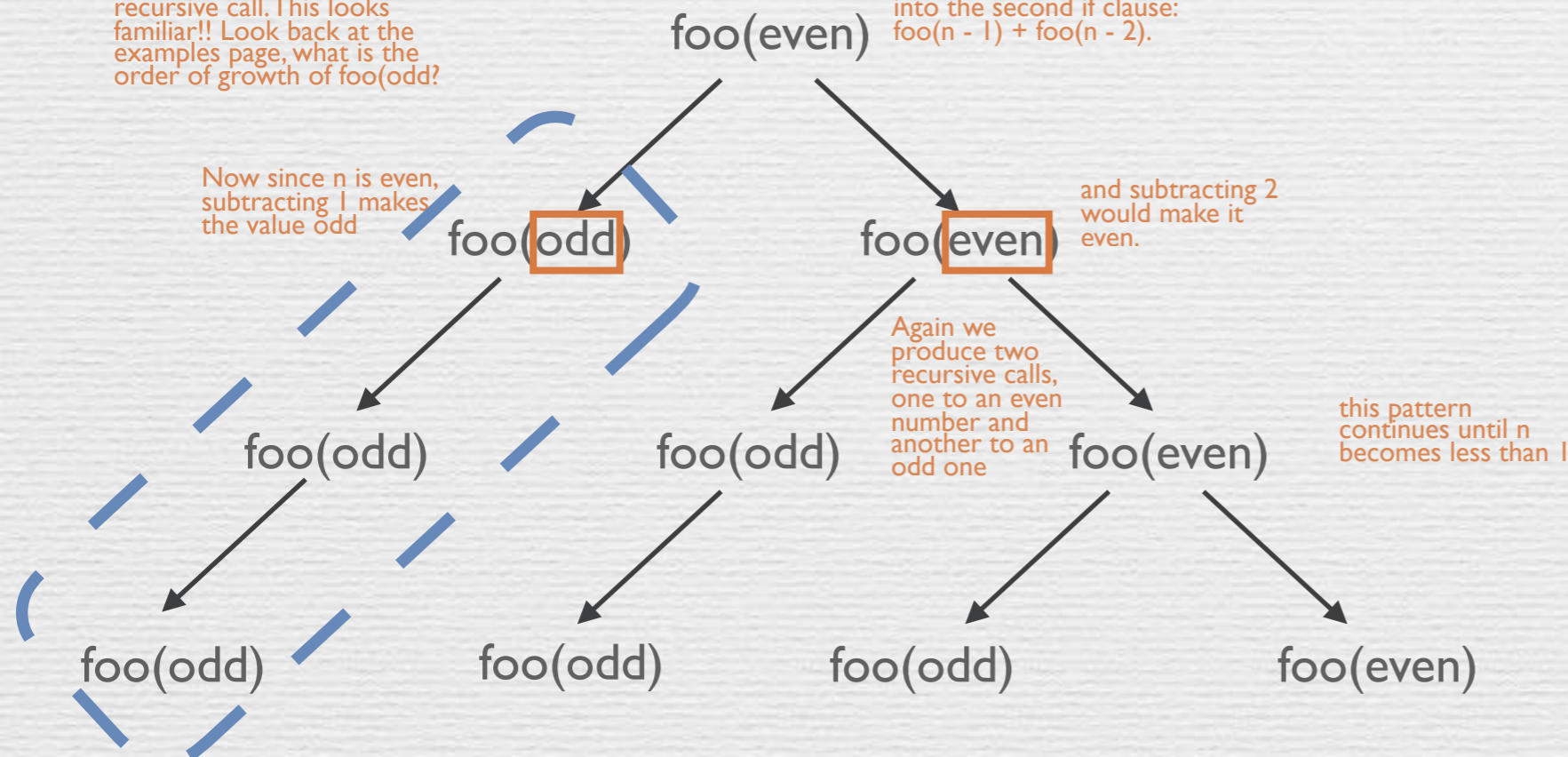
first we call foo on an even number since n is even, we go into the second if clause:  $\text{foo}(n - 1) + \text{foo}(n - 2)$ .

Now since n is even, subtracting 1 makes the value odd

and subtracting 2 would make it even.

Again we produce two recursive calls, one to an even number and another to an odd one

this pattern continues until n becomes less than 1



# 1.2 #5

## What is the order of growth of $\text{foo}(\text{bar}(n))$ ?

```
def bar(n):  
    if n % 2 == 1:  
        return n + 1  
    return n
```

```
def foo(n):  
    if n < 1:  
        return 2  
    if n % 2 == 0:  
        return foo(n - 1) \ + foo(n - 2)  
    return 1 + foo(n - 2)
```

This problem is a bit tricky!

Lets draw out the function calls

Before we even go into foo, it is important to know that bar will always return an even number. So our first input to foo will be even.

! Look at the last if statement !  
Once n becomes odd, we will fall into this case for each recursive call. This looks familiar!! Look back at the examples page, what is the order of growth of foo(odd)?

first we call foo on an even number since n is even, we go into the second if clause:  $\text{foo}(n - 1) + \text{foo}(n - 2)$ .

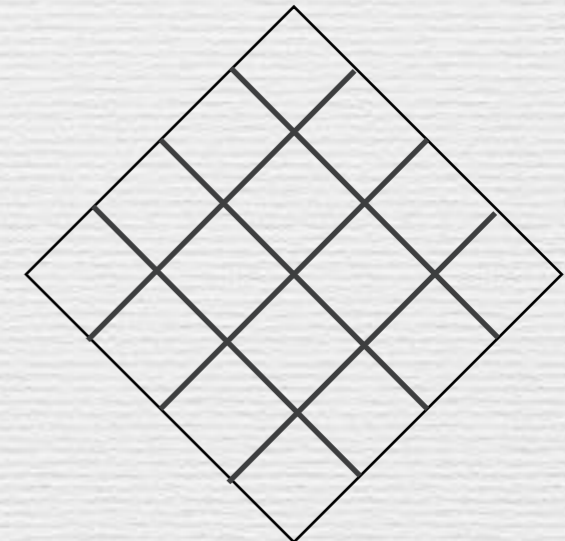
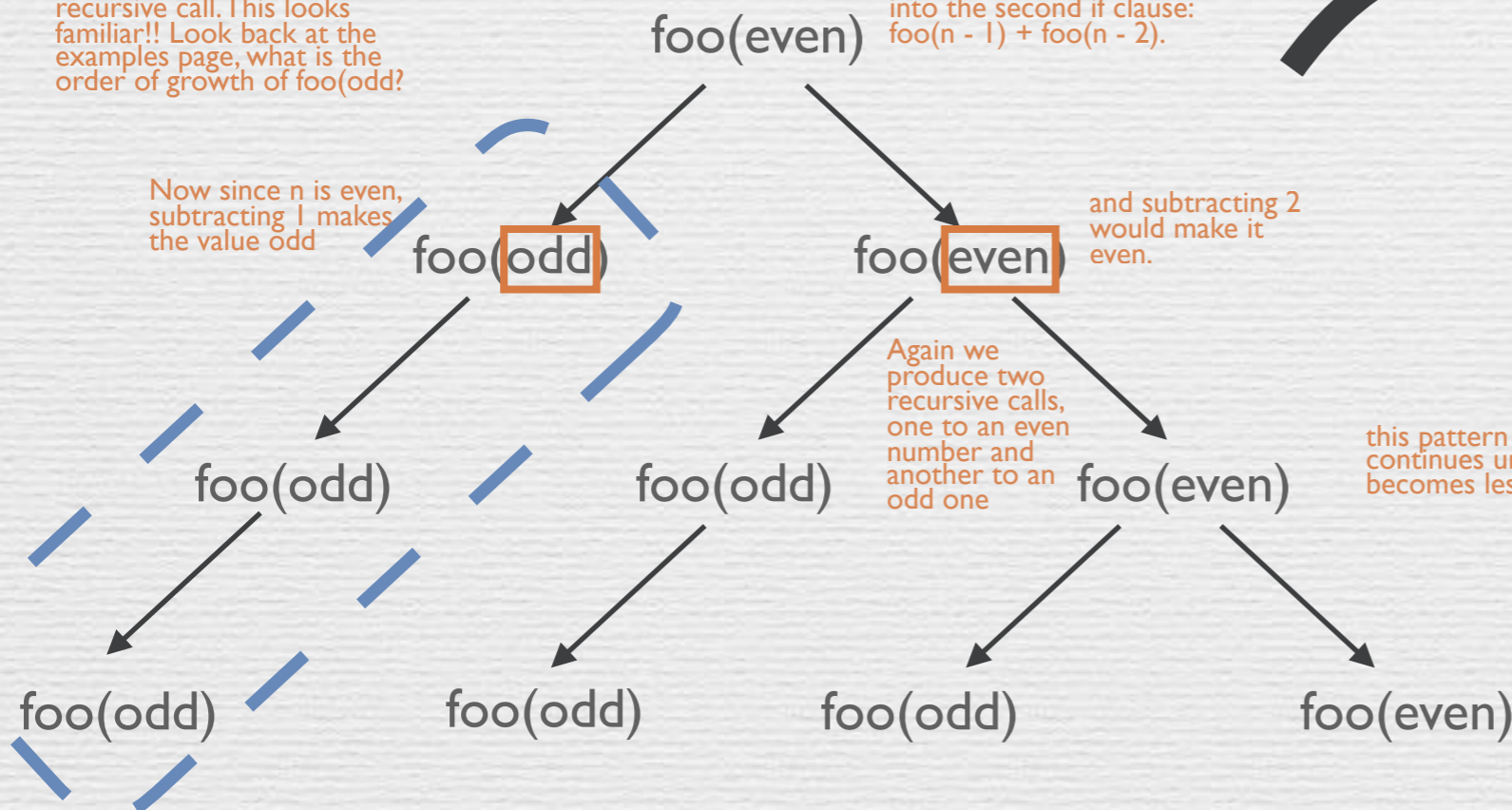
Try to visualize the number of recursive calls as a grid. Each recursive call is represented by the intersection of two lines

Now since n is even, subtracting 1 makes the value odd

and subtracting 2 would make it even.

Again we produce two recursive calls, one to an even number and another to an odd one

this pattern continues until n becomes less than 1





# 1.2 #5

## What is the order of growth of $\text{foo}(\text{bar}(n))$ ?

```
def bar(n):  
    if n % 2 == 1:  
        return n + 1  
    return n
```

```
def foo(n):  
    if n < 1:  
        return 2  
    if n % 2 == 0:  
        return foo(n - 1) \ + foo(n - 2)  
    return 1 + foo(n - 2)
```

This problem is a bit tricky!

Lets draw out the function calls

Before we even go into foo, it is important to know that bar will always return an even number. So our first input to foo will be even.

! Look at the last if statement !  
Once n becomes odd, we will fall into this case for each recursive call. This looks familiar!! Look back at the examples page, what is the order of growth of foo(odd)?

first we call foo on an even number since n is even, we go into the second if clause:  $\text{foo}(n - 1) + \text{foo}(n - 2)$ .

Try to visualize the number of recursive calls as a grid. Each recursive call is represented by the intersection of two lines

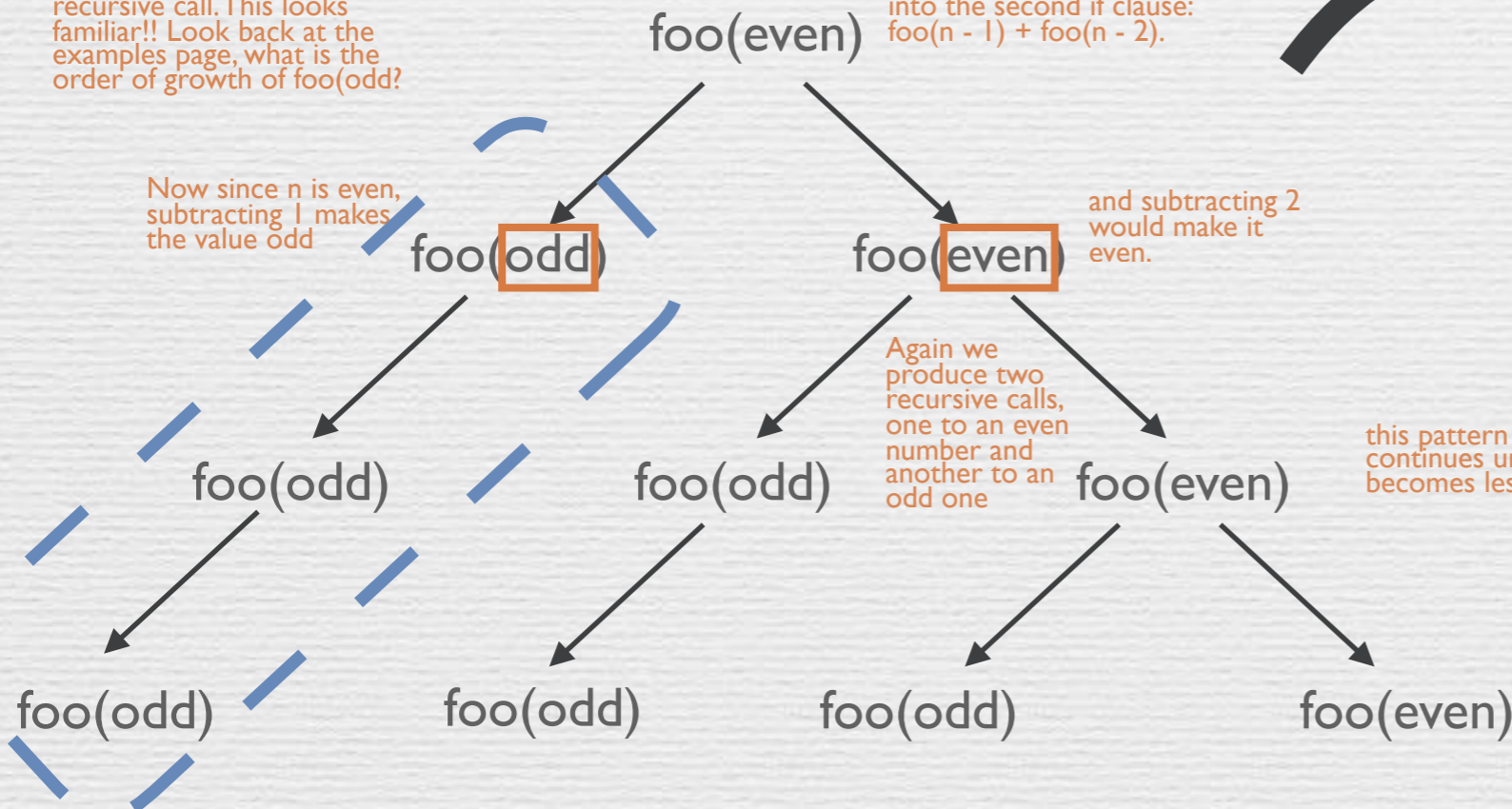
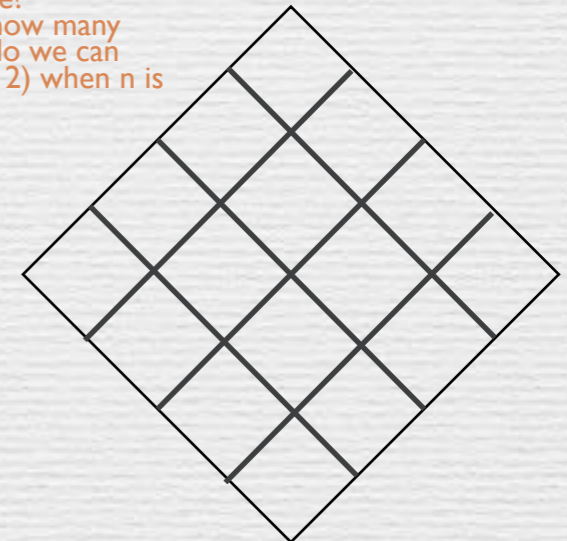
Now since n is even, subtracting 1 makes the value odd

and subtracting 2 would make it even.

How many recursive calls are there on this side?  
(Hint: how many times do we can  $\text{foo}(n - 2)$  when n is odd?)

Again we produce two recursive calls, one to an even number and another to an odd one

this pattern continues until n becomes less than 1



# 1.2 #5

## What is the order of growth of $\text{foo}(\text{bar}(n))$ ?

```
def bar(n):  
    if n % 2 == 1:  
        return n + 1  
    return n
```

```
def foo(n):  
    if n < 1:  
        return 2  
    if n % 2 == 0:  
        return foo(n - 1) \ + foo(n - 2)  
    return 1 + foo(n - 2)
```

This problem is a bit tricky!

Lets draw out the function calls

Before we even go into foo, it is important to know that bar will always return an even number. So our first input to foo will be even.

! Look at the last if statement !  
Once n becomes odd, we will fall into this case for each recursive call. This looks familiar!! Look back at the examples page, what is the order of growth of foo(odd)?

first we call foo on an even number since n is even, we go into the second if clause:  $\text{foo}(n - 1) + \text{foo}(n - 2)$ .

Try to visualize the number of recursive calls as a grid. Each recursive call is represented by the intersection of two lines

Now since n is even, subtracting 1 makes the value odd

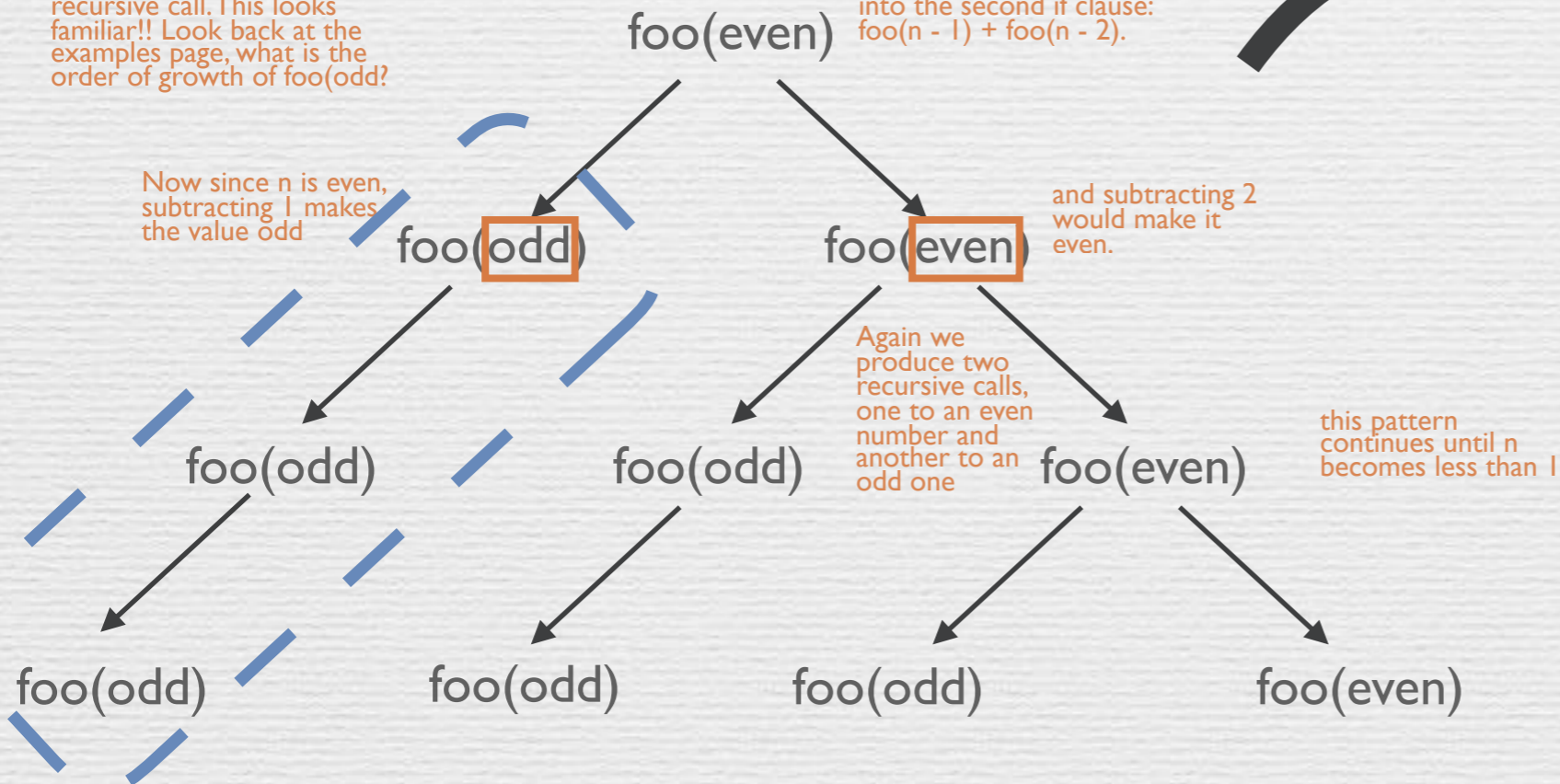
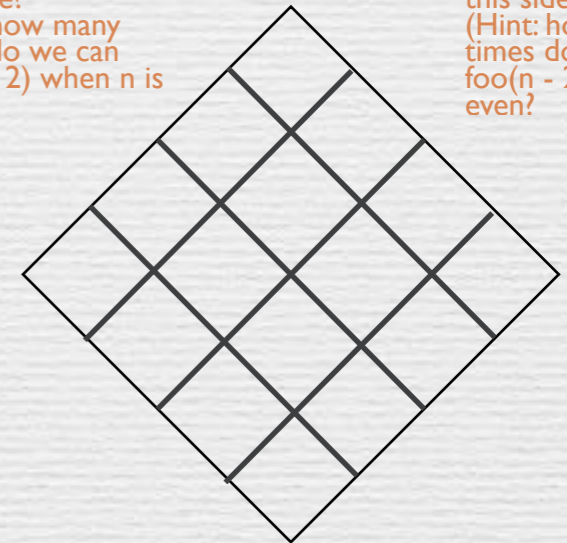
and subtracting 2 would make it even.

How many recursive calls are there on this side?  
(Hint: how many times do we call  $\text{foo}(n - 2)$  when n is odd?)

How many recursive calls are there on this side?  
(Hint: how many times do we call  $\text{foo}(n - 2)$  when n is even?)

Again we produce two recursive calls, one to an even number and another to an odd one

this pattern continues until n becomes less than 1



# 1.2 #5

## What is the order of growth of $\text{foo}(\text{bar}(n))$ ?

```
def bar(n):  
    if n % 2 == 1:  
        return n + 1  
    return n
```

```
def foo(n):  
    if n < 1:  
        return 2  
    if n % 2 == 0:  
        return foo(n - 1) \ + foo(n - 2)  
    return 1 + foo(n - 2)
```

This problem is a bit tricky!

Lets draw out the function calls

Before we even go into foo, it is important to know that bar will always return an even number. So our first input to foo will be even.

! Look at the last if statement !  
Once n becomes odd, we will fall into this case for each recursive call. This looks familiar!! Look back at the examples page, what is the order of growth of foo(odd)?

first we call foo on an even number since n is even, we go into the second if clause:  $\text{foo}(n - 1) + \text{foo}(n - 2)$ .

Try to visualize the number of recursive calls as a grid. Each recursive call is represented by the intersection of two lines

Now since n is even, subtracting 1 makes the value odd

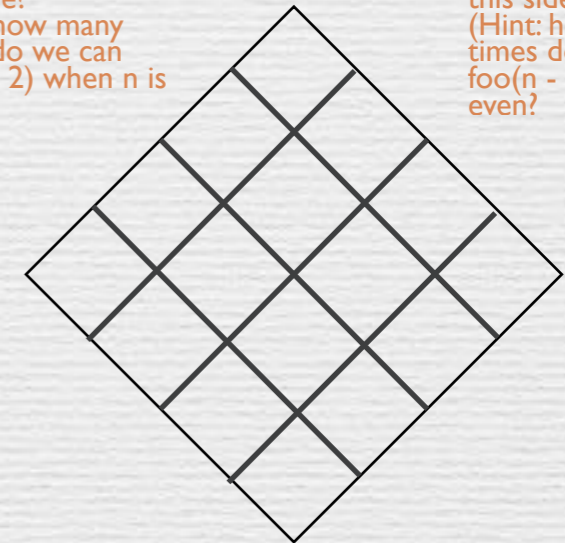
and subtracting 2 would make it even.

How many recursive calls are there on this side?  
(Hint: how many times do we call  $\text{foo}(n - 2)$  when n is odd?

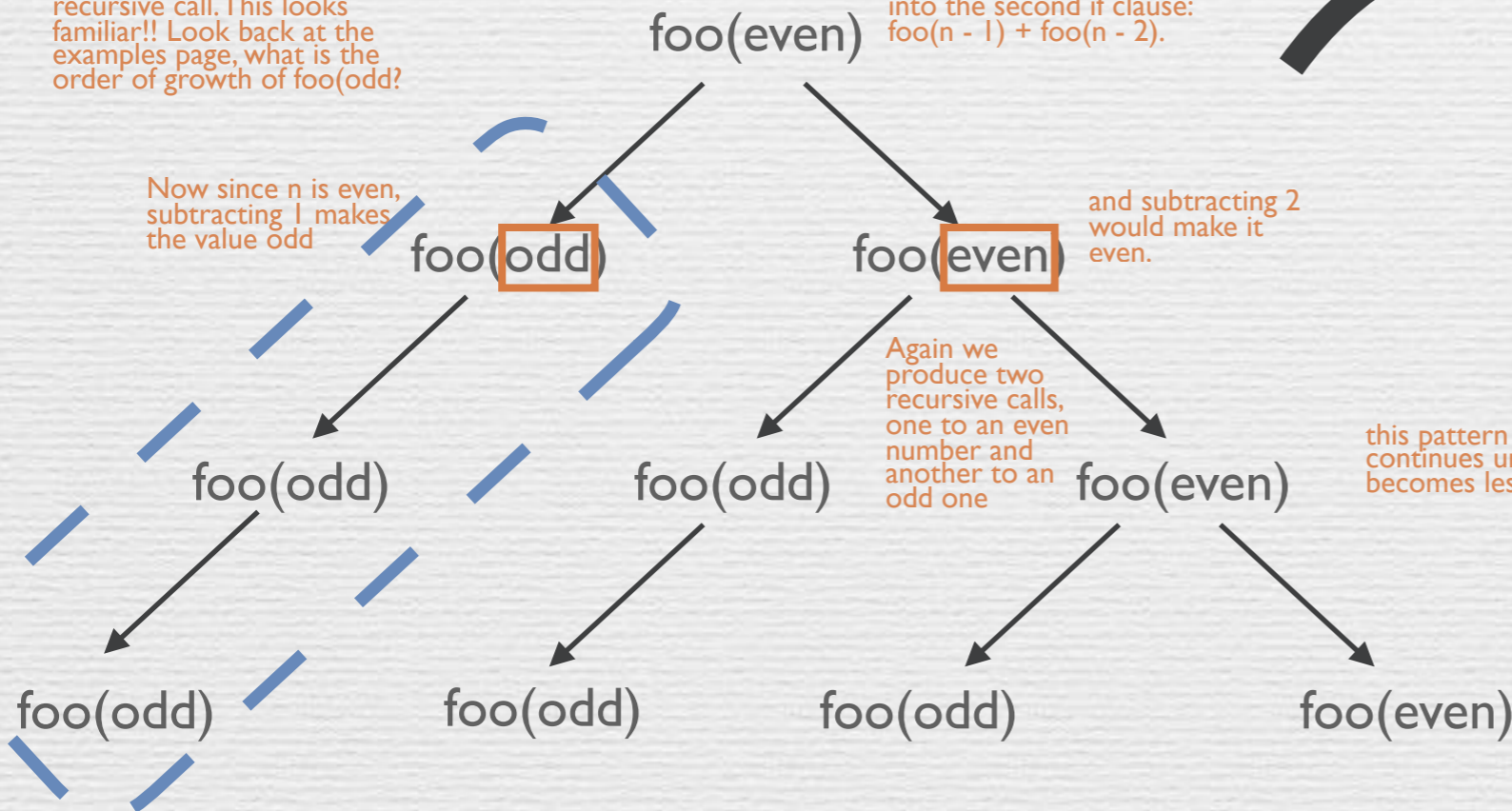
How many recursive calls are there on this side?  
(Hint: how many times do we call  $\text{foo}(n - 2)$  when n is even?

Again we produce two recursive calls, one to an even number and another to an odd one

this pattern continues until n becomes less than 1



On the left since we decrement n by 2 each time, it would be something like  $n / 2$  calls to foo until n becomes less than 1. The same can be said for the right. How many intersection points are there total?



# 1.2 #5

## What is the order of growth of $\text{foo}(\text{bar}(n))$ ?

```
def bar(n):  
    if n % 2 == 1:  
        return n + 1  
    return n
```

```
def foo(n):  
    if n < 1:  
        return 2  
    if n % 2 == 0:  
        return foo(n - 1) \ + foo(n - 2)  
    return 1 + foo(n - 2)
```

This problem is a bit tricky!

Lets draw out the function calls

Before we even go into foo, it is important to know that bar will always return an even number. So our first input to foo will be even.

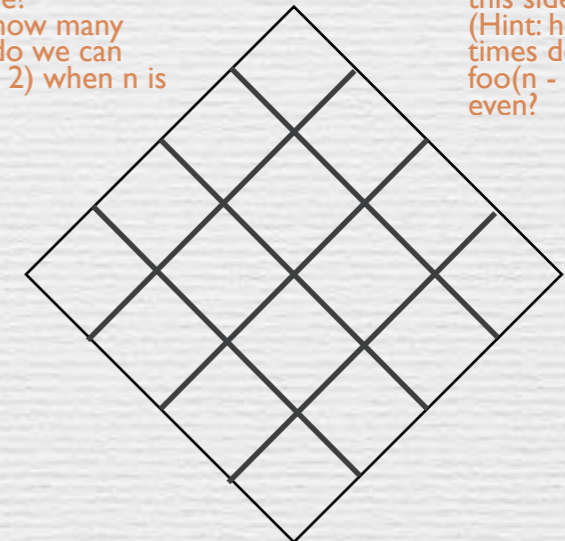
! Look at the last if statement !  
Once n becomes odd, we will fall into this case for each recursive call. This looks familiar!! Look back at the examples page, what is the order of growth of foo(odd)?

first we call foo on an even number since n is even, we go into the second if clause:  $\text{foo}(n - 1) + \text{foo}(n - 2)$ .

Try to visualize the number of recursive calls as a grid. Each recursive call is represented by the intersection of two lines

How many recursive calls are there on this side?  
(Hint: how many times do we call  $\text{foo}(n - 2)$  when n is odd?

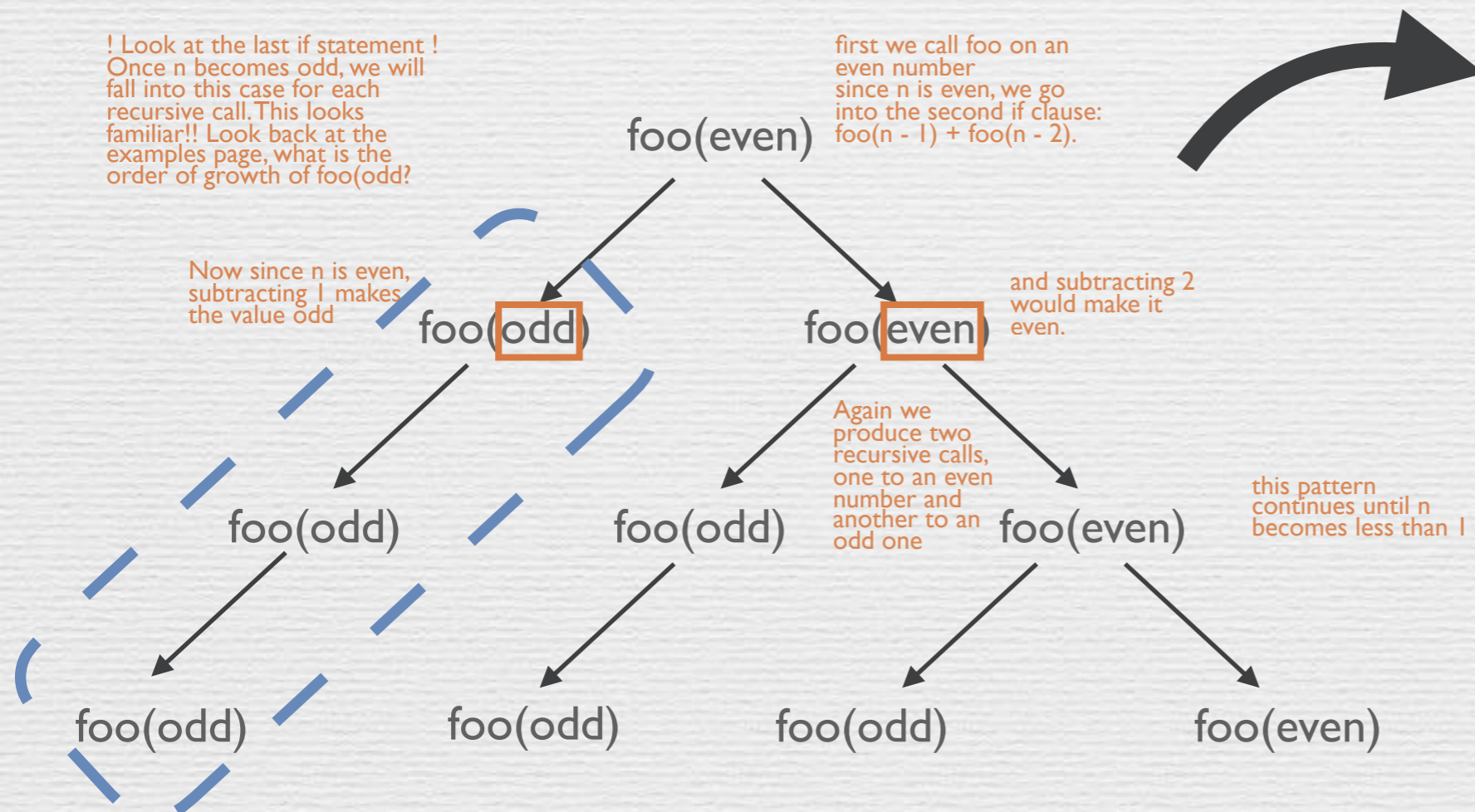
How many recursive calls are there on this side?  
(Hint: how many times do we call  $\text{foo}(n - 2)$  when n is even?



this pattern continues until n becomes less than 1

On the left since we decrement n by 2 each time, it would be something like  $n / 2$  calls to foo until n becomes less than 1. The same can be said for the right. How many intersection points are there total?

$$(n / 2) * (n / 2) = n^2 / 4 = \mathbf{O(n^2)}$$



# Data Abstraction

Don't assume you know how something works.

Abstraction is all around us; to turn on your computer you just press the power button. You don't have to understand what signals are transferred through the wires on the inside of the computer.

The same is with data abstraction. To access the root of a tree, I don't have to know if we used a list, or a string, or anything else to build it. I can just call a function that knows about this internal structure and happily use whatever it returns.