Intro

Good code is two things. It's elegant, and it's fast. In other words, we got a need for speed. We want to find out what's fast, what's slow, and what we can optimize. First, we'll take a tour of what orders of growth really are, and then we'll see a nice 3-step procedure for analyzing the code you write.

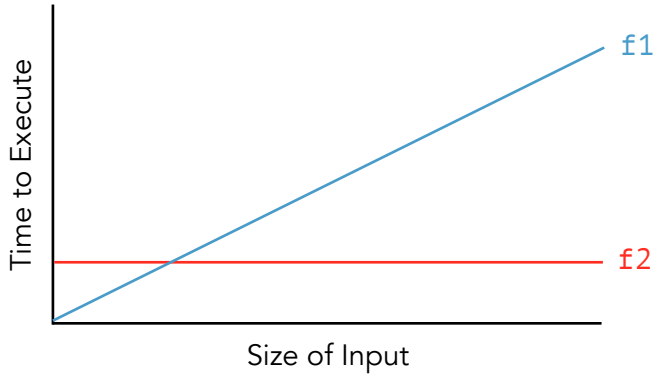
Speed V Growth

There are many ways to solve a problem. Some are faster than others. For example, look at these two definitions of the `identity` function. The one on the left is basically instant, but the one on the right has to count from `0` all the way up to `x`, before it finishes.

```
def identity(x):
    return x
```

```
def identity(x):
    i = 0
    while i < x:
        i += 1
    return i
```

Naturally, situations like this lead us to ask, "Which function takes longer?". Let's pretend we have two functions `f1` and `f2`. It doesn't really matter what they do, but they do the same thing. Here's a graph of how long each one takes to execute, as a function of the size of the arguments we pass in.



On small inputs, `f1` is faster. On big inputs, `f2` is faster. There's no definitive answer! This is the clue that we're asking the *wrong* question.

Instead we need to ask, "Which function grows faster?". Essentially, how do the functions scale as the inputs get bigger? In the example above, `f1` takes longer and longer as the input gets bigger and bigger. `f2`, on the other hand, takes the same amount of time no matter what. `f2` is the winner, because it grows slower than `f1` does. By thinking about growth instead of speed, we can think long-term about the performance of our functions.

The Orders of Growth

There are 5 main orders of growth, each describing how fast a function's runtime grows, as its inputs get bigger. We use the symbol theta to denote an order of growth.

$\Theta(1)$	Constant growth.
$\Theta(\log n)$	Logarithmic growth.
$\Theta(n)$	Linear growth.
$\Theta(n^b)$	Quadratic growth.
$\Theta(b^n)$	Exponential growth.

Θ*(1) : Constant growth*
No matter how big the input gets, a constant function always takes the same amount of time. Here are some examples.

- Arithmetic operators: `+`, `-`, `*`, `/`, `**`, `%`, `//`
- Variable assignment and function declaration
- Boolean operators: `not`, `and`, `or`
- Comparisons: `<`, `>`, `<=`, `>=`, `==`, `!=`
- `print`

Θ*(log n) : Logarithmic growth*
The input is cut to a fraction of its size, over and over again. For example, repeatedly cutting a list in half or dividing a number by some constant. Here's an example, which returns whether `value` occurs in an already ordered list `lst`.

```
def search(lst, value):
    if len(lst) == 1:
        return lst[0] == value
    mid = lst[len(lst) // 2]
    if value < mid:
        return search(lst[:mid], value)
    else:
        return search(lst[mid:], value)
```

Θ*(n) : Linear growth*
It takes an amount of time directly proportional to the input of the function. Usually, this means a single `while` or `for` loop. Here's an example.

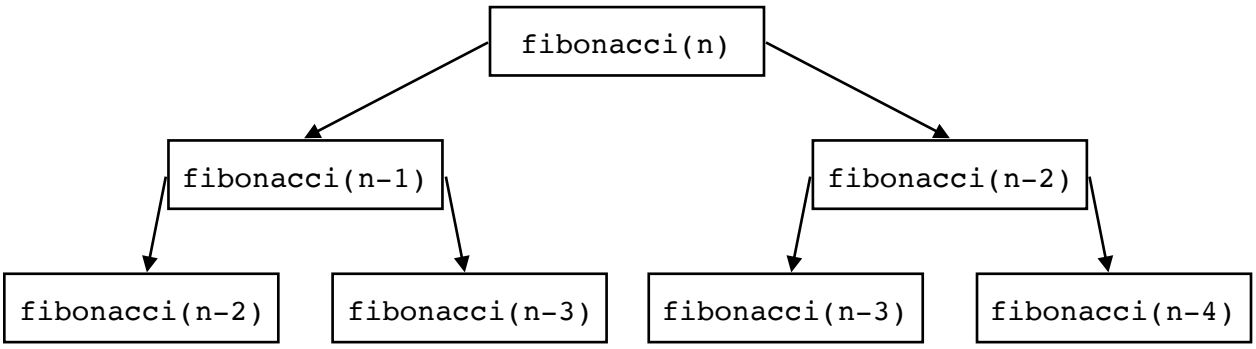
```
def count(n):
    for i in range(n):
        print(i)
```

$\Theta(n^b)$ *: Quadratic growth*
When the size of the input goes up by one, the number of operations goes up by a factor of n. In the following example, if we increment `max_score` from 4 to 5, that increases the total number of operations from 16 to 25. Commonly, we see quadratic growth due to nested `while` or `for` loops.

```
def all_combos(max_score):
    for score_0 in range(max_score):
        for score_1 in range(max_score):
            print(score_0, score_1)
```

$\Theta(b^n)$ *: Exponential Growth*
One function call splits into many more very similar function calls. For the next example, notice there are about 2^n function calls, so when we increment n by 1, we double the number of operations we have to do.

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



Simplifying

Now with a brief into to the different orders of growth, let's revisit the problem of speed versus growth. Recall we care about how a function's runtime grows, as a function of its input's size, and we don't care so much about how long the function actually takes to run. In other words, we want to know the general shape of the graph, not the nitty gritty. Is it linear? Is it curved up? Does it flatten out? To answer questions like that, constant multipliers don't really matter. $\Theta(5n)$ has the same shape as $\Theta(n)$, it's just a bit more squished. With that in mind, here are some simplifications we can make for the purpose of analyzing orders of growth:

- Drop constant multipliers. We only care about the shape of the graph, not the nitty gritty.
- Drop the smaller terms, if it's a sum. Remember, computers are fast. If our input is small, then everything is basically instant so this chapter is meaningless. But when inputs are large, that's

when orders of growth matter. So, we can assume n is really really big. In that way, it's like a limit in calculus. The smaller terms eventually get overshadowed by the bigger ones, so we can leave them out.
- Omit the logarithm bases. There's a snazzy mathematical formula that says you can change the base of a logarithm to whatever you like, as long as you multiply by the right constant multiplier. If we don't care about the constant multipliers, then we shouldn't care about the base of the logarithms either.

Let's see some examples.

$\Theta(5n + 6 \log(n)) = \Theta(n + \log(n)) = \Theta(n)$
$\Theta(\ln(n) + n \log(n)) = \Theta(\log(n) + n \log(n)) = \Theta(n \log(n))$
$\Theta(3^n + 4^n + n^4) = \Theta(4^n)$

One last thing. You'll notice that some functions seem like they could have two different orders of growth. In the next example, the function grows in $\Theta(1)$ if `predicate` is a `True` value, and otherwise it grows in $\Theta(n)$.

```
def example(x, predicate):
    if predicate:
        return True
    else:
        while x > 0:
            x -= 1
        return x
```

To be safe, we go with the bigger order of growth. That way, we get a "worst case" picture of our code. We would say the example above grows in $\Theta(n)$.

A Few Tricky Problems

Tricky Problem 1: $\Theta(1)$ Function Calling
When you call a function inside another one, clearly that affects how long your function is going to take. Most of the time, it also affects how fast your function grows — but not always. In the example below, let's pretend `f` grows in $\Theta(n^2)$. Then, since `g` simply returns a call to `f`, we know that `g` also grows in $\Theta(n^2)$.

```
def g(x):
    return f(x)
```

But now, look at the next example. Every time we call `f`, we call it on the same exact input: `10`. And `f(10)` is always going to take the same amount of time to execute. Let's pretend it takes 5 seconds. That means we've added a constant 5 seconds to the running time of `g`. That

doesn't affect how fast `g` grows. The call to `f` still only adds 5 seconds, no matter how big `x` gets. In the end, it's Θ(1).

```
def g(x):
    return f(10)
```

Tricky Problem 2: Code That Doesn't Run
Take a look at the next example.

```
def example(x):
    if x:
        return True
    if x == True:
        while x > 0:
            x -= 1
        return x
```

The second `if` statement makes it look like Θ(n), but don't be hasty! Notice the `if` conditions. If `x == True`, then we would actually enter the first `if` statement, before seeing the second one. Then the function would terminate at the first `return`, and we never make it to the second `if` at all. In other words, the second `if` statement never gets executed. We can ignore it, so the function actually grows in Θ(1).

Tricky Problem 3: Nested Loops
Like we saw earlier, nested `for` and `while` loops are a good indicator of quadratic growth. However, this isn't always the case.

```
def example(x):
    y = x
    while x > 10:
        while y > 10:
            x -= 1
            y -= 1
    return x
```

Here, the second `while` loop does all the work. By the time it finishes, the first `while` loop's condition is `False`. In fact, we could remove the first `while` loop and it wouldn't even make a difference. This function grows in Θ(n), not $\Theta(n^2)$.

Solving for Orders of Growth

This is the important bit. We're going to talk about how to actually solve orders of growth problems. I will share a method that I find helpful, and which many of my students have found helpful.

The key idea here is abstraction. Woah — that can be a daunting word. Don't worry, I will explain. Instead of analyzing one really big and complicated function, we can break it up into a bunch of smaller processes. Then we just have to add the pieces to get our desired result.

The method is written below. If you don't get it at first, that's fine. We will work through a few examples, and then you can come back to it and check your understanding.

1. Easy stuff first. Go ahead and skip over any `for` or `while` loops, for now. Also don't worry about recursive calls. Right now, we just want the run-time for everything else. Most (but not all) of these lines will be $\Theta(1)$.
2. Iteration. Now take a look at the `for` and `while` loops. We just want to know how many times each of them will run. Multiply this by the sum of the stuff inside the loop. (We'll see examples soon.)
3. Recursion. This is the tricky part. We can usually take care of it by drawing a tree structure though; we'll look at it in more detail later.

Practice: Removing Duplicates

Here's an example from some really old lecture slides. Don't worry about what this code does. For now, let's just apply the method above, so we can see how it works. We want to determine the runtime of `to_set` as a function of `n`, which is the size (specifically, the length) of `an_iterable`.

```
def to_set(an_iterable):
    result = []
    for x in an_iterable:
        L, U = 0, len(results) - 1
        while L <= U:
            M = (L + U) // 2
            if x == result[M]:
                break
            elif x < result[M]:
                U = M - 1
            else:
                L = M + 1
        if L > U:
            result.insert(L, x)
    return result
```

The first step in the method is Easy stuff first. No iteration, no recursion, just the basics for now.

```
def to_set(an_iterable):
    result = []                          Θ(1) since assignment takes constant time.
    for x in an_iterable:                Skip iteration for now.
        L, U = 0, len(results) - 1       Θ(1) for assignment and a call to len.
        while L <= U:                    Skip iteration for now.
            M = (L + U) // 2             Θ(1) since assignment takes constant time.
            if x == result[M]:           Θ(1) for comparison and indexing into a list.
                break                    Θ(1) to exit the while loop.
            elif x < result[M]:          Θ(1) for comparison and indexing into a list.
                U = M - 1                Θ(1) since assignment takes constant time.
            else:                        Θ(1) to enter the else.
                L = M + 1                Θ(1) since assignment takes constant time.
        if L > U:                        Θ(1) for comparison.
            result.insert(L, x)          Θ(1) to insert into a list.
    return result                        Θ(1) to return a value we already know.
```

Now we group these into blocks of code, and we find the order of growth for each block.

```
def to_set(an_iterable):
    result = []                          Θ(1)
    for x in an_iterable:                Skip
        L, U = 0, len(results) - 1       Θ(1)
        while L <= U:                    Skip
            M = (L + U) // 2
            if x == result[M]:
                break
            elif x < result[M]:          Θ(1)
                U = M - 1
            else:
                L = M + 1
        if L > U:                        Θ(1)
            result.insert(L, x)
    return result                        Θ(1)
```

Make sure you understand how we got here, before you continue reading.

Now we look at the Iteration step. How many times does the `while` loop run, at the very most?

- The condition is `L <= U`, where `L` starts at `0` and `U` starts at `n` (the last index of `results`).
- In each iteration of the `while` loop, this gap decreases by at least `1`.
- Thus it takes at most `n` minus `0` iterations, since these were the starting values of `U` and `L`.

```
def to_set(an_iterable):
    result = []                                  Θ(1)
    for x in an_iterable:                        Skip
        L, U = 0, len(results) - 1               Θ(1)
        while L <= U:                            Skip
            M = (L + U) // 2
            if x == result[M]:
                break
            elif x < result[M]:                  Θ(1)
                U = M - 1
            else:
                L = M + 1
        if L > U:
            result.insert(L, x)                  Θ(1)
    return result                                Θ(1)
```

Since the `while` loop runs n times, we are doing this Θ(1) operation n times.

Now we can put the entire `while` loop into the same block. n * Θ(1) = Θ(n).

```
def to_set(an_iterable):
    result = []                                  Θ(1)
    for x in an_iterable:                        Skip
        L, U = 0, len(results) - 1               Θ(1)
        while L <= U:
            M = (L + U) // 2
            if x == result[M]:
                break
            elif x < result[M]:                  Θ(n)
                U = M - 1
            else:
                L = M + 1
        if L > U:
            result.insert(L, x)                  Θ(1)
    return result                                Θ(1)
```

Now that the blocks on the inside of the `for` loop are all at the same indentation level, we can add them together. $\Theta(1) + \Theta(n) + \Theta(1) = \Theta(n)$.

```
def to_set(an_iterable):
    result = []                                  Θ(1)
    for x in an_iterable:                        Skip
        L, U = 0, len(results) - 1
        while L <= U:
            M = (L + U) // 2
            if x == result[M]:
                break
            elif x < result[M]:                  Θ(n)
                U = M - 1
            else:
                L = M + 1
        if L > U:
            result.insert(L, x)
    return result                                Θ(1)
```

We do the same thing for the `for` loop. This is easier, since we don't have to evaluate how long it takes to satisfy a condition. Instead, we just look at the size of whatever we are iterating over. In this case we are iterating over `an_iterable`, which has length `n`, so the `for` loop will execute `n` times. That means we execute the inner $\Theta(n)$ block n times. The total growth for the `for` loop is $n * \Theta(n) = \Theta(n^2)$.

```
def to_set(an_iterable):
    result = []                                  Θ(1)
    for x in an_iterable:
        L, U = 0, len(results) - 1
        while L <= U:
            M = (L + U) // 2
            if x == result[M]:
                break                            Θ(n^2)
            elif x < result[M]:
                U = M - 1
            else:
                L = M + 1
        if L > U:
            result.insert(L, x)
    return result                                Θ(1)
```

Now that the entire inside of the `for` loop is at the same indentation level, we can add together those blocks. In the end, the whole function grows in $\Theta(1) + \Theta(n^2) + \Theta(1) = \Theta(n^2)$.

```
def to_set(an_iterable):
    result = []
    for x in an_iterable:
        L, U = 0, len(results) - 1
        while L <= U:
            M = (L + U) // 2
            if x == result[M]:
                break
            elif x < result[M]:
                U = M - 1
            else:
                L = M + 1
        if L > U:
            result.insert(L, x)
    return result
```

$\Theta(n^2)$

Practice: Foo Bar
This is an example from discussion. We want the runtime of `foo` as a function of `n`.

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

Easy stuff first. We evaluate the runtime for all the lines that don't involve iteration or recursion.

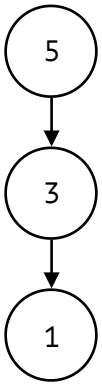
```
def foo(n):
    if n < 2:                       Θ(1) for comparison.
        return 2                    Θ(1) to return a value we already know.
    if n % 2 == 0:                  Θ(1) for comparison and modulo.
        return foo(n-1) + foo(n-2)  Skip recursion for now.
    else:                           Θ(1) to enter the else.
        return 1 + foo(n-2)         Skip for now.
```

Now we group these into blocks of code, and we find the order of growth for each block.

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

The second step is Iteration, but there is none in this problem so we will move on to Recursion. We can approach the recursive calls in any order, but I find it easier to do the simpler ones first. Let's start with the very last line, since this has only one recursive call and the other line has 2.

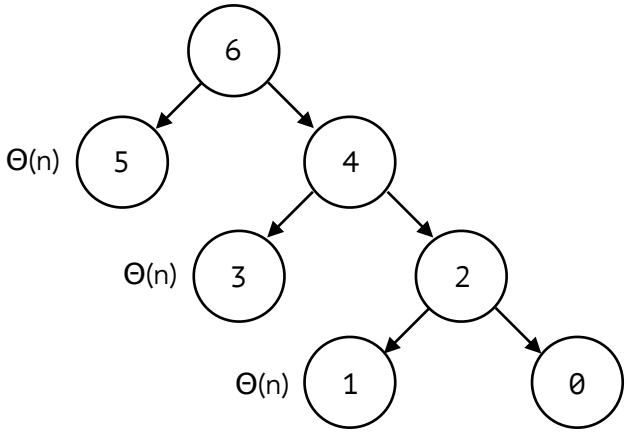
First we need to know when we would ever execute this recursive call. Looking at the `if` conditions, we see that the last line will only execute when `n` is odd and bigger than `1`. Our argument will be `n-2` which is also odd, so it will either terminate in the Θ(1) `if` condition, or it will go to the very last line again. This process repeats a number of times proportional to `n`. It is usually helpful for me to draw the recursive tree:



There will be `(n+1)//2` calls, but this exact value isn't important as long as you see that the number of calls is proportional to the value at the top of the tree, which is `n`. That means the entire `else` case runs in Θ(n) time.

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

Now let's draw the tree for the other recursive call. Note that it only executes when `n` is even, which means `n-1` is odd and `n-2` is even. We have already determined `foo(n-1)` runs in Θ(n) time, so our tree looks like this:



In the other recursive tree, we had a number of Θ(1) calls proportional to `n`. Now we have a number of Θ(n) calls proportional to `n`. Thus the total run-time is n * Θ(n) = Θ(n^2). Again, the exact value is `(n//2)` * Θ(n), but the factor of 1/2 is not important since it's a constant.

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

Θ(1)

Θ(n^2)

Θ(n)

The final growth rate is Θ(1) + Θ(n^2) + Θ(n) = Θ(n^2).

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

Θ(n^2)

Now, suppose we are given a Θ(1) function `bar(n)` that always returns an *odd* number. What is the run-time of `foo(bar(n))`? We just have to figure out what parts of `foo` will actually run. Since `bar(n)` is guaranteed to be odd, we will use the `else` condition, which is Θ(n). That means `foo(bar(n))` runs in Θ(n) time, not Θ(n^2). Try to see why this is intuitive. You may find

it useful to imagine there was a $\Theta(10^{100n})$ block in `foo` — if it was never executed, it wouldn't matter.

Some Last Tips

If you are unsure about a particular line of code, then keep in mind the process of elimination. For example, if something is too fast for $\Theta(n)$ but too slow for $\Theta(1)$, then it can only possibly be $\Theta(\log n)$. Here is the full list that you will encounter in 61A, where b just represents some constant (commonly 2):

$$\Theta(c) < \Theta(\log n) < \Theta(n) < \Theta(n^b) < \Theta(b^n)$$

Also, you will likely come across a confusing recursive tree at some point. There is fortunately a handy formula for calculating the runtime of a function in terms of its recursive tree. It's a bit involved, but don't worry. No problems in 61A will actually require you to use this formula, and there is always a way to solve a problem without it. I include it just in case you ❤️ math, because then it may come in useful here and there.

$$\sum_{\textbf{layers}} (\textbf{work per node}) \times (\textbf{nodes per layer})$$