20200114

There are several complexity classes that we encounter frequently. Here is a table listing the most common ones.

Dominant Term	Big-O class	Description
c (a constant)	O(1)	constant time
$c * \log\ n$	O(log n)	logarithmic time
$c * n$	O(n)	linear time
$c * n * \log n$	O(n * log n)	n log n time
$c * n^2$	O(n^2)	quadratic time or n^2 time
$c * n^3$	O(n^3)	cubic time or n^3 time
$c * n^k$ Where k is a constant	O(n^k)	polynomial time
$c * k^n$ where k is a constant > 1	O(k^n)	exponential time
$c * n!$	O($n!$)	factorial time

Combinations of Functions

If $f_1(n) \in O(g_1(n))$, and $f_2(n) \in O(g_2(n))$ .....

then $f_1(n) + f_2(n) \in O(max(g_1(n), g_2(n)))$

and $f_1(n) * f_2(n) \in O(g_1(n) * g_2(n))$

So far this should all be very familiar. But O classification is just the small first step in the field of computational complexity. There are many other ways of grouping functions together based on the resources (time and/or space) they require. We will consider two more: **Omega** classification and **Theta** classification.

Omega Classification

Big O classification gives us an **upper bound** on the growth-rate of a function (that is, $f(n) \in O(g(n))$ tells us that $f(n)$ grows no faster than $g(n)$ grows), but it doesn't tell us anything about a **lower bound** on the growth-rate of $f(n)$.

Your first reaction to this observation might well be "why would we care about a lower bound on the growth-rate? We use this computational complexity stuff to measure the worst-case running time of an algorithm ... and for worst-case analysis, all we need is an upper bound."

Before we explain why lower-bound analysis is important, we will define exactly what we mean by it and how it works.

Definition: Let $f(n)$ and $g(n)$ be functions. If there exist constants c and n_0 with c > 0 such that

$$f(n) \geq c * g(n) \quad \forall n \geq n_0$$

then $f(n) \in \Omega(g(n))$ (Ω is the Greek letter "Omega")

Note that this is almost exactly the same as the definition of Big O except that the "$\leq c * g(n)$" has become "$\geq c * g(n)$"

As with Big O classification, we can see that $\Omega(g(n))$ is actually a class of functions, all of which grow **at least** as fast as $g(n)$ grows. We can also see that there is a hierarchy of Omega classes, just as there is a hierarchy of Big O classes. For example, suppose $f(n) \in \Omega(n^3)$. This means "growth-rate of $f(n)$" $\geq$ "growth-rate of n^3". But since "growth-rate of n^3" $\geq$ "growth rate of n^2", we can conclude that "growth rate of $f(n)$" $\geq$ "growth rate of n^2", which is equivalent to saying that $f(n) \in \Omega(n^2)$.

In fact, if $f(n) \in \Omega(n^k)$, then $f(n) \in \Omega(n^i) \quad \forall i < k$.

(Note the parallel to Big O: if $f(n) \in O(n^k)$, then $f(n) \in O(n^i) \quad \forall i > k$)

When determining the Big O classification for $f(n)$ we try to find the **smallest** function $g(n)$ such that $f(n) \in O(g(n))$. Conversely, when determining the Ω classification for $f(n)$ we try to find the **largest** function $g(n)$ such that $f(n) \in \Omega(g(n))$.

In class we did a couple of examples. Here's another:

Let $f(n) = 0.0001 * n^2 + (10^6) * n + 3$

We know that $f(n) \in O(n^2)$. It's also very easy to see that $f(n) \in \Omega(n^2)$ ... we can let c = 0.0001 and it is immediately clear that $f(n) \geq c * n^2 \quad \forall n \geq 0$.

Now is it possible that $f(n) \in \Omega(n^3)$?

If this were the case, then there would exist a positive constant c such that

$$f(n) \geq c * n^3 \quad \forall n \geq n_0$$

i.e.

$$0.0001 * n^2 + (10^6) * n + 3 \geq c * n^3$$

$$3 \geq n * (c * n^2 - 0.0001 * n - 10^6)$$

but we can easily see that this is impossible: even if c is very small, as n gets large there will come a point beyond which $c * n^2 - 0.0001 * n - 10^6$ is ≥ 1 so $n * (c * n^2 - 0.0001 * n - 10^6) \geq n$, which would give $3 \geq n \quad \forall n \geq n_0$ ... which is not possible.

Thus $f(n) \notin \Omega(n^3)$

This example illustrates a useful fact: if $f(n)$ is a polynomial, then the Big O class and the Ω class for $f(n)$ are identical.

But this is not always the case. For example, consider this function:

```
A(n):
    if n % 2 == 0:
       for i = 1..n^2:
          print '*'
    else:
       for i = 1..n:
          print '*'
```

Let $T_A(n)$ be the time required to execute A(n). If you plot $T_A(n)$ for n = 1, 2, 3, ... you will see that it has a zig-zag shape. The tops of the zigs occur when n is even, and they grow at the same rate as n^2. It is easy to see that $T_A(n) \in O(n^2)$. However, the bottoms of the zags, which occur when n is odd, do not show this behaviour - they grow at the same rate as n.

Referring back to our previous definitions, we are now able to say that $T_A(n) \in O(n^2)$ and also $T_A(n) \in \Omega(n)$ ... and neither of these can be improved: there is no lower O class for $T_A(n)$, and no higher Ω class for $T_A(n)$.

This example demonstrates that an algorithm's Big O class may be different from its Ω class.

If it turns out that we can show an algorithm's complexity is in $O(g(n))$ **and** in $\Omega(g(n))$, then we get very excited - it means that $g(n)$ gives both an upper and a lower bound on the growth-rate of the time required by the algorithm. Basically it means we know exactly how fast the algorithm's time requirement grows. This is so amazingly wonderful that we give it a special name:

Theta Classification

If $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$, we say $f(n) \in \Theta(g(n))$.