Fantastic Big-O Time Complexities

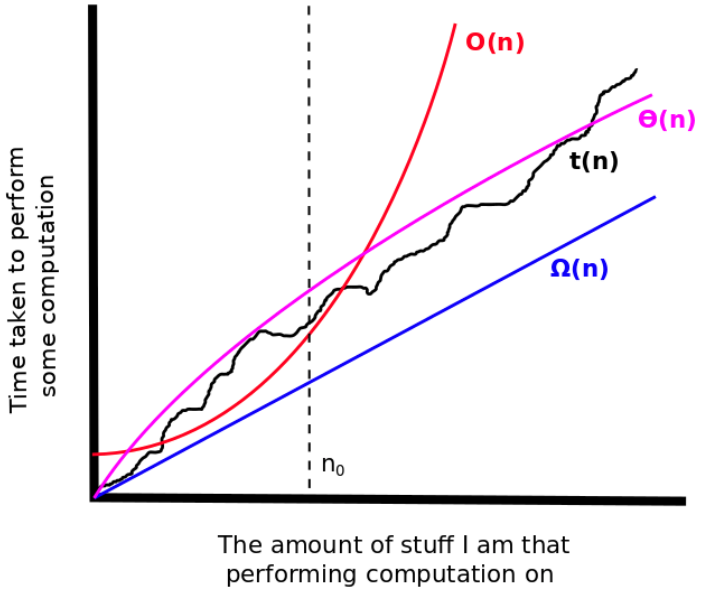
and how to find them

What are time complexities?

Time complexities are a way to indicate how you can expect a program's execution time to increase with an increase in the input size. This is because, naturally if a function performs computation on more data, it can be expected to take more time - and we want to find out how much more this time is. At a high level, it helps us draw insights about how *scalable* an algorithm is.

It doesn't tell you how fast or how slow an algorithm or a piece of code will run, but instead about how it's runtime changes with respect to its input size.

In general, the input size of a program is represented as **n**, and the time complexity is represented as a function of this input size **T(n)**. Sometimes, *T(n)* can be a really



complex function, so we typically like to work with simpler notations - where each notation says different things about the actual T(n).

Refer to the figure on page 1. If we take a piece of code and keep feeding it inputs of different sizes and plotting the amount of times it takes for that piece of code to complete execution, we'll end up graphing a squigly function like *T(n)* (in black) who's mathematical expression is quite complex. However, we can draw other more uniform and simpler expressions to express lines (in red, green and blue) on top of this *T(n)* that are much easier to work with. Each of these lines convey different things about T(n):

1. **Omega - $\Omega(n)$ -** (in blue) represents the *lower bound* for *T(n).*
2. **Theta - $\Theta(n)$ -** (in green) the *best fit* (and still relatively easily expressible) function for *T(n).*
3. **Big-O - $O(n)$ -** (in red) defines the *upper bound* for *T(n)*.

All these definitions are *asymptotic.* This means that they are supposed to work for really large values of *n.* For now, let's just care about the **Big-O** time complexities, and see how we can work with them.

Big-O Time Complexity reduction rules.

Since Big-O time complexities only deal with defining the *upper bound*, the first thing to remember while figuring them out for a given piece of code is assuming the absolute *worst case* execution. What this means is that, for example, if there is an *if* statement in your code that offers two branches of execution, you should assume that your input will always take the longer and more computationally intensive branch.

Now say, following this approach, the *T(n)* of some function comes out to -

$$T(n) = k_1.7^{(n+4)} + n^2 + k_2$$

At this point, we can utilize the fact that big-O notations are **asymptotic**, or that they hold true for very very large values of n.

1. ### Step 1: Get rid of all the less significant terms.

   In the equation above, T(n) is made up of three terms. The first step to reducing T(n) to our big-O notation is getting rid of all the less significant terms - or conversely, **keeping only the most significant term**. How do you decide if a term is the most significant?

Simple - if you can think of a number n_0 such that for all values of $n > n_0$ the term is larger than all the other terms, it is the most significant term.

This may sound more complicated than it is, and you don't really need to figure out the n_0 value every time you figure out a big-O complexity. Here's a handy chart to remember which terms matter more as compared to the others:

← Are more significant							
Term	$n!$	K^n	n^k	$n\ (\log n)$	n	$\log n$	1
n_0	Can be mathematically proven	K log K	Any really big value	e	1	e	-
n_0 value for which term is more significant than all terms below it →							

*e = whatever is the base for *log*. eg. 10 for $\log_{10}(n)$

All you really need to remember here is their order of precedence. Do note that this order of precedence holds even when comparing, say K^n with **1,000,000*n**, because we can just adjust our n_0 value accordingly.

Following these steps, we'll be left with:

$$T(n) = k_1.7^{(n+4)}$$

2. Step 2: Getting rid of all the constant multipliers

This is a side effect of something that we observed towards the end of step 1 - which is that constant multipliers don't really matter. So let's get rid of them!
In our current expression for T(n), we can get rid of k_1. We can also reduce a little further, because $7^{(n+4)}$ can be re-written as $2401*7^{(n)}$, and therefore be further reduced to $7^{(n)}$. Hence, our final big-O time complexity is:

$$O(n) = 7^{(n)}$$

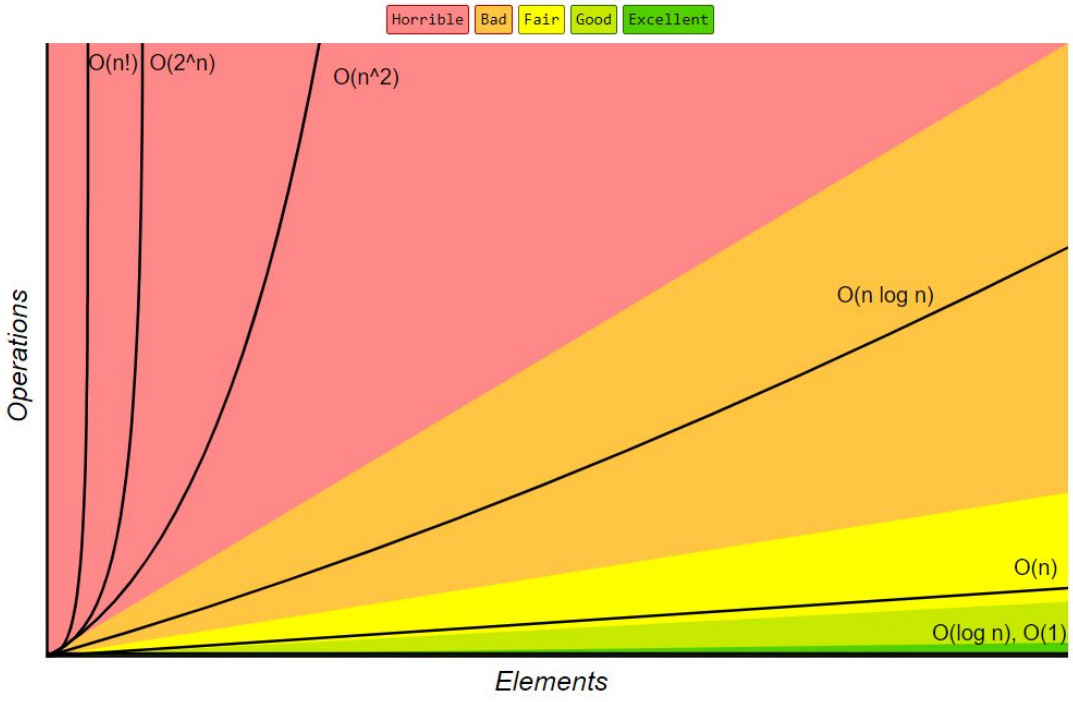
There can only be so many.

One immediate side-effect of the asymptotic nature and the fact that Big-O time complexities define the *upper limit* for time complexities is that there can only be so many kinds of *single term* time complexities you need to worry about (at least for this course). They are listed below:

1. $n!$ - or *factorial time*
2. K^n - or *exponential time*
3. N^k - or *polynomial time*
4. $n (\log n)$
5. N - or *linear time*
6. $\log n$ - or *logarithmic time*
7. 1 - or *constant time*
8. All combinations of the terms above - i.e in product form (f(n)*g(n)) or as a composition (f(g(n)))

The Good, The Bad, and The Ugly.

Some of these time-complexities are acceptable if we want to tractably solve a problem. Others are not. Let's try to graph each of these time complexities to see how they change with an increase in *n*. This should help us do two things: (1) know which time complexities to avoid while writing an algorithm, and (2) help us identify a time complexity for a given piece of code.



(figure from http://bigocheatsheet.com/)

What's what?

Here are some pointers that might help you find out which function/algorithm has what complexity

1. Time Complexities

Time complexities essentially estimate how the compute time of an algorithm can increase given an input. Given a piece of code, we can generalize the answer to this question as follows:

Total time taken = (Amount of time it takes to run once by the function) x (Number of times the function is called)

This is a relatively simple and straight forward approach that works most of the time. Let's take for example the code given below, which computes the following series :

$$1/(n)! + 1/(n/2)! + 1/(n/4)! + \ldots + 1/(4)! + 1/(2)! + 1/(1)!$$

```
def func ( n ):
     if n == 1 :
          return 1
     else :
          return 1 / fact ( n ) + func ( n // 2 )
```

Let's assume the recursive definition for fact() is already given, and hase the time and space complexity of O(n).

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

Amount of time it takes to run once =

For a single call, we notice that func() calls the function fact(n). We know this would take O(n) amount of time (from fact()'s time complexity), plus three operations for performing division and addition in the final return statement and evaluating the *if* condition. It therefore takes the time **n+3** for a given call of func(n).

Amount of times it calls itself =

func(n) calls func(n/2) till n=1. Can we estimate how many times it calls itself for any given value of n?

Let's start plugging in values. For n=4, func will call itself twice. For n=8, thrice. For n=16, four times and so on. Notice that everytime we *double* the value of n, the number of calls doesn't *double*, but just increases linearly. This is a signature behaviour for *logarithmic complexities*. With log(n) complexities, you can typically expect to see *diminishing returns*, i.e. the more you increase *n*, the lesser the *increase* in computation becomes. This becomes clear if you try to plot a log(n) vs n graph - the increase is much slower for higher values of n. Hence, here the number of calls can be expressed as **= log(n)**

Here, since the amount of computation that happens in a single call depends on the stage of computation at which we are, our expression for the total complexity is not as simple as multiplication. Let's write down the amount of computation done in each of the **log(n)** function calls and add them up:

= (n+3) + (n/2 + 3) + (n/4 + 3) + ... + (4 + 3) + (2+ 3) + (1 + 3) [log(n) terms]
= (n) + (n/2) + (n/4) + ... (2) + (1) + log(n)*(3)
= $n(1 - (1/2)^{\log(n)})/(1/2)$ + log(n)*3 [from the GP sum formula]
= 2(n - 1) + 3*log(n)
= O(n) [After reduction]

$$\mathcal{O}(n) = n$$

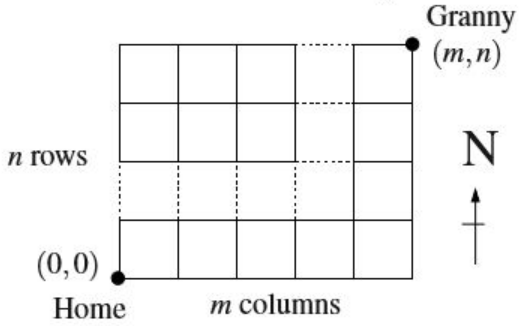
Note: This is very different from simply multiplying *(number of calls)x (operations per call)* - which would have yielded **n.log(n)**. Be careful about this!

Sometimes, the code is not as clear as this. It may provide too little or too much information for us to calculate a clean time complexity. In such cases, it is helpful to take a step back and analyze the algorithm or the approach to solving the problem, rather than the code itself.

For example, let's look at **CS1010fc-midterms-apr14**, question 2B.

Question 2: The Riding Hood [24 marks]

In this problem, we model the problem of the Little Red Riding Hood who wants to visit her sick Granny. In order to get from her home to Granny's place, Riding Hood has to walk through a magical grid-like forest. We model this forest with a $m \times n$ grid as shown in the following figure.



The forest is magical because in each step, Riding Hood can either walk north or east (towards Granny's place) at each intersection. She cannot walk backwards.

A. Fill in the code for the function `paths_granny` that computes the number of distinct paths that Riding Hood can take from her home (at the point $(0,0)$) to reach Granny's place (at the point (m,n)). [6 marks]

B. Suppose $m = n$, what is the time complexity (order of growth) for `paths_granny` in terms of n? [2 marks]

Here, we don't even need to look at the code to estimate what time complexity we can expect!

Given that home is at (0,0) and granny is at (n,n), and the fact that we can't move diagonally, we can tell that all the paths will be 2n boxes long. Say we start constructing different paths starting with some reference path of length 2n. At each box along this path, we will have 2 options - the option of going up, or to the right. Following this logic, we can say that we can construct 2^{2n} unique paths, or that the function will have to explore 2^{2n} unique paths - resulting in 2^{2n} function calls. As long as the function doesn't perform any variable number of operations inside itself, we can expect the time complexity to be $O(2^{2n})$. And that, in fact, is the correct answer!

2. Space Complexities

We can follow a similar approach to solving space complexities. When the code isn't to complicated, all you need to do is

---

Total space taken = space taken by a single function call x longest chain of function calls

---

This formula is similar to what we saw for time complexities, except we only care about the *'longest chain'* of recursive calls instead of the number of recursive calls in total. This is because if we think about space used by recursive calls, the only thing we use *variable* amount of space for are all our *pending/deferred operations*. This is not to be confused with the *total* number of function calls - because once we finish a pending operation, we don't care about storing information about that function call anymore!

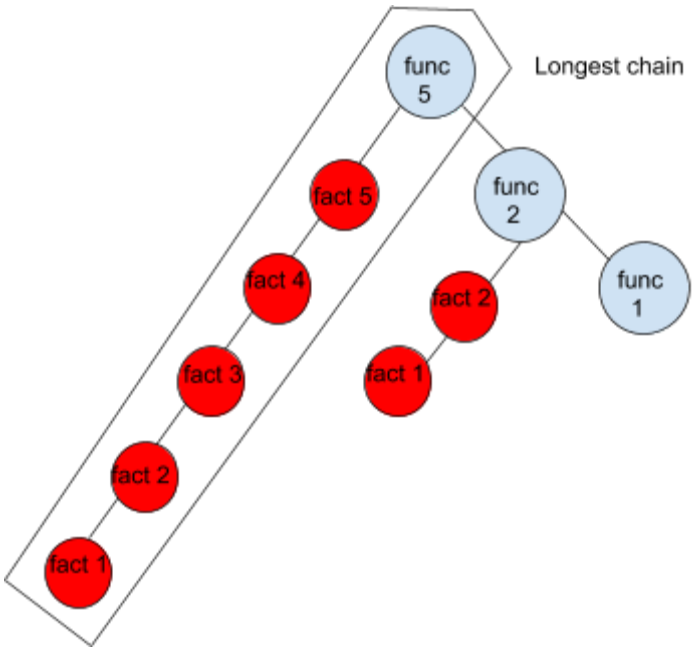
Let's use this approach to figure out the space complexity of the function `func()` defined by us earlier.

Space taken by single function calls=

A single function call for func() creates no new variable except for making a call to fact(). Therefore, the space taken by a single call is **constant**.

Longest chain of recursive calls=

If we examine the function, the longest chain of recursive calls we'll get is '**n+1**'. This will happen while calculating fact() for the very first call of func(n) - where we'll end up tracking 'n' recursive calls of fact(). The recursion tree maintained for a sample call of func(5) is on the right.



Hence, the space complexity is **O(n)** too!

Here's a screenshot from pythontutor showing how these recursive calls are stored in the interpreter. In fact, it is encouraged that you run different pieces of python code on pythontutor yourself to understand how space complexity works. This will be especially useful while dealing with space complexities of functions that deal with strings or maintain other complex data structures.