

## 6.00 Notes On Big-O Notation

Sarina Canelake

See also [http://en.wikipedia.org/wiki/Big\\_O\\_notation](http://en.wikipedia.org/wiki/Big_O_notation)

- We use big-O notation in the analysis of algorithms to describe an algorithm's usage of computational resources, in a way that is independent of computer architecture or clock rate.
- The worst case running time, or memory usage, of an algorithm is often expressed as a function of the length of its input using big O notation.
  - In 6.00 we generally seek to analyze the worst-case running time. However it is not unusual to see a big-O analysis of memory usage.
  - An expression in big-O notation is expressed as a capital letter “O”, followed by a function (generally) in terms of the variable  $n$ , which is understood to be the size of the input to the function you are analyzing.
  - This looks like:  $O(n)$ .
  - If we see a statement such as:  $f(x)$  is  $O(n)$  it can be read as “ $f$  of  $x$  is big Oh of  $n$ ”; it is understood, then, that the number of steps to run  $f(x)$  is linear with respect to  $|x|$ , the size of the input  $x$ .
- A description of a function in terms of big O notation only provides an *upper bound* on the growth rate of the function.
  - This means that a function that is  $O(n)$  is also, technically,  $O(n^2)$ ,  $O(n^3)$ , etc
  - However, we generally seek to provide the tightest possible bound. If you say an algorithm is  $O(n^3)$ , but it is also  $O(n^2)$ , it is generally best to say  $O(n^2)$ .
- Why do we use big-O notation? big-O notation allows us to compare different approaches for solving problems, and predict how long it might take to run an algorithm on a very large input.

With big-O notation we are particularly concerned with the *scalability* of our functions. big-O bounds may not reveal the fastest algorithm for small inputs (for example, remember that for  $x < 0.5$ ,  $x^3 < x^2$ ) but will accurately predict the long-term behavior of the algorithm.

- This is particularly important in the realm of scientific computing: for example, doing analysis on the human genome or data from Hubble involves input (arrays or lists) of size well into the tens of millions (of base pairs, pixels, etc).

- At this scale it becomes easy to see why big O notation is helpful. Say you're running a program to analyze base pairs and have two different implementations: one is  $O(n \lg n)$  and the other is  $O(n^3)$ . Even without knowing how fast of a computer you're using, it's easy to see that the first algorithm will be  $n^3/(n \lg n) = n^2/\lg n$  faster than the second, which is a BIG difference at input that size.

big-O notation is widespread wherever we talk about algorithms. If you take any Course 6 classes in the future, or do anything involving algorithms in the future, you will run into big-O notation again.

- Some common bounds you may see, in order from smallest to largest:
  - $O(1)$ : Constant time.  $O(1) = O(10) = O(2^{100})$  - why? Even though the constants are huge, they are still *constant*. Thus if you have an algorithm that takes  $2^{100}$  discreet steps, regardless of the size of the input, the algorithm is still  $O(1)$  - it runs in constant time; it is *not dependent upon the size of the input*.
  - $O(\lg n)$ : Logarithmic time. This is faster than linear time;  $O(\log_{10} n) = O(\ln n) = O(\lg n)$  (traditionally in Computer Science we are most concerned with  $\lg n$ , which is the base-2 logarithm - why is this the case?). The fastest time bound for search.
  - $O(n)$ : Linear time. Usually something when you need to examine every single bit of your input.
  - $O(n \lg n)$ : This is the fastest time bound we can currently achieve for sorting a list of elements.
  - $O(n^2)$ : Quadratic time. Often this is the bound when we have nested loops.
  - $O(2^n)$ : Really, REALLY big! A number raised to the power of  $n$  is slower than  $n$  raised to any power.
- Some questions for you:
  1. Does  $O(100n^2) = O(n^2)$ ?
  2. Does  $O(\frac{1}{4}n^3) = O(n^3)$ ?
  3. Does  $O(n) + O(n) = O(n)$ ?

The answers to all of these are Yes! Why? big-O notation is concerned with the long-term, or *limiting*, behavior of functions. If you're familiar with limits, this will make sense - recall that

$$\lim_{x \rightarrow \infty} x^2 = \lim_{x \rightarrow \infty} 100x^2 = \infty$$

basically, go out far enough and we can't see a distinction between  $100x^2$  and  $x^2$ . So, when we talk about big-O notation, we always *drop coefficient multipliers* - because they don't make a difference. Thus, if you're analysing your function and you get that it is  $O(n) + O(n)$ , that doesn't equal  $O(2n)$  - we simply say it is  $O(n)$ .

One more question for you: Does  $O(100n^2 + \frac{1}{4}n^3) = O(n^3)$ ?

Again, the answer to this is Yes! Because we are only concerned with how our algorithm behaves for very large values of  $n$ , when  $n$  is big enough, the  $n^3$  term will always dominate the  $n^2$  term, regardless of the coefficient on either of them.

In general, you will always say a function is big-O of its largest factor - for example, if something is  $O(n^2 + n \lg n + 100)$  we say it is  $O(n^2)$ . Constant terms, no matter how huge, are always dropped if a variable term is present - so  $O(800 \lg n + 73891) = O(\lg n)$ , while  $O(73891)$  by itself, with no variable terms present, is  $O(1)$ .

See the graphs generated by the file `bigO_plots.py` for a more visual explanation of the limiting behavior we're talking about here. Figures 1, 2, and 3 illustrate why we drop coefficients, while figure 4 illustrates how the biggest term will dominate smaller ones.

Now you should understand the What and the Why of big-O notation, as well as How we describe something in big-O terms. But How do we get the bounds in the first place?? Let's go through some examples.

1. We consider **all mathematical operations to be constant time ( $O(1)$ )** operations. So the following functions are all considered to be  $O(1)$  in complexity:

```
def inc(x):
    return x+1

def mul(x, y):
    return x*y

def foo(x):
    y = x*77.3
    return x/8.2

def bar(x, y):
    z = x + y
    w = x * y
    q = (w**z) % 870
    return 9*q
```

2. Functions containing for loops that go through the whole input are generally  $O(n)$ . For example, above we defined a function `mul` that was constant-time as it used the built-in Python operator `*`. If we define our own multiplication function that doesn't use `*`, it will not be  $O(1)$  anymore:

```
def mul2(x, y):
    result = 0
    for i in range(y):
        result += x
    return result
```

Here, this function is  $O(y)$  - the way we've defined it is dependent on the size of the input `y`, because we execute the `for` loop `y` times, and each time through the `for` loop we execute a constant-time operation.

3. Consider the following code:

```
def factorial(n):
    result = 1
    for num in range(1, n+1):
        result *= num
    return num
```

What is the big-O bound on `factorial`?

4. Consider the following code:

```
def factorial2(n):
    result = 1
    count = 0
    for num in range(1, n+1):
        result *= num
        count += 1
    return num
```

What is the big-O bound on `factorial2`?

5. The complexity of conditionals depends on what the condition is. The complexity of the condition can be constant, linear, or even worse - it all depends on what the condition is.

```
def count_ts(a_str):  
    count = 0  
    for char in a_str:  
        if char == 't':  
            count += 1  
    return count
```

In this example, we used an `if` statement. The analysis of the runtime of a conditional is highly dependent upon what the conditional's condition actually is; checking if one character is equal to another is a constant-time operation, so this example is linear with respect to the size of `a_str`. So, if we let  $n = |a\_str|$ , this function is  $O(n)$ .

Now consider this code:

```
def count_same_ltrs(a_str, b_str):  
    count = 0  
    for char in a_str:  
        if char in b_str:  
            count += 1  
    return count
```

This code looks very similar to the function `count_ts`, but it is actually very different! The conditional checks if `char in b_str` - this check requires us, in the *worst case*, to check every single character in `b_str`! Why do we care about the worst case? Because big-O notation is an upper bound on the *worst-case running time*. Sometimes analysis becomes easier if you ask yourself, what input could I give this to achieve the maximum number of steps? For the conditional, the worst-case occurs when `char` is **not** in `b_str` - then we have to look at every letter in `b_str` before we can return `False`.

So, what is the complexity of this function? Let  $n = |a\_str|$  and  $m = |b\_str|$ . Then, the `for` loop is  $O(n)$ . Each iteration of the `for` loop executes a conditional check that is, in the worst case,  $O(m)$ . Since we execute an  $O(m)$  check  $O(n)$  time, we say this function is  $O(nm)$ .

6. While loops: With while loops you have to combine the analysis of a conditional with one of a for loop.

```
def factorial3(n):
    result = 1
    while n > 0:
        result *= n
        n -= 1
    return result
```

What is the complexity of `factorial3`?

```
def char_split(a_str):
    result = []
    index = 0
    while len(a_str) != len(result):
        result.append(a_str[index])
        index += 1
    return result
```

In Python, `len` is a constant-time operation. So is string indexing (this is because strings are immutable) and list appending. So, what is the time complexity of `char_split`?

If you are curious, there is a little more information on Python operator complexity here:

<http://wiki.python.org/moin/TimeComplexity> - some notes: (1) CPython just means “Python written in the C language”. You are actually using CPython. (2) If you are asked to find the worst-case complexity, you want to use the Worst Case bounds. (3) Note that operations such as slicing and copying aren’t  $O(1)$  operations.

7. Nested for loops - anytime you’re dealing with nested loops, work from the inside out. Figure out the complexity of the innermost loop, then go out a level and multiply (this is similar to the second piece of code in Example 5). So, what is the time complexity of this code fragment, if we let  $n = |z|$ ?

```
result = 0
for i in range(z):
    for j in range(z):
        result += (i*j)
```

8. Recursion. Recursion can be tricky to figure out; think of recursion like a tree. If the tree has lots of branches, it will be more complex than one that has very few branches.

Consider recursive factorial:

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

What is the time complexity of this? The time complexity of `r_factorial` will be dependent upon the number of times it is called. If we look at the recursive call, we notice that it is: `r_factorial(n-1)`. This means that, every time we call `r_factorial`, we make a recursive call to a subproblem of size  $n-1$ . So given an input of size  $n$ , we make the recursive call to subproblem of size  $n-1$ , which makes a call to subproblem of size  $n-2$ , which makes a call to subproblem of size  $n-3$ , ... see a pattern? We'll have to do this until we make a call to  $n-n=0$  before we hit the base case - or,  $n$  calls. So, `r_factorial` is  $O(n)$ . There is a direct correlation from this recursive call to the iterative loop `for i in range(n, 0, -1)`.

In general, we can say that any recursive function `g(x)` whose recursive call is on a subproblem of size  $x-1$  will have a linear time bound, assuming that the rest of the recursive call is  $O(1)$  in complexity (this was the case here, because the `n*` factor was  $O(1)$ ).

How about this function?

```
def foo(n):
    if n <= 1:
        return 1
    return foo(n/2) + 1
```

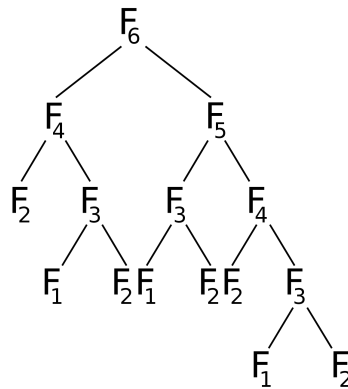
In this problem, the recursive call is to a subproblem of size  $n/2$ . How can we visualize this? First we make a call to a problem of size  $n$ , which calls a subproblem of size  $n/2$ , which calls a subproblem of size  $n/4$ , which calls a subproblem of size  $n/(2^3)$ , ... See the pattern yet? We can make the intuition that we'll need to make recursive calls until  $n=1$ , which will happen when  $n/2^x=1$ .

So, to figure out how many steps this takes, simply solve for  $x$  in terms of  $n$ :

$$\begin{aligned}\frac{n}{2^x} &= 1 \\ n &= 2^x \\ \log_2 n &= \log_2(2^x) \\ \therefore x &= \log_2 n\end{aligned}$$

So, it'll take  $\log_2 n$  steps to solve this recursive equation. In general, we can say that if a recursive function  $g(x)$  makes a recursive call to a subproblem of size  $x/b$ , the complexity of the function will be  $\log_b n$ . Again, this is assuming that the remainder of the recursive function has complexity of  $O(1)$ .

Finally, how do we deal with the complexity of something like Fibonacci? The recursive call to Fibonacci is  $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$ . This may initially seem linear, but it's not. If you draw this in a tree fashion, you get something like:



The *depth* of this tree (the number of levels it has) is  $n$ , and at each level we see a branching factor of two (every call to `fib` generates two more calls to `fib`). Thus, a loose bound on `fib` is  $O(2^n)$ . In fact, there exists a tighter bound on Fibonacci involving the Golden Ratio; Google for “Fibonacci complexity” to find out more if you’re interested in maths : D