COMP 161 - Lecture Notes - 16 - Complexity and Big-O

In these lecture notes we look at algorithm complexity and the use of Big-O notation to express complexity classes.

Complexity

Let's recall a few key points about the time complexity classification of a procedure:

- It's a measurement of elementary operations needed as a function of the size of the vector[1]. For an input with size n we'll often denote this function as $\mathcal{T}(n)$.
- It's based on the worst case, or maximum number of operations, for the size. It's meant as an upper bound.

[1] or some similar property of the input

Recall that we use complexity to capture the big picture of efficiency by classifying procedures[2] into key categories. For example, if we're talking about motor vehicles, then calling something and SUV versus a Hybrid car implies something about the fuel consumption efficiency of the vehicle. We recognize that these two classes of vehicles perform very differently and that within each class there's more variability still. Computational complexity classes essentially do the same thing. They set aside the specific details of the procedure's performance to emphasize overall characteristics. What we want to do is equate the time complexity function for a procedure to a well understood function. By doing so we can reduce the infinite possible actual time

[2] really the underlying algorithm

Big-O

There is a precise mathematical tool call Big-O notation that we can use to express the relationship we're looking for when expressing procedure complexity.

Definition 1. *Let the functions f and g for positive integers. Then the function f is on the order of g if there exists constants $n_0 \geq 0$ and α such that for all $n \geq n_0$*

$$f(n) \leq \alpha g(n)$$

We write this as $f(n) = O(g(n))$, or sometimes just $f = O(g)$,[3] and often say "f is Big-O of g" as opposed to "f is on the order of g".

[3] O is uppercase letter o, not the number zero

Let's pick this apart so that we understand what saying $f = O(g)$ really means. First notice it's clearly an upperbound relationship. The value $\alpha g(n)$ is at least as big as $f(n)$. The α term is allowing us

to focus not on the specific value of $g(n)$ as our bound but nearly any multiple of $g(n)$. We're more or less saying that "$f(n)$ is no larger than some multiple of $g(n)$". Finally, by throwing in n_0 we're emphasizing "larger" values of n for whatever large means to us. Putting this all together we see that $f = O(g)$ really means that "for all n past some point[4], we can draw some multiple[5] of g on or above f". The notion of Big-O gives us the right mixture of specificity and flexibility for capturing complexity and complexity classes.

[4] n_0
[5] α

In future courses you'll explore this formalism in more depth. Right now we want to learn some basic rules and relationships that follow such that we understand how Big-O communicates complexity classes. There are three basic consequences of definition 1 that are essential to working with Big-O in both a formal and informal manner. I'll state them formally and then provide some insight as to why they are so essential to complexity analysis.

The order of a sum is the sum of the orders.

Theorem 1. *Let f and g be functions over the positive integers. Then,*

$$O(f + g) = O(f) + O(g)$$

All procedures can be broken down into a series of steps. This theorem (1) tells us that it is enough to know the order of each step. This let's us greatly simplify complexity analysis by doing it piecewise rather than on the whole.

We have two properties that deal with produces and Big-O. The order of a product is the product of the orders.

Theorem 2. *Let f and g be functions over the positive integers. Then,*

$$O(fg) = O(f)O(g)$$

The order of a function is invariant under multiplication by a constant.

Theorem 3. *Let f and g be functions over the positive integers. Then for all real-values α,*

$$O(\alpha g) = O(g)$$

A great number of procedures involve repetition and repetition induces a product.[6]. When the repetition is done a fixed number of times, then theorem (3) tells us that the order of the repetition is the same as the the order of the thing being repeated. Similarly, if

[6] Repeating k operators t times takes tk operations in total

we repeat some fixed, constant amount of work, some function of our input size number of times, then theorem (3) tells us that the order of the repetition is the order of them number of function that determines the number of repetitions. If, however, the repetition and the work are both functions of our input size, then theorem (2) tells us that we can at least analyze the order of the repetition and the order of the repeated work in parts but cannot really simplify to one order or the other.

Using Big-O we can establish a strict ordering of some functions. This will be true of the functions on which we'll base our complexity classes. It turns out that larger orders dominate smaller orders.

Theorem 4. *For functions f and g. If $O(f) < O(g)$, then*

$$O(f) + O(g) = O(g)$$

Once again, sums arise from the discrete steps of a computation. When one step requires an order of magnitude more computation than another step, then that step dominates the total running time of the computation to the point that the whole computation behaves as if it were only of that larger complexity class. What theorem 4 lets us do is focus solely on thethe critical step(s) that account for most of the work.[7].

Hopefully you're picking up on the fact that these rules have close ties to the constructs of programs. The step-wise nature of induces sums in our time complexity functions and those sums can be more easily managed using theorem (1). Repetition induces products in our time complexity functions and those produces can be managed with theorem (2) and (3). The analysis of conditionals is simplified by the fact that we're only concerned with the worst case. If we know which branch has the highest order, then we can focus on that branch.

[7] I tend to think of this rule in terms of spending money. If you're paying thousands of dollars for something, then adding on tens of dollars doesn't really change the big picture– you're paying thousands of dollars for something.If you're paying $5000, then you'll probably also pay 5050

Some Complexity Classes

Remember our intended use for Big-O notation is the classification of the time complexity of procedures in order to address the questions like whether or not one procedure is more or less complex than another or if a given procedure is too complex to be usable in practice. A large majority of the procedures we encounter have a time function which is on the order of one of the following functions. These functions, in turn, establish a clear ordering of classes.

1. Constant

If the complexity function f is constant, i.e. $f(n) = c$ for some fixed value c, then we say $f = O(1)$.

2. LOGARITHMIC
   If the complexity function f is logarithmic, i.e. $f(n) = \log_b n$, then we say $f = O(\log n)$. Notice that we leave off the base of the logarithm. It can be shown that the base of the logarithm actually doesn't matter in Big-O notation[8] so we typically use 2. You'll often see $\log_2$ written as lg

[8] $O(\log_a n) = O(\log_b n)$ for all a and b

3. LINEAR
   If the complexity function f is linear, i.e. $f(n) = n$, then we say $f = O(n)$.

4. LINEARITHMIC[9]
   If the complexity function f is the product of linear and logarithmic function, i.e. $f(n) = n \log n$, then we say $f = O(n \log n)$.

[9] a blend of linear and logarithmic

5. QUADRATIC
   If the complexity function f is a quadratic function, i.e. $f(n) = n^2$, then we say $f = O(n^2)$.

6. CUBIC
   If the complexity function f is a cubic function, i.e. $f(n) = n^3$, then we say $f = O(n^3)$.

7. EXPONENTIAL
   If the complexity function f is a exponential for some constant $c > 1$, i.e. $f(n) = c^n$. We express this as $f = O(c^n)$.

8. FACTORIAL
   If the complexity function f is the factorial, i.e. $f(n) = n!$, then we say $f = O(n!)$.

Within each class there is an awful lot of nuance. For example, pick any time function where $\{(n) = an + b$, then $f = O(n)$. As you may remember from math class, the larger the value of a the greater the slope of the line and the faster it grows. Big-O notation completely ignores this reality and instead groups all lines together. That's ok, the point of Big-O and complexity classes are not specific, nuanced details, but big picture, order of magnitude details.

The complexity classes we use are represented by basic, well understood functions for which there is a strict least to greatest growth rate order.

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(c^n) < O(n!)$$

If you have two different procedures that solve the same problem and one has work $O(n)$ and the other has work $O(n \log n)$, then you know the first, the linear work option, is *a whole order of magnitude*

better than the later. This is what we get from Big-O and complexity classes. We get the ability to partition the infinite world of procedures into simple efficiency complexity classes that allow us to quickly and easily identify order of magnitude differences.

On the practical side of things, we can also quickly identify when a procedure will be impractical. Computer science has drawn a line in the sand at POLYNOMIAL COMPLEXITY such that any complexity class that is a polynomial or better is theoretically efficient. This includes everything on our list but $O(2^n)$ and $O(n!)$. It turns out that this somewhat arbitrary choice works pretty well in practice. However, cubic and quadratic functions do grow pretty quickly and can easily be impractical for even modest n. Superlinear or better is almost always practical, even for very large n. You'll get a very real sense of this in your final project, but let's look at some numbers to get you thinking.

First lets look at how each function grows as n grows.

	lg(n)	n	nlg(n)	n^2	n^3	2^n
1	0	1.00	0	1.00	1.00	2.00
10	3.00	10.0	33.0	100.	1.00×10^{3}	1.02×10^{3}
25	5.00	25.0	116.	625.	1.56×10^{4}	3.36×10^{7}
50	6.00	50.0	282.	2.50×10^{3}	1.25×10^{5}	1.13×10^{15}
75	6.00	75.0	467.	5.63×10^{3}	4.22×10^{5}	3.78×10^{22}
100	7.00	100.	664.	1.00×10^{4}	1.00×10^{6}	1.27×10^{30}
500	9.00	500.	4.48×10^{3}	2.50×10^{5}	1.25×10^{8}	3.27×10^{150}
1000	10.0	1.00×10^{3}	9.97×10^{3}	1.00×10^{6}	1.00×10^{9}	1.07×10^{301}
2500	11.0	2.50×10^{3}	2.82×10^{4}	6.25×10^{6}	1.56×10^{10}	3.76×10^{752}
5000	12.0	5.00×10^{3}	6.14×10^{4}	2.50×10^{7}	1.25×10^{11}	1.41×10^{1505}
7500	13.0	7.50×10^{3}	9.65×10^{4}	5.63×10^{7}	4.22×10^{11}	5.31×10^{2257}
10 000	13.0	1.00×10^{4}	1.33×10^{5}	1.00×10^{8}	1.00×10^{12}	2.00×10^{3010}
25 000	15.0	2.50×10^{4}	3.65×10^{5}	6.25×10^{8}	1.56×10^{13}	5.62×10^{7525}
50 000	16.0	5.00×10^{4}	7.80×10^{5}	2.50×10^{9}	1.25×10^{14}	3.16×10^{15051}
75 000	16.0	7.50×10^{4}	1.21×10^{6}	5.63×10^{9}	4.22×10^{14}	1.78×10^{22577}
100 000	17.0	1.00×10^{5}	1.66×10^{6}	1.00×10^{10}	1.00×10^{15}	9.99×10^{30102}
250 000	18.0	2.50×10^{5}	4.48×10^{6}	6.25×10^{10}	1.56×10^{16}	3.15×10^{75257}
500 000	19.0	5.00×10^{5}	9.47×10^{6}	2.50×10^{11}	1.25×10^{17}	9.95×10^{150514}
750 000	20.0	7.50×10^{5}	1.46×10^{7}	5.63×10^{11}	4.22×10^{17}	3.14×10^{225772}
1 000 000	20.0	1.00×10^{6}	1.99×10^{7}	1.00×10^{12}	1.00×10^{18}	9.90×10^{301029}

Figure 1: The value of key complexity class functions for increasing values of n

Now imagine you have a computer that performs 1 operation every nanoseconds[10] Then, we can attach some time amounts to these complexity classes.

[10] also known as a 1 Gigaflop GFLOP computerThis is actually fairly modest by today's standards. Your average laptop can probably do two to five times better than this.

	lg(n)	n	nlg(n)	n^2	n^3	2^n
1	0 ns	1.00 ns	0 ns	1.00 ns	1.00 ns	2.00 ns
10	3.00 ns	10.0 ns	33.0 ns	100. ns	1.00 µs	1.02 µs
25	5.00 ns	25.0 ns	116. ns	625. ns	15.6 µs	33.6 ms
50	6.00 ns	50.0 ns	282. ns	2.50 µs	125. µs	1.86 wk
75	6.00 ns	75.0 ns	467. ns	5.62 µs	422. µs	1.20×10^{6} yr
100	7.00 ns	100. ns	664. ns	10.0 µs	1.00 ms	4.02×10^{13} yr
500	9.00 ns	500. ns	4.48 µs	250. µs	125. ms	1.04×10^{134} yr
1000	10.0 ns	1.00 µs	9.97 µs	1.00 ms	1.00 s	3.40×10^{284} yr
2500	11.0 ns	2.50 µs	28.2 µs	6.25 ms	15.6 s	1.19×10^{736} yr
5000	12.0 ns	5.00 µs	61.4 µs	25.0 ms	2.08 min	4.48×10^{1488} yr
7500	13.0 ns	7.50 µs	96.5 µs	56.2 ms	7.03 min	1.68×10^{2241} yr
10 000	13.0 ns	10.0 µs	133. µs	100. ms	16.7 min	6.33×10^{2993} yr
25 000	15.0 ns	25.0 µs	365. µs	625. ms	4.34 h	1.78×10^{7509} yr
50 000	16.0 ns	50.0 µs	780. µs	2.50 s	1.45 days	1.00×10^{15035} yr
75 000	16.0 ns	75.0 µs	1.21 ms	5.63 s	4.88 days	5.63×10^{22560} yr
100 000	17.0 ns	100. µs	1.66 ms	10.0 s	1.65 wk	3.17×10^{30086} yr
250 000	18.0 ns	250. µs	4.48 ms	1.04 min	5.95 mo	1.00×10^{75241} yr
500 000	19.0 ns	500. µs	9.47 ms	4.17 min	3.96 yr	3.16×10^{150498} yr
750 000	20.0 ns	750. µs	14.6 ms	9.38 min	13.4 yr	9.95×10^{225755} yr
1 000 000	20.0 ns	1.00 ms	19.9 ms	16.7 min	31.7 yr	3.14×10^{301013} yr

Figure 2: The time needed to compute key complexity class functions at 1GFLOP