

COMP 161 - Lecture Notes - 15 - Complexity and Big-O

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

Efficiency, Complexity, and Computational Resources

Being efficient means optimizing the use of one or more resources. A fuel efficient car will travel far while minimizing the consumption of fuel. This means the ratio of miles traveled per gallon of fuel is high relative to a less efficient vehicle¹. When comparing cars for fuel efficiency we tend to look at things like the expected miles per gallon or perhaps the highest achievable miles per gallon. In both cases, we're operating the car not under a guarantee of peak efficiency but with an understanding that we can expect a certain level of efficiency when driving. In short, we understand that manufactures typically provide us with theoretically achievable efficiency for our vehicle and that in practice, mileage may vary.

¹ conversely the ratio of gallons consumed per mile is lower

Of course efficiency is not a singular measure. It is relative to the consumption of a specific resource. Perhaps the consumption of fuel is not your concern, but time is. A time efficient car would exhibit a higher acceleration rate and/or top speed. We typically expect that this increased time efficiency came at the cost of fuel efficiency. So not only is efficiency subjective, but the overall efficiency of a vehicle involves trade offs. Less time means more fuel, less fuel means more time.

Complexity

Enough about cars. What about computers and computation? The most discussed and emphasized computational resource around is TIME. Time efficient computations take less time. We then tend to look at other resources in terms of their relationship with time. By increasing the SPACE² needs of a computation we can often decrease the *time* needs. Conversely, decreasing space often requires increased time. In this way space might be viewed as the fuel of computation. There are still more resources used in computation. When we get into parallel computation, we start to think about things like COMMUNICATION as a resource. The less computers working in parallel have to communicate, the less time they need and vice versa.

² memory

COMPLEXITY THEORY is the branch of theoretical computer science in which problems are classified by the resource needs of their solutions. Go look at the documentation for C++ library procedures

and class methods on cplusplus.com and you'll see a section title *Complexity*. This is exactly the complexity theoretic view of the efficiency of that procedure. It provides the programmer with a broad classification of the efficiency of the procedure in favor of a fine-grained view. From the programmer's perspective, complexity measures of efficiency provide you with an expectation of efficiency in much the same way that a car manufacturer's reported miles per gallon does for fuel efficiency. The specifics of your usage of a procedure, the actual inputs, the actual state of the system, will have an impact on the time needed but the complexity provides some general guarantees.

Unless otherwise specified, complexity classifications are based on the **WORST CASE** resource needs of the procedure. The worst and best case are often generated by a very specific variation of the input and it's easier to be concrete about their resource needs. For example, the basic search procedure's best case is when the item we're looking for is the first item we look at and it's worst case is when it's the last. The average case is much less specific; it's when the item we're looking for is neither the first nor the last. Given a choice between best case and worst case thinking, we err on the side of worst case. Knowing that your program needs at most some amount of time means that on average we can expect that efficiency or better. This **UPPER BOUND** on the resource needs is ultimately more useful to the programmer than the *lower bound* offered by the best case.

Complexity theory is carried out in a machine and language agnostic fashion. We measure time by counting basic operations, not by the clock. We measure space sometimes by bits and bytes but sometimes by abstract, discrete units of data. By classifying procedures in this way we are able to rule out the language used and the machine executing the code and instead focus on some underlying abstract model of computation. In doing so, the assessment of a procedure's complexity becomes an assessment of the underlying *algorithm* and not the actual implementation. As a discipline of study, complexity theory is smack dab at the intersection of mathematics and computer science³. For our purposes, we simply wish to apply the basic principles, broad categorization of program efficiency based on the the expected worst case performance of the procedures, to program design and evaluation.

³ The work itself is done in mathematics where the objects of study are those of computer science. The distinction between the two disciplines gets awful blurry in this realm

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 be functions 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 $f = O(g)$,⁴ and often say “ f is Big-O of g ” as opposed to “ f is on the order of g ”.

⁴ 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:

For all n past some point⁵, we can draw some multiple⁶ of g on or above f

⁵ n_0
⁶ α

The notion of Big-O gives us the right mixture of specificity and flexibility for capturing complexity and complexity classes.

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 products 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.⁷ 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 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.

⁷ Repeating k operators t times takes tk operations in total

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 the critical step(s) that account for most of the work.⁸

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.

⁸ 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 notation⁹ so we typically use 2. You'll often see \log_2 written as lg

⁹ $O(\log_a n) = O(\log_b n)$ 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. LINEARITHMIC¹⁰

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

¹⁰ 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 an 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 $f(n) = an + b$, 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 | $n \lg(n)$ | 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 nanoseconds¹¹ Then, we can attach some time amounts to these complexity classes.

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

¹¹ also known as a 1 GigaFlop GFLOP computer. This is actually fairly modest by today's standards. Your average laptop can probably do two to five GigaFlops. The time needed to compute key complexity class functions at 1GFLOP