COMP 340 - Lecture Notes - 04 - Efficiency

Spring 2014

These notes begin discussing material from chapter 2 of *The Algorithm Design Manual*.

Efficient in Theory

Remember our goal is to study and develop algorithms in such a way that our results are meaningful and useful beyond the current state of computing technology. This is why we don't jump to a full fledged programming language when writing down our algorithms and why we use more general mathematics to model problem information.

When we turn our eye to efficiency it might be tempting to say, "Implement the algorithm, run it on actual inputs, and record performance data." This has several problems:

1. We're measuring both the hardware/software stack running the algorithm and the algorithm itself. We really just want to know if the algorithm itself is efficient.

2. Problems regularly have an infinite number of possible inputs, so the best we can manage with this is some statistical guarantees. The leaves us worrying about the outliers and how they'll behave.

3. We don't know if the algorithm is even easy to implement. Remember CORRECT→EFFICIENT→Easy to Implement. Jumping to implementation is a bit like putting the cart before the bull.

So, what we need is a theoretical framework in which we can build accurate models for predicting algorithm efficiency. As science always does, we turn to Mathematics as the bedrock of our theory.

Machine Model

First we need a reasonable model for a sequential computer[1]. This model should be close to what actual hardware does, but not so encumbered by architecture specifics. The RAM MODEL[2] has proven to be remarkable robust. It's simple, elegant, and produces accurate predictions of real-world performance more often than not. The RAM model assumes some predetermined set of primitive operations which require exactly one unit of work to carry out. We then add an infinite size memory space where each memory access costs one unit of work regardless of where it is. From this, we can build a picture of the resource usage of our algorithms where the key resources are TIME[3] and SPACE[4].

[1] we're only looking at sequential algorithms, we'd need a parallel computer model for parallel algorithms

[2] Random Access Machine pg 31

[3] measured in Units of work

[4] measured in Bytes or some fixed unit of size

Performance Model

There are some really surprising and interesting results about the relationship between the TIME-SPACE usage of algorithms. We'll probably encounter some, but our present concern is understanding our one measures the efficiency of an algorithm in terms of these two resources. The first important observation is that any problem, and therefore any algorithm input, has some measurable size attribute. All of our combinatorial structures have obvious sizes: the number of elements. Numerical problems often boil down to the magnitude of the number or the size of the representation of the number in binary. Either way, we can always attribute a size to the inputs to our algorithms. From this, we can construct performance functions.

Let n be the size of the input to our algorithm. Then $T(n)$ is the function relating input size to total units of work and $S(n)$ relates size to memory usage. Our goal is now to understand[5] $T(n)$ and $S(n)$ for our algorithm. Simple enough, right? Actually, the reality of these functions is often hard to pin down and muddied by details. We can, sometimes, find the exact performance functions, but more often than not what we really need is a way to ballpark the performance. We want to understand the big picture, not the minutiae. Thankfully, mathematics has just the tools we need to do this: ASYMPTOTICS. Our favorite asymptotic in CS is "BIG OH"

[5] and minimize!

One more detail before we dig in to Big Oh. Let's say we're looking at a problem dealing with permutations of numbers. It would be nice if our algorithm behaved the exact same for all permutations of size n, but this is rarely the case. Some permutations will invoke different behaviors. Put more generally: *there are often different performance cases for a given input size*. This means we not only need to think about the performance function in general, but different cases. The most common case to consider is the WORST CASE. It's so common that when reading/writing a performance analysis you assume worst case unless someone says otherwise. Why would this be?

Your first gut reaction might be to consider an *average case* performance measure. This is great in theory, but in practice it's often too difficult to rigorously define what the average case means. Do we mean the statistical expected value or just the unweighted average. The later requires that we know a lot about the frequency of one instance over another. The former completely disregards such information. So, average case is often just too difficult to deal with or just not uniform enough in what it tells us.

Best case sounds nice, but then you realize that too often we don't get the best case scenario. So knowing just how good it could get is a bit of a tease. It's probably better that you know how bad it could get instead. Thus we turn to the *worst case*. Worst cases is usually easy enough to pin down and it tells us an upper bound on the algorithm performance. Thus we get a more useful point of comparison.

> Standard efficiency analysis looks at the WORST CASE performance as a function of the size of the input.

Big Oh

Let's say I was going to sell you a car for "hundreds of dollars" and someone else was going to sell you the exact same car for "thousands of dollars". Without knowing the exact price, you can quickly tell that I'm likely to give you the better deal. It's this kind of *orders of magnitude* thinking we need to bring to the analysis of our algorithms. This means we need a systematic way of grouping arbitrary functions into classes such that each class corresponds to an order of magnitude. You might be tempted to look at specific numerical values. Say, this algorithm takes hundreds of units of works for input sizes in the hundreds. This is too limiting though: why? Well, it's too discrete. We need to understand performance for all possible inputs, not just discrete intervals of inputs like size in the hundreds. What we need to understand is the general *growth pattern* of $T(n)$[6]. Put this all together now: we need a way to section out all possible performance functions into orders of magnitude based on how they grow as the input grows. This is what asymptotics and "Big Oh" allow us to do: reduce the growth rate of our performance function to a simple "pattern" suitable for comparison to other performance functions.

[6] or $S(n)$ relative to n

Dominance Relations

Let's do things a little backwards and consider the most common categories in to which we'll pigeon hole our performance functions[7]. In least to greatest order:

[7] Skiena gives you some clues as to where we see these on pg 39

1. CONSTANT $f(n) = 1$
2. LOGARITHMIC $f(n) = \log n$
3. LINEAR $f(n) = n$
4. SUPERLINEAR $f(n) = n \log n$
5. QUADRATIC $f(n) = n^2$
6. CUBIC $f(n) = n^3$
7. EXPONENTIAL $f(n) = c^n$ for some constant $c > 1$

8. Factorial $f(n) = n!$

So, as n gets larger it's always the case that:

$$1 << \log n << n << n \log n << n^2 << n^3 << c^n << n!$$

Notice we said, "as n gets larger". We generally not interested in small n as most of these function produce similar results for small n and in the era of "Big Data" we're extra concerned with very large values of n. However, as n gets bigger and bigger these functions sort out into the above hierarchy. Now, if we can just say something like "our performance function grows like f" then we're in business.

Big Oh by the books

First things first, big O is really one of three asymptotics we use in algorithm analysis. It roughly corresponds to $\geq$. The other two correspond to $\leq$[8] and $=$[9].To effectively use these asymptotics as a communication tool, we need to be certain we understand what they do and do not tell us. Thus, the definitions found in the text should be committed to memory[10].

[8] Big Omega

[9] Big Theta

[10] pg 35

1. $T = O(f)$
   For all $n \geq n_0$, $T \leq c * f$ for some constant c

2. $T = \Omega(f)$
   For all $n \geq n_0$, $T \geq c * f$ for some constant c

3. $T = \Theta(f)$
   For all $n \geq n_0$, $T \leq c_1 * f$ for some constant c_1 and $T \geq c_2 * f$ for some constant c_2.

New Question If we say $T(n) = O(f(n))$ then we're really saying something along the lines of, "past some minimal value of n, n_0, the Time performance function T is bounded above by some kind of f like function" More generally, "the function T grows no faster than something in the set of fs past n_0." The first thing to note is that $O(f(n))$ really denotes a whole family of functions. When we say $O(n^2)$ we should really be thinking not of n^2, but the family of functions of the form $an^2 + bn + c$. What's more, the whole big oh thing is kind of weak if you think about it. For example, all of these are true statements:

- $n = O(n!)$
- $n = O(2^n)$
- $n = O(n^2)$

- $n = O(n)$

The last one seems the most interesting in that it is the tightest upper bound of the four. We typically expect people to give us the tightest upper bound possible, but based on its definition, Big Oh leaves a lot of room for loose bounds.

Now let's look at the other side of things. If we say $T(n) = \Omega(f(n))$, then we're trying to say that, "the function T grows faster than some kind of f-like function past n_0". Did you see what happened there? That sounds like Big oh but the f and T changed places. You can, in fact show that, $T = \Omega(f) \leftrightarrow f = O(T)$. What we're really doing is establishing a lower bound. Once again, our expectation is that the lower bound is tight, but the definition leaves wiggle room for pretty loose bounds. Just like we preferred worst case over best case analysis, we prefer O over Ω for most of what we're doing. The use of Ω is, however, important for get super tight bounds on our worst case.

If $a \leq b$ and $a \geq b$ then it must be the case that $a = b$, right? The same is true with asymptotics. If $T = O(f)$ and $T = \Omega(f)$ then T is most definitely in the f family of functions. This is *Big Theta*, or $T = \Theta(f)$. It turns out that outside of algorithms research, people get kind of sloppy with their asymptotics and use O when they mean Θ. Perhaps the fact that we're doing upper bounds on our cases lends itself to focusing on the upper bound asymptotic. What you really want though is a good, tight Θ bound on the worst case as this solves the wiggle room problem we noticed earlier.

Reasoning about performance with O, Ω, and Θ

There are some key rules to working with asymptotics[11]. Here's the Big Oh version:

[11] pg 40

1. $O(O(f) + O(g)) \rightarrow O(max(f, g))$
2. $O(c * f) \rightarrow O(f)$
3. $O(f) * O(g) \rightarrow O(f * g)$

You may know these as:

1. Drop the higher order terms from a sum
2. Drop the multiplicative constants
3. Big Oh of a product is the product of the Big Ohs

One way to put these to work in algorithm analysis is to take the whole, long detailed performance function and simplify by these rules. This is fine and has the benefit of giving you the complete picture and the coarse grained picture in Big Oh land. However, we don't often need the complete picture. So the real shining moment of these rules is their ability allow us to simplify as we go.

A sequential algorithm is a sequence of statements. Each statement has an associated cost. So, the total cost is the sum of the cost of the statements. Because we're pretty much always dealing with a sum, we can always apply the first rule as we go. The second and third rules usually pop in the course of analyzing loops or recursion[12].

[12] they multiply the cost of a single recursive function evaluation or a single loop iteration by the number function calls or iterations