CS 170: Efficient Algorithms and Intractable Problems

Note 1: Asymptotics

Vishnu Iyer January 23, 2019

Before we can begin a proper discussion of algorithms, we need a notion of algorithmic efficiency. Our definition of efficiency can depend on various things like how fast an algorithm runs, or how much memory it takes to execute. Regardless, we want to formalize the notion of how the resources used by an algorithm changes as the input size grows. To do this, we will introduce **asymptotics**.

1 Asymptotic Notation

The main idea of asymptotics is to capture how fast a function grows while abstracting away minute details. For example, we can say the function $f(n) = 3n^2 + 4n + 1$ grows roughly as n^2. That is, if the input doubles, the value of the functions quadruples. While it is not exact, as the input gets very large, our estimate does a very good job of approximating the growth of the function. We now seek to formalize this notion.

1.1 O Notation

O notation (pronounced "big-Oh") captures the idea of an upper bound on a function.

Definition 1.1 (O notation)**.** Consider two functions $f(n)$ and $g(n)$ from $\mathbb{N}$ to $\mathbb{N}$. We say $f(n) = O(g(n))$ if there exists constants c, M such that for all $n \geq M$, $f(n) \leq cg(n)$.

Let's break this definition down. What purpose do c and M serve? Well, c exists to abstract away any constant factors in front of our function. Thinking back to our earlier example, we can surely say that $3n^2 + 4n + 1$ grows as $3n^2$, but, really, it also grows as n^2. The constant multiple plays no role in the fact that the function quadruples when the input doubles. This is the abstraction we were talking about earlier.

What about M? What does it do? Well, we can interpret the statement "for all $n \geq M$" as "for all n *sufficiently large.*" We want to analyze the behavior of the function as the inputs get really large (thus the word asymptotic) without getting bogged down in the details of how it grows for smaller inputs. After all, variations in the function growth for smaller inputs don't really capture the essence of the function's overall behavior.

Example 1.2. Show that $3n^2 + 4n + 1 = O(n^2)$.

Solution. We mentioned earlier that the function grows like n^2. Let's show this formally by finding c and M. Intuitively, $c \leq 3$ won't work, but to keep things simple, we'll take $c = 4$ (remember, we just have to show *one* such pair c and M). With a calculation, we can verify that when $n \geq 5$, $n^2 \geq 3n^2 + 4n + 1$. Thus, taking $c = 4$ and $M = 5$, we're done. □

The "calculation" mentioned above involves use of differential calculus to find critical points. This, along with the length of the argument needed to make a rather simple claim makes these problems seem overly formal. However, it is important to keep this formality when introducing the topic. For some problems, formality will be necessary. From now on, however, we will take it as granted that a polynomial $f(n)$ of degree at most c satisfies $f(n) = O(n^c)$ (even though we haven't proven this entirely).

Example 1.3. Find as tight an upper bound as is possible for the following function.

$$f(n) = \begin{cases} 2^n & n < 100 \\ 7n^3 + 4n + 3 & n \geq 100 \end{cases}$$

Solution. We wouldn't be wrong in saying that $f(n) = O(2^n)$. Both "segments" of $f(n)$ are $O(2^n)$. However, we want as tight a bound as possible. Notice that we only care about *sufficiently large values of* n. That is, it doesn't matter how the function appears before $n = 100$ but rather how it behaves as n approaches infinity. This translates to taking $M \geq 100$. For simplicity, we take $M = 100$ and $c = 8$ and see that because $7n^3 + 4n + 3 < 8n^3$ for $n \geq 100$, $f(n) = O(n^3)$. □

In the above example we see exactly how M plays a role in the definition and what it means to be sufficiently large. Furthermore, we essentially also proved $7n^3 + 4n + 3 = O(n^3)$ but with a lot less formality than in our first example. This is an important point in asymptotics - using previous results to simplify proofs for more complicated claims. Such abstraction will be important throughout the course.

1.2 Ω Notation

Ω (pronounced "big Omega") notation describes lower bounds for functions in the same way that O notation describes upper bounds.

Definition 1.4 (Ω notation)**.** Consider two functions $f(n)$ and $g(n)$ from $\mathbb{N}$ to $\mathbb{N}$. We say $f(n) = \Omega(g(n))$ if there exists constants c, M such that for all $n \geq M$, $g(n) \leq cf(n)$.

Notice that this definition extremely similar to the one for O notation (which was great for me because I could just copy and paste the LaTeX). Again, c and M exist serve the same purpose as they did in the definition of O notation. Now let's look at an example.

Example 1.5. Show that $3n^2 - 7n + 2 = \Omega(n^2)$. **Solution.** Take $c = 2$ and $M = 5$. By the above definition, we are done. □

When we ask for *tight* upper and lower bounds on an algorithm, it means we want a bound that in some sense can't go any further. For example, the function $3n^2 - 7n + 2$ is upper bounded by $O(n^2)$ and lower bounded by $\Omega(n^2)$ but we can't change those bounds to, say $O(n^{1.99})$ and $\Omega(n^{2.01})$. This is because n^2 is both an upper and lower bound! So there's no way a lower bound can be higher than any upper bound and an upper bound can be lower than a lower bound.

1.3 Θ notation

The interesting thing in the previous example is that $3n^2 - 7n + 2$ is both $O(n^2)$ and $\Omega(n^2)$. This occurrence is so special that it warrants its own notation.

Definition 1.6 (Θ notation)**.** Consider two functions $f(n)$ and $g(n)$ from $\mathbb{N}$ to $\mathbb{N}$. We say $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

In many cases, you'll be asked to give as tight a bound as possible, and Θ notation is often the way to go about it. Sure, we can say that $3n^2 - 7n + 2$ is $O(n^3)$ and $\Omega(n)$ but the most precise characterization is that it is $\Theta(n^2)$.

1.4 o and ω notation

The notations O, Θ, and Ω correspond roughly to a function growing "as fast or slower than," "roughly as fast as," and "as fast or faster than" a reference. We also have notation for "strictly faster than" and "strictly slower than."

Definition 1.7 (o notation)**.** Consider two functions $f(n)$ and $g(n)$ from $\mathbb{N}$ to $\mathbb{N}$. We say $f(n) = o(g(n))$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Definition 1.8 (ω notation)**.** Consider two functions $f(n)$ and $g(n)$ from $\mathbb{N}$ to $\mathbb{N}$. We say $f(n) = \omega(g(n))$ if

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

Thus, just as Ω, Θ, and O are analogs of $\leq$, $=$, and $\geq$, the symbols o and ω are analogs of $<$ and $>$. This interpretation is a nice way to understand the notation intuitively.

1.5 Useful Results

We will now provide a list of some useful facts about asymptotic notation. It is a highly useful (and recommended) exercise to prove the statements. While some may seem intuitive, the practice with formal machinery will make solving other problems easier.

Useful Asymptotics Facts
Let $f : \mathbb{N} \to \mathbb{N}$, $g : \mathbb{N} \to \mathbb{N}$, and $h : \mathbb{N} \to \mathbb{N}$ be functions.

1. $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.
2. $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.
3. If $f(n) = o(g(n))$ then $f(n) = O(g(n))$ but not necessarily vice-versa.
4. If $f(n) = \omega(g(n))$ then $f(n) = \Omega(g(n))$ but not necessarily vice-versa.
5. If $f(n) = \Theta(g(n))$ and $h(n) = O(f(n))$ then $f(n) + h(n) = \Theta(g(n))$.
6. If $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$ then $f_1(n)f_2(n) = \Theta(g_1(n)g_2(n))$.
7. Either $f(n) = O(g(n))$ or $f(n) = \Omega(g(n))$ or both.

2 The Master Theorem

Now we turn our attention to a master theorem that can solve for the growth rate of an arbitrary recurrence relation.

Theorem 2.1 (Master Theorem)**.** Consider the recurrence

$$T(n) = aT(n/b) + f(n)$$

and take $c^* = \log_b a$

1. If $f(n) = O(n^c)$ where $c < c^*$, then $T(n) = \Theta(n^{c^*})$.
2. If $f(n) = \Theta(n^{c^*} \log^k n)$ for $k \geq 0$, then $T(n) = \Theta(n^{c^*} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^c)$ where $c > c^*$, and $af(n/b) < f(n)$ for all n sufficiently large, then $T(n) = \Theta(f(n))$.

While we won't prove the master theorem here, we will take a look at what each point means. In particular, we will consider the theorem in the scope of the "divide and conquer" paradigm (discussed in depth in the next note) of splitting a problem into smaller problems, or subproblems, and then combining their solutions.

The number $c^* = \log_b a$ offers a measure of the "efficiency" of splitting a problem up. Smaller subproblems (large b) and fewer subproblems (small a) will both reduce the value of c^*. Furthermore, the left term $aT(n/b)$ of the recurrence represents the work of splitting the problem while the right term $f(n)$ represents putting the subproblem solutions together.

1. In this first case, the cost of splitting and solving subproblems outweighs the cost of putting them together. This is represented by the fact $f(n) = O(n^c)$ for $c < c^*$. Therefore, the overall runtime is dominated by the splitting and we can effectively ignore the $f(n)$ term in the recursion, giving us $T(n) = \Theta(n^{c^*})$.
2. The second case is the most complicated. Here, the cost of splitting the problems is very similar to the cost of combining them, being only a logarithmic factor apart. The proof of the final result here is tricky but just know that the total runtime has an overhead of $\log n$.
3. This case is the opposite of case 1, and combining subproblems takes the most work. Thus similar to case 1, we can effectively ignore the first term of the recursion and obtain that $T(n) = \Theta(f(n))$.

Now none of the above discussion is formal, but it does clue us in on a good interpretation of the master theorem. The lesson here is that for a theorem involving a multitude of cases and results, it often helps to simplify the underlying logic before committing it to memory.

Example 2.2. Find a tight bound on the following recurrences:

1. $T(n) = 2T(n/2) + n$
2. $T(n) = 3T(n/2) + 4n^2$

3. $T(n) = 6T(n/6) + 2n \log n$

4. $T(n) = 11T(n/10) + n \log n$

Solution. We apply the master theorem.

1. Here we have $c^* = \log_2 2 = 1$ and $f(n) = n = O(n)$. Thus, applying the second case of the master theorem, we obtain $T(n) = \Theta(n \log n)$ (note: this particular recurrence shows up a lot).

2. In this case, $c^* = \log_2 3 < 2$, so $4n^2 = \Omega(n^{c^*})$. Thus we apply the third case of the master theorem and obtain $T(n) = \Theta(n^2)$.

3. This is similar to problem 1, since $c^* = 1$, but we have an additional factor of $\log n$ in $f(n)$. Thankfully, the master theorem accounts for this, and case 2 gives us that $T(n) = \Theta(n \log^2 n)$.

4. This problem is a little tricky, because $c^* = \log_{10} 11 > 1$. Is it true that $n \log n = O(n^{c^*})$? Well this is equivalent to saying $\log n = O(n^{c^*-1})$. An important fact here is that for any constant $\epsilon > 0$, $\log n = O(n^\epsilon)$. Since $c^* - 1 > 0$, $n \log n = O(n^{c^*})$ and the first case of the master theorem gives us that $T(n) = \Theta(n^{\log_{10} 11})$