COMP 161 - Lecture Notes - 22 - A high-level View of Complexity

Spring 2014

In these lecture notes we look at algorithm complexity and the use of Big-O notation in computing. The issues covered in these notes are also discussed at length in chapter 10 of the text.

Putting Measurements in Perspective

So you've written some code and used a tool like callgrind to measure the number of instructions executed. What can you do with that number? You can compare it to numbers gathered by different procedures solving the same problem and assess the efficiency differences between the sources of those numbers. In the same manner, you can use it to direct program optimization where you attempt to lower that number though changes to the original code. But, what we haven't really talked about is comparing that number against your efficiency expectations for your code. Put another way, what was your efficiency goal and did you reach it?

Another, more detailed way of approaching this issue would be to imagine a plot of instructions executed[1] by a procedure on a vector versus the size of the vector. As the vector gets bigger, what should happen to the instructions executed? If computational *work* isn't random, then there should be some function $\mathcal{W}(n)$ which maps vector size n to instructions executed. Let's call $\mathcal{W}$ the work function for a procedure[2].

[1] or some other resource measure

[2] Often time is used interchangeably with work and in that case we talk about $\mathcal{T}(n)$, the time function, instead

Knowing the exact specification for the work function $\mathcal{W}$ is probably overkill, it would reflect too many implementation specific details. For example, programs typically involve conditionals, which means their work function is also conditional. This means it's not smooth and is really a whole set of functions, one for each condition. How should we handle this? What's most useful to a programmer? Knowledge of the best case performance, worst case performance, or average case performance?

In general, you'd like to know the *worst case* for the work function $\mathcal{W}$ because it provides the upper bound, or the maximum amount of work. Sure, we expect to see the average case the most and hope to see the best case a lot, but given the choice between a procedure that will take no more than 5 secs and no less than 1 sec and one that takes no more than 5 minutes but no less than 0.00001 sec, the smart program will hedge his or her bets and go with the $[1, 5]$ sec guarantee over the chance at something less than a second. As this example illustrates, if possible, its useful to establish the best case as

well because when combined with the worst case, we get both the upper and lower bound on $\mathcal{W}$.

We've established that having an upper-bound on $\mathcal{W}$ is generally more useful than having the lower bound or some kind of average. Now the question is how much detail do we really need? Sure, we can do some hard analysis and figure out exactly how many instructions will get executed, but this requires knowledge of the compiler and the computer and we really just want to look at our code. Rather than count instructions, let's count *operations*, i.e. the things C++ does. Each operation in C++ should result in some fixed number of machine instructions. If *operator+* takes 5 instructions, and we do 135 *operator+*s, isn't it instruction count just a fixed linear scaling factor? Plus, we can generalize operator counts to different systems and different computers where *operator+* might take more or less than 5 instructions. What we've decided here is that we'll allow for some kind of constant multiplying factor on the work function $\mathcal{W}$ if for no other reason to allow for different operator to instruction translations by different compilers and for different hardware.

Let's recap. We want to get a ballpark feeling for the work function $\mathcal{W}$ that tells us the amount of computational work carried out as a function of vector size. To simplify matters, we'll focus our energy on the worst case scenario and thereby establish an upper-bound for $\mathcal{W}$. We also don't care so much about actual instructions and will instead focus on C++ operations[3] with an understanding that actual work, as in number of machine instructions, is approximately some constant multiple of $\mathcal{W}$. The last step is to realize that we really don't care about $\mathcal{W}$ at all. What's more useful is to find the *slowest growing, basic function that can be drawn above* $\mathcal{W}$. Take a minute to think about that.

[3] or some abstract notion of operations if need be

If I told you I could draw a line such that beyond some vector size n_0 that line was always above the worst case of $\mathcal{W}$, then you'd know that $\mathcal{W}$ must be at worst a line. If it were some other faster growing function like n^2, then it would eventually cross our imaginary bounding line. If we can draw a line above $\mathcal{W}$, then it really doesn't matter what kind of line it is, the thing that matters is that it's a line. Remember, we allow for constant multiples of $\mathcal{W}$ so we can allow for the same thing with our line. This is subtle but significant shift in thought. Our goal is not really to establish the exact details of $\mathcal{W}$ but to find a simple function that acts as an upper-bound on the upper-bound for $\mathcal{W}$. This language is captured in the mathematical notation of Big-O and is the lingua franca[4] for expressing the COMPLEXITY, of program performance.

[4] go look up lingua franca if you don't know the term

A procedure's, or algorithm's, COMPLEXITY is characterized by this smallest basic function that bounds $\mathcal{W}$ above. We can catego-

rize procedure's by complexity such that all procedures that can be bounded by a line are considered equally complex. Essentially, we can reduce the infinite complexity of all possible $\mathcal{W}$ functions to a few basic categories that relate work to easy to understand, common, basic mathematical functions. Mathematical Big-O notation captures this perfectly.

Complexity Classes

We classify the complexity of a procedure by saying something like, "The work of the procedure as a function of the vector size is on the order of f", or $W(n) = O(f(n))$[5]. The function f is the upper bound on the work and it's growth rate therefore characterizes the growth rate of of W. It is also common to say that the complexity of the procedure is f or that the procedure is a "f time"[6] procedure.

[5] so you can read $O(f)$ as "the order of f"

[6] here time is equated to work

A large majority of the procedures we encounter have a work function bounded above by one of the following functions:

1. The CONSTANT complexity procedures where $f(n) = c$ for some fixed value c. We express this as $\mathcal{W}(n) = O(1)$ or just $O(1)$. We use 1 as the representative constant because we're not concerned about constant multipliers.
2. The LOGARITHMIC complexity procedures where $f(n) = \log_b n$. The base of the logarithm actually doesn't matter in Big-O notation so we typically use 2. We express this as $\mathcal{W}(n) = O(\log n)$ or just $O(\log n)$.
3. The LINEAR complexity procedures where $f(n) = n$. We express this as $\mathcal{W}(n) = O(n)$ or $O(n)$.
4. The LINEARITHMIC[7] complexity procedures where $f(n) = n \log n$. We express this as $\mathcal{W}(n) = O(n \log n)$.
5. The QUADRATIC complexity procedures where $f(n) = n^2$. We express this as $\mathcal{W}(n) = O(n^2)$
6. The CUBIC complexity procedures where $f(n) = n^3$. We express this as $\mathcal{W}(n) = O(n^3)$
7. The EXPONENTIAL complexity procedures where $f(n) = 2^n$. We express this as $\mathcal{W}(n) = O(2^n)$
8. The FACTORIAL complexity procedures where $f(n) = n!$. We express this as $mathcal W(n) = O(n!)$.

[7] a blend of linear and logarithmic

Within each class there is an awful lot of nuance. For example, pick any work function where $\mathcal{W}(n) = an + b$, then $W(n) = O(n)$. 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.

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

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 decided that POLYNOMIAL complexity is the benchmark for efficiency. So any complexity class that is a polynomial or better is theoretically efficient. This class includes everything we've seen but $O(2^n)$. 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. To convince yourself of this, go run some representative algorithms on larger and larger n and see what happens[8].

[8] See project 2!

Big-O

We've been using Big-O is pure notation, as a way of naming complexity classes. It is not pure notation. It has a precise mathematical definition.

> $W(n) = O(f(n))$ if and only if there exists some constants n_0 and c such that for all $n \geq n_0$, $W(n) \leq c * f(n)$.

Reading this closely, we see that the mathematical definition matches exactly with how we use it in the context of algorithm complexity. In algorithms research, we really dig into the mathematics of Big-O and other related definitions[9]. For basic understanding of programming complexity we simply need be sure we understand what Big-O brings to the party and what it's supposed to mean in the context of algorithms.

[9] for lower bound, exact bound, tight upper bound, and tight lower bound

When talking about algorithm complexity, people usually use Big-O when referring to the smallest upper bound. So, if $W(n) = 5n - 3$,

then saying $W(n) = O(n)$ is both accurate and helpful in that a linear upper bound is the tightest upper bound you can draw. However, it would not be incorrect to say $W(n) = O(n^2)$ because if you can draw a line above W then you can certainly draw a parabola. Even though the definition of Big-O allows for loose upper bounds, we rarely see it used to present loose bounds. When it is, the author/presenter is usually clear about it and the reason is usually that W is very hard to pin down. For the tasks you're most likely to encounter as a programmer, this is rarely the case.

Discussion of complexity focus on the worst case. Don't confuse an upper bound on the worst case with an exact measure of the expected performance. For example, When sorting n items, bubble sort will do $O(n^2)$ work in the worst case. However, the algorithm is able to determine if the data being sorted is already sorted. When it is, the algorithm terminates after $O(n)$ work. This means that the true work function of bubble sort isn't really quadratic, it's somewhere in the linear to quadratic spectrum. Don't confuse the upper bound of the worst case[10] with the exact, tight bound.

[10] i.e. the greatest upper bound

Reasoning with Big-O

When determining the order of the work function of an algorithm, it's easier to determine the order of each piece and then combine the sum total of the pieces. To do this, we need to know a few basic rules:

We can drop multiplicative constants.

$$O(c * f(n)) = O(f(n)) = c * O(f(n))$$

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

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

In a sum, drop the lower order terms. Where $O(f(n)) < O(g(n))$,

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

Group products.

$$O(f(n) * g(n)) = f(n) * O(g(n))$$

To better understand these let's look at an example.

Example: Linear Search

As the name implies, linear search is a $O(n)$ algorithm for finding a specific key value in a vector. Here we see a pretty standard C++ implementation for integer vectors.

```
int find(const vector<int> &v, int key){

for(unsigned int i(0); i < v.size(); i++){
       if( v[i] == key )
          return i;
}
return -1;
}
```

First we notice that all of the following expressions require constant, $O(1)$ work.

```
unsigned int i(0)
i < v.size()
i++
v[i] == key
```

The *return*s on cost as much as the computation for the return value, and no computation is needed for the literal -1 or accessing the variable i. The *if* only costs as much as the conditional check and the work done if and when the condition is true.

We've established we have several chunks of $O(1)$ work. Now we need to determine when and how often they're executed. Let's start with the *for* loop header.

1. The declaration of i happens once and only once.
2. The conditional check fir $i < v.size()$ happens every time there is a loop *and* one more time when the loop terminates.
3. The $i++$ occurs once per loop.

Now the body of the *for* loop.

1. The conditional check for $v[i] == key$ happens once per loop.

So, the big question is how many loops does the for loop do? It stops when $v[i] == key$ *or* if $i == v.size()$. Remember we're interested in the worst case and that happens when $v[i]$ is never equal to *key*. In that case the loop will loop $v.size()$ times; let's go ahead and say $n = v.size()$.

We can no express the total work of *find* as the sum total of all of that work which is:

$$O(1) + (n+1) * O(1) + n * O(1) + n * O(1)$$

It's time to use those rules:

$$\begin{aligned} O(1) + (n+1) * O(1) + n * O(1) + n * O(1) &= O(1) + O(n+1) + O(n) + O(n) \\ &= O(3n+2) \\ &= O(3n) \\ &= O(n) \end{aligned}$$

First we grouped products. Then we restated the sum of orders as the order of a sum. Next we dropped the lower order term. Finally we dropped the multiplicative constant. The end result was the expected one, our linear search is $O(n)$. Had it been anything else we'd need to check our math or seriously reconsider the implementation because linear search should never be anything other than $O(n)$ in the wost case.

Recap

Worst case complexity is a way of establishing the big picture. Saying a procedure has *quadratic*, or $O(n^2)$, work[11] complexity, merely states that beyond some value of n, the amount of work the procedure will do is bounded above by a quadratic function. Yet despite leaving out so much information, worst case complexity has proven to be an invaluable tool for evaluating and communicating the expected performance of a procedure or algorithm.

[11] or time