2 Runtime Complexity and Asymptotic Analysis

2.1 Asymptotic Analysis

For many of you this section is review. However, there is more formalism here then you have probably seen. But bear with me for a bit, as I promise you that it will be helpful!

Let's form a *partial* ordering on the set of functions $\mathbb{N} \to \mathbb{R}^+$ (functions from natural numbers to positive reals). Let's say for $f, g : \mathbb{N} \to \mathbb{R}^+$, that $f \leq g$ if for all but finitely many n, $f(n) \leq g(n)$.[1] Formally this means the following:

(a) (reflexivity) $f \leq f$ for all f.

(b) (antisymmetry) If $f \leq g$ and $g \leq f$ then $f = g$.[2]

(c) (transitivity) If $f \leq g$ and $g \leq h$ then $f \leq h$.

What differentiates a partial ordering from a *total* ordering is that there is necessity that f and g are comparable. It might be that $f \leq g$, $f \geq g$ or perhaps neither. In a total ordering, we guarantee that $f \leq g$, or $f \geq g$, perhaps both.

By defining this partial ordering, we've given ourselves the ability to define *complexity equivalence classes*.

Definition 2.1 (Big O Notation)**.** Let $f, g : \mathbb{N} \to \mathbb{R}^+$. We say $f \in O(g)$ and (equivalently) $g \in \Omega(f)$ if $f \leq cg$ for some $c > 0$. Here we use '$\leq$' as described previously. Also,

$$O(g) = \{f : \exists c > 0, f \leq cg\} \qquad \Omega(f) = \{g : \exists c > 0, f \leq cg\} \tag{2.1}$$

[1] You might see this in the notation $\exists\ n_0 \in \mathbb{N}$ such that for all $n > n_0$, $f(n) \leq g(n)$. These are in fact equivalent. If $f(n) \leq g(n)$ for all but finitely many n (call them $n_1 \leq \cdots \leq n_m$) then for all $n > n_m$, $f(n) \leq g(n)$. Setting $n_0 = n_m$ completes this proof. For the other direction, let the set of finitely many n for which it doesn't satisfy be the subset of $\{1, \ldots, n_0\}$ where $f(n) > g(n)$.

[2] Careful here! When we say $f = g$ we don't mean that f and g are equal in the traditional sense. We mean they are equal in the asymptotic sense. Formally this means that for all but finitely many n, $f(n) = g(n)$. For example, the functions $f(x) = x$ and $g(x) = \lceil \frac{x^2}{x+10} \rceil$ are asymptotically equal.

First recognize that $O(g)$ and $\Omega(f)$ are *sets* of functions. Additionally, recognize that since $c > 0$, we can divide by it to yield the equivalent definition $g \in \Omega(f)$ if $g \geq c'f$ for some $c' > 0$. Let's discuss equivalence classes and relations for a bit.

Definition 2.2 (Equivalence Relation)**.** We say $\sim$ is an equivalence relation on a set X if for any $x, y, z \in X$, $x \sim x$ (reflexivity), $x \sim y$ iff $y \sim x$ (symmetry), and if $x \sim y$ and $y \sim z$ then $x \sim z$ (transitivity).

Our altered definition for '=' fits very nicely into this definition for equivalence relations. But equivalence relations are more general than that. They work with the definition of equality in a partial ordering above as well. Now, we can bring up the notion of an equivalence class.

Definition 2.3 (Equivalence Class)**.** We call the set $\{y \in X \text{ s.t. } x \sim y\}$, the equivalence class of x in X and notate it by $[x]$.

What does any of this have to do with runtime complexity? The point of all of these definitions about partial ordering and equivalence classes is that $f \in O(g)$ is a partial ordering as well! Go through the process of checking this as an exercise.

Definition 2.4. We say $f \in \Theta(g)$ and (equivalently) $g \in \Theta(f)$ if $f \in O(g)$ and $g \in O(f)$.

$$\Theta(g) = \{f : \exists\, c_1, c_2 > 0, c_1 g \leq f \leq c_2 g\} \tag{2.2}$$

This means that $f \in \Theta(g)$ is an equivalence relation and in particular $\Theta(g)$ is an equivalence class. By now, perhaps you've gotten an intuition as to what this equivalence class means. It is the set of functions that have the same *asymptotic computational complexity*. This means that asymptotically, their values only deviate from each by a constant factor.

This is an incredibly powerful idea! We're now defined ourselves with the idea of equality that is suitable for this course. We are interested in asymptotic equivalence. If we're looking for a quadratic function, we're happy with finding any function in $\Theta(n^2)$. This doesn't mean per se that we don't care about constant factors, its just that its not the concern of this course. A lot of work in other areas of computer science focus on the constant factor. What we're interested in is how to design algorithms with smallest complexity. For example, we will look at problems that look exponentially hard but in reality might have

polynomial time algorithms. That jump is far more important than a constant factor.

We can also define o, ω notation. These are stronger relations. We used to require the existence of some $c > 0$. Now we require it to be true for all $c > 0$.

Definition 2.5 (Little O Notation)**.** Let $f, g : \mathbb{N} \to \mathbb{R}^+$. We say $f \in o(g)$ and (equivalently) $g \in \omega(f)$ if $f \leq cg$ for all $c > 0$. Here we use '$\leq$' as described previously. Equivalently,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0. \tag{2.3}$$

We can also discuss asymptotic analysis for functions of more than one variable. I'll provide the definition here for Big O Notation but its pretty easy to see how the other definitions translate.

Definition 2.6 (Multivariate Big O Notation)**.** Let $f, g : \mathbb{N}^k \to \mathbb{R}^+$. We say $f \in O(g)$ and (equivalently) $g \in \Omega(f)$ if $f(x_1, \ldots, x_k) \leq cg(x_1, \ldots, x_k)$ for some $c > 0$ for all but finitely many tuples $(x_1, \ldots, x_k)$.[3]

A word of warning. Asymptotic notation can be used incredibly abusively. For example you might see something written like $3n^2 + 18n = O(n^2)$. In reality, $3n^2 + 18n \in O(n^2)$. But the abusive notation can be helpful if we want to 'add' or 'multiply' big O notation terms. You might find this difficult at first so stick to more correct notations until you feel comfortable using more abusive notation.

2.2 Random Access Machines and the Word Model

Okay, so we've gotten through defining Big O Notation so now we need to go about understanding how to calculate runtime complexity. The question to ask is 'what computer are we thinking about when calculating runtime complexity?'. Are we allowed to use a parallel computing system here? These are all good questions and certainly things to be thinking about. However, for the intents of our class, we are not going to be looking at parallelization. We are going to assume a single threaded machine.[4] Is this a quantum

[3] It really helps me to think of this graphically. Essentially, this definition is saying that the region for which $f \not\leq cg$ is bounded.

[4] If you consider a multi threaded machine with k threads, then any computation that takes time t to run on the multi-threaded machine takes at most kt time to run on the single threaded machine. And conversely, any computation that takes time t to run on the single threaded machine takes at most t time

machine? Also, interesting but in this case outside the scope of this course.

The most general model we could use would be a single headed one tape Turing machine. Although equivalent in terms of *decidability*, we know that this is not an efficient model particularly because the head will move around too much and this was incredibly inefficient. It was a perfectly reasonable model for us to use in CS 21 because the movement of the head can be argued to not cause more than a polynomial deviation in the complexity which was perfectly fine with us as we were really only concerned about the distinctions between P,NP,EXP, etc. Now, however, we do care about the degrees of polynomials.

To define a model for computation, we need to define the costs of each of the basic operations we will use in our other proofs. We can start from the ground up and define the time it takes to flip a bit, the time it takes to move the head to a new bit to flip, etc. and build up our basic operations of addition, multiplication from there and then move on to more complicated operations and so forth. This we will quickly find becomes incredibly complicated and tedious. However, this is the only actual method of calculating the exact time of an algorithm. What we will end up using is a simplification, but one that we are content with. When you think about an algorithm's complexity, you must always remember what model you are thinking in. For example, I could define a model where sorting is a $O(1)$ operation. This wouldn't be a very good model but I could define it anyways. Luckily, the models we're going to use have some logical intuition behind them.

We are going to be using two different models in this class. The most common model we will be working in is the *Random-Access Machine (RAM) model*. In this model, instructions are operated on sequentially with no concurrency. Furthermore, a location in memory is editable an arbitrarily many number of times and the access of any part of the memory is done in constant time.[5] We further assume that reading and writing a single bit takes constant time.

to run on the multi-threaded machine. If k is a constant, this doesn't affect asymptotic runtime. The number of threads needs to scale with the problem in order for it to change the complexity. An interesting alg. to the effect is the ability to calculate determinants of $n \times n$ matrices in polylog(n) time on $\mathbf{poly}(n)$ threads.

[5] This is the motivation of the name Random-Access. A random bit of memory can be accessed in constant time. In a Turing machine only the adjacent bits of the tape can be accessed in constant time.

Recall that number between 0 and $n-1$ can be stored using $O(\log n)$ bits. So addition and subtraction take naïvely $O(\log n)$ time and multiplication takes $O((\log n)^2)$ time.[6] This model is the most accurate because it most closely reflects how a computer works.

However, as I said before this can get really messy. We will also make a simplification which we call the *word model*. In the word model, we assume that all the words can be stored in $O(1)$ space. There are numerous intuitions behind the word model but the most obvious is how most programming languages allocate memory. When you allocate memory for an integer, languages usually allocate 32 bits (this varies language to language). These 32 bits allow you to store integers between -2^{31} and $2^{31}-1$. This is done irrespective of the size the integer. So, operations on these integers are irrespective of the size and are moreover $O(1)$.

This model is particularly useful if we want to consider the complexity of higher order operations. A good example is matrix multiplication complexity. A naïve algorithm for matrix multiplication runs in $O(n^3)$ for the multiplication of two $n \times n$ matrices. By this we mean that the number of multiplication and addition operations applied on the elements of the matrices is $O(n^3)$.[7] The complexity of the multiplication of the elements isn't directly relevant to the matrix multiplication algorithm itself and can be factored in later.

For the most part, you will be able to pick up on whether the RAM model or the Word model should be used. In the example in the previous chapter, we used the Word model. Why? Because, the problem gave no specification as to the size of the elements $x_1, \ldots, x_n$. **If specified, then it implies the RAM model. Otherwise, use the word model.** In situations where this is confusing, we will do the best to clarify which model the problem should be solved in.

[6] You can also think about this as adding m bit integers takes $O(m)$ time. And you can store numbers as large as 2^m using m bits.

[7] Later we will show how to get this down to $O(n^{\log_2 7})$.