Introduction to Asymptotic Analysis

Max Lin

There are an infinite number of ways to write code that does a particular task. The question then becomes: what is the most efficient way to do this particular task? We would need a way of telling how efficient some code is, and that is where asymptotic analysis comes in. It is a way of classifying code based on the general number of computer steps some code requires from start to finish. Perhaps you've seen notation similar to $\mathcal{O}(E \log V)$, $\Omega(1)$, or $\theta(n^3)$. These are all examples of orders of growth or, in other words, how many computer steps a program takes as some n (usually a variable relating to the input) increases in size. We write the number of steps in terms of mathematical equations and classify it by one of three notations $(\mathcal{O}, \Omega, \theta)$, which we'll get into later on.

Understanding n

When we think about how long some code runs for, typically we think about it in terms of n. We can think of n as something related to the input for some code, whether its the actual input itself or something else like the number of bits the input takes up. Sometimes we'll see V and E, instead of n, for analysis where the code is related to graphs (because we have 2 types of variables, vertices and edges). n is just the standard variable computer scientists like to use when analyzing code. It is replaceable by any other letter whenever convenient, and we can introduce multiple relevant variables if needed, like for graphs analysis.

Having this variable point of reference is important because we can represent the growth of some code in terms of n and simple mathematical equations. Basically, a program will take some amount of computer steps usually depending on n or some other, possibly multiple, variable(s). But it is also possible for a program to run a finite number of steps regardless of any variable. We'll get into this in the next section, but for now understand that we can model the number of computer steps as a mathematical equation such as $n^2 + 3n$ or 2^n.

Runtime Analysis

As we said earlier, we can model the number of computer steps with some equation with respect to n. Realistically, we are formulating an equation that is representative of an algorithm's growth as n increases in size. Here we will look at the standard classifications of runtime analysis.

Convention

Whenever we do asymptotic analysis, we want to look at simple equations of growth. It is much harder to analyze and classify messy equations. For example, it is difficult to say whether $100n^{23} + 3n^{10} \log^{13}(n)$

or $50n^22 + 40n^{20}\log^9(3n)$ grows faster as n increases. So when we are doing runtime analysis, we typically drop lower degree terms (or we keep the fastest growing term) since the fastest growing term will overshadow the effects of the slower growing terms. For example, the effects of n^2 is negligible in the equation $n^{100} + n^2$. We also drop the constant scalar in front of the fastest growing term. This is because any two equations that are on the same order of growth grows at the same rate. The only difference could be a scalar factor, but the rate of growth is still the same. Since we are classifying equations based on their growth rate, the scalar is unimportant. As an example, n^2 is the same order of growth as $100n^2$, but, for example, $n \log n$ is not on the same order of growth as n or $\log n$. Notice how we said we can drop constant scalars. Neither n nor $\log n$ are constants. $n \log n$ is its own order of growth.

Constant Time

Constant time algorithms are the class of algorithms that take a finite number of steps regardless of the input. That is, regardless of what n is, the algorithm will take an independent constant number of steps. For example, an algorithm that always runs in 1000 steps, regardless of the input, we say that algorithm takes constant time. So if our input is $n = 1$, the algorithm will take 1000 steps. If our input is $n = 1000000000$, our algorithm will still run in 1000 steps. So our model function is $f(n) = 1000$. Recall that we can drop constant scalars, so we can simply to $f(n) = 1$. This is what it means to be a constant time algorithm and we write the runtime as $\mathcal{O}(1)$. Don't worry about the $\mathcal{O}$ notation, we'll talk about it later on. For now, just think of $\mathcal{O}$ as one notation for order of growth.

Example:

```
def even_or_odd(n):
    """Determines if n is even or odd"""
    parity = n % 2  //Find n modulo 2
    if parity is 0:
        return "Even"
    return "Odd"
```

Notice how regardless of what n is, we just find n modulo 2 and return either even or odd. No matter how large or small n is, this algorithm will always take 3 steps, or a constant number of steps irrespective to n: (1) performing n modulo 2, (2) checking if the parity is even, and (3) returning either even or odd. Therefore we say that this algorithm runs at constant time, or $\mathcal{O}(1)$.

As a note, constant time is the best runtime computer scientists can hope for in a program, however its difficult, and usually impossible, to bring complex algorithms down to constant time. But there are some widely used constant time algorithms that exists, like hashing.

Linear Time

Unlike constant time algorithms, where each program takes a finite number of steps irrespective of n, linear time algorithms take a proportional number of steps to n. For example, consider finding the largest number in a list of length 300. The way we would do this would be to have a variable that tells us the

largest number thus far and we would update this variable every time we see a larger number as we traverse the list. So for each number in the list, we compare it with the largest number so far, and update it only if we need to. So at most, that's 2 steps for every number in the list, or at most 600 steps. But this isn't constant time! Remember that we specified that our list is length 300. But what if it its some arbitrary length n. Then we need to do at most 2 steps for each n numbers, or $2n$ steps. We can see that this algorithm is actually linear.

```
def largest_number(list):
    """Finds largest number in the list of size n"""
    largest_number_so_far = -infinity
    for every number in list:
        if current number larger than largest_number_so_far:
            largest_number_so_far = current number
    return largest_number_so_far
```

Another great example is dealing with graphs. Suppose we have a graph with $|V|$ vertices and $|E|$ edges and we wanted to do a graph traversal like DFS or BFS. For those that don't know either DFS or BFS, both of these are graph traversal algorithms. As an overly simplified explanation, you start at an arbitrary vertex and you move to another vertex based on the connecting edges between vertices you've visited already until you've reach every vertex. The order in which vertices are visited vary between the two algorithms. It is important to note that you visit each vertex once, and you consider each edge once (there is some way of checking whether you've been to a vertex or not . . . if you've visited a vertex already, there is no need to visit it again). Since we visit each vertex and each edge once, these traversal algorithms are linear or grows proportional to the number of vertices or edges in the graph.

Polynomial Time

Polynomial algorithms encompass many categories of code. We consider any algorithm that runs at $\mathcal{O}n^k$ where $k > 0$ to be polynomial. That means that linear algorithms are also polynomial time algorithms. But any algorithms that takes steps proportional to a power of n is a polynomial time algorithm. The most common way to recognize polynomial algorithms is with nested loops where each loop runs proportionally to n. For example, suppose we wanted to make a list of all pairwise lock combinations from a list of numbers. Then we would do a nested loop:

```
def pairwise_combination(list):
    """Makes all pairwise combinations from the input list of size n"""
    for all numbers α in list:
        for all numbers β in list:
            print "(α, β)"
```

Linear runtime algorithms would only need one loop, instead of a nested loop.

There are also some pretty interesting analysis tricks when dealing with polynomial and eventually ex-

ponential time algorithms and that's using convergence of series. Take for example the selection sort algorithm. Given a list of numbers, we would like to return a sorted version of the list. The selection sort algorithm goes through the list and finds the smallest element and swaps it with the first element of the list. With the updated list, find the minimum of the sublist starting at the second element, and again swap this minimum with the second element, and do the same for the third element and so on until you have a sorted list. How many operations have we done? Well when we were finding the first minimum, we needed to look through a list of size n. But when we needed to find the second minimum (or second smallest number), we ignored the first element (since we knew it was the minimum). So then we needed to look through a list of size $n-1$, then $n-2$, and so on until we had a list of size 1, which meant that we were done and we had a sorted list. So we had a total number of $n + (n-1) + (n-2) + \ldots + 1$. For those that know series, this converges to $\frac{n(n+1)}{2}$. For those that don't, lets say that this series equals some total number of operations or steps z. So $z = n + (n-1) + (n-2) + \ldots + 1$. But also by the commutative property of addition, $z = 1 + 2 + \cdots + (n-1) + n$. If we add these two equations together:

$$\begin{array}{rcccccccc} z = & n & + & (n\text{-}1) & + \ldots + & 2 & + & 1 \\ z = & 1 & + & 2 & + \ldots + & (n\text{-}1) & + & n \\ \hline 2z = & (n{+}1) & + & (n{+}1) & + \ldots + & (n{+}1) & + & (n{+}1) \end{array}$$

$$2z = n(n+1)$$

$$z = \frac{n(n+1)}{2}$$

Notice that $\frac{1}{2}n(n+1) = \frac{n^2+n}{2}$. Recall about the convention we use when analyzing these equations: (1) drop lower degree terms and (2) drop scalar factors. So really, we see that selection sort has a runtime of $\mathcal{O}(n^2)$. There are plenty of analysis that requires these types of tricks found in math. This is just one such example.

Logarithmic Time

Logarithmic time algorithms are the algorithms that lie between polynomial time and constant time algorithms. Specifically, the base logarithmic runtime $\mathcal{O}(\log n)$ lies between linear and constant time, but it's common to see a combination of polynomial and logarithmic runtimes, for example $n \log n$. Logarithmic algorithms are common in algorithms when we are able to break down our search area. For example, given a sorted list of books (according to author name), we can quickly find a specific book in logarithmic time. Instead of scanning through all the authors from start to finish (which is a linear runtime), we want to start from the middle of our list. Does the author we are looking for come before the middle, after the middle, or is she exactly the middle author. Once we've determine this, we can ignore half of the books! Why? Suppose the middle author name is Lin, and we are looking for Xiang. We have no reason to look at the books that came before Lin because we know Xiang comes later than Lin alphabetically.

```
def binary_searching(list, desired_author):
    left_index_bound = 0
    right_index_bound = len(list) - 1
    middle = (right_index_bound - left_index_bound) / 2 //floor divide to get an integer
    if list[middle] is desired author: //if we found the author, then return the position
        return middle
```

```
if list[middle] comes before desired author: //adjusts the list to latter half via splicing
    return binary_searching(list[middle: right_index_bound], desired_author)
if list[middle] comes after desired author: //adjusts the list to the former half via splicing
    return binary_searching(list[left_index_bound:middle], desired_author)
```

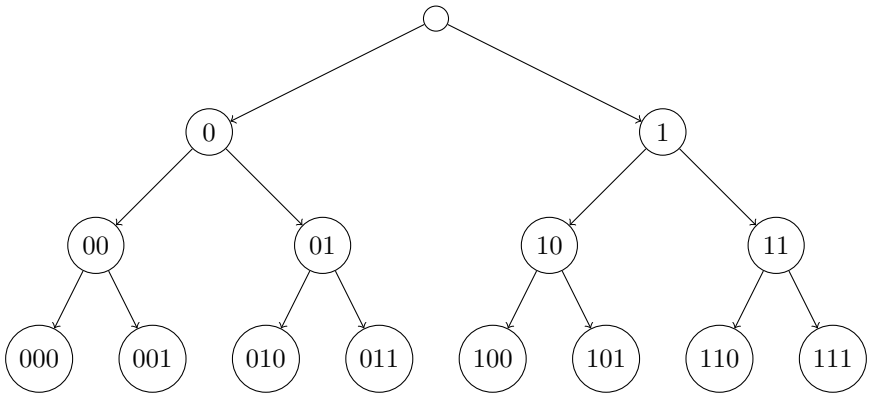
Logarithms are also commonly seen in sorting algorithms (better algorithms that the selection sort algorithm mentioned previously), however we wont discuss them here. If interested, I'd suggest looking up mergesort or heapsort.

Another convention

There is another convention that should be mentioned for logarithms. As we know from math, there are many different logarithmic bases. However, when we do asymptotic analysis, all these bases are nearly the same in how they grow. So we treat all logarithms regardless of base the same. That is, $\mathcal{O}(\log_2(n)) = \mathcal{O}(\log_{100}(n)) = \mathcal{O}(\log(n))$.

Exponential Time

The opposite of logarithmic runtime, as in mathematics, is exponential runtimes. These are the worst and fastest growing algorithms in computer science. These kinds of algorithms are the ones computer scientists try to avoid since they take a really long time to complete as n becomes increasingly large. One of the basic exponential time algorithm is the recursive calculation of the Fibonacci sequence without memoization. Algorithms that deal with power sets like considering all possible n bit numbers are also considered exponential. The best way to visualize exponential time algorithms are with trees that grow by the base factor at each level. For example, with binary numbers, we can draw the following tree:



Each level of the above tree shows all possible binary numbers with n bits on level n. The above tree only goes up the level 3, or all possible binary numbers with 3 bits, but you could imagine how this tree could grow really wide really quickly. In this examples, the number of binary numbers with n bits is 2^n.

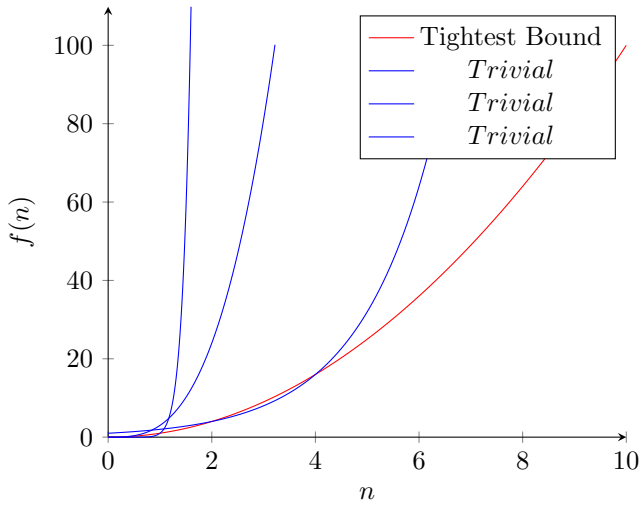
$\mathcal{O}$, Ω, and θ

In asymptotic analysis, there are three ways to classify an algorithm; we can use $\mathcal{O}$, Ω or θ. These notations are used to describe or classify the limits of an algorithm's behavior. We've seen the $\mathcal{O}$ notation al-

ready, but never discussed what it means. We will do so here.

Big-$\mathcal{O}$ Notation

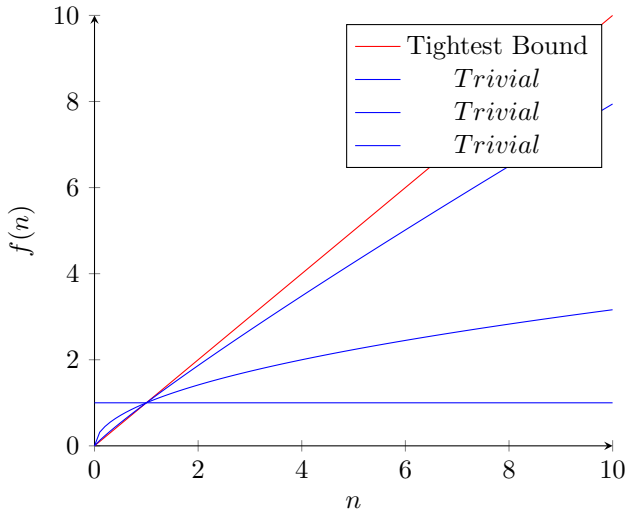
The most common way of thinking about $\mathcal{O}$ is that it symbolizes an upper-bound to how rate of growth or number of steps increases as n increases. That is, for some algorithm with any input n, there are mathematical functions that have **at least** the same growth as the mathematical function representation for the growth of the algorithm. We can think of Big-$\mathcal{O}$ as the subset of all functions that grow faster than or equal to the algorithm we are analyzing. There are infinitely many trivial mathematical functions that grow faster than some given function (and can all be correctly written with $\mathcal{O}$), so we should aim to find the tightest bound.



The tightest bound should be thought of as the mathematical function that is the ***worst case*** for some algorithm. To clarify, some algorithms have certain inputs that they don't do well on, like insertion sort on reverse sorted sets of objects (which runs $\mathcal{O}(n^2)$). It should be noted that some algorithms would have the same runtime or mathematical representation regardless of the input, so the worst case is simply the average case.

Big-Ω Notation

Similarly, we have Ω, which is the lower-bound to how fast some algorithm grows. For some algorithm with some dependency on n, there are mathematical functions that have **at most** the same growth as the mathematical representation of the algorithm's growth. In other words, Big-Ω is the class of functions that grow slower than or equal to the algorithm we are analyzing. Like Big-$\mathcal{O}$, there are infinitely many trivial mathematical functions that grow slower than some given function, which are all validly Ω, we aim to find the tightest bound.



In similar fashion, the tightest bound when dealing with Big-Ω is the mathematical function that is the ***best case*** for some algorithm. Some inputs allow for quicker completion, like already sorted lists for insertion sort (which is $\Omega(n)$)

Big-θ Notation

Now, what happens if the worst case and the best case are the same? That is where Big-θ comes in. Officially, whenever θ is used, it means that the worst case and the best case are the same order of growth. One example is mergesort, which always runs at $n \log n$, no matter the input. So we can say that mergesort has a runtime of $\theta(n \log n)$. We use θ when we want an asymptotically tight bound for our algorithm.

Just to point out, there are also algorithms where best case and worse case are different. If you've noticed, in the discussion of the $\mathcal{O}$ and Ω, insertion sort has a different best case and worse case. So we cannot establish a θ runtime for it.