Asymptotic Analysis[1]

Last week, we discussed how to present algorithms using pseudocode. For example, we looked at an algorithm for singing the annoying song "99 Bottles of Beer on the Wall" for arbitrary values of 99.

```
BottlesOfBeer(n):
  for i ← n down to 1
      Sing "i bottles of beer on the wall, i bottles of beer,"
      Sing "Take one down, pass it around, i − 1 bottles of beer on the wall."

  Sing "No bottles of beer on the wall, no bottles of beer,"
  Sing "Go to the store, buy some more, n bottles of beer on the wall."
```

Along with describing your algorithms, you also need to justify their correctness with a proof, and analyze their running time. There's nothing to justify here; we can just declare that these are the lyrics to the song by definition! However, we can still discuss running time.

It's hard to give the *exact* running (singing?) time of this song, because different people may sing at different speeds. Perhaps I sing $r\ e\ a\ l\ l\ y\ r\ e\ a\ l\ l\ y\ s\ l\ o\ w\ l\ y$ while you sing *reallyreallyfast*. What we could try, however, is to count the number of times we perform certain instructions or reach certain milestones in the 'code'. For example, it's easy to see that we sing the word "beer" three times for each value of i plus three more times at the end for $3n + 3$ times total.

But let's look at a more interesting example. The following song is usually sung as the Christmas holiday approaches.[2]

[1] Supplementary lecture notes by Kyle Fox for UT Dallas course CS 4349.004. August 27, 2018.

[2] Traditional entries in the $gifts$ array include turtle doves, French hens, calling birds, golden rings, geese a-layin', swans a-swimmin', maids a-milkin', lords a-leapin', ladies dancin', pipers pipin', and drummers drummin'.

```
NDaysOfChristmas(gifts[2..n]):
  for i ← 1 to n
      Sing "On the ith day of Christmas, my true love gave to me"
      for j ← i down to 2
          Sing "j gifts[j]"
      if i > 1
          Sing "and"
      Sing "a partridge in a pear tree."
```

In this case, a good approximation on the singing time would be the number of times the singer mentions the name of a gift, equal to $\sum_{i=1}^{n} i = n(n+1)/2$. We might also observe that $\sum_{i=1}^{n}\sum_{j=1}^{i} j = n(n+1)(n+2)/6$ gifts are given in total; fortunately, the singer does not have to count each individual gift, so the first expression more accurately represents the singing time.

But let's say now we want to know which song will take longer to sing. If we only look at small values of n, say $n < 6$, we might assume BottlesOfBeer takes longer to sing; there are strictly more beers mentioned than gifts. However, for $n > 6$, there are always more gifts mentioned than beers. Intuitively, the number of gifts starts smaller than the number of beers, but it grows with increasing n at a much higher rate thanks to the $n^2/2$ term in the total number of gifts. We really only care about singing time for larger values of n ("5 Bottles of Beer on the Wall" is a pretty short song), so we need some way to emphasize these quickly growing terms and compare singing/running times for large values of n.

1 Asymptotic Notation

1.1 Θ-notation

The most common way to express algorithm running times is to express them using asymptotic notation. Let $g(n) : \mathbb{N} \to \mathbb{R}^+$ be a positive function over the natural numbers. We'll begin with a way to express exactly (up to constant factors) how other functions grow relative to $g(n)$ called ***Θ-notation***. Formally, $\Theta(g(n))$ is a set of functions defined as

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}.$$

In other words, $f(n) \in g(n)$ if and only if they differ by constant factors for all sufficiently large values of n. In these cases, we say $g(n)$ is an ***asymptotically tight bound*** for $f(n)$. Often, we'll simplify things by writing $f(n) = \Theta(g(n))$ claiming $f(n)$ *is* $\Theta(g(n))$ whenever we formally mean to say $f(n) \in \Theta(g(n))$. One special case that arises often is $g(n) = 1$. Whenever we have a function $f(n)$ with $f(n) = \Theta(1)$, we can conclude $f(n)$ lies sandwiched between two constants $c_1, c_2 > 0$ for all sufficiently large values of n.

It is not difficult to verify (with a bit of algebra) that the first song given above mentions beer $\Theta(n)$ times while the second song names a gift $\Theta(n^2)$ times. And because the singing times of these two songs are proportional to these two events, the singing times must be $\Theta(n)$ and $\Theta(n^2)$, respectively, as well. But we still don't have a formal handle on which song takes longer to sing.

1.2 O and Ω-notation

To better compare different functions and running times, we need to give possibly loose upper and lower bounds. The first of these comes from ***O-notation*** (pronounced big-oh notation), which you are likely the most familiar with.

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \\ \text{such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}.$$

Again, we may write $f(n) = O(g(n))$ when we really mean $f(n) \in O(g(n))$. With a bit of algebra, we can verify $3n + 3 = O(n(n + 1)/2)$. In other words, it takes *at most* as long to sing BottlesOfBeer as it does to sing NDaysOfChristmas for large values of n. Note, however, that it *is not* the case that $3n + 3 = \Theta(n(n + 1)/2)$. O-notation merely gives an upper bound on function or running time growth. There is no guarantee that this upper bound is tight, much like how $1 \leq 3$ does not imply $1 = 3$.

This loose upper bound can be useful, however, especially when expressing the running time of algorithms. Given an array of n integers, the well-known sorting algorithm InsertionSort runs in $\Theta(n^2)$ time in the *worst case*, but if the input is already sorted, then the running time is actually $\Theta(n)$. It is much easier to just claim the running time as $O(n^2)$, because that statement is true for *all* inputs of size n. *In general, you should express your algorithm running times using O-notation, and using as simple a function as possible while still giving the best bound you can.* State that your algorithm runs in $O(n^3)$ time, not $O(n^3 - 23n^2 + 4n + 3)$ time.

Rarely, we may want to give asymptotic lower bounds as well. For example, we may want to declare that an algorithm uses a certain minimum running time for all inputs or that *any* algorithm to a problem has that minimum running time. For these instances, we use ***Ω-notation*** (big-omega notation).

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \\ \text{such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}.$$

Again, we may write $f(n) = \Omega(g(n))$ when we really mean $f(n) \in \Omega(g(n))$. As you might expect, $n(n + 1)/2 = \Omega(3n + 3)$, so it takes *at least* as long to sing NDaysOfChristmas as it does to sing BottlesOfBeer for large values of n.

1.3 o and ω-notation

Intuitively, O and Ω-notation provide a loose inequality between the growth of functions or algorithm running times. Occasionally, we'll want strict inequalities instead. For these cases, we use o-notation (little-oh notation) or ω-notation (little-omega notation).

$$o(g(n)) = \{f(n) : \text{for any constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}$$

$$\omega(g(n)) = \{f(n) : \text{for any constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}$$

In other words, no matter which constant c you choose, $g(n)$ will eventually become bigger or smaller than $f(n)$ and they will continue to separate as n grows larger. For example, $2n = o(n^2)$, because $2n < cn^2$ whenever $n > 2/c$, no matter which $c > 0$ you choose. However, it is not the case that $2n^2 = o(n^2)$, because you cannot claim $2n^2 < 3n^2$ for any value of $n > 0$. A bit of algebra implies $3n + 3 = o(n(n + 1)/2)$. We finally have formal confirmation that BottlesOfBeer takes strictly less time to sing than NDaysOfChristmas assuming n is sufficiently large.

2 Analyzing Algorithms

A key advantage of doing asymptotic analysis for algorithms instead of counting the exact number of operations is that we can be a little lazier with our math. For example, we don't need to know there are exactly $n(n + 1)/2$ mentions of the name of a gift in NDaysOfChristmas. An easier analysis would be to say there are n iterations of the outer loop, and each iteration names at most n gifts, so there are at most n^2 gifts named. The running time is $O(n^2)$. Similarly, we can provide a lower bound on the singing time by observing that at least $n/2$ iterations of the outer loop involve at least $n/2$ gift names, implying there are at least $n^2/4 = \Omega(n^2)$ gift names. As stated earlier, the total singing time is in fact $\Theta(n^2)$.

Not all algorithms will be this straightforward to analyze. However, we will go over a variety of techniques to help us in analyzing different styles of algorithms. For example, we will discuss solving recurrences in one of the next two lectures, and doing so will help us analyze divide-and-conquer algorithms. Hopefully through a combination of seeing examples from class and your practice on the homework, the runtime analysis part of algorithm design will become relatively easy compared to the rest of the process.

3 Useful Facts

We'll conclude with some additional facts about asymptotics and common functions that will come up throughout the course. Unless specified otherwise, we'll assume throughout

that any functions we mention are positive for sufficiently large n.

3.1 Comparing functions

Many of the properties are you used to seeing from comparison of real numbers also apply to asymptotics.

Transitivity $f(n) = x(g(n))$ and $g(n) = x(h(n))$ implies $f(n) = x(h(n))$ where x is any one of the five $\Theta, O, \Omega, o, \omega$ discussed above.

Reflexivity $f(n) = \Theta(f(n))$, $f(n) = O(f(n))$, and $f(n) = \Omega(f(n))$.

Symmetry and transpose symmetry $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. Similarly, $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$, and $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.

3.2 "Algebra"

Suppose $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$. Then,

$$\begin{aligned} c \cdot f_1(n) &= \Theta(f_1(n)) \text{ for any constant } c, \\ f_1(n) + f_2(n) &= \Theta(g_1(n) + g_2(n)), \\ f_1(n) \cdot f_2(n) &= \Theta(g_1(n) \cdot g_2(n)), \text{ and} \\ f_1(n) + f_2(n) &= \Theta(\max\{g_1(n), g_2(n)\}). \end{aligned}$$

The first equation applies to O and Ω as well. The remaining equations apply to all five pieces of asymptotic notation.

3.3 Important growth functions

A ***polynomial in n of degree d*** is a function $p(n) = \sum_{i=0}^{d} a_i n^i$ where each a_i is a ***coefficient*** of the polynomial and $a_d \neq 0$. We have $p(n) = \Theta(n^d)$. For any real constants $\beta > \alpha \geq 0$, we have $n^\alpha = o(n^\beta)$. A function $f(n)$ is ***polynomially bounded*** if $f(n) = O(n^k)$ for some constant k.

Exponential functions grow faster than polynomial functions, and the base of the exponential matters. In particular, $n^k = o(a^n)$ for any constant k and constant $a > 1$. Also, $a^n = o(b^n)$ for any constants $b > a > 0$.

On the other hand, logarithmic functions grow slower than polynomials, even ones that are in $\Theta(n^a)$ for really small constant $a > 0$. A function $f(n)$ is ***polylogarithmically bounded*** if $f(n) = O(\log^k n)$ for some constant k. We have $\log^k n = o(n^a)$ for any constant k and constant $a > 0$. We will generally use log to denote the logarithm of base 10, lg to

denote the logarithm of base 2, and ln to denote the natural logarithm of base e (Euler's constant). However, from earlier facts we may observe that

$$\log_a n = \frac{\log_b n}{\log_b a} = \Theta(\log_b n)$$

for any constants $a, b > 1$. In other words, the base of the logarithm does not matter when doing asymptotics, as long as that base is a constant.

In the context of algorithm design, correct algorithms with exponential running times are usually easy to find; these algorithms often correspond to some kind of "brute force" solution where we try every possible output and return the best one. We usually don't consider an algorithm to be "efficient" unless it has a polynomial running time, but even then, we prefer to get the running time as close to linear ($O(n)$) as possible. When the input is very special, say a sorted array, we may be able to avoiding reading the whole thing and achieve an even better polylogarithmic running time.