CS 124 Section #1 **Big-Oh and the Master Theorem** 1/30/2017

1 Big-Oh Notation

1.1 Definition

Big-Oh notation is a way to describe the rate of growth of functions. In CS, we use it to describe properties of algorithms (number of steps to compute or amount of memory required) as the size of the inputs to the algorithm increase. The idea is that we want something that is impervious to constant factors; for example, if your new laptop is twice as fast as your old one, you don't want to have to redo all your analysis.

$f(n)$ is $O(g(n))$	if there exist c, N such that $f(n) \le c \cdot g(n)$ for all $n \ge N$.	f "$\le$" g
$f(n)$ is $o(g(n))$	if $\lim_{n\to\infty} f(n)/g(n) = 0$.	f "$<$" g
$f(n)$ is $\Theta(g(n))$	if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$.	f "$=$" g
$f(n)$ is $\Omega(g(n))$	if there exist c, N such that $f(n) \ge c \cdot g(n)$ for all $n \ge N$.	f "$\ge$" g
$f(n)$ is $\omega(g(n))$	if $\lim_{n\to\infty} g(n)/f(n) = 0$.	f "$>$" g

Note that in the definition above, all we care about is the **long term** behavior of f versus g. That is why in the capital letters O and Ω, we have $n \ge N$, and in the lowercase letters o and ω, we have are taking the limit as $n \to \infty$.

Keep in mind that $o \Rightarrow O$ and $\omega \Rightarrow \Omega$.

1.2 Notes on notation

When we use asymptotic notation within an expression, the asymptotic notation is shorthand for an unspecified function satisfying the relation:

- $n^{O(1)}$ means $n^{f(n)}$ for some function $f(n)$ such that $f(n) = O(1)$.
- $n^2 + \Omega(n)$ means $n^2 + g(n)$ for some function $g(n)$ such that $g(n) = \Omega(n)$.
- $2^{(1-o(1))n}$ means $2^{(1-\epsilon(n))\cdot g(n)}$ for some function $\epsilon(n)$ such that $\epsilon(n) \to 0$ as $n \to \infty$.

When we use asymptotic notation on both sides of an equation, it means that for all choices of the unspecified functions in the left-hand side, we get a valid asymptotic relation:

- $n^2/2 + O(n) = \Omega(n^2)$ because for all f such that $f(n) = O(n)$, we have $n^2/2 + f(n) = \Omega(n^2)$.
- But it is not true that $\Omega(n^2) = n^2/2 + O(n)$ (e.g. $n^2 \ne n^2/2 + O(n)$).

The above is quite subtle, but it is important to think of the equal sign as more of a "belongs to the family of functions described by the right hand side" rather than a strict equality.

Exercise 1. Using Big-Oh notation, describe the rate of growth of each of the following:

- n^{124} vs. 1.24^n
- $\sqrt[124]{n}$ vs. $(\log n)^{124}$
- $n \log n$ vs. $n^{\log n}$

- $\sqrt{n}$ vs. $2^{\sqrt{\log n}}$
- $\sqrt{n}$ vs. $n^{\sin n}$

Solution
All the bounds below can be made "looser" if desired:

- $\boxed{n^{124} = o(1.24^n)}$ because as we saw in class, polynomials grow slower than exponentials (with base > 1). Note that we choose little o because it is the best bound we can provide. Using big O would be correct, but is not the best we can say about their relationship.
- $\boxed{\sqrt[124]{n} = \omega((\log n)^{124})}$ by a similar reasoning that polynomials are asymptotically larger than any logarithmic (or power of a logarithm).
- $\boxed{n \log n = o(n^{\log n})}$ because when $n > 4$, then $n^{\log n} > n^2$. We know that $n \log n = o(n^2)$ because $\log n = o(n)$. Therefore, we have that $n \log n = o(n^{\log n})$. This method of finding an intermediate comparison (i.e. n^2 in this case) is quite useful.
- $\boxed{\sqrt{n} = \omega(2^{\sqrt{\log n}})}$ by using a substitution. It is easy to see that if I chose n in a particular form, namely 2^{k^2}, for some integer k, then the two sides simplify nicely.

  $$\sqrt{n} \to 2^{k^2/2} \text{ and } 2^{\sqrt{\log n}} \to 2^k$$

  And now, taking the limit and making the comparison is easy:

  $$\lim_{k\to\infty} \frac{2^{k^2/2}}{2^k} = \lim_{k\to\infty} 2^{k^2/2-k} = \infty$$

  and therefore the correct relation is ω.
- $\boxed{\sqrt{n} \ ? \ n^{\sin n}}$ aren't related. In particular, $n^{\sin n}$ is bounded by 0 and n, but oscillates between the two. It is **not** true that between any two functions f and g, there must a big-Oh relationship between then.

Exercise 2. Which of the following hold?

- $f(n) = \omega(2^n)$ implies that $f(n) = \Omega(n^2)$
- If $g(n) = o(f(n))$, then $f(n) + g(n) = \Theta(f(n))$ (You can assume f and g are positive functions)

Solution
We have:

- $\boxed{True}$. Let $f(n) = \omega(2^n)$. Then, we have that $f(n)/2^n \to \infty$ as $n \to \infty$. But then for all $c > 0$, there exists N such that $f(n) \geq c2^n$ for $n \geq N$. Pick $c = 1$ and the appropriate N. Then, $f(n) \geq n^2 \leq 2^n$ for all $n \geq \max(2, N)$, so $f(n) = \Omega(n^2)$.
- $\boxed{True}$. We claim that $f(n) + g(n)$ is bounded by $2f(n)$ and $f(n)$ for large values of n. We know by the definition of little o that $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$. Another way of saying this is that the proportion $\frac{g(n)}{f(n)}$ can be made arbitrarily close to 0 by making n large. This means that for large enough n, we have: $g(n) \leq f(n)$, and thus $f(n) + g(n) \leq 2f(n)$. Of course, we have

$f(n) + g(n) \geq f(n)$ and thus we have shown that $f(n) + g(n) = \Theta(f(n))$.

Exercise 3. Give a counterexample to the following statement. Then propose an easy fix.

If $f(n)$ and $g(n)$ are positive functions, then $f(n) + g(n) = \Theta(\min(f(n), g(n))$

Solution
This is false. Let $f(n) = n$, $g(n) = 1$. Then, $\min(f(n), g(n)) = 1$ but $f(n) + g(n) = n + 1$ is not $\Theta(\min(f(n), g(n))) = \Theta(1)$.

However, it is true that $f(n) + g(n) = \Theta(\max(f(n), g(n)))$. To prove this, note that:

$$\max(f(n), g(n)) \leq f(n) + g(n) \leq 2\max(f(n), g(n)).$$

In addition, we know that $\max(f(n), g(n)) \leq f(n) + g(n)$ so indeed the Theta relationship is correct.

Exercise 4. (Challenge) Let $f(n) = 1^k + 2^k + \ldots + n^k$ for some constant $k \in \mathbb{N}$; find a 'simple' function $g : \mathbb{N} \to \mathbb{N}$ such that $f(n) = \Theta(g(n))$. Find a constant c such that $\frac{f(n)}{cg(n)} \to 1$ as $n \to \infty$.

Solution
Observe that (plot the function x^k, and represent the sum $f(n)$ in two ways as some area on the plot - one for each direction!)

$$\int_0^n x^k\,dx < 1^k + 2^k + \ldots + n^k < \int_0^{n+1} x^k\,dx$$

from which we conclude that

$$\frac{n^{k+1}}{k+1} < f(n) < \frac{(n+1)^{k+1}}{k+1}$$

Now it's easy to conclude that $f(n) = \Theta(n^{k+1})$ and the hidden constant is $\frac{1}{k+1}$.

2 Recurrence relations

2.1 General guidelines

There is no single best method for solving recurrences. There's a lot of guesswork and intuition involved. That being said, here are some tips that will help you out most of the time:

- **Guess**! You can get surprisingly far by comparing recurrences to other recurrences you've seen, and by following your gut. Often quadratic, exponential, and logarithmic recurrences are easy to eyeball.
- **Graph it**. Try plugging in some values and graphing the result, alongside some familiar functions. This will often tell you the form of the solution.
- **Substitute**. If a recurrence looks particularly tricky, try substituting various parts of the expression until it begins to look familiar.

- **Solve for constants**. Using the previous methods, you can often determine the form of the solution, but not a specific function. However, there is often enough information in the recurrence to solve for the remainder of the information. For instance, let's say a recurrence looked quadratic. By substituting $T(n) = an^2 + bn + c$ for the recurrence and solving for a, b, and c, you'll likely have enough information to write the exact function.

Exercise 5. Solve the following recurrence relations, assuming that $T(n) = 1$ for $n \leq 1$.

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

Solution

- Try some values of n. We know that $T(27) = T(9)+2 = T(3)+2+2 = T(1)+2+2+2 = 7$. It seems like the number of 2's that we add is equal to how many times we can divide the number by 3 before it drops to 1 or less. This suggests that the solution is: $\boxed{T(n) = 2\log_3(n) + 1}$.
- Again, let's try out some values of n and see a pattern. We have: $T(4^3) = T(64) = 3 \cdot T(16) = 3 \cdot 3 \cdot T(4) = 3 \cdot 3 \cdot 3 \cdot T(1) = 3^3$. This suggests that the solution is: $T(n) = 3^{\log_4 n}$. We can rewrite $\log_4 n = \frac{\log_3 n}{\log_3 4} = \log_3 n \cdot \log_4 3$. Plugging in, we get: $T(n) = 3^{\log_3 n \cdot \log_4 3} = n^{\log_4 3}$. Thus, $\boxed{T(n) = n^{\log_4 3}}$.
- We guess that the answer will be some kind of cubic because the difference between consecutive terms is quadratic. In a sense, n^2 is similar to the *derivative* of $T(n)$ or the finite differences of $T(n)$. Therefore, we guess $T(n) = an^3 + bn^2 + cn + d$. Now, plugging in our guess, we get:

$$\begin{aligned} an^3 + bn^2 + cn + d &= a(n-1)^3 + b(n-1)^2 + c(n-1) + d + n^2 \\ &= an^3 - 3an^2 + 3an - 3a + bn^2 - 2bn + b + cn - c + d + n^2 \\ &= an^3 + (-3a + b + 1)n^2 + (3a - 2b + c)n + (-a + b - c + d) \end{aligned}$$

  We need to find a, b, c, d such that the above is *always* satisfied, so we need the polynomials to be the exact same. Matching up the coefficients, we get:

$$a = a \tag{1}$$
$$b = -3a + b + 1 \tag{2}$$
$$c = 3a - 2b + c \tag{3}$$
$$d = -a + b - c + d \tag{4}$$

  Equation (1) gives us no information. Equation (2) gives us $a = \frac{1}{3}$. Equation 3 tells us $b = \frac{1}{2}$ and finally equation 4 tells us $c = \frac{1}{6}$. To figure out d, we need to use the initial conditions: $T(1) = a+b+c+d = 1$, and so $d = 0$. Therefore, the final solution is: $\boxed{T(n) = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n}$.

  Some students may have memorized that: $1^2 + 2^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$, which you can check is exactly the same expression we got for $T(n)$.

2.2 The Master Theorem

The previous section stressed methods for finding the exact form of a recurrence, but usually when analyzing algorithms, we care more about the asymptotic form and aren't too picky about being exact. For example, if we have a recurrence $T(n) = n^2 \log n + 4n + 3\sqrt{n} + 2$, what matters most is that $T(n)$ is $\Theta(n^2 \log n)$. In cases like these, if you're only interested in proving Θ bounds on a recurrence, the Master Theorem is very useful. The solution to the recurrence relation

$$T(n) = aT(n/b) + cn^k$$

where $a \geq 1$, $b \geq 2$ are integers, and c and k are positive constants, satisfies

$$T(n) \text{ is } \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^k) & \text{if } a < b^k \end{cases}$$

In general, the idea is that the relationship between a and b^k determines which term of the recurrence dominates; you can see this if you expand out a few layers of the recurrence. For more information about the proof, see the supplement posted with the section notes.

Exercise 6. Use the Master Theorem to solve $T(n) = 4T(n/9) + 7\sqrt{n}$.

Solution
$a = 4, b = 9, k = \frac{1}{2}$. Since $4 > 9^{\frac{1}{2}}$, we are in the first case and get $T(n) = \Theta(n^{\log_9 4}) \approx \Theta(n^{0.631})$

Exercise 7. How might you remember the above cases if you forgot them during an exam?

Solution
Try making a and b large! If $a \to \infty$ with b and k fixed, then this means we are surely going to be in the first case where $a > b^k$ and $T(n) = \Theta(n^{\log_b a})$. a represents the number of times we recurse in the recurrence relation, so if a gets large, then $T(n)$ will get larger.

On the other hand, if b gets bigger, we will be dividing the problem into more and more parts. In this case, we are dividing the problem into such small pieces that each recursive call takes very little time to run. Therefore, if b gets large, then $\log_b a$ gets smaller so the $T(n)$ gets smaller, if $a > b^k$. If we take $b \to \infty$, then we get $T(n) = a \cdot T(0) + cn^k = \Theta(n^k)$. This makes sense because no matter how many small the parts your recurse on are, you still need to perform this fixed cost of n^k so the run-time cannot be better than $\Theta(n^k)$.

Exercise 8. For each of the following algorithms, come up with a recurrence relationship in the form of Master Theorem and solve it. Verify that your answer makes sense.

- Mergesort
- Binary Search
- Finding the maximum element of a list by recursively finding the maximum element of the left and right halves and then taking the max of those two.

Solution

- $T(n) = 2T(n/2) + n$, which solves to $T(n) = \Theta(n \log n)$. This is indeed the run-time of mergesort.
- $T(n) = T(n/2) + 1$, which solves to $T(n) = \Theta(\log n)$, which is indeed the run-time of binary searching a list of size n.
- $T(n) = 2T(n/2) + 1$, which solves to $T(n) = \Theta(n)$. This makes sense because this recursive procedure for finding the maximum element cannot be faster than $\Theta(n)$ because if you're trying to find the maximum element of a list, you have to look through all the elements!