CS 4349 Lecture—August 23rd, 2017

Main topics for #lecture include #induction, and #asymptotic_notation.

Followup

- Insertion-Sort(A):
    - for j = 2 to A.length
        - key = A[j]
        - i = j - 1
        - Insert key into sorted sequence A[1.. j-1]. (comments are welcome to help make your algorithm more clear)
        - while i > 0 A[i] > key
            - A[i+1] = A[i]
            - i = i - 1
        - A[i + 1] = key
    - output A
- Last time, we discussed how to describe an algorithm, proofs of correctness using induction, and started in on asymptotic notation
- Before I continue, any questions on Monday's material or class administration?
- I want to follow up on some discussion we had last Monday on inductive proofs, and also offer another example of an inductive proof.
- Let's define a tree as a connected acyclic graph.
- And let's prove the following:
- **Theorem: Let T be a tree with n vertices. T has n - 1 edges.**
- Here, we may be tempted to use the following inductive "proof".
- "Proof":
    - Base case: A tree of one vertex has no edges.
    - Inductive hypothesis: A tree of k vertices has k - 1 edges for any $1 \leq k < n$.
    - Inductive step: Let T be a tree of k = n - 1 vertices. Create a new tree T' by adding new vertex and connecting it to T as a leaf. T' has n vertices and n - 1 edges.
- Everything I said was fine, but I never actually proved the theorem. The problem is that I now need to argue that we can construct *any* tree by adding on a new leaf like we did. Yes, that is possible, but I don't think it is obvious.
- Here is a better proof:
    - Inductive step: Let T be a tree of n > 1 vertices. Every pair of vertices is connected by a path since T is connected, so let uv be an arbitrary edge on one of these paths. Consider removing uv to make T \ uv. There is no path from u to v avoiding uv or T would have a cycle, so T \ uv has at least two components. On the other hand, we only

need add uv back to connect T \ uv so T \ uv has at most two components. Each component has fewer than n vertices, say k_1 < n and k_2 < n where k_1 + k_2 = n. Also T is acyclic so both components are acyclic and therefore trees. By the inductive hypothesis, T has k_1 -1 + k_2 - 1 + 1 = n - 1 edges.

- So again, aim down. This is also why it is good practice to use n and things less than n in inductive proofs instead of n and n + 1.

Asymptotic Analysis Continued

- So last time we defined Theta-notation.
- Given g(n) : N ➜ R^+, we defined Theta(g(n)) = {f(n) : there exist positive constants c1, c2, and n0 such that 0 ≤ c_1 g(n) ≤ f(n) ≤ c_2 g(n) for all n ≥ n0}.
- If f(n) in Theta(g(n)) we can write f(n) = Theta(g(n)).
- g(n) is an *asymptotically tight bound* on f(n)
- Sometimes we write Theta(1) to mean a constant or a constant function with respect to some variable. f(n) = Theta(1) means f(n) lies sandwiched between two constants c1, c2 > 0 for all sufficiently large n.

- Now, sometimes we don't want asymptotically tight bounds! Take Insertion-Sort for instance. a n^2 + bn + c = Theta(n^2) was a good upper bound on its performance, but it is not tight for all inputs.
- Can somebody name an instance where the algorithm runs faster than \Theta(n^2)?
- If the array is already sorted, we never even enter the while loop. There are at most bn + c = Theta(n) operations in this case.
- So it's not quite right to say the running time of Insertion-Sort is Theta(n^2), because Theta-notation is supposed to provide both an upper bound *and* a lower bound.

- For running times we usually just want an asymptotic upper bound, so we use big-oh notation.
- O(g(n)) = {f(n) : there exist positive constants c and n0 such that 0 ≤ f(n) ≤ c g(n) for all n ≥ n0}.
- [draw another figure]
- As before, if f(n) in O(g(n)), then we can write f(n) = O(g(n)).
- We may say f(n) = O(1) if f(n) ≤ c for some constant c once n grows large enough.
- big-oh notation is **only** an upper bound. So **Theta(g(n)) subset O(g(n))**, meaning a n^2 + bn + c = O(n^2), but also n = O(n^2) as well.
- It's pretty common to miss this distinction. Often in other classes or even algorithms literature you'll see people claim a function if O(g(n)) as a tight bound or even a lower bound. They may say things like "this problem has a lower bound of O(n log n)", but taken

literately, the lower bound could be 1 = O(n log n). If your goal is to describe a lower bound, use the big-omega notation I will define in a little bit.

- But loose upper bounds can be nice, because we can easily describe the running time of algorithms like Insertion-Sort. Here are some O(1) time operations that occur at most n^2 times, maybe fewer. So Insertion-Sort takes O(n^2) time. It's fine that some inputs are faster, because big-oh is only an upper bound.
- And since it is only an upper bound, we can even say Insertion-Sort runs in O(n^400) time, but that wouldn't be very useful.
- Formally, when we say the running time is O(f(n)), then we mean all running times on inputs of length n including the worst-case running time is O(f(n)) even if some inputs have better running times.
- When analyzing algorithms you usually want to use big-oh notation just in case the algorithm runs faster on some inputs. But you still want to give the slowest growing function you can. So the best analysis of Insertion-Sort is that it runs in O(n^2) time.

- Now, we have upper bounds, so it makes sense to have asymptotic lower bounds as well. We use big-omega notation.
- Omega(g(n)) = {f(n) : there exist positive constants c and n0 such that 0 ≤_ cg(n) ≤ f(n) for all n ≥ n0}.
- I like to imagine the Omega arms holding up the functions from below.
- [picture]
- Similar to before, f(n) = Omega(1) means f(n) never dips below some constant c > 0 once n is big enough.
- We have an asymptotic lower bound and an asymptotic upper bound. If they're both the same, then that must mean the function is asymptotically tight.
- In other words, **f(n) = Theta(g(n)) if and only if both f(n) = O(g(n)) and f(n) = Omega(g(n))**
- For running times, we use big-oh to present the optimistic view. The running time for insertion sort will never be worst than n^2.
- big-omega provides a more pessimistic view.
- If the running time of an algorithm is Omega(f(n)), then **every** input of size n takes Omega(f(n)) time. Maybe more.
- Take Insertion-Sort for example. We perform n - 1 operations just iterating over the different values for j. Therefore, **Insertion-Sort takes Omega(n) time.**

- When analyzing algorithms, we'll often have to evaluate certain expressions that come out to be the running time. For example, we might notice that there are n - 1 iterations of that for loop, each of which takes O(n) time. So we would like to say running time is at most O(1) + (n - 1) * O(n) = O(n^2).

- So let's make this concrete.
- Consider a more simple example 2n^2 + 3n + 1 = 2n^2 + Theta(n).
- We interpret this equation as saying there exists a function f(n) such that f(n) = Theta(n) and makes the equation true.
- More generally, you should be able to substitute in *some* function equal to each piece of asymptotic notation for each time the asymptotic notation appears on the *right hand side* of an equation.
- This means each time it *appears* in writing, so you only substitute in one function for an expression like \sum_{i=1}^n O(i).
- Here's another example: 2n^2 + Theta(n) = Theta(n^2).
- If the asymptotic notation appears on the left hand side, then the equation must be true for *every* function equal to the notation. So that expression is true only if 2n^2 + 100000 n = Theta(n^2), which is itself true because there *exists* something for the right hand side, namely 2n^2 + 100000 n.
- So in short, think of the left hand side being a big for all expression and the right hand side holding a there exists.
- We can even chain equations using these rules
    - 2n^2 + 3n + 1 = 2n^2 + Theta(n) = Theta(n^2).
    - Yep, 2n^2 + 3n + 1 = Theta(n^2) like we would expect.

- I'll introduce two more pieces of notation for when you want bounds that are not asymptotically tight.
- little-oh: o(g(n)) = {f(n) : for *any* positive constant c > 0, there exists a constant n0 > 0 such that 0 ≤_ f(n) < c g(n) for all n ≥_ n0}.
- In big-oh, g may only get away only if we chose a big enough constant c.
- But in little-oh, g will always get away no matter what constant c we choose.
- In other words, there does not exist a constant so that g is smaller. f(n) ≠ \Omega(g(n)) and therefore f(n) ≠ Theta(g(n)).
- Also, if f(n) = o(g(n)), then lim_{n ➔ infinity} f(n)/g(n) = 0.
- Examples: 2n = o(n^2), but 2n^2 ≠ o(n^2).
- f(n) = o(1) means f(n) approaches 0 in the limit.
- Insertion-Sort runs in o(n^400) time but *not* o(n^2) time since you may actually have ~n^2 operations in the worst case.
- Finally, little-omega: omega(g(n)) = {g(n): for any positive constant c > 0, there exists a constant n0 > 0 such that 0 ≤_ c g(n) < f(n) for all n ≥_ n0}. Now f will always get away, and f(n) ≠O(g(n)).
- Or, if f(n) = omega(g(n)), then lim_{n ➔ infinity} f(n)/g(n) = infinity.
- Examples: n^2/2 = omega(n) but n^2/2 ≠ omega(n^2).
- f(n) = omega(1) means f(n) approaches infinity in the limit, even if it grows very, very slowly.

- We're almost ready to discuss algorithms again, but first we should go over some properties of functions and asymptotics that will come up when analyzing and designing algorithms.
- First, asymptotics transpose in a natural way. For example
  - if f(n) = Theta(g(n) and g(n) = Theta(h(n)), then f(n) = Theta(h)
  - if f(n) = O(g(n)) and g(n) = O(h(n)), then f(n) = O(h(n))
  - and this pattern holds for the others as well.
- Big Theta is symmetric, so f(n) = Theta(g(n)) if and only if g(n) = Theta(f(n))
- The others are transpose symmetric, so 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))
- So asymptotics feel like comparisons between real numbers
  - f(n) = O(g(n)) is like a ≤ b
  - f(n) = Omega(g(n)) is like a ≥ b
  - f(n) = Theta(g(n)) is like a = b
  - f(n) = o(g(n)) is like a < b (f(n) is asymptotically smaller) and
  - f(n) = omega(g(n)) is like a > b (f(n) is asymptotically larger)
- But, it's not the case that we can always compare functions. i.e. neither f(n) = O(g(n)) nor f(n) = Omega(g(n)) may be true. For example, we cannot compare n and n^{1 + sin n}, because the later keeps oscillating between growing faster and then slower than n.

- There are a few natural classes of functions that tend to pop up when describing running times.
- We say a function f(n) is *polynomially bounded* if f(n) = O(n^k) for some constant k. So, the running time of Insertion-Sort is polynomially bounded.
- Some other functions, such as 2^n, grow exponentially. Algorithms with exponentially large running times are very slow in the worst-case.
- The base matters! For constants b > a > 1, a^n = o(b^n). So 2^n = o(2.01^n).
- But any base is bad for large enough n. For any real constants a and b with a > 0, n^b = o(a^n). So n^100 = o(1.01^n).
- Some other functions have logarithmic growth. These have some interesting properties if we apply the normal log rules.
- For example, log n^k = k log n = Theta(log n) for any constant k > 0.
- Also, log_b n = log n / log b = Theta(log n) for any constant b > 0, meaning the base doesn't matter for asymptotic growth if it is constant.
- You might see me write lg n sometimes to mean log_2 n. Dividing problems into 2 parts or writing in binary is common enough that it's useful to have the notation. And we can safely throw lg n or log n into our asymptotic functions (usually), because Theta(lg n) = Theta(log n).

- But be careful, because the base may matter if the lg appears in an exponent. $2^{\lg n} \neq \Theta(2^{\log n})$.
- We say a function $f(n)$ is *polylogarithmically bounded* if $f(n) = O(\lg^k n)$.
- These functions grow really slowly usually. For example, the number of bits needed to store the number n is only $O(\log n)$. That's why our 64-bit computers can still address far more bytes of ram than we'll ever fit in there anytime in the near (maybe far?) figure.
- In fact, for any constants a, b > 0, $\lg^b n = o(n^a)$. So $\lg^{100} n = o(n^{0.01})$.
- Our ideal goal for this class will be to write algorithms that runs in polylogarithmic time, but this is usually only possible if the input has a nice structure like a sorted array or you're doing something with data structures. Otherwise, we'll usually try to find something polynomial, and the smaller the exponent, the better.