CS 4349 Lecture—August 21st, 2017

Main topics for #lecture include #administrivia, #algorithms, #asymptotic_notation.

Welcome and Administrivia

- Hi, I'm Kyle!
- Welcome to CS 4349. This a class about algorithms. It's also a theory course. This means I'll be emphasizing correctness in the algorithms we discuss, how to prove their correctness, and how to accurately analyze their running times.
- So, in the exams, and especially the homework, I will expect you to prove things. I do this for a couple reasons. Most importantly, I want you to learn how to *reason* about the algorithms you're designing so you're confident they are correct and so you understand that the "first" approach isn't always the correct one. Even if you never write another proof in your life, I believe these exercises will help you.
- That said, I'll be teaching some useful tools for designing algorithms that you'll likely encounter in real life. You may have heard of things like divide-and-conquer or dynamic programming. We'll discuss these techniques in detail with plenty of examples and exercises so you'll hopefully feel inclined to use them outside the classroom.
- If nothing else, many companies like to see your algorithm chops before they hire you, and sometimes interviews actually will feel like a short quiz from this class, maybe with fewer proofs.
- OK! So, I hopefully got across why you're here. Let's talk a bit about how the class is structured and then we'll get to some material before we run out of time.

- Lectures will be every Monday and Wednesday from 7:00pm to 8:15pm. You probably figured this one out already.
- I plan to run as much as I can through the class website at https://utdallas.edu/~kyle.fox/courses/cs4349fa17/. I'll put announcements up whenever there are new homework assignments or upcoming exams or whatever else feels announcement-worthy.
- All new homework assignments will be put on the website *only*, and I'll put solutions up shortly after assignments are due.
- Please take some time to read through the About this Course page on the website as well as the course syllabus. There's useful info on homework, exams, and grading that I may not get to during this lecture.

- I plan to assign homework about once a week. I'll put out the assignment just before class on Wednesdays and make it due in class the following Wednesday. I want to get the

solutions to you as soon as possible, so I won't accept late homework. However, I will drop your lowest homework score so one bad week won't hurt your grade.
    - This may change depending on how much help I get for grading, etc.
- I encourage you to collaborate while working on the homework. But I also want you to write up solutions by yourselves, in your own words, and without use of other sources.
- That said, the homework is your best opportunity to learn the material, so I'd rather you consult sources before giving up entirely. **If you must, you may use other sources including writing solutions together at no penalty, but you must cite the source or person so I can find them, and you must describe the answer in your own words.** So don't just copy paste, even if you're citing something. And don't just rephrase a solution without giving proper credit. I'll consider those two things acts of academic dishonesty. Citing is easy and has no cost to your grade, so please when in doubt, cite.
- There will be two exams. First is a midterm scheduled for Wednesday, October 4th. It will start at 7:00pm like this class, but run until 9pm so you have time to solve the problems. The other exam is a comprehensive final that will take place in the normally scheduled finals slot. I'll announce the time when I know it.
- The website gives details on how grades are computed. In short, I will create a curve and find a reasonable distribution of grades in the curve. However, there are some minimum grade thresholds you can shoot for. If you hit the minimum for any grade, then you will *at least* as well as that minimum. So, in theory, everybody could get an A.

- And the last thing I think is office hours and the TA. The class will have one TA who will hold office hours, but I am still waiting for that assignment. I was thinking of holding why own office hour on Mondays from 5:30 until I need to leave for class in case you are not available until the evening. But just in case, what would you prefer? Would you prefer something during the normal business day, or does that evening plan sound better?
- Alright, so does anybody have any more questions about administrative stuff?

Describing Algorithms

- Then let's get to some material.
- This is a class on algorithms, so it's probably good to start with what an algorithm **is**, how to describe one, and what I expect from you when you describe and analyze an algorithm.
- An algorithm is an unambiguous set of instructions. For this class, an algorithm will take a set of value called the *input* and produce a set of values called the *output*.
- So, for example, an algorithm may take an input a sequence of numbers A = <a_1, a_2, …, a_n> and output the maximum value a in the sequence.
- So, how do we describe algorithms? Really, any way that is unambiguous is fine, including English prose!

- So, an algorithm to find the maximum might look like
  - "Set a value max to a_1. For each value i from 2 to n, if a_i is greater than max, then set max equal to a_i. Return max as the output."
- Often, though, it's easier to describe an algorithm in terms of pseudocode.
  - Max(A):
    - max = A[1]
    - for i = 2 to A.length
      - if A[i] > max, then max = A[i]
    - output max
- I am not going to be picky on how your pseudocode works. But whether you're using English or pseudocode, you need to make algorithms' instructions clear and unambiguous.

- Let's look at a more interesting example.
- An algorithm for the *sorting problem* takes an input a sequence of numbers A = <a_1, a_2, …, a_n> and output a permutation A' = <a'_1, a'_2, …, a'_n> of A such that a'_1 ≤ a'_2 ≤ … ≤ a'_n.
- Let's take a running example for the rest of this lecture: The Insertion-Sort algorithm which solves the sorting problem. Here's it's probably easiest to start with the pseudocode
  - 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
- Or in English: for each value j from 2 to the length of A, set the variable key to be A[j]. Set a index i equal to j - 1. While i > 0 and A[i} is strictly greater than key, copy A[i] to A[i+1] and then decrement i. When the while loop terminates, set A[i+1] to key.
- In other words, we are pushing everything greater than key up by one position to insert key into the already sorted sequence A[1.. j-1].
- One thing you may notice is that neither the instructions in the pseudocode nor instructions in English alone was all that illuminating. I believe, maybe you disagree, that combining both and giving the little intuitive explanation helped clarify the whole thing.
- So, what wouldn't be a suitable algorithm?

    - If A[2] is greater than A[1], swap them. Then, move A[3] to between A[1] and A[2] if it lies between them in sorted order or before both if A[3] < A[1]. Continue in that manner. Or in pseudocode:
    - Insertion-Sort2(A):
        - key = A[2]
        - if A[1] > key, then A[2] = A[1], A[1] = key
        - key = A[3]
        - if A[2] > key
            - A[3] = A[2]
            - if A[1] > key, then A[2] = A[1], A[1] = key
            - else A[2] = key
        - and so on…
    - So like I said, we successfully sort the first two values, and then we sort the first three, and then what?
- This algorithm is not suitable, because it is not unambiguous.
- Pattern matching is hard. Maybe you'll find patterns while designing algorithms, but that's kind of the point. *You* found the pattern; now describe the algorithm. You can't expect somebody reading your partial algorithm or especially a computer to come to the same conclusions!
- So always explicitly spell out what happens at each step of an iteration and what the bounds for the iteration are.
- This is also not a suitable algorithm:
    - Insertion-Sort3(A):
        - for j = 2 to A.length
            - Insert A into the sorted sequence A[1.. j-1]
- I mean, it's correct in a sense, but that inner step is a doozy. You can't expect somebody to understand how to do that middle step until you've already explained it once.
- That means the middle step is actually a *reduction*. We'll return to those in a couple lectures.
- As a general rule of thumb, each step of your algorithm should be something you would reasonably expect a computer to do in a constant number of operations. Do not make large leaps unless you mean for them to be reductions to problems we already know how to solve.

An Inductive Proof

- So we've discussed describing an algorithm. But one focus of this course is not just to describe any algorithm, but a *correct* algorithm that *solves* the problem we have in mind.
- After describing an algorithm, we should prove its correctness, so let's do that for

insertion-sort.

- The textbook proves correctness using a method they call “loop invariants”. This is really just induction in disguise.
- You’ll be using induction a lot in this course, especially when we get to recursive algorithms, so why don’t we try proving correctness of insertion-sort using induction directly.
- We’ll prove correctness using induction on the outer loop variable, j. specifically, let’s prove the following.
  - **Lemma: For all 2 ≤_j ≤ n+1, at the beginning of iteration j, A[1.. j-1] is sorted.**
  - We’ll say the for loop terminates at the beginning of iteration j = n+1. By the lemma, the whole sequence is sorted.
  - Induction proofs generally have three steps. The first is the base case. What is the base case here? **BEAT**
  - Right j = 2. A[1] is trivially a sorted array.
  - Next is the inductive hypothesis. Now, it’s tempting to decrement the index by one. In other words, we’ll assume that **at the beginning of iteration j-1, A[1.. j-2] is sorted**. Doing so would be something called *weak induction*.
  - Instead, we could use *strong induction* and say **at the beginning of iteration j’, A[1.. j’-1] is sorted for all 2 ≤ j’ < j**. Yeah, maybe we don’t strictly need it here, but it gives us another j-1 values to work with. When we discuss divide-and-conquer algorithms next week, we’ll need some of them.
  - **never ever ever use weak induction since strong induction does the same thing but better**
  - Finally, we look at the inductive step. **[lots of writing]** So, per the lemma, we are beginning loop j, meaning we just finished loop j-1. At the beginning of loop j-1, A[1.. j-2] is sorted per the inductive hypothesis. Let i be the greatest index such that A[i] ≤_A[j-1]. We set key = A[j-1] and then the while loop moves all values A[i+1] through A[j-2] one space to the right, so the first j -1values are now the old A[1.. i] * A[j-1] * A[i+1… j-2]. The first set is in order by the inductive hypothesis, the middle value comes after that first set by definition of i, and the last set is in order by the inductive hypothesis.
  - The first j-1 values are sorted, proving the lemma.
- So what did we do here? We described the algorithm in an unambiguous way, and then we proved via induction that sorts the array as we’d like.
- OK, if we wanted to be really formal, we’d also want to show the while loop does shift things like I claimed, but now we’re getting way too tedious for a lecture.
- The general rule for when you write proofs: Unless the problem states otherwise, feel free to use anything seen in lecture, a prerequisite class, or the textbook as a given. It’s been proven or the algorithm is known and does not need further explanation.

- There's one last focus in this course and one last step you need to take when describing an algorithm, and that is in analyzing the algorithm.
- There are many ways to analyze algorithms such as looking at average case running time or expected running time if the algorithm uses randomness, but
- **in this course, we will usually be concerned with worst-case running time and sometimes space**.
- In other words, the running time you claim should be greater than the running time for every possible input to the algorithm.
- But running time is a weird thing. How long do individual operations last? Does the line A[i+1] = A[i] take one step? Three steps? What about i = i - 1? And how tedious is it going to be to count individual operations in our algorithms?
- On sequences of length n, it looks like each of the inner loop's operation occur at most n^2 times, once for each possible pair (j, i). There are some operations that occur at most n times outside the while loops. And there's some operations that happen a constant number of times.
- So we could say the running time is at most a n^2 + b n + c for some constants a, b, and c.
- But what happens when n gets really large? the a n^2 term takes over, no matter how large b and c are. We really only care about that n^2 term.
- So let's step away from Insertion-Sort and discuss how to formalize that idea with asymptotic notation.

Asymptotic Analysis

- Let f(n) : N ➔ R+ be a positive function over the natural numbers.
- Let's start with a way to fully characterize its growth called Theta-notation.
- Formally, given g(n) : N ➔ R+, Theta(g(n)) is a set of functions. We'll loosen the formalism in a bit.
- Theta(g(n)) = { f(n) : there exist positive constant c1, c2, and n0 such that 0 ≤ c_1 g(n) ≤ f(n) ≤ c_2 g(n) for all n ≥ n0}.
- In other words, we have the following situation. [draw c1 g, f, and c2g on the same axes.] The function f is sandwiched between c1g and c2g at all points to the right of n0.
- You would formally write f(n) in Theta(g(n)), but it's common to write and say f(n) = Theta(g(n)), and that's fine for this class.
- If f(n) equals Theta(g(n)) , then g(n) is an *asymptotically tight bound* for f(n)
- Theta notation lets us throw away the less important or *lower order* terms in a function.
- For example, n^2 + bn + c = Theta(n^2) for any b, c with a > 0.
    - A tedious proof: Set c_1 = a/4, c_2 = 7a/4, and n0 = 2 * max{|b| / a, \sqrt{|c| / a}}.
    - For n ≥ n0, 0 ≤ a/4 n^2 = a n^2 - a / 2 n^2 - a / 4 n^2 ≤ a n^2 + b n + c ≤ a n^2 + a /

2 n^2 + a / 4 n^2 = 7a/4

- Meaning our running time bound for Insertion-Sort was \Theta(n^2)
- Theorem without proof: For any polynomial p(n) = \sum_{i=0}^d a_i n^i where each a_i is a constant and a_d > 0, we have p(n) = Theta(n^d).*
- I’ll teach some more rules for how derive facts like that probably on Wednesday.