CS 6363.005.19S Lecture 1—August 20, 2018

Main topics are #algorithms, and #administrivia#.

Introduction and Etymology

- Hi, I'm Kyle!

- And welcome to CS 6363, Design and Analysis of Computer Algorithms.
- Today, we're going to discuss what you should expect from a course on algorithm design and analysis, and what's expected when describing an algorithm in this course.
- In short, we'll be working on the design and analysis of algorithms that are provably correct for every input and giving running time bounds which also hold for every input.

Administrivia

- But let's start with some administrative stuff.
- This is a section of CS 6363. All sections have essentially the same goals of understanding of designing algorithms of various types.
- It's also the section devoted specifically for the Algorithms QE. So I'm going to put extra emphasis on making sure you can design and describe algorithms yourself, along with performing algorithm related proofs, as that is the format of the QE exam.
- There's a rather long page on course policies and suggested readings on the course webpage and in the syllabus at http://utdallas.edu/~kyle.fox/courses/cs6363.005.19s/. I'll go over a few key points now.
- All sections of 6363 share a "required" textbook by Cormen and others called "Introduction to Algorithms". It's usually called CLRS. This textbook is an excellent resource used by many, and it contains far more material than I might possibly be able to cover during the semester.
- However, I'm personally more fond of a freely available textbook by Jeff Erickson. I think it strikes a better balance between rigor and readability than CLRS and puts extra emphasis on cutting to the core of what you need to know and do to design algorithm using different techniques.
- I'll generally follow Erickson when designing lectures, although I may pull examples or proofs from CLRS. I promise I'll always write out each homework problem I want to assign instead of pointing directly to either book's problem sets.
- Grades will be determined by weighing homework 30%, two midterms at 20% each, and a final exam 30%.
- DO YOUR HOMEWORK. Designing algorithms and proving things about them is hard. Like

really really hard. I can show you what others have done before. I can show the paradigms others have designed to make algorithm design easier. But sadly, I cannot truly "teach" you to design algorithms by just talking at you. The only way to truly learn the art is to practice practice practice, and that's what the homework is for.

- Because I so want you to actually do the homework, I'm offering a 48 hour grace period on each deadline. If you miss the deadline, you get another 48 hours, penalty free, to turn things in, during which I may send annoying emails because you're technically late. After that, though, I really need to put out solutions so I won't accept it anymore.
- I understand that we all have bad weeks, though, so I will drop your lowest homework score also.
- Grades will be assigned based on how well you do relative to the class average and the difficulty of assignments and exams. I'll stay cognizant that this is the QE section and you should be stronger than the average graduate students. So in principle, everybody can get top marks regardless of the class average if everybody does well enough. Talk to me if you're concerned about your grades.
- You'll turn in homework as individuals, but I want you to collaborate as well. And if even that isn't enough, finding the solution somehow is still better than giving up.
- So, if you need outside help you, may seek it. However, you **must** cite all outside sources, including other students you work with, and write solutions **in your own words**. Otherwise, I consider it an act of plagiarism.
- Finally, I'm going to go over it today, but be sure to read up on the website what I mean by "describe an algorithm". This includes justifying correctness through a short proof sketch and analyzing running time. You may get to skip the proof part on midterms due to time constraints; I'll let you know.
- OK, let's actually learn something about algorithms now.

Algorithms

- An algorithm is an explicit, precise, unambiguous, mechanically-executable sequence of elementary instructions.
- Erickson gives a pretty good summary on where the term came from, but one interesting point is that*algorithm* at one point referred to general pencil-and-paper methods for numerical calculations. And the people who did these calculations were called *computers*.
- Of course, now we have electronic computers to run our algorithms for us, but the principal remains the same. The algorithms we design in this class should be simple enough to be read and executed by humans, although it might get a bit tedious in some cases.

Simple Examples

- So when I said an algorithm is a sequence of instructions, I never specified they had to do something useful on a computer. For example, here's an algorithm for singing an annoying song titled "99 Bottles of Bear on the Wall" (for arbitrary values of 99).
- BottlesOfBeer(n) is a procedure that sings n Bottles of Beer on the wall where n is a non-negative integer. It's also an excuse to describe how I write pseudocode.

```
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."
```

- OK, so algorithms are generally used for more computery or numeric things.
- Here's an algorithm for multiplying large numbers sometimes called *Russian peasant multiplication*. Please know that today I'm just going through examples of what algorithms look like and how to write them. I'm definitely not expecting you to memorize this particular algorithm or learn to sing drinking songs.
- PeasantMultiply(x, y) takes two non-negative integers x and y and returns their product x * y. x, y, and x * y can be expressed as an array of digits.

```
PEASANTMULTIPLY(x, y):
  prod ← 0
  while x > 0
      if x is odd
          prod ← prod + y
      x ← ⌊x/2⌋
      y ← y + y
  return p
```

x	y		$prod$
			0
123	+456	=	456
61	+912	=	1368
30	~~1824~~		
15	+3648	=	5016
7	+7296	=	12312
3	+14592	=	26904
1	+29184	=	**56088**

- So this algorithm expects a bit more of you than n bottles of beer on the wall, but it still breaks down multiplication into four simple operations that can be mechanically performed: (1) determining parity, (2) addition, (3) doubling a number, and (4) halving a number (rounding down)
- The general rule is that if you can expect nearly any CS student to do something by hand without further explanation, it's safe to write it as a single line of pseudocode.

Designing and Analyzing Algorithms

- My main goal in this course is to teach you how to design and analyze your own algorithms for a variety of problems. Mastering this skill is vital if you want to pass the QE.
- But after designing and analyzing the algorithm, you still need to describe it to others, including me and the TA.

- And it turns out both skills complement each other nicely.
- When describing an algorithm, you generally need to specify four things:
  - What: A precise specification of the problem that the algorithm solves.
  - How: A precise description of the algorithm itself.
  - Why: A proof that the algorithm solves the problem it is supposed to solve.
  - How fast: An analysis of the running time of the algorithm.
- You may *develop* these concepts in any particular order. For example, you may start thinking about running time and realize a recursive solution is best. But then you need to tweak the specification so the recursion works.
- But it's usually easiest to write these four things separately, and usually in that order.
- Like other kinds of writing, how you describe an algorithm depends upon who your audience is.
- You may be surprised to learn that your audience *is not* a computer for this class. Instead, I want to you aim at a competent but skeptical program who is not as experienced as you are. Think about yourself before you joined the graduate program here.
- This programer you're writing for is a novice and will interpret the how of your algorithm exactly as written. They don't want to solve anything for you or even do much high level thinking. So you're forced to work through the finer details yourself and present them.
- The programmer is also skeptical that you're correct; you'll need to develop robust arguments and proofs for correctness of your algorithm and its running time.
- During lectures, I'm want to treat you all as skeptical novices as well. Please hold me to this. If I talk about something you haven't seen, press me for details. If I don't explain why something is correct, press me for details. Feel free to point out mistakes or omissions when I'm proving things. Assume I'm trying to lie to you.
- Let's go over each of these four things to specify in more detail for the rest of today and Thursday. Some of this may be review of things you've seen before, but I want to warn you that you may be seeing them in a slightly different way from what you've seen before.

Specifying the Problem

- Often, you'll be asked to design an algorithm for a real world scenario, or at least something I want to pretend is a real world scenario.
- And before you can precisely describe an algorithm for that scenario, you need to specify what *problem* the algorithm is supposed to solve.
- So when asked to do something in terms of real world object, first restate the problem in terms of mathematical objects like numbers, arrays, lists, graphs, trees, and so on that you can reason about formally.
- In particular, specify your algorithms' inputs and output and their types.
  - BottlesOfBeer(n) sings the song n bottles of beer where n is a non-negative integer.

    - PeasentMultiply(x, y) returns the product of x and y where x and y are both non-negative integers represented as (say) an array of digits.
- Generally, your specification should give enough information that somebody could use your algorithms as a *black-box* or *subroutine* without needing to understand how or why the algorithm works.
- Pretend you're library authors who are actually good at documentation.

Describing the Algorithm

- Computer programs are concrete representations of algorithms, but algorithms are not programs, and they should not be described in any particular pogromming language.
- The algorithms you design should work in any programming language, and if you describe them using a particular language like C++ or perl, then you'll focus more on the language's nuances instead of what is really going on. Do you really want me chasing pointers around when I could be reading the higher level idea? Also, I can't read perl at all.
- On the other hand, pure English isn't great either. There's lot's of subtleties and ambiguities in the language, and it's really hard to describe conditionals, loops, and recursion without being confusing.
- So use pseudocode, maybe combined with a very brief high level description of what you're going for. Pseudocode uses the structure of programming languages and math to break algorithms into primitive steps, but the primitive steps themselves can be written in English, math, or some combination of the two; whatever is the clearest.

Proving Correctness

- Often, it's enough to design algorithm that work for "most" inputs or work "well enough" for all inputs.
- In this class, though, we'll be focusing on algorithms that are correct for *all* inputs. Meaning they find the thing we're looking for or find a best solution *every, single, time*.
- With very few exceptions, no algorithm I present or algorithm you're asked to describe will be *obviously* correct.
- So you'll need to prove correctness of your algorithm. This *is not* the same as restating your pseudocode descriptions in plain English.
- Instead, you'll need to explain why what you did is correct. Why did adding numbers in this way give the correct thing? What led to this iterative procedure? Why does this recursion terminate?
- The most important proof technique we'll need is *induction* Nearly *everything* comes back to induction, so I hope you're comfortable with it.
- Honestly, though, most people aren't as comfortable with induction as we'd like, so I'll be

going over it on Thursday. Please please come to this lecture, even if you've mastered using that template you were given in discrete math. It's likely the template has taught you some bad habits that make it useless for this course.

Analyzing Running Time

- There may be many different algorithms that solve the same problem, and the most common way of ranking those algorithms is by running time.
- I guess I *could* ask you to count exactly how many operations your algorithm performs. For example, we mention beer exactly 3n + 3 times when performing BottlesOfBeer(n).
- But counting operations doesn't tell you too much. Singing time is highly variable depending on the tempo. Time spent running a computer algorithm is highly variable depending on the speed of the computer.
- And for any algorithm that's the least bit sophisticated, we probably can't compute an exact count with the mathematics we understand or hope to express it as a close-form solution.
- Therefore, we'll rely on asymptotic notation to describe running times. In particular, we'll use big-Oh notation **a lot**.
- Your final job after specifying the problem, describing the algorithm, and proving correctness, will be to tell me the running time using big-Oh notation and explain why you're correct.
- For example, BottlesOfBeer(n) takes O(n) time to sing. In fact, it's Theta(n) if you're family with that notation. Why? Because there's a single for loop over n beers, and it takes constant time to sing each iteration.