CS 6363.003.21S Lecture 1—January 19, 2019

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 I'll be expecting from you.
- 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 and designing algorithms of various types.
- As far as I'm able to, I'm only going to assume the same knowledge base as in other 6363 sections. So you should have taken the prerequesites or equivalents here or elsewhere. I expect you to understand basic matters on discrete structures, proofs, asymptotic notation, and so on, and to have at least been exposed to some basic commonly used algorithms. However, I'll still go over many of these basics again with an emphasis on their use in algorithm design. Please don't be surprised when we revisit some things you may have seen a while ago, including sorting. Sometimes seemingly basic things make for great examples of complicated topics.
- That said, we're in the section devoted specifically for the Algorithms QE. I'm going to put extra emphasis on making sure you can design and describe algorithms yourself with minimal hand holding, along with performing algorithm related proofs. The QE exams consists almost entirely of describing, analyzing, and proving correctness of novel algorithms.

- There's a rather long page on course policies and suggested readings on the course webpage and in the syllabus at https://personal.utdallas.edu/~kyle.fox/courses/cs6363.003.21s/. I'll go over a few key points now.
- This course has the "remote" modality, meaning we won't be meeting in person excepting special circumstances.
- All lectures and office hours will be held over MS Teams. If you haven't figured that out yet,

I'm not sure how you're listening to me.

- I also encourage you to use MS Teams as a discussion board. There's probably tools in eLearning for this sort of thing too, but that system is such a pain to use that I'd rather avoid it as much as we can.
- I will use eLearning to email announcements, accept homework submissions, and post grades, though.
- I will be teaching this course using regular lectures, likely on an iPad or possibly whiteboard. Your responsibilities with the material are to watch the lectures, read any lecture notes I write myself, read any relevant portions of other lecture notes or books I link to.
- If you can, please attend the live lectures on MS Teams so you can ask me questions as we go. I'm wearing headphones and frequently checking my computer screen so I can stay in sync with you all.
- If you cannot attend live lectures, I completely understand, and I don't expect you to give me any notice or explanations. An MS Streams recording of each lecture will appear in the Lectures channel on Teams as soon as I can make it available.

- 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%.
- I'll release homework once every couple weeks and make it due about two weeks later. We'll probably do four, maybe five assignments.
- You may work in formal groups of up to two people if you'd like, but it is not a requirement to group up. Each group should turn in **one** copy of the homework on eLearning. I'll give everybody in the group the same grade.
- If you need extra time past the deadline, you must ask for an extension. I will automatically approve all extensions of up to 48 hours, but you still need to ask. I might give you longer if there are extenuating circumstances. It's 2021, which is practically 2020. Please don't be

afraid to ask.

- 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 used to make algorithm design easier. But sadly, I cannot truly "teach" you to design algorithms by just talking at you. In other words, there is no algorithm for designing algorithms. The only way to truly learn the art or even to understand the examples I provide is to practice practice practice, and that's what the homework is for.
- Grades will be assigned based on how well you do relative to the class average and the difficulty of assignments and exams. 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 or pairs, but I want you to collaborate en large 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. And if you use a website, please actually give me the full address and not just the domain.
- 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'll be expected to go through the whole design process in several homework problems. You'll often get to skip the proof part on midterms due to time constraints, though; 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. 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 *computators* or more simply, *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 actually executing the algorithm by hand could take quite a long time.

Simple Examples

- So when I said an algorithm is a sequence of instructions, I never specified they had to do something useful on a digital 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 like to 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 two numbers x and y together which probably resembles the one you learned in grammar school, although the operations may be happening in a different order from what you learned. This family of multiplication algorithms is called *lattice multiplication*.
- The input is two arrays of decimal digits X[0 .. m - 1] and Y[0 .. n - 1] representing the natural numbers

$$x = \sum_{i=0}^{m-1} X[i] \cdot 10^i \quad \text{and} \quad y = \sum_{j=0}^{n-1} Y[j] \cdot 10^j,$$

.

- Note that I'm explicitly telling you how I'm indexing the arrays. I usually start arrays on index 1. **gasp** But that 10^i there makes it more convenient to start on index 0.
- The output will be a single array Z[0 .. m + n - 1] representing the product

$$z = x \cdot y = \sum_{k=0}^{m+n-1} Z[k] \cdot 10^k.$$

.

- Here's the pseudocode. All I am assuming is that you know how to add two numbers together and multiply two decimal *digits* together. The version of lattice multiplication I'm showing is actually due to Fibonacci who's numbers you may have heard. I'm not expecting you to learn this particular algorithm. It's just a nice example of pseudocode doing some math.

```
FIBONACCIMULTIPLY(X[0..m-1], Y[0..n-1]):
  hold ← 0
  for k ← 0 to n + m - 1
      for all i and j such that i + j = k
          hold ← hold + X[i] · Y[j]
      Z[k] ← hold mod 10
      hold ← ⌊hold/10⌋
  return Z[0..m + n - 1]
```

- 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 while also showing you some of the "classics". They'll serve and good examples and be directly useful in case you need to use or refer to them in the future.
- Of course, 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.
- For certain homework or exam problems, I may ask you to do only some of these things, but all four are necessary if all you know is the what you're trying to design an algorithm for.
- Now, 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 programmer who is not as experienced as you are. Think about yourself before you started this semester.
- 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 for correctness of your algorithm and its running time.
- During lectures, I want to treat you all as skeptical novices as well, at least as much as time allows. 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. Rarely, I'll ask you to just trust me on something (OK, I already did that with FibbonaciMultiplication), but especially when I'm describing an algorithm you've not seen before, act as if I might be making things up.
- Let's go over each of these four things to specify in some more detail for the rest of today, and then I'll speak a bit more about the main "technique" you'll want to master to do well in algorithm design.

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.
  - FibonacciMultiply(X[0 .. m - 1], Y[0 .. n - 1]) returns the product of x and y where x and y are both non-negative integers represented by X and Y.
- 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 writing 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 are lots of subtleties and

ambiguities in the language, and it's really hard to describe conditionals, loops, and recursion without being confusing.

- So I highly suggest you use pseudocode, preferably 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.
- I chose those two examples earlier to help you see what the right balance between math and English is. Erickson has many more examples, so please look at them.

Proving Correctness

- Often, it's enough to design an 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 justify 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?

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.
- You've probably already seen this and done it using big-Oh notation. We'll discuss running time analysis in more detail next Monday.