6.080/6.089 GITCS Feb 5, 2008

Lecture 1

Lecturer: Scott Aaronson *Scribe: Yinmeng Zhang*

1 Administrivia

Welcome to Great Ideas in Theoretical Computer Science. Please refer to the syllabus for course information.

The only prerequisite for the class is "mathematical maturity," which means that you know your way around a proof. What is a proof? There's a formal definition of proof where each statement must follow from previous statements according to specified rules. This is a definition we will study in this course, but it's not the relevant definition for when you're doing your homework. For this class, a proof is an argument that can withstand all criticism from a highly caffeinated adversary.

Please interrupt if anything is ever unclear; the simplest questions are often the best. If you are not excited and engaged then complain.

2 What is computer science?

Computer science is not glorified programming. Edsger Dijkstra, Turing Award winner and extremely opinionated man, famously said that computer science has as much to do with computers as astronomy has to do with telescopes. We claim that computer science is a mathematical set of tools, or body of ideas, for understanding just about any system—brain, universe, living organism, or, yes, computer. Scott got into computer science as a kid because he wanted to understand video games. It was clear to him that if you could really understand video games then you could understand the entire universe. After all, what is the universe if not a video game with really, really realistic special effects?

OK, but isn't physics the accepted academic path to understanding the universe? Well, physicists have what you might call a top-down approach: you look for regularities and try to encapsulate them as general laws, and explain those laws as deeper laws. The Large Hadron Collider is scheduled to start digging a little deeper in less than a year.

Computer science you can think of as working in the opposite direction. (Maybe we'll eventually meet the physicists half-way.) We start with the simplest possible systems, and sets of rules that we haven't necessarily confirmed by experiment, but which we just suppose are true, and then ask what sort of complex systems we can and cannot build.

3 Student Calibration Questions

A *quine* is a program that prints itself out. Have you seen such a program before? Could you write one?
Here's a quine in English:

Print the following twice, the second time in quotes.
"Print the following twice, the second time in quotes."
Perhaps the most exciting self-replicating programs are living organisms. DNA is slightly different from a quine because there are mutations and also sex. Perhaps more later.

Do you know that there are different kinds of infinities? In particular, there are more real numbers than there are integers, though there are the same number of integers as even integers. We'll talk about this later in the course, seeing as it is one of the crowning achievements of human thought.

4 How do you run an online gambling site?

This is one example of a "great idea in theoretical computer science," just to whet your appetite.

Let's see what happens when we try to play a simple kind of roulette over the Internet.

We have a wheel cut into some even number of equal slices: half are red and half are black. A player bets n dollars on either red or black. A ball is spun on the wheel and lands in any slice with equal probability. If it lands on the player's color he wins n dollars; otherwise he loses n dollars, and there is some commission for the house. Notice that the player wins with probability 1/2—an alternate formulation of this game is "flipping a coin over the telephone."
What could go wrong implementing this game? Imagine the following.

Player: I bet on red.
Casino: The ball landed on black. You lose.
Player: I bet on black.
Casino: The ball landed on red. You lose.
Player: I bet on black.
Casino: The ball landed on red. You lose.
Player: I bet on red.
Casino: The ball landed on black. You lose.
Player: This #$%îng game is rigged!
So actually the player could probably figure out if the casino gives him odds significantly different from 50-50 if he plays often enough. But what if we wanted to guarantee the odds, even if the player only plays one game?

We could try making the casino commit to a throw before the player bets, but we have to be careful.

Casino: The ball landed on black.
Player: That's funny, I bet black!

We also need to make the player commit to a bet before the casino throws. At physical casinos it's possible to time it so that the throw starts before the Player bets and lands after. But what with packet-dropping and all the other things that can go wrong on them intertubes, it's not clear at all that we can implement such delicate timing over the Internet.

One way to fix this would be to call in a trusted third party, which could play man-in-the-middle for the player and casino. It would receive bet and throw information from the two parties, and only forward them after it had received both. But who can be trusted?

Another approach, one that has been extremely fruitful for computer scientists, is to assume that one or both parties have limited computational power.

5 Factoring is Hard

Multiplying two numbers is pretty easy. In grade school we learned an algorithm for multiplication that takes something like N^2 steps to multiply two N-digit numbers. Today there are some very clever algorithms that multiply in very close to $N \log N$ time. That's fast enough even for thousand-digit numbers.

In grade school we also learned the reverse operation, factoring. However, "just try all possible factors" does not scale well at all. If a number X is N bits long, then there are something like 2^N factors to try. If we are clever and only try factors up to the square root of X, there are still $2^{N/2}$ factors to try. So what do we do instead? If you had a quantum computer you'd be set, but you probably don't have one. Your best bet, after centuries of research (Gauss was very explicitly interested in this question), is the so-called *number field sieve* which improves the exponent to roughly a constant times $N^{1/3}$.

Multiplication seems to be an operation that goes forward easily, but is really hard to reverse. In physics this is a common phenomenon. On a microscopic level every process can go either forwards or backwards: if an electron can emit a photon then it can also absorb one, etc. But on a macroscopic level, you see eggs being scrambled all the time, but never eggs being unscrambled. Computer scientists make the assumption that multiplying two large prime numbers is an operation of the latter kind: easy to perform but incredibly hard to reverse (on a classical computer).
For our purposes, let's make a slightly stronger assumption: not only is factoring numbers hard, it's even hard to determine if the last digit of one of the factors is a 7. (This assumption is true, so far as anyone knows.) Now, to bet on red, the player picks two primes that don't end in 7 and multiplies them together. To bet on black, the player picks two primes, at least one of which ends in 7, and multiplies them together to get X.

Player: sends X to the casino.
Casino: announces red or black.
Player: reveals factors to casino.
Casino: checks that factors multiply to X.

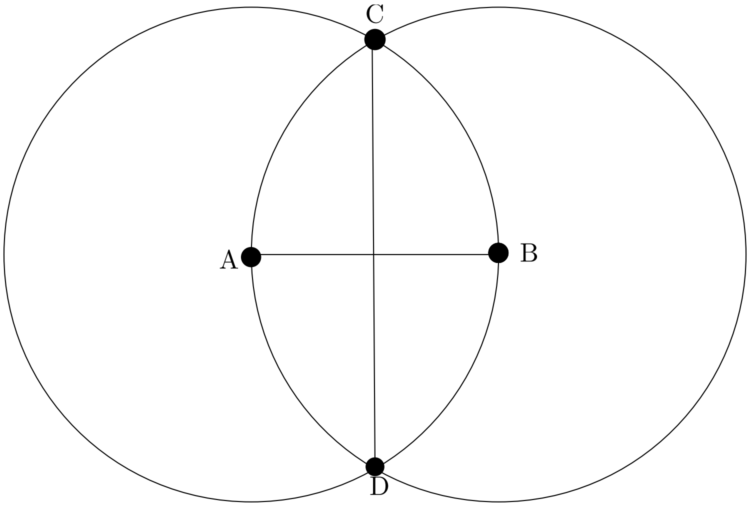
Is this a good protocol? Can the casino cheat? Can the player? The player might try to cheat by sending over a number which is the product of three primes. For example, suppose the factors were A, B, and C, and they ended in 1, 3, and 7 respectively. Then if the casino announces red, the player could send the numbers AB and C; if the Casino announces black, the player sends A and BC – the player wins both ways. But all is not lost. It turns out that checking if a number has non-trivial factors is a very different problem from actually producing those factors. If you just want to know whether a number is prime or composite, there are efficient algorithms for that—so we just need to modify the last step to say "Casino checks that the factors are primes which multiply to X."

This is a taste of the weird things that turn out to be possible. Later, we'll see how to convince someone of a statement without giving them any idea why it's true, or the ability to convince other people that the statement is true. We'll see how to "compile" any proof into a special format such that anyone who wants to check the proof only has to check a few random bits—regardless of the size of the proof!—to be extremely confident that it's correct. That these counterintuitive things are possible is a discovery about how the world works. Of course, not everyone has the interest to go into these ideas in technical detail, just as not everyone is going to seriously study quantum mechanics. But in Scott's opinion, any scientifically educated person should at least be aware that these great ideas exist.

6 Compass and Straightedge

Now let's go back to the prehistory of computer science – the time of the Ancient Greeks. The Greeks were very interested in a formal model of computation called *compass-straightedge constructions*: what kind of figures can you draw in the plane using a compass and straightedge? The rules are as follows.

We start with two points. The distance between them defines the unit length.
We can draw a line between any two points.
We can draw a circle given its center and a point on its circumference.
We can draw a point at the intersection of any two previously constructed objects.



By applying these rules over and over, we can construct all kinds of things. Above is the construction of the perpendicular bisector of a line segment. [Can you prove that it works?] In 1796, Gauss constructed a regular 17-gon, which he was so proud of that he asked to have it inscribed on his tombstone. (The carver apparently refused, saying it would look just like a circle.) In principle you could construct a regular 65535-sided poygon, though presumably no one has actually done this. Instead of actually drawing figures, we can reason about them instead. The key to fantastically complicated constructions is *modularity.* For example, once we have an algorithm for constructing perpendicular lines, we encapsulate it into the perpendicular lines subroutine. The next time we need a perpendicular line in a line of reasoning, we don't have to build it from scratch, we get to assume it.

By building on previous work in this way over the course of centuries, people erected a veritable cathedral of geometry. And for centuries, this manipulation of production rules was the canonical example of what it meant to think precisely about something. But the game pointed its way to its own limitations. Some constructions eluded geometers—among them, famously, squaring the circle, trisecting an angle, and doubling the cube.

Today we'll talk about doubling the cube. In this problem, you're given the side length of a cube, and asked to construct the side length of a new cube that would have twice the volume of the old one. In other words, given a unit length line segment, construct a line segment of length $\sqrt[3]{2}$. You can do this if you assume you have some extra production rules, and you can approximate it arbitrarily well, but no one managed to give an exact construction with just a straightedge and compass.

In the 1800's geometers stepped back and started asking meta-questions about the fundamental limitations of the rules, a very computer science-y thing to do. They were able to do so due to a couple of revolutionary ideas that had occurred in the years since Rome annexed the Grecian provinces.

The first of these ideas was Cartesian coordinates, named for Descartes in the 1600's. This moves the game to the Cartesian plane. The initial points are $(0, 0)$ and $(1, 0)$. A nonvertical line through (a, b) and (c, d) is described by the function $y = \frac{d-b}{c-a}x + \frac{ad-bc}{a-c}$. A circle centered at (a, b) through (c, d) has the function $(x - a)^2 + (y - b)^2 = (a - c)^2 + (b - d)^2$. Intersection points are the solutions to systems of equations. For the intersection of lines, this is simply a linear system, which is easy to solve. For a line and circle or circle and circle, we get a quadratic system, and the quadratic formula leads us to the solution.

The upshot is that no matter how many systems of equations we solve and new points we draw, all we're doing is taking the original coordinates and applying $+, -, \times, \div$, and taking square roots. In fact, it would be sufficient for our purposes to reinterpret the problem as follows. We start with the numbers 0 and 1, and apply the above operations as often as we'd like. (Note that we can't divide by 0, but square roots of negative numbers are fine.) Can we construct the number $\sqrt[3]{2}$ with these operations?

It seems like we shouldn't be able to muck around with square roots and produce a cube root, and in fact we can't. There's a really nifty proof using Galois theory which we won't talk about because it requires Galois theory.

Even though this example was historically part of pure math, it illustrates many of the themes that today typify theoretical computer science. You have some well-defined set of allowed operations. Using those operations, you build all sorts of beautiful and complex structures—often reusing structures you previously built to build even more complex ones. But then certain kinds of structures, it seems you *can't* build. And at that point, you have to engage in metareasoning. You have to step back from the rules themselves, and ask yourself, what are these rules *really* doing? Is there some fundamental reason why these rules are never going to get us what we want?

As computer scientists we're particularly interested in building things out of ANDs and ORs and NOTs or jump-if-not-equal's and other such digital operations. We're still trying to understand the fundamental limitations of these rules. Note when the rules are applied *arbitrarily many times*, we actually understand pretty well by now what is and isn't possible: that's a subject called *computability theory*, which we'll get to soon. But if we limit ourselves to a "reasonable" number of applications of the rules (as in the famous P vs. NP problem), then to this day we haven't been able to step back and engage in the sort of metareasoning that would tell us what's possible.

7 Euclid's GCD Algorithm

Another example of ancient computational thinking, a really wonderful non-obvious efficient algorithm is Euclid's GCD algorithm. It starts with this question.

How do you reduce a fraction like 510/646 to lowest terms?

Well, we know we need to take out the greatest common divisor (GCD). How do you do that? The grade school method is to factor both numbers; the product of all the common prime factors is the GCD, and we simply cancel them out. But we said before that factoring is believed to be hard. The brute force method is OK for grade school problems, but it won't do for thousand-digit numbers. But just like testing whether a number is prime or composite, it turns out that the GCD problem can be solved in a different, more clever way—one that *doesn't* require factoring.

Euclid's clever observation was that if a number divides two numbers, say 510 and 646, it also divides any integer linear combination of them, say 646 – 510. [Do you see why this is so?] In general, when we divide B by A, we get a quotient q and a remainder r connected by the equation $B = qA + r$, which means that $r = B - qA$, which means that r is a linear combination of A and B!

So finding the GCD of 510 and 646 is the same as finding the GCD of 510 and the remainder when we divide 646 and 510. This is great because the remainder is a smaller number. We've made progress!

$$\text{GCD}(510, 646) = \text{GCD}(136, 510)$$

And we can keep doing the same thing.

$$\text{GCD}(136, 510) = \text{GCD}(102, 136) = \text{GCD}(34, 102) = 34$$

Here we stopped because 34 divides 102, and we know this means that 34 is the GCD of 34 and 102. We could also take it a step further and appeal to the fact that the GCD of any number and 0 is that number: $\text{GCD}(34, 102) = \text{GCD}(0, 34) = 34$.

```
GIVEN: natural numbers A,B
Assume B is the larger (otherwise swap them)
If A is 0 return B
Else find the GCD of (B % A) and A
```

The numbers get smaller every time, and we're working with natural numbers, so we know that we'll arrive at 0. So Euclid's algorithm will eventually wrap up and return an answer. But exactly how many remainders are we going to have to take? Well, exactly how much smaller do these numbers get each time? We claim that $(B \mod A) < B/2$. Can you see why? (Hint: case by whether A is bigger, smaller, or equal to $B/2$.) So the numbers are getting *exponentially* smaller. Every other equal sign, the numbers are half as big as they were before: $102 < 510/2$ and $136 < 646/2$, and so on. This means Euclid's algorithm is pretty great.

QUESTION: Could you speed up Euclid's algorithm if you had lots of parallel processors?

We can speed up the work within each step—say with a clever parallel division algorithm, but it seems impossible to do something more clever because each step depends on the output of the previous step. If you find a way to parallelize this algorithm, you will have discovered the first really striking improvement to it in 2300 years.

8 For further reference

Check out the Wikipedia articles on "Compass and straightedge", "General number field sieve", "Edsger Dijkstra", etc.

6.045J / 18.400J Automata, Computability, and Complexity
Spring 2011