6.080/6.089 Great Ideas in Theoretical Computer Science

MIT, Spring 2008

http://stellar.mit.edu/S/course/6/sp08/6.080/

Place and Time	TR2:30-4PM, 32-155
Instructor	Scott Aaronson aaronson@csail.mit.edu www.scottaaronson.com 32-G638 Office hours: Mondays 1-3PM or by appointment
TA	Yinmeng Zhang ynz@csail.mit.edu 32-G614 Office hours: Wednesdays 6-8PM or by appointment

This course provides a challenging introduction to some of the central ideas of theoretical computer science. It attempts to present a vision of "computer science beyond computers": that is, CS as a set of mathematical tools for understanding complex systems such as universes and minds. Beginning in antiquity—with Euclid's algorithm and other ancient examples of computational thinking—the course will progress rapidly through propositional logic, Turing machines and computability, finite automata, Gödel's theorems, efficient algorithms and reducibility, NP-completeness, the P versus NP problem, decision trees and other concrete computational models, the power of randomness, cryptography and one-way functions, computational theories of learning, interactive proofs, and quantum computing and the physical limits of computation. Class participation is essential, as the class will include discussion and debate about the implications of many of these ideas.

Requirements. Students taking 6.080 will be graded on the following basis:

2/7 problem sets (number to be determined, probably 4-6)
1/7 first quiz
1/7 second quiz
2/7 final exam
1/7 class participation

Students taking 6.089 will be graded on the following basis:

2/9 problem sets
1/9 first quiz
1/9 second quiz
2/9 final exam
1/9 class participation
2/9 scribe notes

The only differences are that 6.089 includes a scribe notes requirement whereas 6.080 does not, and that 6.089 is 12-unit whereas 6.080 is 9-unit. Students should be aware that scribe notes are a nontrivial responsibility (basically a small project), requiring maturity and strong command of the material. Thus, the "default recommendation" is for underclassmen to take 6.080 and for upperclassmen to take 6.089.

Textbooks. While the course will not closely follow any textbook, we will sometimes use *Complexity Theory: A Modern Approach* by Sanjeev Arora and Boaz Barak. A draft of this new book is available for free on the web (www.cs.princeton.edu/theory/complexity), and a printed version will be distributed in a class. In addition, Sipser's *Introduction to the Theory of Computation* (second edition) is highly recommended though not required. Several library copies should have been placed on reserve.

Scribe Notes. Each student taking 6.089 is expected to prepare two scribe notes. A draft of the scribe notes will be due at midnight, one week after the class is given. Notes must be done in LaTeX; a template will be available under the Materials tab on the web page. Please email your draft to Yinmeng, and set up a meeting to discuss them. A polished draft, on which you will be graded, is expected by the end of the course.

Collaboration Policy. Students are welcome to collaborate on problem sets. However, if they do so, they must list the names of collaborators. Collaboration policy for scribe notes will depend on class enrollment.

Prerequisites. This course is designed for undergraduates (both under- and upperclassmen) in computer science and related areas of science and engineering. Others are welcome to take the course but might wish to discuss with the instructor. There are no formal prerequisites. The only prerequisite is some facility with mathematical reasoning: that is, the ability to construct valid mathematical arguments (including proofs by contradiction, induction, etc.) and to recognize errors in invalid ones. Such facility might be gained, for example, by taking 6.042. Programming experience is helpful but not essential; the course has no programming assignments.

Relation to other courses. There is significant overlap between 6.080 and 6.045. The main difference is that 6.080 is an experimental course, which presents many ideas normally postponed until graduate courses, whereas 6.045 provides a more traditional introduction to computability, formal languages, and complexity theory. Students who seek a solid grounding in formal languages, suitable for further work in compilers and other areas, are advised to take 6.045, whereas those wishing to learn about recent insights in theoretical computer science might prefer 6.080. There is also significant overlap between 6.080 and 6.840. However, as a graduate course, 6.840 is able to cover topics in much more mathematical depth. It is hoped that many students taking 6.080 will go on to take 6.840. The overlap between 6.080 and 6.006 (or between 6.080 and 6.046) is smaller.

Organization. The course will consist of three units, lasting approximately one month each. These are:

- Logic, Math, and Machines
- Computational Complexity
- Randomness, Adversaries, and the Physical World

At the end of the first unit, there will be a quiz focusing on that unit only, and similarly at the end of the second unit. At the end of the third unit there will be a cumulative final exam.

Syllabus (extremely approximate).

Logic, Math, and Machines

Ancient computational thinking (Euclid et al.)
Propositional and first-order logic
Finite automata
Turing machines and the halting problem
Oracles and computability
Basic set theory
Gödel's completeness and incompleteness theorems
Philosophical considerations (Penrose and "strong AI")

Computational Complexity

Polynomial time and its justification
Nontrivial examples of polynomial-time algorithms
The concept of a reduction
P, NP, and NP-completeness; the Cook-Levin Theorem
The P versus NP problem and why it's hard
Decision trees and circuits

Randomness, Adversaries, and the Physical World

The power of probabilistic algorithms
Private-key cryptography and one-way functions
Public-key cryptography and trapdoor functions
Pseudorandom number generators
Does randomness really help? The P versus BPP question
Interactive proofs
Zero-knowledge proofs
Computational learning theory
Quantum computing
The ultimate physical limits of computation

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.

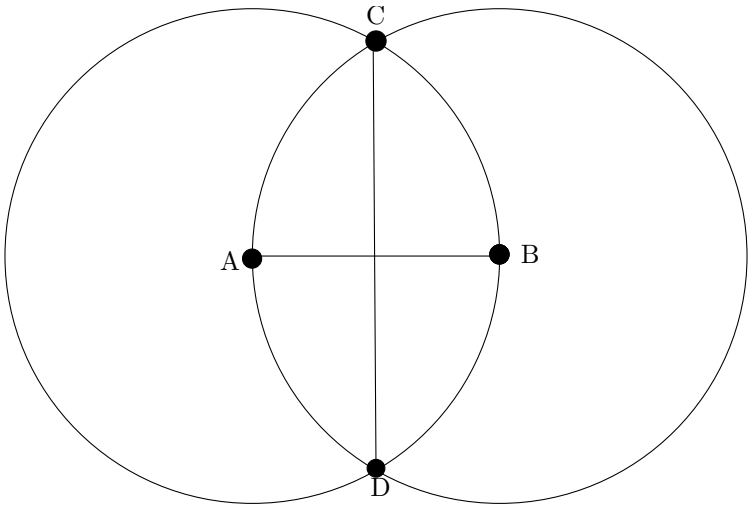
Is this a good protocol? Can the casino cheat? Can the player? The player might try to cheat by sending over a number which is the product of three primes. For example, suppose the factors were A, B, and C, and they ended in 1, 3, and 7 respectively. Then if the casino announces red, the player could 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.

6.080/6.089 GITCS Feb 07, 2008

Lecture 2

Lecturer: Scott Aaronson *Scribe: Mergen Nachin*

Administrative announcements:

- Two scribe notes per student needed.
- If you've already taken 6.840, there's really no reason for you to take this class, unless you enjoy my jokes. But if you take this class, there's reason to take 6.840.

1 Review of last lecture

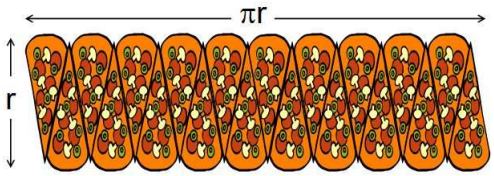
Last week, we talked about online gambling and computation in the ancient world. What I find interesting, incidentally, is that so many basic mathematical ideas were discovered independently by multiple cultures (the Greeks, Mayans, Indians, Chinese, etc). To me, this is a striking empirical refutation of the idea that math is just a "cultural construct." To give one example, Pascal's triangle was discovered in China around 1000 AD – and is instantly recognizable as soon as you see it (a print of the ancient Chinese Pascal's triangle is passed around in class).

We also talked about Euclidean geometry as a model of computation. Euclid laid down a set of simple, clear rules that one can repeatedly apply to construct complicated objects. We also talked about Euclid's GCD algorithm, which was one of the first non-trivial algorithms known to humankind.

Here is some digression. Area of a circle is $A = \pi r^2$. It's obvious that the area of a circle should go like the r^2; the question is why the constant of proportionality (π) should be the same one that relates circumference to diameter - $2\pi r$.

- *Student: can be observed using differential equation.*
- *Scott: yes, that can be one way. But was calculus invented in that time? If you don't have calculus, you can do this by "Sicilian pizza argument".*

Proof by pizza: Cut a circle of radius r into thin pizza slices, and then "Sicilianize" (i.e. stack the slices into a rectangle of height r and length πr).



(Figure taken from http://www.scottaaronson.com/democritus/lec2.html)

2 Today: Logic and Proof

These might seem like dry topics, but they're prerequisites to almost everything else in the course. What does it mean to think logically? Can we formalize what we mean by logical thought, and subject it to logical scrutiny itself?

The credit for being the first logician is usually given to Aristotle, who formalized the concept of the syllogism.

All men are mortal, Socrates is man, therefore Socrates is a mortal.

This is a syllogism. In more modern language, we call it transitivity of implications. In general, a syllogism is

If A $\Rightarrow$ B is valid and B $\Rightarrow$ C is valid, then A $\Rightarrow$ C is valid.

Remark: What do we mean by "$\Rightarrow$"? "A $\Rightarrow$ B" is valid if A is false or B is true or both. If A is true, B has to be true. If A is false, B could be either true or false.

You all know a false statement implies anything? "I am one, the Pope is one, therefore I and the Pope are one." ("proof" of 1+1=1?)

How many of you have seen the puzzle with four cards?

B 5 2 J

Each card has a number on one side and a letter on the other. Which cards would you have to turn over, to test the rule that if there's a J on one side there's a 5 on the other side? You would have to turn J and the 2, not the 5. 80-90 percent of college students get this wrong.

On the other hand, suppose you ask people the following: you say, you're a bouncer in a bar, and you want to make sure the rule "If you are under 21 the you are not drinking". Who do you have to check to test this rule: someone who is drinking, someone who isn't drinking, someone who's over 21, someone who's under 21?

And then, of course, almost everyone gets it right. Even though this problem is logically equivalent in every way to the other problem.

And this brings up a fundamental point about our brains. We're designed for spearing small animals. Not for proving theorems. This class is all about doing things that your brains are *not* designed for. The trick is to co-opt parts of your brain that evolved for something else. "You over there: you're supposed to track leopards leaping across the savanna? Well, now those leopards are going to be arbitrary vectors $v \in \Re^3$. Deal with it."

Remark: Implication is actually going on at two different levels. There are these "implication" arrows inside the statements that the sentence is talking about, then there's also the sentence that's talking about them. These three sentences you can think of as meaningless pieces of code, the sentence is addressed to us; it's telling us one of the rules of the code.

Was Aristotle the first person in history to apply such an inference? Obviously he wasn't. As we all see in our everyday life, everyone draws inferences all the time. However, what he was (as far as I know) was the first person in the historical record to (as it were) draw a box around the inference rule, to say that this is a general law of thought.This is crucial because it allows us to reason about the rule itself.

2.1 Leibniz and the Calculus Ratiocinator: "Gentlemen, let us calculate!"

Today, Leibniz's dream that we could use an automatic reasoning machine to settle legal disputes strikes us as naïve, especially when we think about the sorts of arguments that actually get used in courtrooms: "If it does not fit, you must acquit," etc.

More to the point, in real court cases, often the difficulty is not that people don't draw the right inferences from the facts, it's that they don't agree about the facts! Or the facts are probabilistic and uncertain, and people don't agree how much weight different facts entered into evidence should be assigned. On top of that, the laws themselves are necessarily vague, and people disagree in their preferred interpretation of the law.

Nevertheless, this idea of Leibniz that we could automate even part of human thought was extremely bold for its time.

Moreover, Leibniz had what today we would see as the right picture of what such a machine would be. To him, it's not that you would take some lifeless clump of clay, and utter some mystical incantation that would magically imbue it the power of speech – like in the legend of Pinocchio, or the legend of the Golem, or even in a lot of science fiction. Leibniz's idea, rather, was that you'd "merely" be building a complicated machine. Any individual gear in the machine would not be thinking – it's just a gear. But if you step back and consider *all* the gears, then it might look like it was thinking.

On this view, the role of logic is to tell us what are the "atoms of thought" that we would need to build a reasoning machine.

If you know A⇒B and B⇒C, and you conclude A⇒C, you really haven't done much in the way of thinking. We do this sort of thinking every morning: "My socks go on my feet, these are my socks, therefore these go on my feet."

Yet suppose you strung together hundreds or thousands of these baby steps. Then maybe you'd end up with the most profound thought in the history of the world! Conversely, if you consider the most profound thoughts anyone ever had, how do we know they *can't* be analyzed into thousands of baby steps? Maybe they can! Many things seem magical until you know the mechanism. So why not logical reasoning itself?

To me, that's really the motivation for studying logic: to discover "The Laws of Thought." But to go further, we need to roll up our sleeves, and talk about some real examples of logical rule systems.

Maybe the simplest interesting system is the one where every statement has the form

- $A \Rightarrow B$
- $\neg A \Rightarrow B$
- $A \Rightarrow \neg B$ or
- $\neg A \Rightarrow \neg B$

and the only rules are:

- Given $A \Rightarrow B$ and $B \Rightarrow C$, you can deduce $A \Rightarrow C$.
- Given $\neg A \Rightarrow A$ and $A \Rightarrow \neg A$, you can deduce a contradiction.

Can we have both $\neg A \Rightarrow A$ and $A \Rightarrow \neg A$ valid at the same time? If we assign A=false then $A \Rightarrow \neg A$ is valid. But $\neg A \Rightarrow A$ is not valid. Similarly if we assign A=true then $A \Rightarrow \neg A$ is not valid. Now consider the following example.

- $A \Rightarrow B$

- $\neg C \Rightarrow A$
- $\neg A \Rightarrow \neg C$
- $B \Rightarrow \neg A$

Can these sentences simultaneously be satisfied? I.e. is there some way of setting A,B,C,D to "true" or "false" that satisfies all four sentences?

No. By applying the rules, we can reach a contradiction! You agree that if we reach a logical contradiction by applying the rules, then the sentences can't all be valid?

Suppose a set of sentences is inconsistent (i.e., there's no way of setting the variables so that all of them are satisfied). Can we always discover the contradiction always by applying the above rules?

Yes, we can always discover the contradiction. You can just imagine a big graph. The nodes are the variables and their negations. A,¬ A,B,¬ B,C, ¬ C.... Place directed edge from node A to node B whenever $A \Rightarrow B$. Whenever we apply a rule, we can think as if we were walking through the graph. So $A \Rightarrow B$, $B \Rightarrow C$ then $A \Rightarrow C$ actually means C is reachable from A. Start with A=true and if we reach ¬ A then it means $A \Rightarrow \neg A$. If we also end up connecting $\neg A$ and A, in other words if we have cycle, then we have discovered a contradiction.

What we're talking about are two properties of logical systems called "soundness" and "completeness."

Soundness: Any statement you get by cranking the rules is true. (That's a pretty basic requirement.)

Completeness: Any statement that's true, you can get by cranking the rules.

In the next lecture: a bit of first-order logic!

6.080/6.089 GITCS Feb 12, 2008

Lecture 3

Lecturer: Scott Aaronson *Scribe: Adam Rogal*

1 Administrivia

1.1 Scribe notes

The purpose of scribe notes is to transcribe our lectures. Although I have formal notes of my own, these notes are intended to incorporate other information we may mention during class - a record for future reference.

1.2 Problem sets

A few comments on the problem sets. Firstly, you are welcome to collaborate, but please mark on your problem sets the names of whom you worked with. Our hope is to try all the problems. Some are harder than others; there are those marked challenge problems as well. If you can't solve the given problem, be sure to state what methods you tried and your process up to the point you could not continue. This is partial credit and much better than writing a complete, but incorrect solution. After all, according to Socrates *the key to knowledge is to know what you don't know.*

1.3 Office hours

We will have office hours once a week.

2 Recap

2.1 Computer science as a set of rules

We can view computer science as the study of simple set of rules and what you can and can't build with them. Maybe the first example of that could be considered Euclidian geometry. And the key to discovering what processes we can build is that these rules are well-defined.

2.2 Logic

The field of logic focuses on automating or systematizing not just any mechanical processes, but rational thought itself. If we could represent our thoughts by manipulations of sequences of symbols, then in principle we could program a computer to do our reasoning for us.

We talked the simplest logical systems which were the only ones for thousands of years. Syllogisms and propositional logic, the logic of Boolean variables that can be either true or false, and related to each other through operators like and, or, and not. We finally discussed first order logic.

2.2.1 First order logic

The system of first order logic is built up of sentences. Each of these sentences contain variables, such as x, y, and z. Furthermore we can define functions which take these variables as input.

For example: let's define a function $Prime(x)$. Given an integer, it will return true if the number is prime, false if it is composite. Just like functions in any programming languages, we can build functions out of other functions by calling them as subroutines. In fact, many programming languages themselves were modeled after first order logic.

Furthermore, as in propositional logic, symbols such as $\wedge$ (and), $\vee$ (or), $\neg$ (not), and $\rightarrow$ (implies) allow us to relate objects to each other.

Quantifiers are a crucial part of first order logic. Quantifiers allow us to state propositions such as "Every positive integer x is either prime or composite."

$$\forall x.Prime(x) \vee Composite(x)$$

There's a counterexample, of course, namely 1. We can also say: "There exists an x, such that something is true."

$$\exists x.Something(x)$$

When people talk about first-order logic, they also normally assume that the equals sign is available.

2.2.2 Inference rules

We want a set of rules that will allow us to form true statements from other true statements.
Propositional tautologies:

$$A \vee \neg A$$

Modus ponens:

$$A \wedge (A \rightarrow B) \rightarrow B$$

Equals:

$$Equals(X, X)$$

$$Equals(X, Y) \iff Equals(Y, X)$$

Transitivity property:

$$Equals(X, Y) \wedge Equals(Y, Z) \rightarrow Equals(X, Z)$$

Furthermore, we have the rule of change of variables. If you have a valid sentence, that sentence will remain valid if we change variables.

2.2.3 Quantifier rules

If $A(x)$ is a valid sentence for any choice of x, then for all x, $A(x)$ is a valid sentence. Conversely, if $A(x)$ is a valid sentence for all x, then any $A(x)$ for a fixed x is a valid sentence.

$$A(X) \iff \forall x.A(x)$$

We also have rules for dealing with quantifiers. For example, it is false, that for all x, $A(x)$ iff there exists an x, $\neg A(x)$.

$$\neg\forall x.A(x) \iff \exists\neg A(x)$$

2.2.4 Completeness theorem

Kurt Gödel proved that the rules thus stated were all the rules we need. He proved that if you could not derive a logical contradiction by using this set of rules, there must be a way of assigning variables, such that all the sentences are satisfied.

3 Circuits

Electrical engineers views circuits to be complete loops typically represented in figure 1. However, in computer science, circuits have no loops and are built with logic gates.

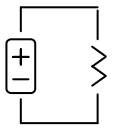


Figure 1: A simple EE circuit.

3.1 Logic gates

The three best-known logic gates are the *NOT*, *AND*, and *OR* gates shown in figure 2.

Figure 2: The logical gates *NOT*, *AND*, and *OR*.

Though primitive on their own, these logic gates can be strung together to form complex logical operations. For example, we can design a circuit, shown in figure 3, that takes the majority of 3 variables: x, y, and z. We can also use De Morgan's law to form a *AND* gate from an *OR* gate and vice versa as shown figure 4.

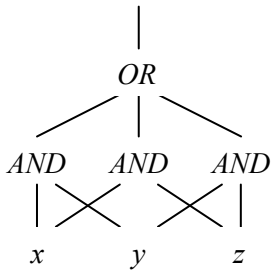


Figure 3: The majority circuit.

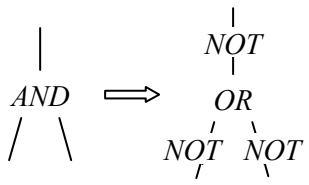


Figure 4: An *AND* gate can be constructed from an *OR* and three *NOT* gates by using De Morgan's law.

These logic gates can also be combined to form other gates such as the *XOR* and *NAND* gates shown in figure 5. Conversely, by starting with the *NAND* gate, we can build any other gate we want.

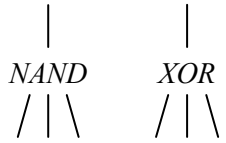


Figure 5: *NAND* and *XOR* gates.

On the other hand, no matter how we construct a circuit with *AND* and *OR* gates, if the input is all 1's we can never get an output of 0. We call a Boolean function that can be built solely out of *AND* and *OR* gates a *monotone* Boolean function.

Are there any other interesting sets of gates that *don't* let us express all Boolean functions? Yes: the *XOR* and *NOT* gates. Because of their linearity, no matter how we compose these gates we can never get functions like *AND* and *OR*.

4 Puzzle

Here's an amusing puzzle: can you compute the NOT's of 3 input variables, using as many AND/OR gates as you like but only 2 NOT gates?

4.0.1 Limitations

Although we have discovered that circuits can be a powerful tool, as a model of computation they have some clear limitations. Firstly, circuits offer no form of storage or memory. They also have no feedback; the output of a gate never gets fed as the input. But from a modern standpoint, the *biggest* limitation of circuits is that (much like computers from the 1930s) they can only be designed for a fixed-size task. For instance, one might design a circuit to sort 100 numbers. But to sort 1000 numbers, one would need to design a completely new circuit. There's no *general-purpose* circuit for the sorting task, one able to process inputs of arbitrary sizes.

5 Finite automata

We'll now consider a model of computation that *can* handle inputs of arbitrary length, unlike circuits – though as we'll see, this model has complementary limitations of its own.

5.1 Description

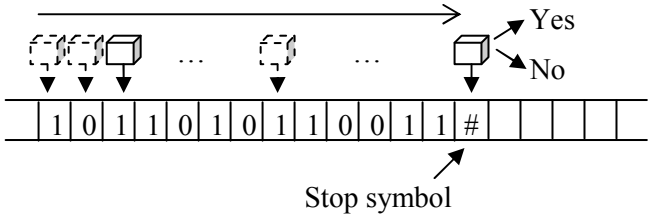


Figure 6: At any given time, the machine ahs some unique state. The machine reads the tape in one motion (in this case left to right) and the state changes depending on the value of the current square. When the reaches the stop state (signaled by the # sign, the machine returns a yes or no answer - an accept or reject state respectively.)

The simple way of thinking of a finite automaton is that it's a crippled computer that can only move along memory in one direction. As shown in figure 6, a computer with some information written along a tape, in some sort of encoding, will scan this tape one square at a time, until it reaches the stop symbol. The output of this machine will be a yes or no - accept or reject. This will be the machine's answer to some question that it was posed about the input.

5.2 State and internal configuration

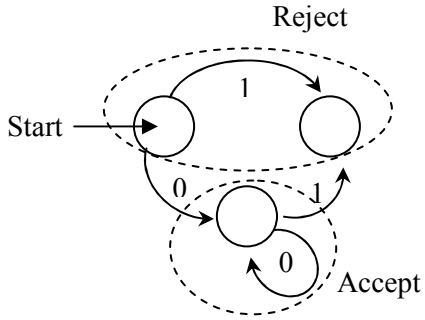


Figure 7: This simple machine has 3 states. Given an input of 0 or 1, the state will transition to a new state. The final state will determine its output - accept or reject.

It is unnecessary to determine what the internal configuration of this machine is. We can abstract this notion into the statement that this machine will have some state and the ability to transition between states given a certain input. The machine will begin with a start state, before it has read any input. When the machine reads the stop symbol, the correct state will determine if the machine should output an accept or reject.

It is crucial that we define the machine as having a finite number of states. If the machine had an infinite number of states, then it could compute absolutely *anything*, but such an assumption is physically unrealistic.

5.3 Some examples

Let us design a machine that determines if any 1's exist in a stream given the alphabet of 0 or 1. We define two states of the machine - 0 and 1. The 0 represents the state that the machine has not seen a 1 yet. The 1 state represents the state that the machine has seen a 1. When the machine has transitioned to the 1 state, neither a 1 or 0 will ever change the state back to 0. That is, regardless of input or length of input, our question, "Are there any 1's in the stream?" has been answered. Therefore, the 1 state should produce an accept, while the 0 state should produce a reject when a stop symbol has been reached.

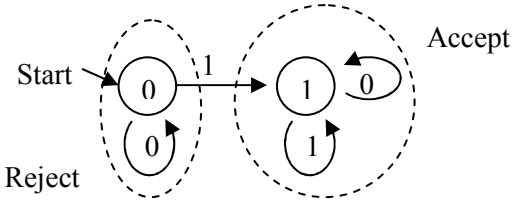


Figure 8: This FA determines if any 1's exist in our data stream.

Let us now design a machine that determines if the number of 1's is even or odd in the stream. We define two states again - 0 and 1. The 0 state represents a machine that has seen an even number of 1's and the 1 state describes a machine that has seen an odd number of 1's. An input of 0 will only transition the state to itself. That is, we are only concerned about the number of 1's in this stream. At each input of a 1, the machine will alternate state between 0 and 1. The final state will determine if the data stream has seen an even or odd number of 1's, with 1 being set as the acceptance state.

It should be noted that regardless of input size, this machine will determine the correct answer to the question we posed. Unlike with circuits, our machine size was not dictated by the size of the input.

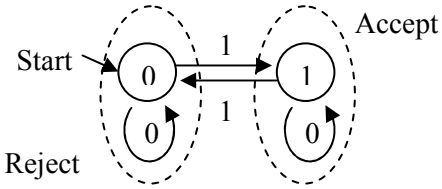


Figure 9: This FA determines if there are an even or odd number of 1's in our data stream.

5.4 Palindromes

Let us now explore if we could create a finite machine that can determine if an input string is a palindrome, a string that reads the same backwards and forwards. The input will be finite, and there will be a terminator at the end. We begin by defining the possible states of the machine. If we let our machine contain 2^N states, then as shown in figure 10, we could just label each final leaf as an accept or reject for every possible sequence of 1's and 0's.

The question still remains, can we create a machine with a finite number of states that can

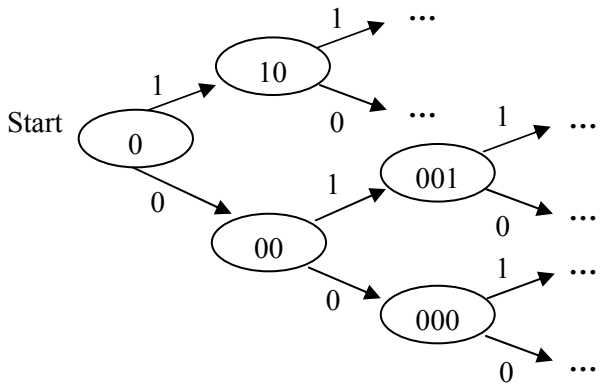


Figure 10: For a stream of N bits, a finite automaton, intended to determine if the stream is a palindrome, grows exponentially. For N bits, 2^N states are required.

act as a palindrome detector. The answer lies in using the Pigeonhole Principle to analyze the limitations of finite automata.

5.5 The Pigeonhole Principle

The Pigeonhole Principle states that if we have N pigeons and we want put them into $N-1$ holes, at least one hole will have two or more pigeons. Although very simple, this principle allows us to prove that no finite automaton can act as a palindrome detector.

5.5.1 A digression: proving the pigeonhole principle

Even though the pigeonhole principle is simple, it is non-trivial to prove in simple systems of logic. We can express the requirements that every pigeon goes into some hole, and that no two pigeons go into the same hole, using propositional logic. The challenge is then to prove that not all the statements can be true, using only mechanical logical manipulation of the statements (and not higher-order reasoning about what they "mean").

In other words, the pigeonhole principle seems obvious to us because we can stand back and see the larger picture. But a propositional proof system like the ones we saw in the last lecture can't do this; it can only reason locally. ("Let's see: if I put this pigeon here and that one there ... darn, *still* doesn't work!") A famous theorem of Haken states that any proof of the Pigeonhole Principle based on "resolution" of logical statements, requires a number of steps that increases exponentially with N (the number of pigeons). This is an example of something studied by a field called *proof complexity*, which deals with questions like, "does any proof have to have a size that is exponentially larger than the theorem we are trying to prove?"

5.6 Using the Pigeonhole Principle for palindromes

We use the Pigeonhole Principle to prove that no finite automaton that can be constructed such that we can detect if any string is a palindrome.

To begin this proof, let us split a palindrome down the middle. We will ignore everything about the finite automaton except its state at the middle point; any information that the automaton will carry over to the second half of the string, must be encoded in that state.

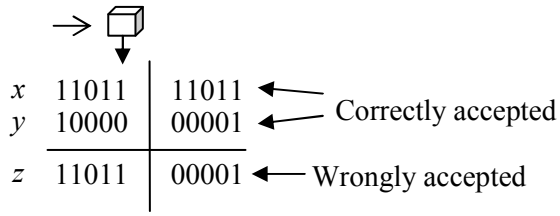


Figure 11: By using the Pigeonhole principle, we can show that we can split two strings at their reflection points such that a finite automaton will be in at the same state for both sub strings. We can then cross the two strings to form a new string that "tricks" the machine into thinking that it has correctly accepted a string as a palindrome.

A finite automaton must have a fixed number of states. On the other hand, there are infinitely many possibilities for the first half of the string. Certainly, you can't put infinitely many pigeons into a finite number of holes without having at least one hole with at least two pigeons. This means that there is at least one state that does "double duty," in that two different first halves of the string lead to the same state.

As shown in figure 11, we consider two palindromes x and y. If the machine works correctly, then it has to accept both of them. On the other hand, for some x, y pair, the machine will lie in the same state for both x and y when it's at the halfway point. Then by crossing the remaining halves of x and y, we can create a new string, z, which is accepted by the machine even though it's not a palindrome. This proves that no finite automaton exists that recognizes all and only the palindromes.

5.7 Regular expressions

Regular expressions allow us to search for a keyword in a large string. Yet, they are more powerful than simply searching for the keyword 110 in the string 001100. We can use regular expressions to locate patterns as well.

For example, we can create an expression like (0110)|(0001) which will either match the keyword 0110 or 0001. We can also create expressions that will find any 3 bit string with a 1 in the middle: (0|1)1(0|1).

We can also use more advanced characters such as the asterisk to represent repetition. (0|1)1(0|1)0* searches for any 3 bit string with a 1 in the middle followed by any number of 0's. We can also repeat larger patterns such as [(0|1)1(0|1)]*. This states that we would like to match any number of 3 bit strings with 1's in the middle. It should be noted that each time the pattern repeats, the 0 or 1's can be chosen differently.

We can now state (without proof) a very interesting theorem: any language is expressible by a regular expression, if and only if it's recognized by a finite automaton. Regular expressions and finite automaton are different ways of looking at the same thing.

To give an example: earlier we created a finite automaton that was able to recognize all strings with an even number of 1's. According to the theorem, there must be regular expression that generates this same set of strings. And indeed there is: 0*(0*10*1)*.

6 Nondeterministic finite automata

Nondeterministic finite automata represent machines that can not only transition between states, but between sets of states. As before, we have a machine that reads a tape from left to right with a finite number of states. When the machine reads an input, each state that the machine is now on, is allowed to transition to any other states emanating from the previous states based on the input. The machine is in acceptance if any final state is an accept state.

You might guess that NDFA's (nondeterministic finite automata) would be much more powerful than DFA's (deterministic finite automata). This is not the case, however: given an NDFA with N states, we can always simulate it by a DFA with 2^N states, by creating a single state in the DFA to represent each *set* of states in the NDFA.

6.080/6.089 GITCS	Feb 14, 2008
Lecture 4	
Lecturer: Scott Aaronson	*Scribe: Aseem Kishore*

1 Previously in 6.089...

Last lecture, we talked about two different models of computation, *finite automata* and *circuits*. Finite automata allowed us to recognize many properties of an arbitrarily long input, while circuits allowed us to represent any Boolean function.

However, both models had significant limitations. Circuits were limited in hardware–we have to know how big an input is before we can make a circuit for it–while finite automata were limited by their memory–which also had to be known in advance of a problem.

2 Turing Machines

How can we generalize finite automata to overcome their limitations? A first idea is to let them move backwards on the tape as well as forwards. This is a good start, but by itself it actually provides no extra power. To see why, suppose a two-way finite automaton is in some state a, going forward at some point x on the tape. At this point, if it goes backwards on the tape and then returns to x, that simply induces some function $a \Rightarrow f(a)$, where f depends on the earlier part of the tape. But this means that we can simulate the two-way machine by a one-way machine, which simply keeps track of the whole function f instead of just the state a.[1] Thus, being able to move in two directions provides no additional power on its own.

What we really need is a machine that can not only move backwards and forwards, but also *write* to the tape and *halt* at any time of its choosing. And that's what a Turing machine is. The ability to write essentially gives Turing machines an unlimited memory, since any information that can't fit in the machine's internal state can always be written to the tape. The ability to halt at discretion means that Turing machines aren't "tied to the input" the way finite automata are, but can do as much auxiliary computation as they need.

So at any given point on the tape, a Turing machine faces three questions:

1. Change state?
2. Write to the tape?
3. Move left, move right, or halt?

The machine's answer to these questions is a function of its current state and the current input on the tape.

A clear example of these features overcoming the limitations of finite automata is a Turing machine's ability to solve the palindrome problem. By using a simple back-and-forth process, a

[1] Note that the number of functions $f(a)$ mapping states to states grows exponentially with the number of states in the original machine.

Turing machine can repeatedly check that a letter at one end exists at the opposite end, and by marking letters that it has seen, the machine ensures it continuously narrows its scope. (Here we're assuming that additional symbols are available besides just 0 and 1.) This algorithm takes $O(n^2)$ time (interestingly, there is a proof that argues that this is the best a Turing machine can do).

Likewise, addition of integers is also possible, as are multiplying and some other mathematical operations. (We won't prove that!) Searching for non-regular patterns also becomes possible. But perhaps the most interesting thing a Turing machine can do is to emulate another Turing machine!

3 Universal Turing Machines

In his 1936 paper "On Computable Numbers" (in some sense, the founding document of computer science), Turing proved that we can build a Turing machine U that acts as an interpreter for other Turing machines. In other words, U's input tape can contain a description of *another* Turing machine, which is then simulated step by step. Such a machine U is called a *Universal Turing machine*. If a universal machine *didn't* exist, then in general we would need to build new hardware every time we wanted to solve a new problem: there wouldn't even be the concept of *software*. This is why Professor Aaronson refers to Turing's universality result as the "existence of the software industry lemma"!

A question was brought up in class as to how this can be, if the machine being interpreted may require more states than the interpreting Turing machine has. It turns out that universal Turing machines aren't limited by their states, because they can always keep extra state on blank sections of the tape. They can thus emulate a machine with any number of states, but themselves requiring only a few states. (In fact, there is a popular parlor-type competition to find a universal Turing machine that uses as few states and symbols as possible. Recently, one student actually came up with such a machine that uses only two states and a three-symbol alphabet. To be fair, however, the machine required the inputs in a special format, which required some pre-computation, so a question arises as to how much of the work is being done by the machine versus beforehand by the pre-computation.)

4 The Church-Turing Thesis

Related to the idea of universal machines is the so-called *Church-Turing thesis*, which claims that anything we would naturally regard as "computable" is actually computable by a Turing machine. Intuitively, given any "reasonable" model of computation you like (RAM machines, cellular automata, etc.), you can write compilers and interpreters that translate programs back and forth between that model and the Turing machine model. It's never been completely clear how to interpret this thesis: is it a claim about the laws of physics? about human reasoning powers? about the computers that we actually build? about math or philosophy?

Regardless of its status, the Church-Turing Thesis was such a powerful idea that Gödel declared, "one has for the first time succeeded in giving an absolute definition to an interesting epistemological notion."

But as we'll see, even Turing machines have their limitations.

5 Halting is a problem

Suppose we have a Turing machine that never halts. Can we make a Turing machine that can detect this? In other words, can we make an infinite loop detector? This is called the *Halting problem*.

The benefits of such a machine would be widespread. For example, we could then prove or disprove Goldbach's Conjecture, which says that all even numbers 4 or greater are the sum of two primes. We could do this by writing a machine that iterated over all even numbers to test this conjecture:

```
for i = 2 to infinity:
  if 2*i is not a sum of two primes
    then HALT
```

We would then simply plug this program into our infinite-loop-detecting Turing machine. If the machine detected a halt, we'd know the program must eventually encounter a number for which Goldbach's conjecture is false. But if it detected no halt, then we'd know the conjecture was true.

It turns out that such an infinite loop detector can't exist. This was also proved in Turing's paper, by an amazingly simple proof that's now part of the intellectual heritage of computer science.[2]:

We argue by contradiction. Let P be a Turing machine that solves the halting problem. In other words, given an input machine M, $P(M)$ accepts if $M(0)$ halts, and rejects if $M(0)$ instead runs forever. Here $P(M)$ means P run with an encoding of M on its input tape, and $M(0)$ means M run with all 0's on its input tape. Then we can easily modify P to produce a new Turing machine Q, such that $Q(M)$ runs forever if $M(M)$ halts, or halts if $M(M)$ runs forever.

Then the question becomes: what happens with $Q(Q)$? If $Q(Q)$ halts, then $Q(Q)$ runs forever, and if $Q(Q)$ runs forever, then $Q(Q)$ halts. The only possible conclusion is that the machine P can't have existed in the first place.

In other words, we've shown that the halting problem is *undecidable*–that is, whether another machine halts or not is not something that is *computable* by Turing machine. We can also prove general uncomputability in other ways. Before we do so, we need to lay some groundwork.

6 There are multiple infinities

In the 1880's, Georg Cantor discovered the extraordinary fact that there are different degrees of infinity. In particular, the infinity of real numbers is greater than the infinity of integers.

For simplicity, let's only talk about positive integers, and real numbers in the interval $[0, 1]$. We can associate every such real number with an infinite binary string: for example, 0.0011101001.... A technicality is that some real numbers can be represented in two ways: for example, $0.100\overline{0}$ is equivalent to $0.011\overline{1}$. But we can easily handle this, for example by disallowing an infinity of trailing 1's.

To prove that there are more real numbers than integers, we'll argue by contradiction: suppose the two infinities are the same. If this is true, then we must be able to create a one-to-one association, pairing off every positive integer with a real number $x \in [0, 1]$. We can arrange this association like so:

[2]This proof also exists as a poem by Geoffrey K. Pullum entitled "Scooping the Loop Snooper": http://www.ncc.up.pt/∼rvr/MC02/halting.pdf

```
1: 0.0000... (rational)
2: 0.1000...
3: 0.0100...
4: 0.101001000100001... (irrational)
5: 0.110010110001001...
```

We can imagine doing this for all positive integers. However, we note that we can construct another real number whose n^{th} digit is the opposite of the n^{th} digit of the n^{th} number. For example, using the above association, we would get 0.11110...

This means that, contrary to assumption, there were additional real numbers in $[0, 1]$ not in our original list. Since every mapping will leave real numbers left over, we conclude that there are more real numbers than integers.

If we try to apply the same proof with rational numbers instead of real numbers, we fail. This is because the rational numbers are *countable*; that is, each rational number can be represented by a finite-length string, so we actually can create a one-to-one association of integers to rational numbers.

7 Infinitely many unsolvable problems

We can use the existence of these multiple infinities to prove that there are uncomputable problems. We'll begin by showing that the number of possible Turing machines is the smallest infinity, the infinity of integers.

We can define a Turing machine as a set of states and a set of transitions from each state to another state (where the transitions are based on the symbol being read). A crucial aspect of this definition is that both sets are finite.

Because of this, the number of Turing machines is *countable*. That is, we can "flatten" each machine into one finite-length string that describes it, and we can place these strings into a one-to-one association with integers, just as we can with rational numbers.

The number of *problems*, on the other hand, is a greater infinity: namely, the infinity of real numbers. This is because we can define a problem as a function that maps every input $x \in 0, 1^*$ to an output (0 or 1). But since there are infinitely many inputs, to specify such a function requires an infinite number of bits. So just like with Cantor's proof, we can show that the infinity of problems is greater than the infinity of Turing machines.

The upshot is that there are far more problems than there are Turing machines to solve them. From this perspective, the set of computable problems is just a tiny island in a huge sea of unsolvability. Admittedly, most of the unsolvable problems are not things that human beings will ever care about, or even be able to define. On the other hand, Turing's proof of the unsolvability of the halting problem shows that at least *some* problems we care about are unsolvable.

6.080/6.089 GITCS	21 February 2008
Lecture 5	
Lecturer: Scott Aaronson	*Scribe: Emilie Kim*

1 Administrivia

When it is your turn to do scribe notes, please have a rough draft prepared within one week. Then you have until the end of the course to get the notes in better shape, and you'll want to do that because your final grade depends on it. After you submit your rough draft, schedule a meeting with Yinmeng to discuss your notes.

2 Review Turing Machines

Turing machines can go backwards and forwards on the tape, as well as write to the tape and decide when to halt. Based on this idea of Turing machines comes the "Existence of the Software Industry Lemma", which states that there exist universal Turing machines that can simulate any other Turing machine by encoding a description of the machine on its tape.

The Church-Turing thesis says that Turing machines capture what we mean by the right notion of computability. Anything reasonably called a computer can be simulated by a Turing machine. However, there are limitations associated with Turing machines. For example, no Turing machine can solve the halting problem (Proof by poem). Also, the number of possible problems is far greater than the number of computer programs. Remember the infinity of real numbers versus the infinity of integers.

But who cares? That's why Turing brought up the halting problem; we actually care about that.

3 Oracles

Oracles are a concept that Turing invented in his PhD thesis in 1939. Shortly afterwards, Turing went on to try his hand at breaking German naval codes during World War II. The first electronic computers were used primarily for this purpose of cracking German codes, which was extremely difficult because the Germans would change their codes every day. The Germans never really suspected that the code had been broken, and instead thought that they had a spy among their ranks. At one point they did add settings to the code, which then set the code-breakers back about nine months trying to figure out how to break the new code.

3.1 Oracles

An *oracle* is a hypothetical device that would solve a computational program, free of charge. For example, say you create a subroutine to multiply two matrices. After you create the subroutine, you don't have to think about how to multiply two matrices again, you simply think about it as a

"black box" that you give two matrices and out comes their product.

However, we can also say that we have oracles for problems that are unsolvable or problems that we don't know how to solve. Assuming that the oracle can solve such problems, we can then create *hierarchies of unsolvability*. If given many hard problems, we can use hierarchies of unsolvability to tell us which problems are hard relative to which other problems. It can tell us things like "this problem can't be solvable, because if it were, it would allow us to solve this other unsolvable problem".

Given:

$$A : \{0,1\}^* \to \{0,1\}$$

where the input is any string of any length and the output is a 0 or 1 answering the problem, then we can write

$$M^A$$

for a Turing machine M with access to oracle A. Assume that M is a multi-tape Turing machine and one of its tapes is a special "oracle tape". M can then ask a question, some string x, for the oracle on the oracle tape, and in the next step, the oracle will write its answer, $A(x)$, on the oracle tape.

From this, we can say that given two problems A and B, A is *reducible* to B if there exists a Turing machine M such that M^B solves A. We write this as $A \leq_T B$.

3.2 Example 1: Diophantine equations

Given:

$$x^n + y^n = z^n$$

$$n \geq 3$$

$$xyz \neq 0$$

is there a solution in just integers?

In fact, there is no solution involving just integers, and this was an unsolved problem for 350 years. What if we had an oracle for the halting problem? Could we solve this problem?

To try to solve this problem, we might try every possible solution in order, (x, y, z, n) of integers:

$$\begin{aligned} x + y + z + n &= 1 \\ x + y + z + n &= 2 \\ &= 3 \\ &= ... \end{aligned}$$

and try them all, one by one in order, and then pause when we find the solution. However, if there is no solution, it would just go on forever. So our question to the oracle would be, "Does this program halt or not?", and if so, the Diophantine equation would be solvable.

If we solve the Diophantine problem, can we also then solve the halting problem?

This was an unanswered question for 70 years until 1970, when it was shown that there was no algorithm to solve Diophantine equations because if there were, then there would also be an algorithm for solving the halting problem. Therefore, the halting problem is reducible to this problem.

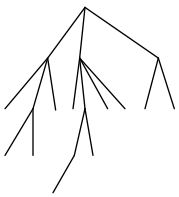
3.3 Example 2: Tiling the plane

Given some finite collection of tiles of all different shapes, can we fill an entire plane just using these tiles? For simplicity, let us assume that all the tiles are 1x1 squares with different-shaped notches in the sides.

The halting problem is reducible to this problem, which means if you could solve this problem, you could then also solve the halting problem. Why is this? Assume you could create a set of tiles that would only link together in some way that encodes the possible actions of a Turing machine. If the Turing machine halts, then you wouldn't be able to add any more tiles. Then the plane would only be tiled if the machine runs forever.

Is this problem reducible to the halting problem? In other words, if you could solve the halting problem, then can you decide whether a set of tiles will tile the plane or not? Can you tile a 100x100 grid? Can you tile a 1000x1000 grid? These questions can be answered by a Turing machine. But suppose every finite region can be filled, why does it follow that the whole infinite plane can be tiled?

In comes **König's Lemma**. Let's say that you have a tree (a computer science tree, with its root in the sky and grows towards the ground) with two assumptions:



1) Every node has at most a finite number of children, including 0.
2) It is possible to find a path in this tree going down in any finite length you want.

Claim: The tree has to have a path of infinite length.

Why is this? The tree has to be infinite because if it were finite, then there would exist a longest path. We don't know how many subtrees there are, but there are a finite number of subtrees at the top level. From that, we can conclude that one of the subtrees has to be infinite because the sum of the number of vertices in all the subtrees is infinite and there is only a finite number of subtrees

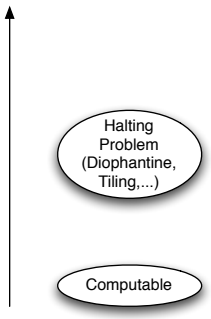
so one of the subtrees has to have infinitely many vertices.

We'll start at the top and move to whichever node has infinitely many descendants. We know there has to be at least one of them, as we just stated previously. If we repeat this, now the next subtree, by assumption, has infinitely many nodes in it with a finite number of children, and each of those children is the top of a subtree and one of its subtrees must be infinite, and we can keep going on forever. The end result is an infinite path.

So how does König's Lemma apply to the tiling problem? Imagine that the tree is the tree of possible choices to make in the process of tiling the plane. Assuming you're only ever placing a tile adjacent to existing tiles, then at each step you have a finite number of possible choices. Further, we've assumed that you can tile any finite region of the plane, which means the tree contains arbitrarily long finite paths. Therefore, König's Lemma tells us that the tree has to have an infinite path, which means that we can tile the infinite plane. Therefore, the tiling problem is reducible to the halting problem.

3.4 Turing degrees

A *Turing degree* is a maximal set of all problems that are reducible to each other. For example, we have so far seen two examples of Turing degrees: computable problems and problems equivalent to the halting problem.

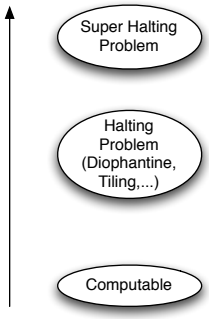


Is there any degree above the halting problem? In other words, if we were given an oracle for the halting problem, is there any problem that would still be unsolvable?

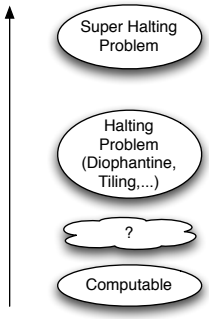
How about if we took a Turing machine with an oracle for the halting problem and asked, "Do you halt?" This can be called the "Super Halting Problem"! To prove this, we can repeat Turing's original halting problem proof, but just one level higher, where all the machines in the proof have this oracle for the halting problem. As long as all of the machines in the proof have the same oracle, then nothing changes and this problem is still the same as the original halting problem. We can follow each machine step by step and ask the oracle, but the oracle won't have the answer for the Super Halting Problem.

If there were the Super Turing Machine that could solve the Super Halting Problem, then you could feed that Super Turing Machine itself as input and cause it to do the opposite of whatever it does, just like an ordinary Turing Machine.

From this, it is clear that we could just keep going up and up with harder and harder problems, the Super Duper Halting Problem, the Super Duper Duper Halting Problem...



Is there any problem that is in an "intermediate" state between computable and the halting problem? In other words, is there a problem that is 1) not computable, 2) reducible to the halting problem, and 3) not equivalent to the halting problem?



This was an open problem called "Post's Problem". In 1956, it was solved by using a technique called the "priority method", which considers all possible Turing machines that might solve a problem A, and all possible Turing machines that might reduce the halting problem to A, and then constructs A in a complicated way that makes all of these machines fail. The problem A that you end up with is not something ever likely to occur in practice! But it does exist.

4 Gödel's Incompleteness Theorem

Gödel's Theorem is a contender (along with quantum mechanics) for the scientific topic about which *the most crap has been written*. Remember systems of logic, containing axioms and rules of inference. You might hope to have a single system of logic that would encompass all of mathematics. In 1930, five years before Turing invented Turing machines, Gödel showed that this was impossible. Gödel's theorems were a direct inspiration to Turing.

Gödel's Incompleteness Theorem says two things about the limits of any system of logic.

First Incompleteness Theorem: Given any system of logic that is consistent (can't prove a contradiction) and computable (the application of the rules is just mechanical), there are going to be true statements about integers that can't be proved or disproved within that system. It doesn't mean these statements are unprovable, but if you want to prove them, you need a more powerful system, and then there will be statements within *that* system that can't be proved, and so on.

Second Incompleteness Theorem: No consistent, computable system of logic can prove its own consistency. It can only prove its own consistency if it is inconsistent. Kind of like how people who brag all the time tend to have nothing to brag about.

(Technical note: Gödel's original proof only worked for a subset of consistent and computable systems of logic, including those that are *sound* and computable. Here "sound" means "unable to prove a falsehood." Soundness is a stronger requirement than consistency. On the other hand, it's also what we usually care about in practice. A later improvement by Rosser extended Gödel's Theorem to all consistent and computable systems, including those that are not sound.)

How did Gödel prove these theorems? He started out with the paradox of the liar: "This sentence is not true." It can't be either true or false! So if we're trying to find an unprovable statement, this seems like a promising place to start. The trouble is that, if we try to express this sentence in purely mathematical language, we run into severe problems. In particular, how do we define the word "true" mathematically?

Gödel's solution was to replace "This sentence is not true" with a subtly different sentence: "This sentence is not *provable*." If the sentence is false, it means that the sentence is provable, and is therefore a provable falsehood! That can't happen if we're working within a sound system of logic. So the sentence has to be true, but that means that it isn't provable.

Gödel showed that as long as the system of logic is powerful enough, you *can* define provability within the system. For, in contrast to truth, whether or not a sentence is provable is a purely "mechanical" question. You just have to ask: starting from the axioms, can you derive this sentence by applying certain fixed rules, or not? Unfortunately for Gödel, the idea of the computer hadn't been invented yet, so he had to carry out his proof using a complicated number theory construction. Each sentence is represented by a positive integer, and provability is just a function of integers. Furthermore, by a trick similar to what we used to show the halting problem was unsolvable, we can define sentences that talk about their *own* provability. The end result is that "This sentence is not provable" gets "compiled" into a sentence purely about integers.

What about the Second Incompleteness Theorem? Given any reasonable logical system S, let

$$G(S) = \text{"This sentence is not provable in } S\text{"}$$

$$Con(S) = \text{"}S \text{ is consistent"}$$

where again, "consistent" means that you cannot prove both a statement and the negation of the statement. Consistency is also a purely mechanical notion because you could just keep turning a crank, listing more and more consequences of the axioms, until you found a statement and its negation both proved. If you never succeed, then your system is consistent. Gödel shows that no sound logical system can prove its own consistency.

The key claim is that $Con(S) \Rightarrow G(S)$. In other words, if S could prove its own consistency, then it actually *could* prove the "unprovable" Gödel sentence, "This sentence is not provable".

Why? Well, suppose $G(S)$ were false. Then $G(S)$ would be provable. But then S would be inconsistent because it would prove a falsehood! So taking the contrapositive, if S is consistent then $G(S)$ must be true. Furthermore, all of this reasoning can easily be formalized within S itself.

So if $Con(S)$ were provable in S, then $G(S)$ would also be provable, and therefore S would prove a falsehood! Assuming S is sound, the only possible conclusion is that $Con(S)$ is *not* provable in S. The system will never be able to prove its own consistency.

6.080/6.089 GITCS	Feb 26, 2008
Lecture 6	
Lecturer: Scott Aaronson	*Scribe: Tiffany Wang*

1 Administrivia

1.1 Scribe Notes

If you are doing the scribe notes for a lecture, remind Professor Aaronson to send you his own lecture notes. Lecture 3 has been posted, and Lecture 4 should follow shortly.

1.2 Problem Sets/Exams

Pset1 is due this coming Thursday. Submit assignments on class Stellar site or send email to Yinmeng. Typed submissions are preferable. Pset2 will be handed out on Thursday.

Midterm exam will an in-class exam on Thursday, April 3.

2 Agenda

Today will be fun! Different structure than previous classes: an open philosophical discussion that will be a good lead into complexity theory. The hope is to motivate more students to get involved in class discussions and to introduce some interesting topics that you should be exposed to at least once in your life.

3 Recap

3.1 Oracles and Reducibility

Oracles are hypothetical devices that solve a given problem without any computational cost. Assuming the existence of such oracles, we establish a hierarchy of insolvability in which problems may be reducible to one another.

So given two problems A and B, A is reducible to B if there exists a Turing machine M such that M^B solves A, or A $\leq_T$ B.

3.2 Turing Degrees

Turing degrees are used to classify all possible problems into groups that are computably equivalent. If given an oracle for one problem in a group, you would be able to solve all other problems of the same Turing degree.

Some examples include the set of all computable problems or the set of problems equivalent to the halting problem, which is incomputable. We also identified that there are problems which are harder than the halting problem: problems that would still be unsolvable even if given an oracle

for the halting problem. There also exist problems of intermediate degrees, which reside between the degrees of computable and the halting problem.

3.3 Gödel's Incompleteness Theorems

Gödel's theorems are among the top intellectual achievements of last century.

3.3.1 First Incompleteness Theorem

Gödel's First Incompleteness Theorem states: For any fixed formal system of logic F, if the system is sound and computable, then there exist true statements about the integers that are not provable within the system, F. In order to prove those statements you would need a more powerful system, which in turn would also have statements that are not provable and require an even more powerful system, and so on.

Gödel's proof involved a mathematical encoding of the sentence:

$G(F) =$ "This sentence is not provable in F."

If $G(F)$ is false, then it is provable, which means F is inconsistent. If $G(F)$ is true, then it is not provable, which means F is incomplete.

3.3.2 Second Incompleteness Theorem

Gödel's Second Incompleteness Theorem states: among the true statements that are not provable within a consistent and computable system F, is the statement of F's own consistency. F can only prove its own consistency if it is inconsistent.

One possible workaround would be to add an axiom to the system which states that F is consistent. However, you would have a new system, F+Con(F), that would not be able to prove Con(F+Con(F)), and so on. You would then establish a hierarchy of more and more powerful theories, each one able to prove consistency of weaker theories but not of itself.

Another proof for Gödel's Incompleteness Theorem is based on the unsolvability of the halting problem. We already established that no Turing Machine exists can solve the halting problem. If we had a proof system that could analyze any Turing machine and prove if it halted or did not halt, we could use this system to solve the halting problem by brute force (also known as the "British Museum algorithm") by trying every possible string that might be a proof. You would either terminate and find a proof that it halts or terminate and find a proof that it doesn't halt. If this proof system was sound and complete, it would violate the unsolvability of the halting problem. Therefore, there is no such sound and complete proof system.

4 Completeness vs. Incompleteness

How can we reconcile Gödel's Incompleteness Theorem with his earlier *Completeness* Theorem? Recall that the completeness theorem states: starting from a set of axioms, you can prove anything logically entailed by those axioms by applying the inference rules of first-order logic.

But doesn't this contradict the Incompleteness Theorems? Look, the same guy proved both of them, so there *must* be a resolution! As it turns out, the two theorems are talking about very subtly different things.

The key is to distinguish three different concepts:

1. *True* (assuming the universe we are talking about is the integers)

   The statement is true in the realm of positive integers.

2. *Entailed by the axioms* (true in any universe where the axioms are true)

   This is a *semantic* notion, or based on the meaning of the statements in question. Simply, the statement is true in any situation where the axioms are true.

3. *Provable from the axioms* (provably by applying rules of inference)

   This is a completely mechanical (or *syntactic*) notion, which just means that the statement is derivable by starting from the axioms and then turning a crank to derive consequences of them.

The Completeness Theorem equates provability with entailment. The theorem says that if something is logically entailed by the set of axioms, then it is also provable from the axioms.

The Incompleteness Theorem differentiates entailment from truth over the positive integers. The theorem implies that there is no set of axioms that captures all and only the true statements about the integers. Any set of axioms that *tries* to do so will also describe other universes, and if a statement is true for the integers but not for the other universes then it won't be provable.

4.1 Implications

The Incompleteness Theorem was a blow to Hilbert and other mathematicians who dreamed of formalizing all of mathematics. It refuted the belief that every well-posed mathematical question necessarily has a mathematical answer.

However, the question arises of whether incompleteness ever rears its head for any "real" problem. In order to prove his theorems, Gödel had to practically invent the modern notion of a computer, but from a purely mathematical standpoint, his sentences are extremely contrived. So you might wonder what the big deal is! Furthermore, we (standing outside the system) "know" that the Gödel sentence $G(F)$ is true, so who cares if it can't be proved within F itself? (This is a point we'll come back to later in this lecture.)

It took until the 1960's for people to prove that there actually exist mathematical questions that mathematicians wanted answers to, but that can't be answered within the current framework of mathematics.

5 Continuum Hypothesis

Recall that Georg Cantor showed there are different kinds of infinity, and specifically that the infinity of real numbers is larger than the infinity of integers.

A related question that Cantor obsessed over (to the point of insanity) until the end of his life was, "Is there any infinity that is intermediate in size between the infinity of real numbers and the infinity of integers?" Cantor formulated the *Continuum Hypothesis* (CH) stating: There is no set whose size is strictly between that of the integers and that of the real numbers.

5.1 Gödel and Cohen's results

In 1939, Gödel showed that the Continuum Hypothesis can be assumed consistently. In other words, the Continuum Hypothesis can be assumed to be true without introducing any inconsistency into set theory.

What is set theory? There are different ways of formalizing a set theory, or choosing the right set of axioms to describe a set. The standard form of axiomatic set theory, and the most common foundation of mathematics, is Zermelo-Fraenkel (ZF) set theory.

By Gödel's incompleteness theorems, ZF cannot prove its own consistency, so how could it possibly prove the consistency of itself *plus* the Continuum Hypothesis? Well, it can't! What Gödel proved was a *relative* consistency statement. Namely, if we assume ZF to be consistent, then adding the Contiuum Hypothesis won't make it inconsistent. Conversely, if the Continuum Hypothesis leads to an inconsistency, this can be converted to an inconsistency in ZF itself. Or in logic notation:

$\text{Con(ZF)} \Rightarrow \text{Con(ZF+CH)}$

Then, in 1963, Paul Cohen showed that you can also assume the Continuum Hypothesis is *false* without introducing an inconsistency. In fact, you can insert as many intermediate infinities as you want.

$\text{Con(ZF)} \Rightarrow \text{Con(ZF+}\neg\text{(CH))}$

5.2 Implications

In George Orwell's novel 1984, the protagonist, Winston Smith, is tortured until he has no will to live. The breaking point is when Smith's assailant is able to get him to admit that $2 + 2 = 5$. Orwell is in a way asserting that the certainty of math is a foundation for our beliefs about everything else. So, should we be concerned about the independence of the Continuum Hypothesis? Does it imply that the answers to seemingly reasonable math problems can depend on how we feel about them?

One response is that we ought to step back and ask, when talking about arbitrary subsets of real numbers, whether we really understand what we mean. In some people's view (and Prof. Aaronson would count himself among them), the one aspect of math that we *really* have a direct intuition about is *computation*. So perhaps the only mathematical questions for which we should definite answers are the ones that we can ultimately phrase in terms of Turing machines and whether they halt. There may be other more abstract problems that do have answers (such as the existence of different levels of infinity), but perhaps we should just consider these as added bonuses.

6 Thinking Machines

The dream of building a "thinking machine" motivated the creation of formal logic and computer science. On the other hand, to this day there's a huge philosophical debate surrounding what a thinking machine would be and how it would be recognized. A surprisingly large portion of this debate was summarized and even anticipated in a *single paper* written by Alan Turing in 1950, "Computing Machinery and Intelligence".

6.1 Turing Test

Turing proposed a criterion called the *Turing Test* to distinguish between humans and machines. If a human interacting with a machine cannot reliably distinguish it from a human, then the machine

ought to be regarded as intelligent, just as the human would be.
Response: But it's just mechanical contrivance! *Clearly* it's not really conscious like I am; it doesn't really have feelings.
Response to the response: Set aside yourself and think about other people. How can you be certain that *other people* are conscious and have feelings?

You infer the consciousness of a person based on interactions. Likewise, if a computer program interacted with you in a way that was indistinguishable from how a person would, you should be willing to make the same inference.

Perhaps Turing himself said it best:

> [Solipsism] may be the most logical view to hold but it makes communication of ideas difficult. A is liable to believe "A thinks but B does not" whilst B believes "B thinks but A does not." Instead of arguing continually over this point it is usual to have the polite convention that everyone thinks.

Question from the floor: Can humans fail the Turing Test?

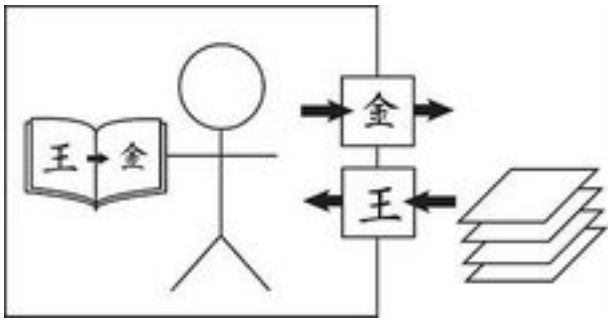
Great question! The Loebner Prize is an annual competition that awards prizes to the most humanlike Chatterbox. On the subject of Shakespeare, a librarian was repeatedly judged to be a machine because people did not believe a human could possibly know so much about Shakespeare.

Many people have argued that passing the Turing Test is a *sufficient but not necessary* condition for intelligence.

6.2 Searle's Chinese Room

Searle's Chinese Room is a thought experiment designed by John Searle (1980) in response to the Turing Test. Searle wanted to dramatize the point that carrying out computations and manipulating symbols does not constitute real awareness or intelligence.

Searle: Suppose you sealed me in a room and fed me slips of paper with Chinese characters written on them, and suppose I had a giant rulebook for producing other slips of Chinese characters, constituting a fluent response to you. By exchanging these slips of paper, I could simulate a Chinese conversation without actually knowing Chinese. Therefore simple symbol manipulation does not constitute understanding.[1]



What's a possible response to this argument?
System Response: The problem with Searle's argument is the need to distinguish between Searle and the system consisting of him and the rulebook. Searle may not understand Chinese, but the system as a whole does understand.

[1] It's a strange experience to explain Searle's thought experiment to students many of whom *would* understand what was on the slips! –SA

Searle: That's ridiculous! (In his writings, Searle constantly appeals to what he regards as common sense.) Just memorize the rulebook, thereby removing the system.
Response: You would then have to distinguish between Searle and the person being simulated by his memory.

Another response is Searle gets a lot of the mileage in his thought experiment from careful choice of imagery: "mere slips of paper"! However, the human brain has immense computational capacity (roughly 10^{11} neurons and 10^{14} synapses, with each neuron itself much more complex than a logic gate), and performs its computations massively in parallel. Simulating the computational power of the brain could easily require enough slips of paper to fill the solar system. But in that case Searle's scenario seems to lose its intuitive force.

6.3 Competing Analogies

The debate surrounding the feasibility of truly thinking machines comes down to competing analogies.

The central argument against the possibility of intelligent machines has always been that "A computer simulation of a hurricane doesn't make anyone wet." But on the other hand, a computer simulation of multiplication clearly *is* multiplication.

So the question boils down to: is intelligence more like a hurricane or like multiplication?

6.4 The "Practical" Question

Setting aside whether or not a machine that passes the Turing Test should be considered conscious, there's also the more "practical" question: can there ever *be* a machine that passes the Turing Test? Or is there some fundamental technological limitation?

In 1950, Turing predicted that by the year 2000, a machine would be able to fool the average person 70% of the time into thinking it was human after a 5-minute conversation. The accuracy of his prediction depends on the sophistication of the "average person" judging the machine.

Already in the 1960s, computer chat programs were easily able to fool unsophisticated people. Chat programs like ELIZA (or more recently AOLiza) are based on parroting the user like a "psychotherapist": "Tell me more about your father." "I would like to go back to the subject of your father." People would pour their hearts out to these programs and refuse to believe they were talking to a machine.

Sophisticated people who know the right thing to look for would easily uncover the identity of the program by asking any commonsense question: "Is Mount Everest bigger than a shoebox?" The program would answer with something like "Tell me more about Mount Everest," and continue parroting the person instead of giving a straight answer.

A current practical issue involves CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart). A CAPTCHA is a test that current computers can generate and grade but cannot pass. These tests were invented to block spambots but actually raise a profound philosophical issue: that of distinguishing between humans from machines.

There is an arms race between the spammers and the CAPTCHA-makers. Some of the commonly used CAPTCHA's have been broken, but in general the CAPTCHA-makers are still winning.

7 Gödel and Thinking Machines

The idea that Gödel's Incompleteness Theorem somehow proves the impossibility of thinking machines is an old one (indeed, Gödel himself might have believed something close to this). Today, though, the idea is most closely associated with Roger Penrose, the famous mathematical physicist who, among other achievements, invented Penrose tiles and (along with Stephen Hawking) showed that General Relativity generically predicts black holes.

7.1 The Emperor's New Mind

In 1989, Penrose wrote a book, *The Emperor's New Mind*, in which he tried to use Gödel's Incompleteness Theorem to argue that computers would never be able to simulate human beings. Consider again the Gödel sentence

G(F) = "This sentence is not provable in F."

Penrose argues that any computer working within the logical system F cannot prove G(F), but we as humans can just "see" that it is true by engaging in meta-reasoning. Therefore humans can do something that computers can't.

Question from the floor: Can you change the statement to "This sentence cannot be proved by Roger Penrose?"?

Great question! One possible response is that the modified sentence can't be compiled into a purely logical form, since we don't yet completely understand how the human brain works. This is basically an argument from ignorance.

Another response: Why does the computer have to work within a fixed formal system F?

Penrose: Because otherwise, the computer would not necessarily be sound and could make mistakes.
Response: But humans make mistakes too!
Penrose: When I perceive G(F) is true, I'm absolutely certain of it.
But how certain are we that G(F) is true? Recall that Con(F) (the consistency of F) implies G(F).

Claim: The inverse is true as well. That is, G(F) implies Con(F).

Proof: If F is inconsistent, then it can prove anything, including G(F). Hence $\neg$ Con(F) implies $\neg$ G(F) (i.e., that G(F) is provable), which is the contrapositive of what we wanted to show.

So the bottom line is that G(F) is simply equivalent to the consistency of F. The question, then, is whether or not human beings can step back and "directly perceive" the consistency of a system of logic, which seems like more of a religious question than a scientific one. In other words, one person might be absolutely certain of a system's consistency, but how could that person ever convince someone else by verbal arguments?

7.2 Views of Consciousness

Penrose devised a classification of views about consciousness:

1. Simulation produces consciousness. (Turing)
2. Consciousness can be simulated, but mere simulation does not produce consciousness. (Searle)
3. Consciousness cannot even be simulated by computer, but has a scientific explanation. (Penrose)
4. No scientific explanation. (99% of people)

Scribe Notes: Introduction to Computational Complexity

Jason Furtado

February 28, 2008

1 Motivating Complexity Theory

Penrose's Argument (continued)

Last lecture, Roger Penrose's argument was introduced. It says that it was impossible for a computer to simulate the human brain. His argument is based on Godel's Incompleteness Theorem. Penrose's argument says that, within any formal system F, we can construct a Godel sentence $G(F)$ that the computer cannot prove but a human can prove it.

Professor Aaronson proposes that there is a simple way to see that there must be a problem with Penrose's Argument or any similar argument. Penrose is attempting to show that no computer could pass the Turing Test. The Turing Test states that a computer is intelligent if it could carry on a textual conversation with a human and the human would be unable to tell that he or she was having a conversation with a computer rather than another human. A Turing Test lasts a finite amount of time (it is a conversation). If you could type incredibly quickly, say 5000 characters per minute, the number of possible instant messaging conversation you could have is finite. Therefore, in principle, a giant lookup table could be constructed that stored an intelligent human response to every possible question that could be asked. Then the computer could just consult the lookup table whenever a question is asked of it.

The problem with this lookup table is that it would have to be incredibly large. The amount of storage necessary would be many times larger than the size of the observable universe (which has maybe 10^{80} atoms). Therefore, such a system would be infeasible to create. The argument has now changed from theoretical possibility to a question of feasible given resources. The argument against such a system has to do with the amount of memory, processing power, etc. needed to create a system that could reliably pass the Turing Test. Since the system is theoretically possible, the question is whether or not the amount of resources necessary is a reasonable amount.

In order to have such a conversation, we will need a way to measure the efficiency of algorithms. The field that studies such issues is called *computational complexity theory*, and will occupy us for most of the rest of the course. In contrast to what we saw before, computational complexity is a field where almost all of the basic questions are still unanswered – but also where some of what we *do* know is at the forefront of mathematics.

Dijkstra Algorithm

In 1960, the famous computer scientist Edsgar Dijkstra discovered an algorithm that finds the shortest path between two vertices of a graph, in an amount of time linear in the number of edges. The algorithm was named after him (Dijkstra's Algorithm), and something like it is used (for example) in Google Maps and every other piece of software to find directions. There's a famous anecdote where Dijkstra (who worked in a math department) was trying to explain his new result

to a colleague. And the colleague said, "but I don't understand. Given any graph, there's at most a finite number of non-intersecting paths. Every finite set has a minimal element, so just enumerate them all and you're done." The colleague's argument does, indeed, show that the shortest path problem is computable. But from a modern perspective, showing a problem is computable does not say much. The important question is whether a problem is solvable *efficiently*.

Efficiency

Computer scientists are interested in finding efficient algorithms to solve problems. Efficiency is typically measured by considering an algorithm's running time as a function of the size of the input. Here *input size* usually (not always) refers to the number of bits necessary to describe the input. This way of describing efficiency avoids being too tied to a particular machine model (Macs, PC's, Turing machines, etc.), and lets us focus on more "intrinsic" properties of the problem and of algorithms for solving it.

2 Circuit Complexity

A good place to begin studying complexity questions is *circuit complexity*. Historically, this is also one of the first places where complexity questions were asked.

Let's pose the following general question:

Given a Boolean function $f : \{0,1\}^n \to \{0,1\}$, what is the size of the smallest circuit that computes it? (That is, how many gates are needed?)

As an example, suppose we're trying to compute the XOR of the n input bits, $x_1, \ldots, x_n$. Then how many gates to do we need? (For simplicity, let's suppose that a two-input XOR gate is available.) Right, $n-1$ gates are certainly sufficient: XOR x_1 and x_2, then XOR the result with x_3, then XOR the result with x_4, etc. Can we show that $n-1$ gates are necessary?

Claim: You need at least $n-1$ gates in order for the single output bit to depend on all n input bits using 2-input gates.

Proof: Using the "Chocolate-breaking" argument as a basis, we can view each input as a group. Each XOR function joins two groups, so you then have one less total group than before. If you continue to join groups until there is only a single group, you would have used $n-1$ XOR functions to join the n inputs.

Note: In lecture, it was brought up that another measure of efficiency in circuits is the *depth* of the circuit. This is the maximum number of gates needed to traverse the circuit from an input to the output. For a XOR circuit of n inputs constructed using two-input XOR gates, one can easily show that a depth of $\log n$ is necessary and sufficient.

Shannon's Counting Argument

Is there a Boolean Function with n inputs that requires a circuit of exponential size in n?

Well, let's look at some possible candidates. Does the Majority function require an exponential-size circuit? No, it doesn't. What about SAT and other NP-complete problems? Well, that's getting ahead of ourselves, but the short answer is that no one knows!

But in 1949, Claude Shannon, the father of information theory, mathematical cryptography and several other fields (and who we'll meet again later), gave a remarkably simple argument for why such a Boolean function must exist.

First, how many Boolean functions are there on n inputs? Well, you can think of a Boolean function with n inputs as a truth table with 2^n entries, and each entry can be either 0 or 1. This leads to 2^{2^n} possible Boolean functions: not merely an exponential number, but a *double*-exponential one.

On the other hand, how many circuits are there with n inputs and T gates? Assume for simplicity that there are only NAND gates in our circuit. Then each gate will have at most $n+T$ choices for the left input and most $n+T$ choices for the right input. Therefore, there can be no more than $(n+T)^{2T}$ circuits possible. Every circuit can compute only one Boolean function. And therefore, if we want to represent all 2^{2^n} Boolean functions, then we need $(n+T)^{2T} \geq 2^{2^n}$. Taking the log of both sides we can estimate that T is $\frac{2^n}{2n}$. If T were any smaller than this, there would not be enough circuits to account for all Boolean functions.

Shannon's argument is simple yet extremely powerful. If we set $n = 1000$, we can say that almost all Boolean functions with 1000 inputs will need to use at least $2^{1000}/2000$ NAND gates in order to compute. That number is larger than 10^{80}, the estimated number of atoms in the visible universe. Therefore, if every atom in the universe could be used as a gate, there are functions with 1000 inputs that would need a circuit much bigger than the universe to compute.

The amazing thing about Shannon's counting argument is that it proves that such complex functions must exist, yet without giving us a *single specific example* of such a function. Arguments of this kind are called *non-constructive*.

3 Hartmanis-Stearns

So the question remains: can we find any *concrete* problem, one people actually care about, that we can prove has a high computational complexity? For this question, let's switch from the circuit model back to the Turing machine model.

In 1965, Juris Hartmanis and Richard Stearns showed how to construct problems that can take pretty much any desired number of steps for a Turing machine solve. Their result basically started the field of computational complexity, and earned them the 1993 Turing Award. To prove their result (the so-called Hierarchy Theorem), they used a "scaled-down" version of the argument that Turing had originally used to prove the unsolvability of the halting problem.

Let's say we want to exhibit a problem that any Turing machine needs about n^3 steps to solve. The problem will be the following:

Question: Given as input a Turing Machine M, input x and integer n does M halt after at most n^3 steps on input x?

Claim: Any Turing machine to solve this problem needs at least n^3 steps.

Proof: Suppose there were a machine, call it P, that solved the above problem in, say, $n^{2.99}$ steps. Then we could modify P to produce a new machine P', which acts as follows given a Turing machine M and input n:

1. Runs forever if M halts in at most n^3 steps given its own code as input.
2. Halts if M runs for more than n^3 steps given its own code as input.

Notice that P' halts in at most $n^{2.99}$ steps (plus some small overhead) if it halts at all – in any case, in fewer than n^3 steps.

Now run $P'(P', n)$. There is a contradiction! If P' halts, then it will run forever by case 1. If P' runs forever then it will halt by case 2.

The conclusion is that P could not have existed in the first place. In other words, not only is the halting problem unsolvable, but in general, there is no faster way to predict a Turing machine's behavior given a finite number of steps than to run the machine and observe it. This argument works for any "reasonable" runtime (basically, any runtime that's itself computable in a reasonable amount of time), not just n^3.

4 Notation for Complexity

In later lectures, it will be helpful to have some notation for measuring the growth of functions, a notation that ignores the "low-order terms" that vary from one implementation to another. What follows is the standard notation that computer scientists use for this purpose.

Big-O Notation

We say $f(n) = O(g(n))$ if there exist constants a, b such that $f(n) < ag(n) + b$ for all n.

The function $g(n)$ is an upper bound on the growth of $f(n)$. A different way to think of Big-O is that $g(n)$ "grows" at least as quickly as $f(n)$ as n goes to infinity. $f(n) = O(g(n))$ is read as "$f(n)$ belongs to the class of functions that are $O(g(n))$".

Big-Omega Notation

We say $f(n) = \Omega(g(n))$ if there exist $a > 0$ and b such that $f(n) > ag(n) - b$ for all n.

Intuitively, the function $f(n)$ grows at the same rate or more quickly than $g(n)$ as n goes to infinity.

Big-Theta

We say $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ AND $f(n) = \Omega(g(n))$.

Intuitively, the function $f(n)$ grows at the same rate as $g(n)$ as n goes to infinity.

Little-o

We say $f(n) = o(g(n))$ if for all $a > 0$, there exists a b such that $f(n) < ag(n) + b$ for all n.

Intuitively, $f(n)$ grows strictly more slowly than $g(n)$. So $f(n) = O(g(n))$ but not $\Theta(g(n))$.

6.080/6.089 GITCS	March 4, 2008
Lecture 8	
Lecturer: Scott Aaronson	*Scribe: Hristo Paskov*

1 Administrivia

There will be two scribes notes per person for the term. In addition, you may now use the book to review anything not clear from lecture since we're on complexity.

2 Recap

In most cases of interest to us, the real question is not what's computable; it's what's computable with a reasonable amount of time and other resources. Almost every problem in science and industry is computable, but not all are *efficiently* computable. And even when we come to more speculative matters – like whether it's possible to build a machine that passes the Turing Test – we seethat with unlimited resources, it's possible almost trivially (just create a giant lookup table). The real question is, can we build a thinking machine that operates within the time and space constraints of the physical universe? This is part of our motivation for switching focus from computability to complexity.

2.1 Early Results

2.1.1 Claude Shannon's counting argument

Most Boolean functions require enormous circuits to compute them, i.e. the number of gates is exponential in the number of inputs. The bizarre thing is we know this, yet we still don't have a good example of a function that has this property.

2.1.2 Time Hierarchy Theorem

Is it possible that everything that is computable at all is computable in linear time? No. There is so much that we don't know about the limits of feasible computation, so we must savor what we do know. There are more problems that we can solve in n^3 than in n^2 steps. Similarly, there are more problems that we can solve in 3^n than in 2^n steps. The reason this is true is that we can consider a problem like:
"Given a Turing Machine, does it halt in $\leq n^3$ steps?"
Supposing it were solvable in fewer than n^3 steps, we could take the program that would solve the problem and feed it itself as input to create a contradiction. This is the finite analog of the halting problem.

2.1.3 Existence of a fastest algorithm

One thing we have not mentioned is a bizarre phenomenon in runtime people discovered in the 60's:

Does there have to be a fastest algorithm for every problem? Or, on the contrary, could there be an infinite sequence of algorithms to solve a problem, each faster, but no fastest one?

2.1.4 Concrete example: Matrix Multiplication

Given two $n \times n$ matrices, find:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \dots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \dots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & \dots & \vdots \\ c_{n1} & c_{n2} & \dots & c_{nn} \end{pmatrix}$$

2.1.5 Straightforward way

The straightforward way takes n^3 steps because of the way we multiply columns and rows. However, there do exist better algorithms.

2.1.6 Strassen's algorithm

In 1968, Strassen found an algorithm that takes only $O(n^{2.78})$ steps. His algorithm used a divide and conquer approach, repeatedly dividing the original matrix into $\frac{n}{2} \times \frac{n}{2}$ matricies and combining in clever way. It's a somewhat practical algorithm; people actually use it in scientific computing and other areas. Strassen's algorithm served as one of the early inspirations for the theory of efficient algorithms: after all, if people had missed something so basic for more than a century, then what else might be lurking in algorithm-land?

2.1.7 Faster and faster algorithms

Sometime later an algorithm that takes $O(n^{2.55})$ time was found. The best currently known algorithm is due to Coppersmith and Winograd, and takes $O(n^{2.376})$ algorithm. Many people believe that we should be able to do better. The natural limit is n^2 since, in the best case, we have to look at all entries of the matrices. Some people conjecture that for all $\epsilon > 0$, there exists an algorithm that takes $O(n^{2+\epsilon})$ time.

2.1.8 Practical considerations

If matrices are reasonably small, then you're better off multiplying them with the naïve $O(n^3)$ algorithm. It's an empirical fact that, as you go to asymptotically faster algorithms, the constant prefactor seems to get larger and larger. You *would* want to switch to a more sophisticated algorithm with larger matrices, since there the constant prefactor constant doesn't matter as much. For all we know, it could be that as you go to bigger and bigger matrices, there's an unending sequence of algorithms that come into play. Or the sequence could terminate; we don't know yet.

Note that we can't obviously just make an algorithm that chooses the best algorithm for a given matrix size, since each new faster algorithm might require some extra insight to come up with. If there *were* an algorithm to produce these algorithms, then we'd have a single algorithm after all.

This illustrates the notion of an infinite progression of algorithms with no best one. Note that this sequence is countable since there's only a countable number of algorithms.

2.1.9 Blum Speedup Theorem

In 1967, Blum showed that we can construct problems for which there's *provably* no best algorithm. Indeed, there exists a problem such that for every $O(f(n))$ algorithm, there's also an $O(log(f(n)))$

algorithm! How could this be? The problem would take a lot of time to solve by any algorithm—some giant stack of exponentials such that taking *log* any number of times won't get it down to a constant. We're not going to prove this theorem, but be aware that it exists.

2.1.10 Summary

So for every problem there won't necessarily be a best algorithm. We don't know how often this weird phenomenon arises, but we do know that it can in principle occur.

Incidentally, there's a lesson to be learned from the story of matrix multiplication. It was intuitively obvious to people that n^3 was the best we could do, and then we came up with algorithms that beat this bound. This illustrates why it's so hard to prove *lower* bounds: because you have to rule out every possible way of being clever, and there are many non-obvious ways.

3 The meaning of efficient

We've been talking about "efficient" algorithms, but what exactly do we mean by that?

One definition computer scientists find useful (though not the only one) is that "efficient" = "polynomial time." That is, an algorithm is efficient if there exists a k such that all instances of size n are solved in $O(n^k)$ time. Things like $\sqrt{n}$ or $logn$ are not polynomial, but there is a polynomial larger than them, hence they're also efficient.

No sooner do you postulate this criterion for efficiency than people think of objections to it. For example, an $n^{10,000}$-time algorithm is polynomial-time, but obviously not efficient in practice. On the other hand, what about $O(1.00000000001^n)$ which is exponential, but appears to be fine in practice for reasonably-sized n?

Theoretical computer scientists are well aware of these issues, and have two main responses:

1. Empirical - in practice efficient usually does translate into polynomial time; inefficient usually translates into exponential time and worse with surprising regularity. This provides the empirical backbone of theory. Who invents $O(n^{10,000})$ algorithms?

2. Subtle response - any criterion for efficiency has to meet the needs of practitioners *and* theorists. It has to be convenient. Imagine a subroutine that takes polynomial time and an algorithm that makes polynomial calls to the subroutine. Then the runtime is still polynomial since polynomials are closed under composition. But suppose "efficient" were instead defined as quadratic time. When composing such an algorithm with another $O(n^2)$ algorithm, the new algorithm won't necessarily be efficient; it could take $O(n^4)$ time. Thus, we need something that's convenient for theory.

What are the actual limits of what we can do in polynomial time? Does it encompass what you would use computers for on a daily basis (arithmetic, spell checking, etc.)? Fortunately these are all polynomial time. For these tasks, it requires some thought to find the best polynomial time algorithm, but it doesn't require much thought to find *a* polynomial time algorithm. Our definition encompasses non obvious algorithms as well. You may be familiar with some of these if you have taken an algorithms class.

4 Longest Increasing Subsequence

4.1 The Problem

Let $X(1), X(2), ..., X(n)$ be a list of integers, let's say from 1 to $2n$. The task is to find the longest subsequence of numbers, not necessarily contiguous, that is increasing, i.e. $X_{i_1} < X_{i_2} < \ldots < X_{i_k}$ where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$. For example:

$$3, 8, 2, 4, 9, 1, 5, 7, 6$$

has several longest subsequences of length 4: (2,4,5,7), (2,4,5,6), (3,4,5,6), (3,4,5,7).

Solving this is manageable when we have 9 numbers, but what about when $n = 1000$? How do we program a computer to do this?

4.2 A Polynomial Time Algorithm

One could try all possibilities, but this approach is not efficient. Trying all possibilities of size k takes $\binom{n}{k}$ time, and $\sum \binom{n}{i} = 2^n$. We'd have an exponential algorithm.

Consider this approach instead:

Iterate through the elements from left to right and maintain an array of size n. For each element i, save the longest subsequence that has i as its last element. Since computing this subsequence for the k^{th} element involves only looking at the longest subsequences for the previous $k-1$ elements, this approach takes $O(n^2)$ time.

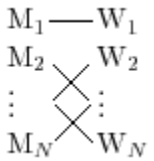
4.3 Dynamic Programming

The above was an example of *dynamic programming*, a general technique that allows us to solve some problems in polynomial time even though they seem to require exponential time. In general, you start with solutions to subproblems and gradually extend to a solution to the whole problem. However, for this technique to work, the problem needs to have some sort of structure that allows us to split it into subproblems.

5 Stable Marriage

5.1 The Problem

Given N men and N women where each man ranks each woman, and vice versa, we want to match people off in a way that will make everyone "not too unhappy."



More specifically, we want to avoid *instabilities*, where a man and woman who are not married both prefer each other over their actual spouses. A matching without any such instabilities is called a *stable marriage*. Our goal is to give an efficient algorithm to find stable marriages, but the first question is: does a stable marriage even always exist?

5.2 Victorian Romance Novel Algorithm

It turns out that the easiest way to show that a solution does exist is to give an algorithm that finds it. This algorithm, which was proposed by Gale and Shapley in the 1950's, has one of the simplest human interpretations among all algorithms.

5.2.1 Algorithm

Start with the men (this was the 50's, after all). The first man proposes to his top ranked woman and she tentatively accepts. The next man proposes, and so on, and each woman tentatively accepts. If there is never conflict, then we're done. If there is a conflict where 2 men propose to the same woman, the woman chooses man she likes better, and the man gets booted, crossing off the woman who rejected him. We keep looping through the men like this, where in the next round, any man who got booted goes to the next woman he prefers. If she hasn't been proposed to, she tentatively accepts. Otherwise, she picks which man she likes best, booting the other one, and so forth. We continue looping like this until all men and women are matched.

5.2.2 Termination

Does this algorithm terminate? Yes: in the worst case, every man would propose once to every woman on his list. (Note that a man never reconsiders a woman who's been crossed off.)

Next question: when the algorithm terminates is everyone matched up? Yes. Suppose for the sake of contradiction there's a couple who aren't matched. This means that there's a single man and a single woman. But in that case, no one else could've proposed to that woman, and hence when the man in question proposed to her (which he would have at some point) she would've accepted him. (Note that the problem statement encodes the assumption that everyone would rather be with *someone* than alone.) This is a contradiction; therefore everyone must be matched.

5.2.3 Correctness

Is the matching found by the algorithm a stable one? Yes. For suppose by contradiction that M_1 is matched to W_1 and M_2 to W_2, even though M_1 and W_2 both prefer each other over their spouses. In that case, M_1 would have proposed to W_2 *before* he proposed to W_1. And therefore, when the algorithm terminates W_2 couldn't possibly be matched with M_2—after all, she was proposed to by at least one man who she prefers over M_2. This is a contradiction; therefore the algorithm outputs a stable marriage.

5.2.4 Runtime

Finally, note that the input consists of $2n$ lists, each with n numbers. The algorithm takes $O(n^2)$ time and is therefore linear-time with respect to the input size.

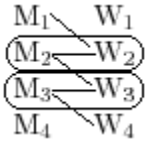
The naïve way to solve this problem would be to loop through all $n!$ possible matchings and output a stable one when found. The above is a much more efficient algorithm, and it *also* provides a proof that a solution exists.

6 Other Examples

After forty years these are some of the problems we know are solvable in polynomial time: Given N men and N woman again, but no preference lists. Instead, we know whether M_i can be matched

to W_j, $1 \leq i, j \leq n$. It is clear that in this case we won't be able to make everyone happy. Take for example the case where no man can be matched with any woman. But even though we won't necessarily be able to make everyone happy, can we make as many people as happy as possible? This is called the "Maximum Matching" problem. More abstractly, given a bipartite graph with n vertices on each side, we want to find a maximal disjoint set of edges. The naïve way looks at how to match 1 person, 2 people, 3 people, etc. but this is slow.

A better solution involves repeatedly matching k people, and then trying to see if we "improve" the match to k+1 people. Examining the graph of men and women, if we find any men and women who are not matched then we tentatively match them off. Of course, following this approach, we might come to a situation where we can no longer match anyone off, and yet there might still exist



a better matching. We can then search for some path that zigzags back and forth between men and women, and that alternates between edges we haven't used and edges we have. If we find such a path, then we simply add all the odd-numbered edges to our matching and remove all the even-numbered edges. In this approach, we keep searching in our graph till no more such paths can be found. This approach leads to an $O(n^3)$ algorithm. As a function of the size of our input, which is n^2, this is an $O(n^{3/2})$ algorithm. Hopcroft and Karp later improved this to an $O(n^{5/4})$ algorithm.

It is worthwhile to note that this problem was solved in Edmonds' original 1965 paper that defined polynomial time as efficient. In that paper, Edmonds actually proved a much harder result: that even in the case where men could also be matched to men, and likewise for women, we can still find a maximal matching in polynomial time. Amusingly (in retrospect), Edmonds had to explain in a footnote why this was an interesting result at all. He pointed out that we need to distinguish between polynomial and exponential-time algorithms since exponential time algorithms don't convey as good an understanding of the problem and aren't efficient.

6.1 Gaussian Elimination

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \dots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

Another example is solving linear system of equations. Instead of trying every possible vector, we can use Gaussian Elimination. This takes $O(n^2)$ time to zero out rows below, and doing this n

times till we get an upper triangular matrix takes a total of $O(n^3)$ time. Essentially, this method is a formalization of the idea of doing substitution.

What about first inverting the matrix A and then multiplying it by the vector y? The problem is that the standard way to invert a matrix is to use Gaussian Elimination! As an aside, it's been proven that whatever the complexity of inverting matrices is, it's equivalent to the complexity of matrix multiplication. A reduction exists between the two problems.

6.2 Linear Programming

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \dots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \leq \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

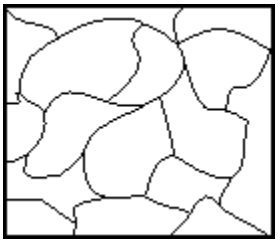
What about solving a system of linear *inequalities*? This is a problem called linear programming, whose basic theory was developed by George Dantzig shortly after World War II. As an operations researcher during the war, Dantzig had faced problems that involved finding the best way to get shipments of various supplies to troops. The army had a large set of constraints on how much was available and where it could go. Once these constraints were encoded as linear inequalities, the problem was to determine whether or not a feasible solution existed. Dantzig invented a general approach for solving such problems called the simplex method. In this method, you start with a vector of candidate solutions and then increment it to a better solution with each step. The algorithm seemed to do extremely well in practice, but people couldn't prove much of anything about its running time. Then, in the 1970's, Klee and Minty discovered contrived examples where the simplex algorithm takes exponential time. On the other hand, in 1979 Khachiyan proved that a different algorithm (though less efficient than the simplex algorithm in practice) solves linear programming in polynomial time.

6.3 Aside

As a student in class pointed out, had it not been for World War II, computer science (like most other sciences) would probably not have made so many advances in such a short period of time. If you're interested in learning more about the history of wartime research (besides the Manhattan Project, which everyone knows about), check out *Alan Turing: The Enigma* by Andrew Hodges or *Endless Frontier* by G. Pascal Zachary. World War II might be described as the first "asymptotic war" (i.e., the first war on such a huge scale that mathematical theories became relevant to winning it).

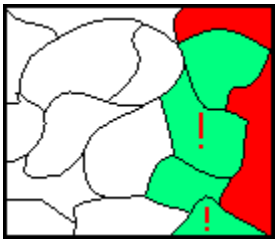
6.4 Primality Testing

Primality testing is the following problem: given a number, is it prime or composite? The goal is to answer in time proportional to the number of digits in the number. This is crucial for generating RSA keys; and fortunately, polynomial-time algorithms for this problem do exist. We'll go into this in more detail later.



6.5 Two-Coloring a Map

We're given a list of countries and told which countries are adjacent to which other ones. Can we color a map of the countries so that no two adjacent countries are colored the same way? A simple algorithm exists: color one country with one of the colors, and then color every adjacent country with the other color. We can keep going this way until either the map is fully colored or else two adjacent countries are forced to be the same color. In the latter case, the map is not two-colorable.



What makes the problem so simple is once we color a country, we are forced in how we can color all the remaining countries. Because of this, we can decide whether the map has a two-coloring in $O(n)$ time.

6.6 Can we solve these in polynomial time?

On the other hand, what if we want to know whether our map has a *three*-coloring? This seems harder since we are no longer always forced to color each country a specific color—in many cases we'll have two choices. It's not obvious how to solve the problem efficiently.

Or consider a problem where we're given a list of people, and also told for any pair of people whether they're friends or not. Can we find a maximal set of people who are all friends with each other?

Stay tuned to next lecture to explore these problems...

6.080/6.089 GITCS	March 6, 2008
Lecture 9	
Lecturer: Scott Aaronson	*Scribe: Ben Howell*

1 Administrivia

1.1 Problem Set 1

Overall, people did really well. If you're not happy with how you did, or there's something you don't understand, come to office hours on Monday. No one got the challenge problem to prove the completeness of resolution, but many people got the problem with the cannonballs. The solution to the cannonball problem involves defining a function on the state of the system that is strictly monotonic. Here are two possible ways to construct that proof:

Solution 1: Let S_i be the number of cannonballs in stack i, and define a function that describes the state of the system: $f = \sum_i (S_i)^2$. Each time a stack is exploded, the function f increases by two. Since explosion is the only defined operation and it always increases the value of f, the system can never return to a previous state.

Solution 2: Let p be the rightmost stack that has been exploded so far. Then stack $p+1$ has more cannonballs than it had at the beginning. The only way for that to go back down is to explode stack $p+1$, which will cause stack $p+2$ to have more cannonballs than it had at the beginning. This leads to an infinite regress, implying that there is no finite sequence of moves that returns to the starting configuration.

2 Review

The central question of theoretical computer science is: "Which problems have an efficient algorithm to solve them?" In computer science, the term "efficient" equates to "polynomial time".

Here are a few examples of problems that can be solved efficiently:

- Arithmetic, sorting, spellcheck, shortest path
- Solving linear systems (and inverting matrices)
- Linear programming
- Dynamic programming
- Stable marriage, maximum matching
- Primality testing
- Finding roots of polynomials
- 2-coloring a graph

And then there are other problems that have no known polynomial-time algorithm, for example:

- 3-Coloring a map
- Max Clique: Given a set of people, find the largest set who are all friends with each other.
- Packing: Given boxes of various dimensions, find the largest set you can pack in your truck.
- Traveling Salesman: Given a set of cities and costs of travel between cities, find the least expensive route that goes through every city.

3 General Problems

The problems listed above seem like puzzles. Let's consider some other problems that seem "deeper" or more "general":

- **The "DUH" Problem:** Given n, k, and a Turing machine M, is there an input y of size n such that $M(y)$ accepts in fewer than n^k steps?
- **The Theorem Problem:** Given a mathematical statement and an integer n, is there a proof in ZF set theory with at most n symbols?

All of these problems have a few things in common:

- They are all computable, but the computation might take an exponential amount of time.
- Given a proposed solution, it is possible to check whether that solution is correct in a polynomial amount of time.

3.1 The Theorem Problem

Let's take a closer look at the Theorem problem. Without the "at most n symbols" constraint, would this problem even be computable?

Claim: If you could solve this problem without that constraint, then you could solve the halting problem.

Proof: Given a Turing machine M, use your algorithm to find the answers to the following questions:

- Is there a proof that M halts? If so, then M halts.
- Is there a proof that M does not halt? If so, then M does not halt.

If neither proof exists, then we can still conclude that M does not halt. If it did halt, then there must be a proof that it does; the most straightforward one being to simulate M until it halts.

Therefore, we can conclude that without the "at most n symbols" constraint, this problem is not computable.

However, this problem computable with the "at most n symbols" constraint. In the worst case, the algorithm could simply search through all possible proofs with at most n symbols.

Is there a better way to solve the problem that doesn't involve a brute-force search? Exactly this question was asked in one of the most remarkable documents in the history of theoretical computer science: a letter that Kurt Gödel sent to John von Neumann in 1956. (The full text of this letter is in the appendix.) In this letter, Gödel concludes that if there were some general way to avoid brute-force search, mathematicians would be out of a job! You could just ask your computer whether there's a proof of Goldbach, the Riemann Hypothesis, etc. with at most a billion symbols.

4 P and NP

After 52 years, we still don't know the answer to Gödel's question. But there's something remarkable that we *do* know. Look again at all of these problems. A priori, one might think brute-force search is avoidable for problem X, not for problem Y, etc. But in the 1970s, people realized that in a very precise sense, *they're all the same problem.* A polynomial-time algorithm for any one of them would imply a polynomial-time algorithm for all the rest. And if you could prove that there was no polynomial-time algorithm for one of them, that would imply that there is no polynomial-time algorithm for all the rest.

That is the upshot of the **theory of NP-completeness**, which was invented by Stephen Cook and Richard Karp in the US around 1971, and independently Leonid Levin in the USSR. (During the Cold War, there were a lot of such independent discoveries.)

4.1 P and NP

P is (informally) the class of all problems that have polynomial-time algorithms. For simplicity, we focus on decision problems (those having a yes-or-no answer).

Formally: **P** is the class of all sets $L \subseteq \{0,1\}^n$ for which there exists a polynomial-time algorithm A such that $A(x) = 1$ iif $x \in L$.

NP (Nondeterministic Polynomial-Time) is (informally) the class of all problems for which a solution can be verified in polynomial time.

Formally: **NP** is the class of all sets $L \subseteq \{0,1\}^n$ for which there exists a polynomial-time Turing machine M, and polynomial p, such that $\forall x \in \{0,1\}^n : x \in L$ iff $\exists y \in \{0,1\}^{p(n)}$ s.t. $M(x,y)$ accepts (here, x is the instance of the problem and y is some solution to the problem).

Why is it called "nondeterministic polynomial-time"? It's a somewhat archaic name, but we're stuck with it now. The original idea was that you would have a polynomial-time Turing machine that can make *nondeterministic transitions* between states, and that accepts if and only if there exists a path in the computation tree that causes it to accept. (Nondeterministic Turing machines are just like the nondeterministic finite automata we saw before.) This definition is equivalent to the formal one above. First have the machine guess the solution y, and then check whether y works. Since it is a *nondeterministic* Turing machine, it can try all of the possible y's at once and find the answer in polynomial time.

Question: is $P \subseteq NP$? Yes! If someone gives you a solution to a P problem, you can ignore it and solve it yourself in NP time.

4.2 NP-hard and NP-complete

NP-hard is the class of problems that are "at least as hard as any NP problem." Formally, L is **NP-hard** if given an oracle for L, you could solve every NP problem in polynomial time. In other words, L is **NP-hard** if $NP \subseteq P^L$.

NP-complete is the class of problems that are both NP-hard and in NP. Informally, NP-complete problems are the "hardest problems in NP" — NP problems that somehow capture the difficulty of all other NP problems.

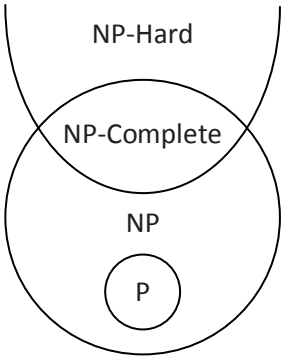


Figure 1: Relationship between P, NP, NP-Hard, and NP-Complete.

Claim: If any NP-complete problem is in P, then all are, and P = NP.
If any NP-complete problem is not in P, then none are, and P $\neq$ NP.

4.3 NP-complete problems

How can we prove that NP-complete problems even exists? Can we find an example of one?

Consider the "DUH" problem from above: "Given a polynomial-time Turing machine M, find an input y that causes $M(y)$ to accept." This problem is NP-complete almost by the definition of NP-complete.

- DUH is in NP: Given a solution y, it is possible to check it in polynomial time simply by running $M(y)$ to see if it accepts.
- DUH is NP-hard: Suppose we have an oracle to solve "DUH". Then given any NP problem, by the definition of NP, there must be some polynomial-time Turing machine $M\prime(x, y)$ for verifying a claimed solution y. So just ask the oracle whether there exists a y that causes M to accept (having hardwired the input x into the description of M).

So the real achievement of Cook, Karp, and Levin was not to show that NP-complete problems exist. That is almost tautological. Their real achievement was to show that many natural problems are NP-complete.

4.4 Proving a problem is NP-complete

To show a given problem L is NP-complete, you only have to do two things:

1. Show L $\in$ NP.
2. Show some problem K already known to be NP-complete reduces to L. This implies that L is NP-hard, since it must be at least as hard as K for K to reduce to L.

5 The Cook-Levin Theorem

Consider another problem: SAT (for Satisfiability). Given an arbitrary Boolean formula (in propositional logic), is there any way of setting the variables so that the formula evaluates to `TRUE`?

Theorem. SAT is a NP-complete problem.

Proof. First, show SAT is in NP. (This is almost always the easiest part.) Given a proposed set of arguments, plug them into the equation and see if it evaluates to `TRUE`.

Next, show that some *known* NP-complete problem reduces to SAT to show that SAT is NP-hard. We know DUH is NP-complete, so let's reduce DUH to SAT. In other words: given some arbitrary polynomial-time Turing machine M, we need to create a Boolean formula F, which will be satisfiable if and only if theres an input y that causes M to accept. And the translation process itself will have to run in polynomial time.

The way to do this is very simple in concept, but the execution is unbelievably complicated, so we'll focus more on the concept.

Remember that a Turing machine consists of this tape of 0s and 1s, together with a tape head that moves back and forth, reading and writing. Given a Turing machine M, and given the input $y = y_1, \ldots, y_r$ that's written on its tape, the entire future history of the machine is determined. That means we can draw what is called a *tableau*, which shows the entire history of M's computation at a single glance. Each row of a tableau indicates the state of the Turing machine, the value of the tape, and the position of the Turing machine on the tape.

Time	State	Tape					
6	[H]	0	1	1	1	1	0
5	B	0	1	[1]	1	1	0
4	A	0	[0]	1	1	1	0
3	B	0	0	[0]	1	1	0
2	A	0	0	0	[1]	1	0
1	B	0	0	0	1	[0]	0
0	A	0	0	0	[0]	0	0

Figure 2: Example tableau for a Turing machine that halts after 6 steps.

The tableau for SAT will have T time steps, where T is some polynomial in n, and *because* it only has T time steps, we only ever need to worry about T tape squares from side to side. This means that the whole tableau is of polynomial size. And let's say that the machine accepts if and only if at the very last step some tape square T_i has a 1.

The question that interests us is whether there's *some* setting of the "solution bits" $y = y_1, \ldots, y_r$ that causes M to accept.

How can we create a formula that's satisfiable if and only if such a setting exists? We should certainly include $y = y_1, \ldots, y_r$ as variables of the formula. But we're going to need a whole bunch of additional variables. Basically, for every yes-or-no question you could possibly ask about this tableau, we're going to define a Boolean variable representing the answer to that question. Then we're going to use a slew of Boolean connectives to force those variables to relate to each other in a way that mimics the actions of the Turing machine.

- For all i, t, let $x_{t,i} = 1$ iif at step t, the i^{th} tape square is set to 1.
- Let $p_{t,j} = 1$ iif at step t the Turing machine head is at the i^{th} square.
- Let $s_{t,k} = 1$ iif at step t the Turing machine is in state k.

Next, look at each "window" in the tableau, and write propositional clauses to make sure the right thing is happening in that window.

- First, $x_{1,j} = y_j$ for all j from 1 to r, and $x_{1,j} = 0$ for all $j > r$.
- $p_{t,j} = 0 \Rightarrow x_{t+1,j} = x_{t,j}$. This says that if the tape head is somewhere else, then the j^{th} square is unchanged. We need such a clause for every t and j.
- $s_{t,k} = 1 \wedge p_{t,j} = 1 \wedge x_{t,j} = 1 \Rightarrow s_{t+1,k\prime} = 1$. This says that if, at time t, the machine is in state k and at tape square j, and 1 is written on that tape square, then at the next time step, it will be in state $k\prime$. We'll also need clauses to make sure that $\forall l \neq k\prime : s_{t+1,l} = 0$
- ... and so on.

We need one final clause to make sure that the machine actually accepts.

- $x_{T,0} = 1$

Finally, we string all of these clauses together to produce a humongous—but still polynomial-size!—Boolean formula F, which is consistent (i.e., satisfiable) if and only if there's some $y = y_1, \ldots, y_r$ that causes M to accept.

So, *supposing* we had a polynomial-time algorithm to decide whether a Boolean formula is satisfiable, we could use that algorithm to decide in polynomial time whether there is an input that causes the Turing machine M to accept. In fact, the two problems are equivalent. Or to put it another way, **SAT is NP-complete**. That is the Cook-Levin Theorem.

5.1 Special-case: 2SAT

The above proof doesn't preclude that *special cases* of the SAT problem could be solved efficiently. In particular, consider the problem from Lecture 2 where you have clauses consisting of only two literals each. This problem is called 2SAT, and as we showed in back in Lecture 2, 2SAT is in P. If your Boolean formula has this special form, then there *is* a way to decide satisfiability in polynomial time.

5.2 Special-case: 3SAT

In contrast to 2SAT, let's consider the problem where each clause has *three* literals, or 3SAT. It turns out that 3SAT is NP-complete; in other words, this special case of SAT encodes the entire difficulty of the SAT problem itself. We'll go over the proof for this in Lecture 10.

Appendix

Kurt Gödel's letter to John von Neumann

To set the stage: Gödel is now a recluse in Princeton. Since Einstein died the year before, Gödel barely has anyone to talk to. Later he'll starve himself because he thinks his wife is trying to poison him. Von Neumann, having previously formalized quantum mechanics, invented game theory, told the US government how to fight the Cold War, created some of the first general-purpose computers, and other great accomplishments, is now dying of cancer in the hospital. Despite von Neumann's brilliance, he did not solve the problem that Gödel presented in this letter, and to this day it remains one of the hallmark unsolved problems in theoretical computer science.

> Princeton, 20 March 1956
>
> Dear Mr. von Neumann:
>
> Since you now, as I hear, are feeling stronger, I would like to allow myself to write you about a mathematical problem, of which your opinion would very much interest me: One can obviously easily construct a Turing machine, which for every formula F in first order predicate logic and every natural number n, allows one to decide if there is a proof of F of length n (length = number of symbols). Let $\Psi(F, n)$ be the number of steps the machine requires for this and let $\varphi(n) = \max_F \Psi(F, n)$. The question is how fast $\varphi(n)$ grows for an optimal machine. One can show that $\varphi(n) \geq k \cdot n$. If there really were a machine with $\varphi(n) \sim k \cdot n$ (or even with $\sim k \cdot n^2$), this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem, the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. After all, one would simply have to choose the natural number n so large that when the machine does not deliver a result, it makes no sense to think more about the problem. Now it seems to me, however, to be completely within the realm of possibility that $\varphi(n)$ grows that slowly. Since
>
> 1. it seems that $\varphi(n) \geq k \cdot n$ is the only estimation which one can obtain by a generalization of the proof of the undecidability of the Entscheidungsproblem and
> 2. after all, $\varphi(n) \sim k \cdot n$ (or $\sim k \cdot n^2$) only means that the number of steps as opposed to trial and error can be reduced from N to $\log N$ (or $\log^2 N$).
>
> However, such strong reductions appear in other finite problems, for example in the computation of the quadratic residue symbol using repeated application of the law of reciprocity. It would be interesting to know, for instance, the situation concerning the determination of primality of a number and how strongly in general the number of steps in finite combinatorial problems can be reduced with respect to simple exhaustive search.

I do not know if you have heard that "Post's problem", whether there are degrees of unsolvability among problems of the form $(\exists y)\varphi(y, x)$, where φ is recursive, has been solved in the positive sense by a very young man by the name of Richard Friedberg. The solution is very elegant. Unfortunately, Friedberg does not intend to study mathematics, but rather medicine (apparently under the influence of his father). By the way, what do you think of the attempts to build the foundations of analysis on ramified type theory, which have recently gained momentum? You are probably aware that Paul Lorenzen has pushed ahead with this approach to the theory of Lebesgue measure. However, I believe that in important parts of analysis non-eliminable impredicative proof methods do appear.

I would be very happy to hear something from you personally. Please let me know if there is something that I can do for you. With my best greetings and wishes, as well to your wife.

Sincerely yours,

Kurt Gödel

P.S. I heartily congratulate you on the award that the American government has given to you.

Source: http://weblog.fortnow.com/2006/04/kurt-gdel-1906-1978.html

6.080/6.089 GITCS Mar 11, 2008

Lecture 10

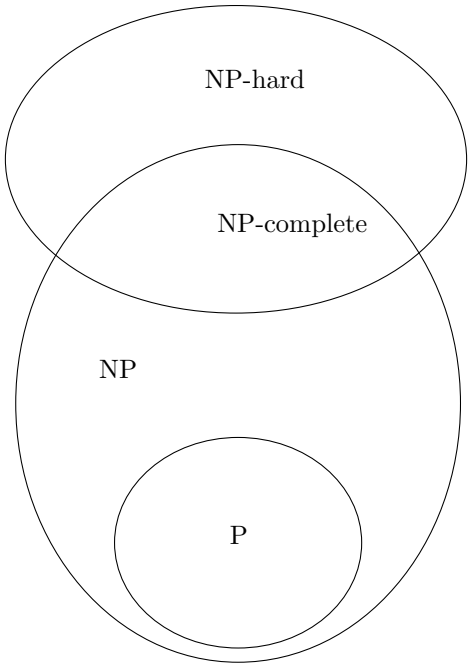
Lecturer: Scott Aaronson *Scribe: Yinmeng Zhang*

1 Administrivia

Work hard on the homework, but don't freak out. Come to office hours. There will be a short (1 week) pset before spring break, and an exam on the Thursday after, which is April 3rd.

2 Recap

Last time we saw a rough geography of the universe of computational problems.



One of the precepts of this course is that this picture is really important. In an ideal world, people would recognize this map the same way they recognize a map of the continents of earth. As part of the popularization effort we are distributing Scott's Scientific American article on the limits of quantum computation, which contains a similar picture, except quantum-ier.

The concept of **NP**-completeness was defined in the '70s. Last time we saw that the DUH problem — given a Turing machine, is there an input on which it halts in polynomial time — is obviously **NP**-complete, duh. We then took a look at our first "natural" **NP**-complete problem, SATISFIABILITY (or SAT for short). We saw Cook and Levin's proof that SAT is **NP**-complete by reduction from DUH. Today we will see some more reductions.

3 SAT reduces to 3SAT

Though SAT is **NP**-complete, specific instances of it can be easy to solve.

Some useful terminology: a *clause* is a single disjunction; a *literal* is a variable or the negation of a variable. A Boolean formula in *conjunctive normal form* (CNF) is one which is the AND of ORs of literals. Then the 2SAT problem (which we saw earlier in the course) is defined as follows:

- Given a CNF formula where every clause involves at most 2 literals, does it have a satisfying assignment?

In Lecture 2 we saw that 2SAT is solvable in polynomial time, because we can draw the implication graph and check it for cycles in polynomial time.

Today, we'll talk about the 3SAT problem.

- Given a CNF formula where every clause involves at most 3 literals, does it have a satisfying assignment?

Unlike 2SAT, the 3SAT problem appears to be hard. In fact, 3SAT is **NP**-complete — this special case encapsulates all the difficulty of generic SAT!

To prove this, we'll show how to convert an arbitrary Boolean formula into a 3SAT problem in polynomial time. Thus, if we had an oracle for 3SAT, then we could solve SAT by converting the input into 3SAT form and querying the oracle.

So how do we convert an arbitrary Boolean formula into a 3SAT formula? Our first step will be to convert the formula into a circuit. This is actually really straightforward – every $\neg$, $\wedge$, or $\vee$ in the formula becomes a NOT, AND, or OR gate in the circuit. One detail we need to take care of is that when we say multiple literals $\wedge$ed or $\vee$ed together, we first need to specify an order in which to take the $\wedge$s or $\vee$s. For example, if we saw $(x_1 \vee x_2 \vee x_3)$ in our formula, we should parse it as either $((x_1 \vee x_2) \vee x_3)$ which becomes $OR(x_1, OR(x_2, x_3))$, or $(x_1 \vee (x_2 \vee x_3))$ which becomes $OR(OR(x_1, x_2), x_3)$. It doesn't matter which one we pick, because $\wedge$ and $\vee$ are both associative.

So every Boolean formula can be converted to an equivalent circuit in linear time. This has a couple of interesting consequences. First, it means the problem CircuitSAT, where we are given a circuit and asked if it has a satisfying assignment, is **NP**-complete. Second, because Cook-Levin gave us a way to convert Turing Machines into Boolean formulas if we knew a bound on the run time, tacking this result on gives us a way to convert Turing Machines into circuits. If the Turing Machine ran in polynomial time, then the conversion will only polynomial time, and the circuit will be of polynomial size.

Ok, so now we have a circuit and want to convert it into a 3SAT formula. Where does the 3 come in? Each gate has at most 3 wires attached to it – two inputs and one output for AND and OR, and one input and one output for NOT. Let's define a variable for every gate. We want to know if there is a setting of these variables such that for every gate, the output has the right relationship to the input/inputs, and the final output is set to true.

So let's look at the NOT gate. Call the input x and the output y. We want that if x is true then y is false, and if x is false then y is true – but we've seen how to write if-then statements as disjunctions! So, we have

$$NOT : (x \vee y) \wedge (\neg x \vee \neg y)$$

We can do the same thing with the truth table for OR. Call the inputs x_1 and x_2 and the output y. Let's do one row. If x_1 is true and x_2 is false, then y is true. We can write this as

$(\neg(x_1 \wedge \neg x_2) \vee y)$. By DeMorgan's Law, this is the same as $(\neg x_1 \vee x_2 \vee y)$. Taking the AND over all possibilities for the inputs we get

$$OR : (\neg x_1 \vee \neg x_2 \vee y) \wedge (\neg x_1 \vee x_2 \vee y) \wedge (x_1 \vee \neg x_2 \vee y) \wedge (x_1 \vee x_2 \vee \neg y)$$

And similarly for the AND gate. The big idea here is that we're taking small pieces of a problem we know to be hard (gates in a circuit) and translating them into small pieces of a problem we're trying to show is hard (CNF formulas). These small pieces are called "gadgets", and they're a recurring theme in reduction proofs. Once we've got these gadgets, we have to somehow stick them together.

In this case, all we have to do is AND together the Boolean formulas for each of the gates, and the variable for the final output wire. The Boolean formula we get from doing this is true if and only if the circuit is satisfiable. Woo.

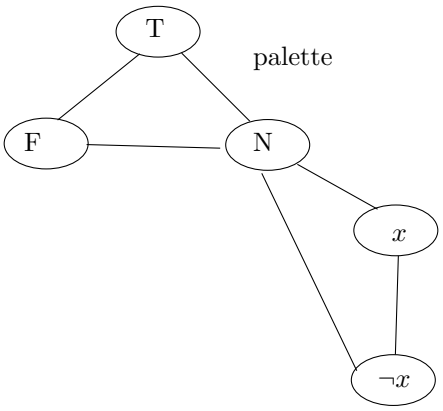
4 3COLOR is NP-complete

- Given a graph, is there a way to color each vertex either Red, Blue, or Green, so that no adjacent vertices end up the same color?

This problem is called 3COLOR, and it is **NP**-complete. We'll prove it by showing a reduction from CircuitSAT. So suppose we're given a circuit, and we have a magic box that, when given a graph, says whether it has a good 3-coloring. Can we make a graph that is 3-colorable iff the Boolean formula is satisfiable? We're going to need some gadgets.

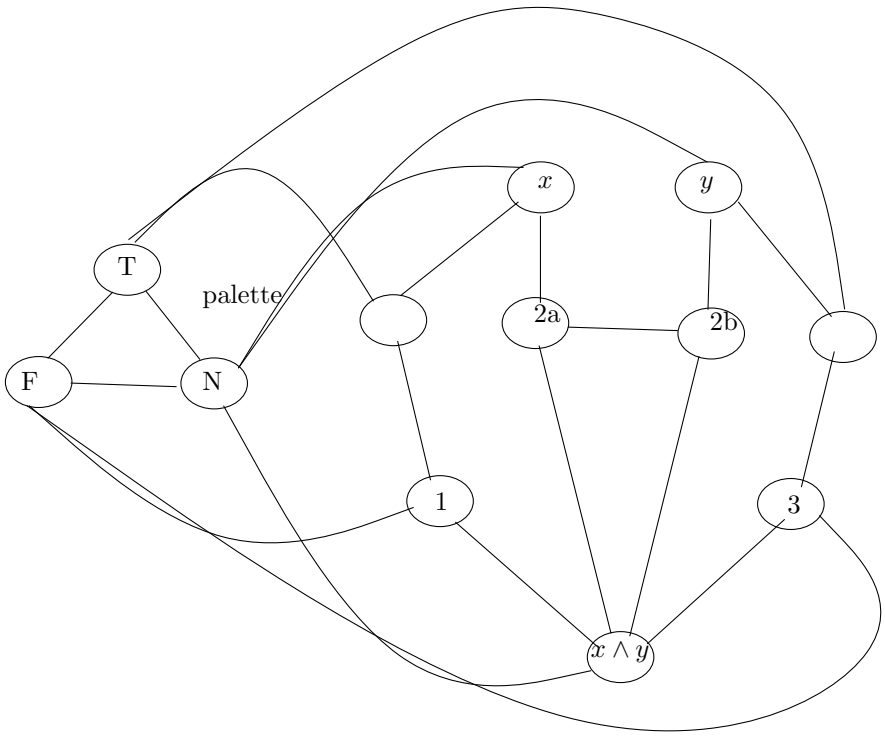
First of all, we're renaming the colors. The first piece of our graph is going to be a triangle, which will serve as a palette. Notice that any good 3-coloring must assign different colors to each vertex of a triangle. So for a given good 3-coloring, whatever color the first vertex gets assigned, we're going to call True; the second is going to be False; and the third is going to be Neither.

Here is a gadget for the NOT gate.



Because x and $\neg x$ are connected to each other, any good coloring will assign them different colors, and because they're both connected to the Neither vertex in the palette, one of them will have to be true, and one of them will be false.

Here is a gadget for the AND gate, courtesy of Alex.



It's obvious it works, right? The main things to note are as follows. The inputs and output are connected to Neither, which forces them to take on boolean values. If x and y are colored the same, then $2a$ and $2b$ will be colored with the other two colors, forcing $x \wedge y$ will have to be colored the same as x and y – that's correct behavior for an AND! If x and y are colored oppositely, then either 1 or 3 will have to be colored True, which forces $x \wedge y$ to be colored False – that's again what we were hoping would happen. Finally, in all cases, we haven't added too many restrictions: good colorings exist for any setting of x and y.

A gadget for the OR gate could be made by composing the AND and NOT gadgets using DeMorgan's Law. There's also a cute way to modify the AND gadget to turn it into the OR gadget. Do you see it?

By composing these gadgets, we can turn any circuit into a graph so that assignment of T/Fs to the wires translates to colorings of the graph. If we then connect the final output vertex to the False vertex in the palette, then the graph we've constructed will have a good 3-coloring iff the circuit had a satisfying assignment.

So we've just proved 3COLOR is **NP**-complete. This is a strange thing. It means, for example, that we can efficiently construct graphs which are 3-colorable iff there's a short proof of the Riemann Hypothesis in ZF set theory. Chew on that.

Looking at the somewhat horrifying AND gadget, you might wonder if there's a way to draw the graph so that no lines cross. This turns out to be possible, and implies that 3PLANAR-COLOR (given a planer graph, is it 3-colorable) is also **NP**-complete. What about 4PLANAR-COLOR? This problem turns out to be easy. In fact, you can just output yes every time! It was a long-standing conjecture that every planar graph could be colored with 4 colors. It was finally proven by Appel and Haken in 1976. There is some drama surrounding this proof that you can read about on Wikipedia. What about 2PLANAR-COLOR? This is easy too. It's a special case of 2COLOR, which we saw a greedy algorithm for last time. When it comes to planar graph-coloring, it seems there's something special about the number 3.

5 In conclusion

In 1972, Karp showed 21 problems were **NP**-complete using the above techniques and a good dose of cleverness. We could spend a month looking at clever gadgets, but we're not going to. The takeaway lesson is that **NP**-completeness is ubiquitous. Today, thousands of practical problems have been proved **NP**-complete. In fact, any time an optimization problem arises in industry, etc., a good heuristic is to assume it's **NP**-complete until proven otherwise! This, of course, is what motivates the question of whether **P**=**NP** — what makes it one of the central questions in all of math and science.

6 Tricky Questions

Cook defined a language A to **NP**-complete when it itself was in **NP**, and every **NP** problem could be solved in polynomial time if given oracle access to A.

- A is in **NP**
- SAT is in $\mathbf{P}^A$

This definition allows us to call the oracle multiple times, but in fact in the above proofs we only ever called it once. The proofs had a very specific form: somehow come up with an equivalent problem instance in the candidate language; call the oracle on it. Karp noticed this and proposed a different notion of **NP**-completeness

- There exists a polynomial time algorithm that maps satisfiable formulas to strings in A, and unsatisfiable formulas to strings outside A.

Are these definitions the same or is the second one more restrictive? This is an open problem, which you are welcome to close for extra credit.

Now, if **P**=**NP**, then every problem in **NP** will be solvable efficiently, and every problem in **NP** will be **NP**-complete: the computational universe will look pretty monotonous. If **P**≠**NP**, then what do we know? All the problems in **P** will be easy, and all **NP**-complete problems will be hard. Would there be anything in between? Would there be problems in **NP** which are neither **NP**-complete nor solvable efficiently? In 1975 Ladner showed that the answer is yes. Tune in next time to find out more.

6.080/6.089 GITCS March 13, 2008

Lecture 11

Lecturer: Scott Aaronson *Scribe: Michael Fitzgerald*

1 NP-Completeness In Practice

Recent lectures have talked a lot about technical issues. This lecture is going to take a step back.

We've shown that **NP**-complete problems do occur in real-life problems and situations, so obviously engineers, scientists, etc. have come across them while doing their jobs. For example, trying to set up an optimal airline flight schedule, or an optimal conflict-minimizing class schedule, or trying to design the optimal airplane wing. When you run up against an **NP**-complete problem, however, do you just give up? No, there are many ways to cope with an encounter with an **NP**-complete problem. You have to change the way you look at the problem; you stop looking for a general-purpose solution and play a different game.

1.1 Alternatives

We'll discuss seven strategies for dealing with **NP**-complete problems.

1.1.1 Brute Force

This works fine if you have a small instance of a problem; you just try every possible solution until you find one that works. Your problem could be **NP**-complete, but if your n is 4, finding the solution is trivial. For example, the sudoku in your Sunday paper is possible to brute force; you can find a solution for a nine by nine grid, but huge ones are much more difficult.

1.1.2 Structure Exploitation

Your particular instance of the **NP**-complete problem might have a structure that lends itself to a solution. Remember, "**NP**-complete" is a statement about the while *class* of problems; it doesn't say anything about the complexity of an individual instance. "Proof With n Symbols" is an **NP**-complete problem, but humans have quite obviously solved many specific instances of it.

1.1.3 Special Casing

You can find special cases of the general problem that both lend themselves to a general solution and contain your instance of the general problem. For example, $2SAT$ is a special case of SAT that is solvable; it has a special form that has a polynomial-time solution algorithm. Every **NP**-complete problem has some special cases that can be solved in polynomial-time; you just have to find a special subset to which your instance belongs.

1.1.4 Taking Advantage of Parameters Besides n

This strategy is related to the previous one. Traditionally, we consider everything to be a function of n, the size of the input. But we can also consider other parameters. As an example, recall the max-clique problem: given a set of people, find the largest subset that are all friends with each

other. This is an **NP**-complete problem. But suppose we add another parameter k, and change the question to whether there's a clique of size k. If k is given as part of the input, then this problem is still **NP**-complete. (As a sanity check, if $k = n/2$, then the number of possibilities to check is something like $\binom{n}{n/2}$, which grows exponentially with n.) If k is a constant, however, the problem *does* have an efficient solution: you can simply check all sets of size k, which takes $\binom{n}{k}$ time. So, for every fixed k, there's a polynomial-time algorithm to solve the problem.

Indeed, in some cases one can actually do better, and give an algorithm that takes $f(k)p(n)$ time, for some function f and polynomial p (so that the parameter k no longer appears in the exponent of n). The study of the situations where this is and isn't possible leads to a rich theory called *parameterized complexity*.

1.1.5 Approximation

If we can't get an optimal solution, sometimes we can get one that's close to optimal. For example, in the max-clique problem, can we find a clique that's at least half the size of the biggest one? In the traveling salesman problem, can we find a route that's at most twice as long as the shortest route? The answers to these questions will in general depend on the problem.

Here's a more in-depth example. We know that $3SAT$ is **NP**-complete, but is there an efficient algorithm to satisfy (say) a 7/8 fraction of the clauses? If we fill in each of the variables randomly, we have a 1/8 chance that each individual clause is false. So on average we're going to satisfy about 7/8 of the clauses, by linearity of expectation.

On the other hand, is there an efficient algorithm to satisfy a $7/8 + \epsilon$ fraction of the clauses, for some constant $\epsilon > 0$? A deep result of Håstad (building on numerous contributions of others) shows that this is already an **NP**-complete problem—that is, it's already as hard as solving $3SAT$ perfectly. This means that 7/8 is the limit of approximability of the $3SAT$ problem, unless **P**=**NP**.

The approximability of **NP**-complete problems has been a huge research area over the last twenty years, and by now we know a quite a lot about it (both positive and negative results). Unfortunately, we won't have time to delve into the topic further in this course.

1.1.6 Backtracking

We can make the brute force method more effective by generating candidate solutions in a smarter way (in other words, being less brute). In the 3-coloring problem, for example, you might know early on that a certain solution won't work, and that will let you prune a whole portion of the tree that you might otherwise have wasted time on. In particular, given a map, you can fill in colors for each of the countries, branching whenever more than one color can work for a given country. If you reach a conflict that invalidates a potential solution, go back up the tree until you reach an unexplored branch, and continue on from there. This algorithm will eventually find a solution, provided one exists—not necessarily in polynomial time, but at least in exponential time with a smaller-order exponential.

1.1.7 Heuristics

The final strategy involves using heuristic methods such as simulated annealing, genetic algorithms, or local search. These are all algorithms that start with random candidate solutions, and then repeatedly make local changes to try and decrease "badness." For example, with 3-coloring, you might start with a random coloring and then repeatedly change the color of a conflicting country, so that the conflicts with its neighbors are fixed. Rinse and repeat.

This sort of approach will not always converge efficiently on a good solution—the chief reason being that it can stuck on local optima.

1.2 In Nature

Nature deals with **NP**-complete problems constantly; is there anything we can learn from it? Biology employs what computer scientists know as genetic algorithms. Sometimes biology simply mutates the candidate solutions, but other times it combines them (which we call sexual reproduction).

Biology has adopted the sexual reproduction technique so widely that it *must* have something going for it, even if we don't yet fully understand what it is. In experiments, local search algorithms that employ only mutations (i.e. "asexual reproduction") consistently do better at solving **NP**-complete problems than do algorithms that incorporate sex.

The best theory people currently have as to why nature relies so heavily on sex is that nature isn't trying to solve a fixed problem; the constraints are always changing. For example, the features that were ideal for an organism living in the dinosaur age are different from the features that are ideal for modern-day organisms. This is still very much an open issue.

As another example of nature apparently solving **NP**-complete problems, let's look at protein folding. Proteins are one of the basic building blocks of a cell. DNA gets converted into RNA, which gets converted into a protein, which is a chain of amino acids. We can think of that chain of amino acids as just encoded information. At some point, however, that information needs to be converted into a chemical that interacts with the body in some way. It does this by folding up into a molecule or other special shape that has the requisite chemical properties. The folding is done based on the amino acids in the chain, and it's an extremely complicated process.

The key computational problem is the following: given a chain of amino acids, can you predict how it's going to fold up? In principle, you could simulate all the low-level physics, but that's computationally prohibitive. Biologists like to simplify the problem by saying that every folded state has potential energy, and the folding is designed to minimize that energy. For idealized versions of the problem, it's possible to prove that finding a minimum-energy configuration is **NP**-complete. But is every cell in our body really solving hard instances of **NP**-complete problems millions of times per second?

There is another explanation. Evolution wouldn't select for proteins that are extremely hard to fold (e.g., such that any optimal folding must encode a proof of the Riemann Hypothesis). Instead, it would select for proteins whose folding problems are computationally easy, since those are the ones that will fold reliably into the desired configuration.

Note that when things go wrong, and the folding *does* get caught in a local optimum, you can have anomalies such as prions, which are the cause of Mad Cow Disease.

2 Computational Universe Geography

We finished previously by mentioning Ladner's Theorem: if $\mathbf{P} \neq \mathbf{NP}$, then there are **NP** problems that are in neither in **P** nor **NP**-complete. The proof of this theorem is somewhat of a groaner; the basic idea is to define a problem where for some input sizes n the problem is to solve SAT (keeping the problem from being in **P**), and for other input sizes the problem is to do nothing (keeping the problem from being **NP**-complete). The trick is to do this in a careful way that keeps the problem in **NP**.

While the problem produced by Ladner's theorem isn't one we'd ever care about in practice, there *are* some problems that are believed to be intermediate between **P** and **NP**-complete and that matter quite a bit. Perhaps the most famous of these is factoring.

2.1 Factoring

We don't yet know how to base cryptography on **NP**-complete problems. The most widely used cryptosystems today are instead based on number theory problems like factoring, which—contrary to a popular misconception—are neither known nor believed to be **NP**-complete.

Can we identify some difference between factoring and the known **NP**-complete problems, that might make us suspect that factoring is *not* **NP**-complete?

One difference involves the number of solutions. If we consider an **NP**-complete problem like 3-coloring, there might be zero ways to 3-color a given map, a few ways, or an enormous number of ways. (One thing we know is that the number of colorings has to be divisible by 6—do you see why?) With factoring, by contrast, there's exactly one solution. Every positive integer has a unique prime factorization, and once we've found it there aren't any more to find. This is very useful for cryptography, but it also makes it extraordinarily unlikely that factoring is **NP**-complete.

2.2 coNP

Why do I say that?

Well, let's forget about factoring for the moment and ask a seemingly unrelated question. Supposing that there's no efficient algorithm to find a 3-coloring of a graph, can we at least give a short proof that a graph isn't 3-colorable? If the backtracking tree is short, you could just write it down, but that would take exponential time in the worst case. The question of whether there exist short proofs of *un*satisfiability, which can be checked in polynomial time, is called the **NP** versus **coNP** question. Here **coNP** is the set of *complements* of **NP** problems: that is, the set $\{\overline{L} : L \in \mathbf{NP}\}$.

Relating this back to the original **P** versus **NP** question, if **P**=**NP** then certainly **NP**=**coNP** (if we can decide satisfiability in polynomial time, then we can also decide unsatisfiability!). We're not sure, however, if **NP**=**coNP** implies **P**=**NP**; we don't know how to go in this direction. Still, almost all computer scientists believe that **NP**≠**coNP**, just as they believe **P**≠**NP**.

2.3 New Universe Map

We now have five main regions on our universe map: **P**, **NP**, **NP**-complete, **coNP**, and **coNP**-complete.

If a problem is in both **NP** and **coNP**, does that mean it's in **P**? Well, consider a problem derived from factoring, like the following: given a positive integer n, does n have a prime factor that ends in 7? This problem is in both **NP** and **coNP**, but we certainly don't know it to be in **P**.

Finally, here's the argument for factoring not being **NP**-complete. Because of the existence of unique factorization, factoring is in **NP**∩**coNP**. So if factoring were **NP**-complete, then an **NP**-complete problem would be in **coNP**. So **NP** itself would be contained in **coNP**, and (by symmetry) would equal **coNP**—thereby making a large part of the computational universe collapse. To put it differently, if **NP**≠**coNP**, then factoring can't be **NP**-complete.

6.080/6.089 GITCS DATE

Lecture 12

Lecturer: Scott Aaronson *Scribe: Mergen Nachin*

1 Review of last lecture

- NP-completeness in practice. We discussed many of the approaches people use to cope with NP-complete problems in real life. These include brute-force search (for small instance sizes), cleverly-optimized backtrack search, fixed-parameter algorithms, approximation algorithms, and *heuristic algorithms* like simulated annealing (which don't always work but often do pretty well in practice).
- Factoring, as an intermediate problem between P and NP-complete. We saw how factoring, though believed to be hard for classical computers, has a special structure that makes it different from any known NP-complete problem.
- coNP.

2 Space complexity

Now we will categorize problems in terms of how much memory they use.

Definition 1 *L is in PSPACE if there exists a poly-space Turing machine M such that for all x, x is in L if and only if M(x) accepts. Just like with NP, we can define PSPACE-hard and PSPACE-complete problems.*

An interesting example of a PSPACE-complete problem is n-by-n chess:

> Given an arrangement of pieces on an n-by-n chessboard, and assuming a polynomial upper bound on the number of moves, decide whether White has a winning strategy.

(Note that we need to generalize chess to an n-by-n board, since standard 8-by-8 chess is a finite game that's completely solvable in $O(1)$ time.)

Let's now define another complexity class called EXP, which is apparently even bigger than PSPACE.

Definition 2 *Deterministic $f(n)$-Time, denoted DTIME($f(n)$), is the class of decision problems solvable by a Turing machine in time $O(f(n))$.*

Definition 3 *Exponential Time, denoted EXP, equals the union of $DTIME(2^{p(n)})$ over all polynomials p.*

Claim 4 $PSPACE \subseteq EXP$

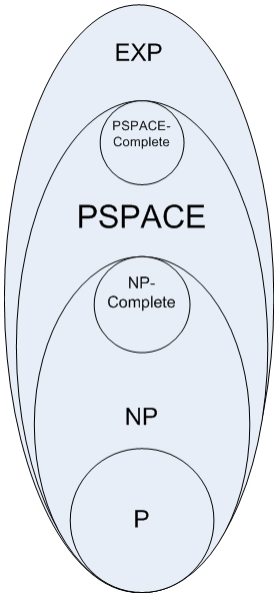


Figure 1: A general picture that we believe

Proof Just like in problem set 2. In order for the machine to halt, any "configuration" must be visited at most once. Assuming an upper bound of $p(n)$ on the number of tape squares, there are $2^{p(n)}$ possible settings of 0's and 1's on the tape, $p(n)$ possible locations for the Turing machine head, and s possible states for the head. So an upper bound on number of steps is $2^{p(n)}p(n)s$, which is exponential in $p(n)$. ■

Claim 5 *$P \subseteq PSPACE$.*

Proof Obviously a P machine can't access more than a polynomial number of memory locations. ■

Claim 6 *$NP \subseteq PSPACE$.*

Proof Enumerate all possible polynomial-size proofs. ■

Remark Does NP=PSPACE? Just like P vs. NP, this is a longstanding open problem! It's conjectured that they're different.

Definition 7 *LOGSPACE is the class of decision problems solvable by a Turing machine with $O(\log n)$ bits of memory.*

Remark But isn't the input already n bits? A technicality: The LOGSPACE machine has *read-only access* to the input. It's only the *read-write* memory that's logarithmically bounded.

Claim 8 *$LOGSPACE \subseteq P$.*

Proof Same as Claim 4, except "scaled down by an exponential." There are at most $s2^{c \log n} c \log n = n^{O(1)}$ possible configurations of a Turing machine with $c \log n$ tape squares. ■

Remark Does LOGSPACE=P? Another longstanding open problem! Again, conjecture is that they are different.

We *can* prove $LOGSPACE \neq PSPACE$, using a Space Hierarchy Theorem similar to the Time Hierarchy Theorem that we saw in Lecture 7. As usual, it's not hard to prove that *more of the same resource* (time, space, etc) provides more computational power than less of it. The hard part is to compare *different* resources (like time vs. space, or determinism vs. nondeterminism).

3 Why do we believe $P \neq NP$ if we can't prove it?

- Hardness of solving NP-complete problems in practice: the empirical case.
- There are "vastly easier" problems than NP-complete ones (like factoring) that we already have no idea how to solve in P.
- As Gödel argued, P=NP would mean mathematical creativity could be automated. God would not be so kind!
- We know that $LOGSPACE \neq PSPACE$. But this means either $LOGSPACE \neq P$, or $P \neq NP$, or $NP \neq PSPACE$! And if one is true, then why not all three?

Incidentally, let me tell you one of the inside jokes of complexity theory. Let LINSPACE be the set of problems solvable in linear space. Then one of the very few separations we can prove is that $LINSPACE \neq P$. Why? Well, suppose $P = LINSPACE$. Then $P = PSPACE$ also. Why? Pad the inputs! But that means LINSPACE=PSPACE, which is ruled out by the Space Hierarchy Theorem!

The punchline is, while we know P and LINSPACE are different, we have no idea which one is not contained in the other one (or as is most likely, whether neither is contained in the other one). We just know that they're different!

4 Why is it so hard to prove $P \neq NP$?

- Because $P \neq NP$!
- Because there really are lots of clever, non-obvious polynomial-time algorithms. Any proof that 3SAT is hard will have to *fail* to show 2SAT is hard, even though the "handwaving intuition" seems the same for both (there are 2^n possible solutions, and clearly each one takes at least constant time to check!). Simple as it seems, this criterion already rules out almost every amateur $P \neq NP$ proof in Prof. Aaronson's inbox...
- Let NPSPACE (Nondeterministic PSPACE) be the class of problems solvable by a PSPACE machine that can make nondeterministic transitions. Then by analogy to $P \neq NP$, you might conjecture that $PSPACE \neq NPSPACE$. But *Savitch's Theorem* shows that this conjecture is false: PSPACE=NPSPACE. So any proof of $P \neq NP$ will have to fail when the polynomially-bounded resource is space rather than time.

- The Baker-Gill-Solovay argument. Techniques borrowed from logic and computability theory, like the ones used to prove the Time Hierarchy Theorem, all seem to *relativize*. In other words, they all seem to go through without change if we assume there's a magical oracle in the world that solves some problem for free. As an example, recall how the proof of unsolvability of the halting problem could easily be adapted to show the unsolvability of the "Super Halting Problem" by "Super Turing Machines". By contrast, any solution to the P vs. NP problem will have to be *non-relativizing*. In other words, it will have to be sensitive to whether or not there's an oracle around. Why? Well, because there are some oracle worlds where P=NP, and others where $P \neq NP$! Specifically, let A be any PSPACE-complete problem. Then I claim that $P^A = NP^A = PSPACE$. (Why?) On the other hand, we can also create an oracle B such that $P^B \neq NP^B$. For every input length n, either there will exist a $y \in \{0,1\}^n$ such that $B(y) = 1$, or else $B(y)$ will equal 0 for all $y \in \{0,1\}^n$. Then given an n-bit input, the problem will be to decide which. Certainly this problem is in NP^B (why?). On the other hand, at least intuitively, the only way a deterministic machine can solve this problem is by asking the oracle B about exponentially many y's. Baker, Gill, and Solovay were able to formalize this intuition (details omitted), thus giving an oracle world where $P \neq NP$.

5 Starting a new unit: Randomness

People have debated the true nature of randomness for centuries. It seems clear what we mean when we say two dice have a 1/36 probability of landing snake-eyes: that if you roll infinitely many times, in the limit you'll get snake-eyes 1/36 of the time. On the other hand, if you go to Intrade.com (at the time of this writing), you can find the market estimates a 70% probability for Obama to win the primary and a 30% probability for Hillary to win. But what does such a probability mean in philosophical terms? That if you literally reran the election infinitely many times, in the limit Obama would win 70% of the time? That seems ridiculous! You can make the puzzle sharper by supposing the election has already happened, and people are trying to guess, what's the probability that when the votes are counted, this candidate will win. Well, the votes are already there! Or at least, what the voting machine says are the votes are there. Or suppose someone asks you for the probability that P=NP. Well, they're either equal or not, and they've been that way since before the beginning of the universe!

Interestingly, in the context of Intrade, probability does have a pretty clear meaning. To say that Obama is estimated by the market to have a 70% chance of winning means, if you want to buy a futures contract that will pay a dollar if he wins and nothing if he loses, then the cost is 70 cents. There's a whole field called decision theory that *defines* probabilities in terms of what bets you'd be willing to make. There are even theorems to the effect that, if you want to be a rational bettor (one who can't be taken advantage of), then you have to act as if you believe in probabilities whether you like them or not.

And then there's the physics question: are the laws of physics fundamentally probabilistic, or is there always some additional information that if we knew it would restore determinism? I'm sure you all know the story of Einstein saying that God doesn't play dice, and quantum mechanics saying God does play dice, and quantum mechanics winning. The world really does seem to have a fundamental random component.

Fortunately, for CS purposes we don't have to worry about many of these deep issues. We can just take the laws of probability theory as axioms.

6.080/6.089 GITCS 20 March 2008

Lecture 13

Lecturer: Scott Aaronson *Scribe: Emilie Kim*

1 Slippery Nature of Probabilities

Why are probabilities so counter-intuitive? Let's look at some examples to hone our intuition for probabilities.

1.1 Two children

Let's assume that boys and girls are equally likely to be born. A family has two children. One of them is a boy. What is the chance that the other one is a boy?

Older child:	B	G	B
Younger child:	G	B	B

The probability that the other child is a boy is 1/3, not 1/2.

The assumption in this question was that "One of them is a boy" meant that at least one of the children was a boy. If the question had instead been phrased such that a specific child was a boy, for example, "The older child is a boy. What is the probability that the younger one is a boy?", then the probability would have been 1/2.

1.2 Three unfortunate prisoners

There are three prisoners. One is randomly selected to be executed the next morning, but the identity of the unlucky prisoner is kept a secret from the prisoners themselves. One of the prisoners begs a guard, "At least tell me which of the other two prisoners *won't* be executed. I know at least one of them won't anyway." But the guard says, "From your perspective, that would increase your own chance of being executed from 1/3 to 1/2."

Is the guard correct? No, he isn't. The chance of being executed is still 1/3. (Why?)

1.3 Monty Hall

Suppose you are on a game show. There are three doors, of which one has a car behind it, and the other two have goats. The host asks you to pick a door; say you pick Door #1. The host then opens another door, say Door #3, and reveals a goat. The host then asks you if you want to stay with Door #1 or switch to Door #2. What should you do?

In probability theory, you always have to be careful about unstated assumptions. Most people say it doesn't matter if you switch or not, since Doors #1 and #2 are equally likely to have the car behind them. But at least in the usual interpretation of the problem, this answer is not correct.

The crucial question is whether the *host* knows where the car is, and whether (using that knowledge) he always selects a door with a goat. It seems reasonable to assume that this is the host's behavior. And in that case, switching doors increases the probability of finding the car from 1/3 to 2/3.

To illustrate, suppose the car is behind Door #1 and there are goats behind Doors #2 and #3.

Scenario 1: You choose Door #1. The host reveals a goat behind either Door #2 or Door #3. You switch to the other door and get a goat.

Scenario 2: You choose Door #2. The host reveals a goat behind Door #3. You switch to Door #1 and get the car.

Scenario 3: You choose Door #3. The host reveals a goat behind Door #2. You switch to Door #1 and get the car.

Here's another way of thinking about this problem. Suppose there are 100 doors, and the host reveals goats behind 98 of the doors. Should you switch? Of course! The host basically just told you where the car is.

1.4 Two envelopes

Suppose you have the choice between two envelopes containing money. One envelope has twice as much money as the other envelope. You pick one envelope, and then you're asked if you want to switch envelopes. The obvious answer is that it shouldn't make any difference. However, let x be the number of dollars in the envelope you originally picked. Then the expected number of dollars in the other envelope would seem to be $\frac{1}{2}(2x + \frac{x}{2}) = \frac{5x}{4}$, which is greater than x! So, can you really increase your expected winnings by switching? If so, then what if you keep switching, and switching, and switching—can you get an unlimited amount of money?

As a sanity check, suppose one envelope had a random number of dollars between 1 and 100, and the other envelope had twice that amount. Then if you believed that *your* envelope had more than 100 dollars, it's clear that you wouldn't switch. This simple observation contains the key to resolving the paradox.

See, the argument for switching contained a hidden assumption: that for every positive integer x, your original envelope is just as likely to have $2x$ dollars as it is to have x. But is that even possible? No, because if you think about any probability distribution over positive integers, it's going to have to taper off at some point as you get to larger numbers, so that larger amounts of money become less likely than smaller amounts.

2 Why Do We Need Randomness in Computer Science?

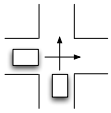
Why is randomness important in computer science? Let's take a look at some examples where randomness seems indispensable—or at the very least, a great help.

2.1 Cryptography

Randomness is absolutely essential to cryptography. Imagine if choosing an encryption key was based on a completely deterministic algorithm! If an eavesdropper can predict exactly what you'll do, then you have a problem.

2.2 Symmetry breaking

Randomness is useful for breaking up the "hallway dance", where you walk towards someone, and you both move to the same side to get out of each other's way, then both to the other side, then both back to the original side, in a continual dance.



Imagine two robot cars at an intersection. If both cars are programmed the same way—and if furthermore they can't distinguish left from right—then they won't be able to move because they'll both be doing the same thing, which would mean entering the intersection at the same time and crashing. On the other hand, if the algorithm governing the cars has a random component, both cars could have a chance of making it through the intersection unscathed. The fascinating field of *distributed computing* (which we unfortunately won't have time to delve into in this course) is full of examples like this. For many distributed computing problems of practical interest, it's possible to prove that there's no solution if all of the agents behave deterministically.

2.3 Database checks

Given two databases, each with 1 terabyte of information, suppose we want to check to see if the contents of the databases are identical. Additionally, suppose the databases are on different continents, so we need to send information over the internet to compare them. How can we do this? Of course, we could just send one database over the internet, but even with today's broadband rates, sending a terabyte probably isn't practical.

Can we just pick a random index and compare that index for both databases? What if the databases differ in just one place? How about summing the contents and comparing the sums? Better, but the sum of the contents of the first database, $x_1 + \cdots + x_n$, could be equal to the sum of the contents of the second database, $y_1 + \cdots + y_n$, even though the actual contents are different.

Comparing checksums is the right idea, but we need to add randomness to make it work. One approach is to interpret the databases as giant integers written in binary:

$$X = x_1 + 2x_2 + 4x_3 + 8x_4 + ...$$

$$Y = y_1 + 2y_2 + 4y_3 + 8y_4 + ...$$

Next, we pick some small random prime number, p. Then we check whether

$$X \bmod p = Y \bmod p.$$

We know that if two numbers are equal, then their remainder modulo anything is also equal. But if two numbers are *not* equal, and we pick a small prime at random and look at the two numbers modulo that random prime, then can we say that with high probability the results will be different?

Incidentally, here's a crucial point about randomized computation: you can never make the assumption that the inputs (in this case the databases) are random. You don't know where an input came from; it's just something someone gave you. The only assumption you can make is that the bits that you specifically *picked* to be random are random.

The question here is: $X - Y \stackrel{?}{=} 0 \bmod p$. This equality holds exactly when p divides $X - Y$. So now the question is how many different prime numbers can divide $X - Y$? Say $X - Y$ is an n-bit number. Then at most n different primes can divide it, since every prime number is at least 2. On the other hand, by the famous Prime Number Theorem, there are at least $\sim \frac{M}{\ln M}$ primes up to a given number M. So let's say we pick a random prime p with at most $10 \log n$ bits. Then p will be at most n^{10}. The number of primes less than n^{10} is quite large (roughly $\frac{n^{10}}{10 \ln n}$), but the number of

possible divisors that $X - Y$ could have is at most n. Therefore, when we pick a random prime, with very high likelihood, we are going to find one that does not divide $X - Y$.

Getting back to our original problem: if we are comparing the contents of two n-bit databases, if we just look at them as integers modulo a random $O(\log n)$-bit prime, then if the two databases are different anywhere, we will see that difference with extremely high probability. This is called **fingerprinting**.

2.4 Monte Carlo simulation

Imagine you are trying to simulate a physical system and you want to get a feel for the aggregate behavior of the system. However, you don't have time to simulate every possible set of starting conditions. By picking various starting conditions at random, you can instead do a random sampling (also called *Monte Carlo simulation*).

But here we already encounter a fundamental question. Do you *really* need random starting conditions? Or would "pseudorandom" starting conditions work just as well? Here pseudorandom numbers are numbers that are meant to *look* random, but that are actually generated by (perhaps complicated) deterministic rules. These rules might initially be "seeded" by something presumed to be random—for example, the current system time, or the last digit of the Dow Jones average, or even the user banging away randomly on the keyboard. But from then on, all further numbers are generated deterministically.

In practice, almost all computer programs (including Monte Carlo simulations) use pseudorandomness instead of the real thing. The danger, of course, is that even if pseudorandom numbers *look* random, they might have hidden regularities that matter for the intended application but just haven't been noticed yet.

2.5 Polynomial equality

Given two polynomials of degree d, each of which is described by some complicated arithmetic formula, suppose we want to test whether they are equal or not.

$$(1-x)^4(x^2-3)^{17} - (x-5)^{62} \stackrel{?}{=} (x-4)^8(x^2+2x)^{700}$$

Sure, we could just expand both polynomials and check whether the coefficients match, but that could take exponential time. Can we do better?

It turns out that we can: we can simply pick random values for the argument x, plug them in, and see if the polynomials evaluate to the same value. If any of the values result in an inequality, then the polynomials are not equal.

Just like in the fingerprinting scenario, the key question is this: if two polynomials are different, then for how many different values of x can they evaluate to the same thing? The answer is d because of the **Fundamental Theorem of Algebra**.

1. Any non-constant polynomial has to have at least one root.
2. A non-constant polynomial of degree d has at most d roots.

When trying to test if two polynomials are equal, we are just trying to determine if their difference is equal to zero or not. As long as x is being chosen at random from a field which is much larger than the degree of the polynomial, chances are if we plug in a random x, the polynomial difference will *not* evaluate to zero, because otherwise the Fundamental Theorem of Algebra would

be violated. So, we know that if two polynomials are different, then with high probability they will differ at any random point where we choose to evaluate them. Interestingly, to this day we don't know how to pick points where the polynomials are different in a deterministic way. We just know how to do it randomly.

3 Basic Tools for Reasoning about Probability

It's time to get more formal about what we mean by probabilities.

A	an event
$Pr[A]$	the probability that event A happens
$Pr[\neg A] = 1 - Pr[A]$	obvious rule
$Pr[A \vee B] = Pr[A] + Pr[B] - Pr[A \wedge B]$	subtract $Pr[A \wedge B]$ so we don't double count
$\leq Pr[A] + Pr[B]$	**Union Bound**

The union bound is one of the most useful facts in computer science. It says that even if there are all sorts of bad things that could happen to you, if each *individual* bad thing has a small enough chance of happening, then overall you're OK. In computer science, there are usually many different things that could cause an algorithm to fail, which are correlated with each other in nasty and poorly-understood ways. But the union bound says we can upper-bound the probability of *any* bad thing happening, by the sum of the probabilities of each *individual* bad thing—completely ignoring the complicated correlations between one bad thing and the next.

$Pr[A \wedge B] = Pr[A]Pr[B]$	if and only if A and B are independent
$Pr[A\|B] = \frac{Pr[A \wedge B]}{Pr[B]}$	definition of conditional probability
$= \frac{Pr[B\|A]Pr[A]}{Pr[B]}$	**Bayes' Rule**

Bayes' Rule comes up often when calculating the probability of some hypothesis being true given the evidence that you've seen. To prove Bayes' Rule, we just need to multiply both sides by $Pr[B]$, to get $Pr[A|B]Pr[B] = Pr[B|A]Pr[A] = Pr[A \wedge B]$.

We'll also need the concepts of *random variables* and *expectations*.

x	a random variable
$Ex[X] = \sum_i Pr[X = i]i$	expectation of X
$Ex[X + Y] = Ex[X] + Ex[Y]$	linearity of expectation
$Ex[XY] = Ex[X]Ex[Y]$	only if X and Y are independent

6.080/6.089 GITCS 1 April 2008

Lecture 14

Lecturer: Scott Aaronson *Scribe: Geoffrey Thomas*

Recap

Randomness can make possible computation tasks that are provably impossible without it. Cryptography is a good example: if the adversary can predict the way you generate your keys, you cannot encrypt messages. Randomness is also good for breaking symmetry and making arbitrary choices.

We also have randomized algorithms. For example, to determine the equality of two polynomials given in nonstandard form, e.g.,

$$(1+x)^2 = 1 + 3x + 3x^2 + x^3$$

pick a few random values and see if they evaluate to the same thing. Since two different polynomials of degree d can only be equal at up to d points per the Fundamental Theorem of Algebra, after evaluating the polynomials at very few values, we can know with high probability whether they are equal.

Can we "derandomize" any randomized algorithm, i.e., can we convert it into a deterministic algorithm with roughly the same efficiency? This (formalized below as **P** versus **BPP**) is one of the central open questions of theoretical computer science.

Useful Probability Formulas

- Union bound: $\mathbf{Pr}[A \vee B] \leq \mathbf{Pr}[A] + \mathbf{Pr}[B]$
- Linearity of expectation: $\mathbf{E}[X+Y] = \mathbf{E}[X] + \mathbf{E}[Y]$ whether or not X and Y are independent
- Markov's inequality:

  $$\mathbf{Pr}[X \geq k\mathbf{E}[X]] \leq \frac{1}{k}$$

  This is true for any distribution X that takes only non-negative values. To prove it: suppose it were false. Then the contribution to $\mathbf{E}[X]$ from X greater than $k\mathbf{E}[X]$ would be so big as to increase the expectation.

Complexity

Can we formalize the concept of a problem that can be solved efficiently with a randomized algorithm? There are several ways to do this.

The complexity class **BPP** (Bounded-Error Probabilistic Polynomial-Time) is the class of languages $L \subseteq \{0,1\}^*$ for which there exists a polynomial time algorithm $M(x,r)$ such that for all inputs x,

- if $x \in L$, then $M(x,r)$ accepts with probability $\geq \frac{2}{3}$

- if $x \notin L$, then $M(x,r)$ accepts with probability $\leq \frac{1}{3}$.

Here r is a random string of polynomial length.

Why $\frac{1}{3}$ and $\frac{2}{3}$? Well, they're just two nice numbers that are separated from each other. If you want more accurate probabilities, you can use *amplification*: run the algorithm many times and take the majority answer. By combining many noisy answers, you can compute a single more accurate answer. So intuitively, it shouldn't matter what probabilities we use to define **BPP**, since we can amplify any success probability to any other with a constant number of repetitions.

But can we be more precise, and bound how many repetitions are needed to amplify a given success probability p to another probability q? This is basically a statistics problem, involving the tails of binomial distributions. Computer scientists like to solve such problems using a rough-and-ready yet versatile tool called the *Chernoff bound*. For n fair coin flips $X_1 \ldots X_n \in \{0,1\}$, let $X = X_1 + \cdots + X_n$. By linearity of expectation, $\mathbf{E}[X] = \frac{n}{2}$. The Chernoff bound says that for all constants $a > 0$,

$$\mathbf{Pr}\left[\left|X - \frac{n}{2}\right| > an\right] \geq c_a^n$$

for some constant $c_a < 1$. In other words, if we repeat our **BPP** algorithm n times, the probability that the majority of the answers will be wrong decreases exponentially in n.

To prove the Chernoff bound, the key idea is to bound the expectation not of X, but of c^X for some constant c:

$$\begin{aligned}
\mathbf{E}\left[c^X\right] &= \mathbf{E}\left[c^{X_1+\cdots+X_n}\right] \\
&= \mathbf{E}\left[c^{X_1}\cdots c^{X_n}\right] \\
&= \mathbf{E}\left[c^{X_1}\right]\cdots\mathbf{E}\left[c^{X_n}\right] \\
&= \left(\frac{1+c}{2}\right)^n
\end{aligned}$$

Here, of course, we've made crucial use of the fact that the X_i's are independent. Now by Markov's inequality,

$$\begin{aligned}
\mathbf{Pr}\left[c^X \geq k\left(\frac{1+c}{2}\right)^n\right] &\leq \frac{1}{k} \\
\mathbf{Pr}\left[X \geq \log_c k + n\log_c\frac{1+c}{2}\right] &\leq \frac{1}{k}
\end{aligned}$$

We can then choose a suitable constant $c > 1$ to optimize the bound; the details get a bit messy. But you can see the basic point: as we increase $d := \log_c k$—which intuitively measures the deviation of X from the mean—the probability of X deviating by that amount decreases *exponentially* with d.

As a side note, can we amplify *any* difference in probabilities—including, say, a difference between $1/2 - 2^{-n}$ and $1/2 + 2^{-n}$? Yes, but in this case you can work out that we'll need an exponential number of repetitions to achieve constant confidence. On the other hand, so long as the inverse of the difference between the two acceptance probabilities is at most a polynomial, we can amplify the difference in polynomial time.

Other Probabilistic Complexity Classes

BPP algorithms are "two-sided tests": they can give errors in both directions. Algorithms like our polynomial-equality test can give false positives but never false negatives, and are therefore

called "one-sided tests." To formalize one-sided tests, we define another complexity class **RP** (Randomized Polynomial-Time). **RP** is the class of all languages $L \subseteq \{0,1\}^*$ for which there exists a polynomial time algorithm $M(x,r)$ such that for all inputs x,

- If $x \in L$, then $M(x,r)$ accepts with probability $\geq \frac{1}{2}$.
- If $x \notin L$, then $M(x,r)$ always rejects regardless of r.

The polynomial-*non*equality problem is in **RP**, or equivalently, the polynomial-equality problem is in co**RP**.

P is in **RP**, co**RP** and **BPP**. **RP** and co**RP** are in **BPP**, because we can just amplify an **RP** algorithm once and reject with probability $0 \leq \frac{1}{3}$ and accept with probability $\frac{3}{4} \geq \frac{2}{3}$. **RP** $\cap$ *co***RP** is also called **ZPP**, for Zero-Error Probabilistic Polynomial-Time. It might seem obvious that **ZPP** = **P**, but this is not yet known to be true. For even given both **RP** and co**RP** algorithms for a problem, you might get unlucky and always get rejections from the **RP** algorithm and acceptances from the co**RP** algorithm.

RP is in **NP**: the polynomial certificate that some $x \in L$ is simply any of the random values r that cause the **RP** algorithm to accept. (Similarly, co**RP** is in co**NP**.) **BPP** is in **PSPACE** because you can try every r and count how many accept and how many reject.

Whether **BPP** is in **NP** is an open question. (Sure, you can generate a polynomial number of "random" numbers r to feed to a deterministic verifier, but how do you convince the verifier that these numbers are in fact random rather than cherry-picked to give the answer you want?) Sipser, Gács, and Lautemann found that $\mathbf{BPP} \subseteq \mathbf{NP}^{\mathbf{NP}}$, placing **BPP** in the so-called *polynomial hierarchy* ($\mathbf{NP}$, $\mathbf{NP}^{\mathbf{NP}}$, $\mathbf{NP}^{\mathbf{NP}^{\mathbf{NP}}}$, ...).

An even more amazing possibility than $\mathbf{BPP} \subseteq \mathbf{NP}$ would be $\mathbf{BPP} = \mathbf{P}$: that is, that every randomized algorithm could be derandomized. Nevertheless, the consensus on this question has changed over time, and today most theoretical computer scientists believe that $\mathbf{BPP} = \mathbf{P}$, even though we seem far from being able to prove it.

Of the several recent pieces of evidence that point toward this conclusion, let us mention just one. Consider the following conjecture:

> There is a problem solvable by a uniform algorithm in 2^n time, which requires c^n circuit size (for some $c > 1$) even if we allow nonuniform algorithms.

This seems like a very reasonable conjecture, since it is not at all clear why nonuniformity (the ability to use a different algorithm for each input size) should help in simulating arbitrary exponential-time Turing machines.

In 1997, Impagliazzo and Wigderson proved that if the above conjecture holds, then $\mathbf{P} = \mathbf{BPP}$. Intuitively, this is because you could use the hard problem from the conjecture to create a *pseudorandom generator*, which would be powerful enough to derandomize any **BPP** algorithm. We'll say more about pseudorandom generators in the next part of the course.

6.080/6.089 GITCS April 8, 2008

Lecture 15

Lecturer: Scott Aaronson *Scribe: Tiffany Wang*

1 Administrivia

Midterms have been graded and the class average was 67. Grades will be normalized so that the average roughly corresponds to a B. The solutions will be posted on the website soon.

Pset4 will be handed out on Thursday.

2 Recap

2.1 Probabilistic Computation

We previously examined probabilistic computation methods and the different probabilistic complexity classes, as seen in Figure 1.

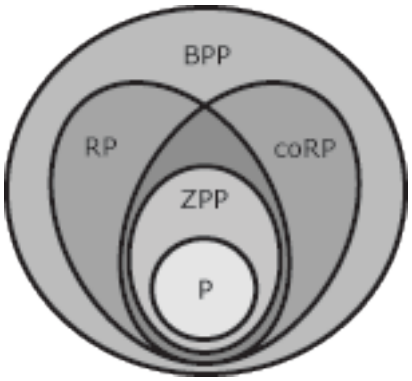


Figure 1. Probabilistic Complexity Classes

P: Polynomial time - Problems that can be solved deterministically in polynomial time.

ZPP: Zero-error Probabilistic Polynomial (Expected Polynomial) time - Problems that can be solved efficiently but with 50% chance that the algorithm does not produce an answer and must be run again. If the algorithm *does* produce an answer it is guaranteed to be correct.

RP: Randomized Polynomial time - Problems for which if the answer is *NO*, the algorithm always outputs *NO*. Otherwise, if the answer is *YES*, the algorithm outputs *YES* at least 50% of the time. Hence there is an asymmetry between *YES* and *NO* outputs.

coRP: Complement of RP - These are problems for which there's a polynomial-time algorithm that always outputs *YES* if the answer is *YES* and outputs *NO* at least 50% of the time if the answer

is *NO*.

BPP: Bounded-error Probabilistic Polynomial time - Problems where if the answer is *YES*, the algorithm accepts with probability $\geq \frac{2}{3}$, and if the answer is *NO*, the algorithm accepts with probability $\leq \frac{1}{3}$.

2.2 Amplification and Chernoff Bound

The question that arises is whether the boundary values $\frac{1}{3}$ and $\frac{2}{3}$ have any particular significance. One of the nice things about using a probabilistic algorithm is that as long as there is a noticeable gap between the probability of accepting if the answer is *YES* and the probability of accepting if the answer is *NO*, that gap can be amplified by repeatedly running the algorithm.

For example, if you have an algorithm that outputs a wrong answer with $\Pr \leq \frac{1}{3}$, then you can repeat the algorithm hundreds of times and just take the majority answer. The probability of obtaining a wrong answer becomes astronomically small (there's a much greater chance of an asteroid destroying your computer).

This notion of amplification can be proven using a tool known as the *Chernoff Bound*. The Chernoff Bound states that given a set of independent events, the number of events that will happen is heavily concentrated about the expected value of the number of occurring events.

So given an algorithm that outputs a wrong answer with $\Pr = \frac{1}{3}$, repeating the algorithm 10,000 times would produce an expected number of $3333.3\ldots$ wrong answers. The number of wrong answers will not be exactly the expected value, but the probability of getting a number far from the expected value (say 5,000) is very small.

2.3 P vs. BPP

There exists a fundamental question as to whether every probabilistic algorithm can be replaced by a deterministic one, or *derandomized*. In terms of complexity, the question is whether P=BPP, which is almost as deep a question as P=NP.

There is currently a very strong belief that derandomization is possible in general, but no one yet knows how to prove it.

3 Derandomization

Although derandomization has yet to be proven in the general case, it *has* been proven for some spectacular special cases: cases where for decades, the only known efficient solutions came from randomized algorithms. Indeed, this has been one of the big success stories in theoretical computer science in the last 10 years.

3.1 AKS Primality Test

In 2002, Agrawal, Kayal, and Saxena of the Indian Institute of Technology Kanpur developed a deterministic polynomial-time algorithm for testing whether an integer is prime or composite, also known as the AKS primality test.

For several decades prior, there existed good algorithms to test primality, but all were probabilistic. The problem was first known to be in the class RP, and then later shown to be in the class ZPP. It was also shown that the problem was in the class P, but only assuming that the Generalized Riemann Hypothesis was true. The problem was also known to be solvable deterministically in $n^{O(loglogn)}$ time (which is *slightly* more than polynomial). Ultimately, it was nice to have the final answer and the discovery was an exciting thing to be alive for in the world of theoretical computer science.

The basic idea behind AKS is related to Pascal's Triangle. As seen in Figure 2, in every prime-numbered row, the numbers in Pascal's Triangle are all a multiple of the row number. On the other hand, in every composite-numbered row, the numbers are *not* all multiples of the row number.

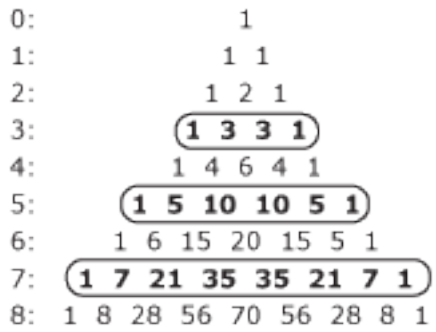


Figure 2. Pascal's Triangle and Prime Numbers

So to test the primality of an integer N, can we just check whether or not all the numbers in the N^{th} row of Pascal's Triangle are multiples of N? The problem is that there are exponentially many numbers to check, and checking all of them would be no more efficient than trial division.

Looking at the expression $(x+a)^N$, which has coefficients determined by the N^{th} row of Pascal's Triangle, AKS noticed that the relationship $(x+a)^N = x^N + a^N \mod N$ holds if and only if N is prime. This is because if N is prime, then all the "middle" coefficients will be divisible by N (and therefore disappear when we reduce mod N), while if N is composite then some middle coefficients will not be divisible by N. What this means is that the primality testing problem can be mapped to an instance of the polynomial identity testing problem: given two algebraic formulas, decide whether or not they represent the same polynomial.

In order to determine whether $(x+a)^N = x^N + a^N \mod N$, one approach would be to plug in many random values of a and see if we get the same result each time. However, since the number of terms would still be exponential, we need to evaluate the expression not only mod N, but also mod a random polynomial:

$$(x+a)^N = x^N + a^N \mod N, x^r - 1.$$

It turns out that this solution method works; on the other hand, it still depends on the use of randomness (the thing we're trying to eliminate).

The tour de force of the AKS paper was to show that if N is composite, then it is only necessary to try some small number of deterministically-chosen values of a and r until a pair is found such that the equation is not satisfied. This immediately leads to a method for distinguishing prime numbers from composite ones in deterministic polynomial time.

3.2 Trapped in a Maze

Problem: Given a maze (an undirected graph) with a given start vertex and end vertex, is the end vertex reachable or not?

Proposed solution from the floor: Depth-first search.

In a maze, this is the equivalent of wandering around the maze and leaving bread crumbs to mark paths that have already been explored. This solution runs in polynomial time, but the problem is that it requires breadcrumbs, or translated into complexity terms, a large amount of memory. The hope would be to solve the undirected connectivity problem in LOGSPACE: that is, to find a way out of the maze while remembering only $O(\log n)$ bits at any one time. (Why $O(\log n)$? That's the amount of information needed even to write down where you are; thus, it's essentially the best you can hope for.)

Proposed solution from the floor: Follow the right wall.

The trouble is that, if you were always following the right wall, it would be simple to create a maze that placed you in an infinite loop.

Simple-minded solution: Just wander around the maze randomly.

Now we're talking! Admittedly, in a *directed* graph it could take an exponential time for a random walk to reach the end vertex. For example, at each intersection of the graph shown in Figure 3, you advance forward with Pr=$\frac{1}{2}$ and return to the starting point with Pr=$\frac{1}{2}$. The chance that you make n consecutive correct choices to advance all the way to the end is exponentially small, so it will take exponential time to reach the end vertex.

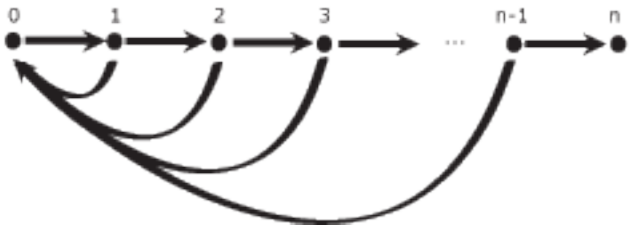


Figure 3. Exponential Time Directed Graph

In 1979, Aleliunas et al. showed that randomly wandering through an undirected graph will get you out with high probability after $O(n^3)$ steps. After $O(n^3)$ steps, with high probability you will

have visited every vertex, regardless of the structure of the graph.

However, this still leaves the question of whether there's a *deterministic* algorithm to get out of a maze using only $O(\log n)$ bits of memory. Finally, in 2005, Omer Reingold proved that by making *pseudorandom* path selections based on a somewhat complicated set of rules, the maze problem can be solved deterministically in LOGSPACE. At each step, the rule is a function of the outcome of the previous application of the rules.

4 New Unit: Cryptography

4.1 History

Cryptography is a 3,000-year old black art that has been completely transformed over the last few decades by ideas from theoretical computer science. Cryptography is perhaps the best example of a field in which the concepts of theoretical computer science have real practical applications: problems are designed to be hard, the worst case assumptions are the right assumptions, and computationally intractable problems are there because we put them there.

For more on the history of cryptography, a great reference is David Kahn's *The Codebreakers*, which was written before people even knew about the biggest cryptographic story of all: the breaking of the German naval code in World War II, by Alan Turing and others.

4.2 Caesar Cipher

One of the oldest cryptographic codes used in history is the "Caesar cipher." In this cryptosystem, a plaintext message is converted into ciphertext by simply adding 3 to each letter of the message, wrapping around to A after you reach Z. Thus A becomes D, Z becomes C, and DEMOCRITUS becomes GHPRFULWXV.

Clearly this encryption system can easily be broken by anyone who can subtract mod 26. As an amusing side note, just a couple years ago, the head of the Sicilian mafia was finally caught after 40 years because he was using the Caesar cipher to send messages to his subordinates.

4.3 Substitution Cipher

A slightly more complicated cryptographic encoding is to scramble the letters of a message according to a random rule which permutes all the letters of the alphabet. For example, substituting every A with an X and every S with a V.

This encoding can also be easily broken by performing a frequency analysis of the letters appearing in the ciphertext.

4.4 One-Time Pad

It was not until the 1920's that a "serious" cryptosystem was devised. Gilbert Sandford Vernam, an American businessman, proposed what is known today as the one-time pad.

Under the one-time pad cryptosystem, the plaintext message is represented by a binary string M which is them XOR-ed with a random binary key, K, of the same length. As seen in Figure 4,

the ciphertext C is equal to the bitwise sum of M and K, mod 2.

Figure 4. One-time Pad Encryption

Assuming that the recipient is a trusted party who shares the knowledge of the key, the ciphertext can be decrypted by performing another XOR operation: $C \oplus K = M \oplus K \oplus K = M$. See Figure 5.

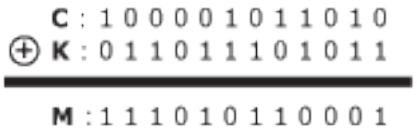


Figure 5. One-Time Pad Decryption

To an eavesdropper who does not have knowledge of the key, the ciphertext appears to be nonsense since XOR-ing any string of bits with a random string just produces another random string. There is no way to guess what the ciphertext may be encoding because any binary key could have been used.

As a result of this, the one-time pad is a provably unbreakable cryptographic encoding, but only if used correctly. The problem with using the one-time pad is that it literally is a "one-time" encryption. If the same key is ever used to encrypt more than one message, then the cryptosystem is no longer secure. For example, if we sent another message M_2 encrypted with the same key K to produce C_2, the eavesdropper could obtain a combination of the messages: $C_1 \oplus C_2 M_1 \oplus K \oplus M_2 \oplus K = M_1 \oplus M_2$. If the eavesdropper had any idea of what either of the messages may have contained, the eavesdropper could learn about the other plaintext message, and indeed obtain the key K.

As a concrete example, Soviet spies during the Cold War used the one-time pad to encrypt their messages and occasionally slipped up and re-used keys. As a result, the NSA, through its VENONA project, was able to decipher some of the ciphertext and even gather enough information to catch Julius and Ethel Rosenberg.

4.5 Shannon's Theorem

As we saw, the one-time pad has the severe shortcoming that the number of messages that can be encrypted is limited by the amount of key available.

Is it possible to have a cryptographic code which is unbreakable (in the same absolute sense that the one-time pad is unbreakable), yet uses a key that is much smaller than the message?

In the 1940s, Claude Shannon proved that a perfectly secure cryptographic code requires the encryption key to be at least as long as the message that is sent.

Proof: Given an encryption function: $e_k : \{0,1\}^n_{plaintext} \rightarrow \{0,1\}^m_{ciphertext}$.

For all keys k, e_k must be an injective function (provide a one-to-one mapping between plaintext and ciphertext). Every plaintext must map to a different ciphertext, otherwise there would be no way of decrypting the message.

This immediately implies that for a given ciphertext, C, that was encrypted with a key of r bits, the number of possible plaintexts that could have produced C is at most 2^r (the number of possible keys). If $r < n$, then the number of possible plaintexts that could have generated C is smaller than the total number of possible plaintext messages. So if the adversary had unlimited computational power, the adversary could try all possible values of the key and rule out all plaintexts that could not have encrypted to C. The adversary would thus have learned something about the plaintext, making the cryptosystem insecure. Therefore the encryption key must be at least as long as the message for a perfectly secure cryptosystem.

The key loophole in Shannon's argument is the assumption that the adversary has unlimited computational power. For a practical cryptosystem, we can exploit computational complexity theory, and in particular the assumption that the adversary is a polynomial-time Turning machine that does not have unlimited computational power. More on this next time...

6.080/6.089 GITCS	April 4th, 2008
Lecture 16	
Lecturer: Scott Aaronson	*Scribe: Jason Furtado*

Private-Key Cryptography

1 Recap

1.1 Derandomization

In the last six years, there have been some spectacular discoveries of deterministic algorithms, for problems for which the only similarly-efficient solutions that were known previously required randomness. The two most famous examples are

- the Agrawal-Kayal-Saxena (AKS) algorithm for determining if a number is prime or composite in deterministic polynomial time, and
- the algorithm of Reingold for getting out of a maze (that is, solving the undirected s-t connectivity problem) in deterministic LOGSPACE.

Beyond these specific examples, mounting evidence has convinced almost all theoretical computer scientists of the following
Conjecture: Every randomized algorithm can be simulated by a deterministic algorithm with at most polynomial slowdown. Formally, $P = BPP$.

1.2 Cryptographic Codes

1.2.1 Caesar Cipher

In this method, a plaintext message is converted to a ciphertext by simply adding 3 to each letter, wrapping around to A after you reach Z. This method is breakable by hand.

1.2.2 One-Time Pad

The "one-time pad" uses a random key that must be as long as the message we want to encrypt. The exclusive-or operation is performed on each bit of the message and key ($Msg \oplus Key = EncryptedMsg$) to end up with an encrypted message. The encrypted message can be decrypted by performing the same operation on the encrypted message and the key to retrieve the message ($EncryptedMsg \oplus Key = Msg$). An adversary that intercepts the encrypted message will be unable to decrypt it as long as the key is truly random.

The one-time pad was the first example of a cryptographic code that can *proven* to be secure, even if the adversary has all the computation time in the universe.

The main drawback of this method is that keys can never be reused, and the key must be the same size as the message to encrypt. If you were to use the same key twice, an eavesdropper could compute $(Enc \oplus Msg1) \oplus (Enc \oplus Msg2) = Msg1 \oplus Msg2$. This would leak information about $Msg1$ and $Msg2$.

Example. Suppose $Msg1$ and $Msg2$ were bitmaps and $Msg1$ had sections that were all the same (say, a plain white background). For simplicity, assume $Msg1$ is all zeros at bit positions 251-855. Then $Msg2$ will show through in those bit positions. During the Cold War, spies were actually caught using this sort of technique.

Also, note that the sender and the recipient must agree on the key in advance. Having shared random keys available for every possible message size is often not practical. Can we create encryption methods that are secure with smaller keys, by assuming our adversary doesn't have unlimited computing power (say, is restricted to running polynomial-time algorithms)?

2 Pseudorandom Generators

A pseudorandom generator (PRG) is a function that takes as input a short, truly random string (called the *seed*) and produces as output a long, seemingly random string.

2.1 Seed Generation

A seed is a "truly" random string used as input to a PRG. How do you get truly random numbers? Some seeds used are generated from the system time, typing on a keyboard randomly, the last digits of stock prices, or mouse movements. There are subtle correlations in these sources so they aren't completely random, but there are ways of extracting randomness from weak random sources. For example, according to some powerful recent results, nearly "pure" randomness can often be extracted from two or more weak random sources that are assumed to be uncorrelated with each other.

How do you prove that a sequence of numbers is random? Well, it's much easier to give overwhelming evidence that a sequence is *not* random! In general, one does this by finding a *pattern* in the sequence, i.e. a computable description with fewer bits than the sequence itself. (In other words, by showing that the sequence has less-than-maximal Kolmogorov complexity.)

In this lecture, we'll simply assume that we have a short random seed, and consider the problem of how to expand it into a long "random-looking" sequence.

2.2 How to Expand Random Numbers

2.2.1 Linear-Congruential Generator

In most programming languages, if you ask for random numbers what you get will be something like the following (starting from integers a, b, and N):

$x_1 = ax_0 + b \bmod N$ $x_2 = ax_1 + b \bmod N$

...

$x_n = ax_{n-1} + b \bmod N$

This process is good enough for many non-cryptographic applications, but an adversary could easily distinguish the sequence $x_0, x_1, \ldots$ from random by solving a small system of equations mod N. For cryptography applications, it must not be possible for an adversary to figure out a pattern in the output of the generator in polynomial time. Otherwise, the system is not secure.

2.2.2 Cryptographic Pseudorandom Generator (CPRG)

Definition: (Yao 1982)

A cryptographic pseudorandom generator (CPRG) is a function $f : \{0,1\}^n \to \{0,1\}^{n+1}$ such that:

1. f is computable in polynomial time.
2. For all polynomial-time algorithms A (adversaries),

$$\left| Pr_{y \in \{0,1\}^{n+1}}[A(y) \text{ accepts}] - Pr_{x \in \{0,1\}^n}[A(f(x)) \text{ accepts}] \right|,$$

the "advantage", is negligibly small.

In other words, the output of the CPRG must "look random" to any polynomial time algorithm.

In the above definition, "negligibly small" means less than $1/p(n)$ for all polynomials p. This is a minimal requirement, since if the advantage of the adversary were $1/p(n)$, then in polynomial time the adversary could amplify the advantage to a constant (see Lecture 14). Of course it's even better if the adversary's advantage decreases exponentially.

The definition above only requires f to stretch an n-bit seed into a random-looking $(n+1)$-bit string. Could we use such an f to stretch an n-bit seed into, say, a random-looking n^2-bit string? It turns out that the answer is yes; basically we feed f its own output n^2 times. (See Lecture 17 for more details.)

2.2.3 Enhanced One-Time Pad

Using such a CPRG $f : \{0,1\}^n \to \{0,1\}^{p(n)}$, we can make our one-time pad work for messages polynomially larger than the original key s:

$k = f(s)$
$e = x \oplus k$
$x = e \oplus k$

Claim. With this construction, no polynomial-time adversary can recover the plaintext from the ciphertext.

Proof. Assume for simplicity that the plaintext consists of just a single repeated random bit (i.e., is either $00\cdots0$ or $11\cdots1$, both with equal probability). Also, suppose by way of contradiction that a polynomial-time adversary could guess the plaintext given the ciphertext, with probability non-negligibly greater than 1/2. We know that if the key k were truly random, then the adversary would *not* be able to guess the plaintext with probability greater than 1/2. But this means that the adversary must be distinguishing a pseudorandom key from a truly random key with non-negligible bias – thereby violating the assumption that f was a CPRG!

The system above is not yet a secure cryptographic system (we still need to deal with the issue of repeated keys, etc.), but hopefully this gives some idea of how CPRG's can be used to construct computationally-secure cryptosystems.

3 Blum-Blum-Shub CPRG

The Blum-Blum-Shub (BBS) CPRG is proven to breakable if and only if a fast (polynomial-time) algorithm exists for factoring. With this generator, the seed consists of integers x and $N = pq$, where p, q are large primes. The output consists of the last bit of $x^2 \bmod N$, the last bit of $(x^2)^2 \bmod N$, the last bit of $((x^2)^2)^2 \bmod N$, etc.

4 $P \neq NP$-based CPRG

Ideally, we would like to construct a CPRG or cryptosystem whose security was based on an NP-complete problem. Unfortunately, NP-complete problems are always about the worst case. In cryptography, this would translate to a statement like "there *exists* a message that's hard to decode", which is not a good guarantee for a cryptographic system! A message should be hard to decrypt with overwhelming probability. Despite decades of effort, no way has yet been discovered to relate worst case to average case for NP-complete problems. And this is why, if we want computationally-secure cryptosystems, we need to make stronger assumptions than $P \neq NP$.

5 One-Way Functions

The existence of one-way functions (OWF's) is such a stronger assumption.

Definition: A one-way function is a function $f : \{0,1\}^n \to \{0,1\}^{p(n)}$ such that:

1. f is computable in polynomial time.
2. For all polynomial-time algorithmss A,

$$Pr_{x \in \{0,1\}^n}[f(A(f(x))) = f(x)]$$

is negligible.

In other words, a polynomial-time algorithm should only be able to invert f with negligible probability. The reason we don't require $A(f(x)) = x$ is to rule out trivial "one-way functions" like $f(x) = 1$.

$CPRG \Rightarrow OWF$?

True. Any CPRG is also an OWF by the following argument: if given the output of a pseudorandom generator we could efficiently find the seed, then we'd be distinguishing the output from true randomness – thereby violating the assumption that we had a CPRG in the first place.

$OWF \Longleftrightarrow CPRG$?

Also true, but this direction took over 20 years to prove! In 1997, Håstad, Impagliazzo, Levin, and Luby showed how to construct a pseudorandom generator from any one-way function, by a complicated reduction with many steps.

6.080/6.089 GITCS Apr 15, 2008

Lecture 17

Lecturer: Scott Aaronson *Scribe: Adam Rogal*

1 Recap

1.1 Pseudorandom Generators

We will begin with a recap of pseudorandom generators (PRGs). As we discussed before a pseudorandom generator is a function that takes as input a short truly random input string and produces an output of a seemingly random string. Formally, a PRG is a polytime-computable function $f : \{0,1\}^n \rightarrow \{0,1\}^{n+1}$ such that for all deterministic polynomial-time algorithms A,

$$\left| \Pr_{y \in \{0,1\}^{n+1}} [A(y) \text{ accepts}] - \Pr_{x \in \{0,1\}^n} [A(f(x)) \text{ accepts}] \right|$$

is negligible.

Given a PRG that stretches n bits to $n+1$ bits, we can create a PRG that stretches n bits to $p(n)$ bits for any polynomial p. To do so, we repeatedly break off a single bit of the PRG's output, and feeding the remaining n bits back into the PRG to get another $n+1$ pseudorandom bits. This process is shown in figure 1. To prove that it works, one needs to show that, could we distinguish the $p(n)$-bit output from random, we could *also* distinguish the original $(n+1)$-bit output from random, thereby violating the assumption that we started with a PRG. Formalizing this intuition is somewhat tricky and will not be done here.

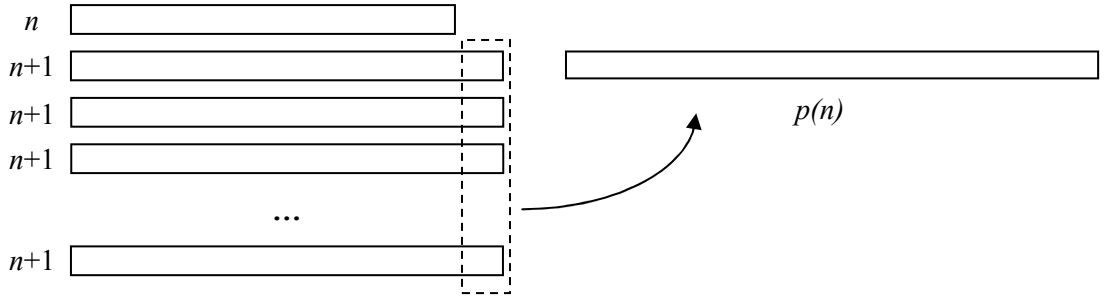


Figure 1: A seemingly random string of size $p(n)$ is generated from an n-bit seed using the feed and repeat method.

1.2 Cryptographic Codes

Using pseudorandom generators, it's possible to create secure cryptographic codes with small key sizes. The details of this are complicated if you want to protect against realistic

attacks (for example, so-called *chosen-message attacks*). But at the simplest level, the intuition is the following: we should be able to simulate a one-time pad (which is provably unbreakable when used correctly) by (1) taking a small random key, (2) stretching it to a longer key using a PRG, and then (3) treating that longer key as the one-time pad. If a polynomial-time adversary could break such a system, that would mean that the adversary was distinguishing the PRG's output from a truly random string, contrary to assumption.

1.3 One-Way Functions

In addition to PRGs, we'll be interested in a closely-related class of objects called OWFs, or *one-way functions*. An OWF is a polytime-computable function $f : \{0,1\}^n \rightarrow \{0,1\}^{p(n)}$ such that for all deterministic polynomial-time algorithms A,

$$\Pr_{x \in \{0,1\}^n}[f(A(f(x))) = f(x)]$$

is negligible.

Or in plainer language, an OWF is a function that's easy to compute but hard to invert.

1.4 Yao's Minimax Principle

As a side note, you might wonder why we assumed the adversary A was determinisic rather than probabilistic. The answer is that it makes no difference! If you're playing rock-paper-scissors, and you *know* the probability distribution over your opponent's move, then there's always some *fixed* move you can make that does as well as any randomized strategy. Similarly, one you fix the probability distribution over inputs – as we do with PRGs and OWFs – there's always a deterministic algorithm whose success probability is as large as any randomized algorithm's. This is (the easy part of) *Yao's Minimax Principle*, one of the most useful facts in theoretical computer science.

1.5 Relation Between PRGs and OWFs

Claim: Every PRG is also an OWF. Why? Because if we could invert a PRG, then it wouldn't be pseudorandom! We'd learn that there was *some* seed that generated the output string, which would be true for a random string with probability at most 1/2.

In 1997, Håstad et al. proved the opposite direction: if OWFs exist then so do PRGs. This direction was much, *much* harder (note that transformations of the OWF are necessary, since it's easy to give examples of OWFs that are not PRGs). Because of this result, we now know that the possibility of private-key encryption with small keys is essentially equivalent to the existence of OWFs.

2 Public-Key Cryptography

2.1 Abstract Problem

Suppose Alice is trying to send Bob a package, so that no third party can open it *en route*. We'll assume that boxes can be "locked," in such a way that you can only open a box if you have the right key.

If Alice and Bob share duplicates of the same key, then this problem is trivial: Alice locks the box with her key and sends it to Bob, who then opens it with his key. But what

if Alice and Bob *don't* share a key? Obviously, we don't want Alice to send the package in a locked box, and the key that opens the lock in an unlocked box! We seem to be faced with an infinite regress.

Fortunately, there's a simple solution. As shown in Figure 2, first Alice puts the package in a box, locks it, and sends it to Bob. Then Bob puts a *second* lock on the box and sends it back to Alice. Then Alice removes her lock and sends the box back to Bob. Finally Bob removes his lock and opens the box.

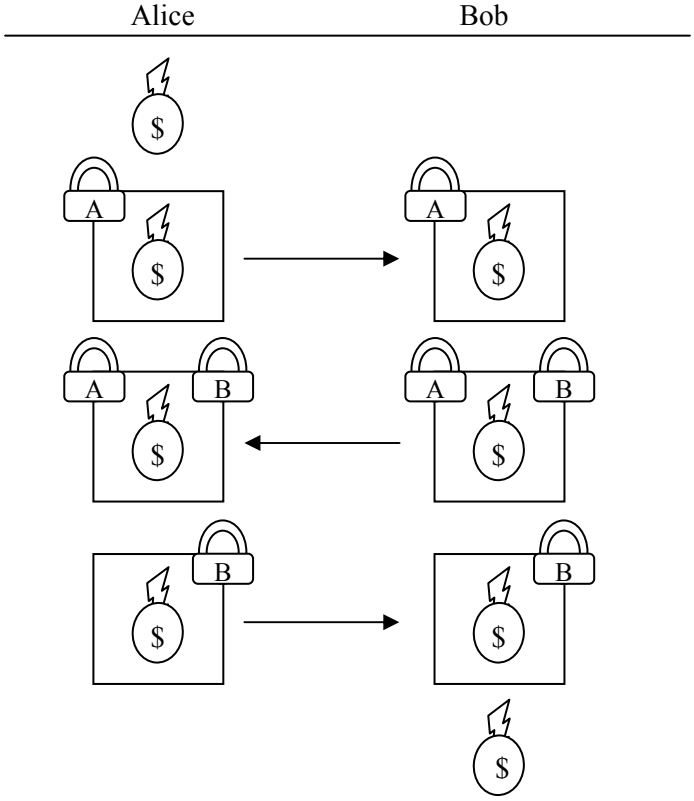
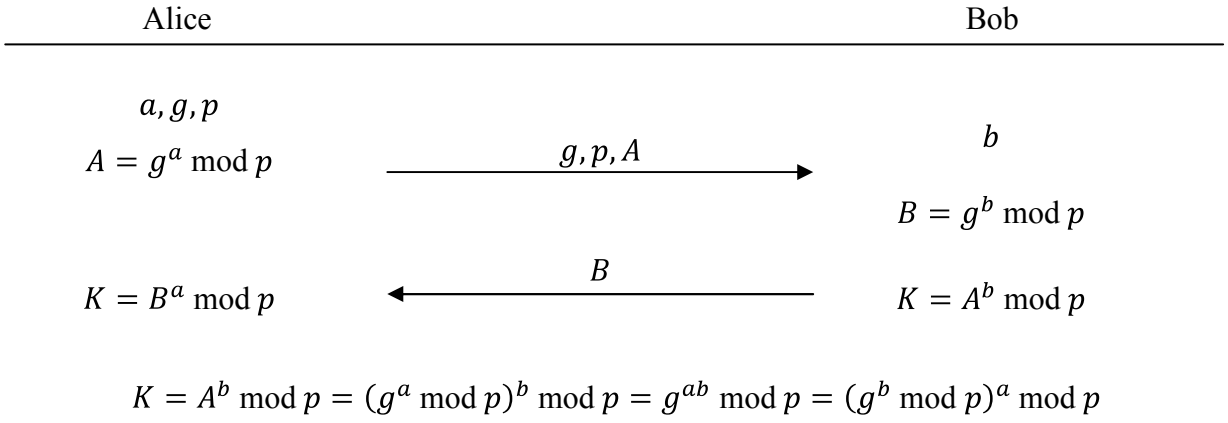


Figure 2: The smarter approach has Alice and Bob passing the package with at least one form of protection at all times. This ensures that only Alice and Bob will be able to open the package.

2.2 Diffie-Hellman

How could we simulate the above protocol, in the situation where Alice and Bob are sending bits of information rather than physical boxes? The first serious proposal in the open literature for how to do this was given by Diffie and Hellman in 1976.



$$K = A^b \bmod p = (g^a \bmod p)^b \bmod p = g^{ab} \bmod p = (g^b \bmod p)^a \bmod p$$

Figure 3: The Diffie-Hellman protocol for creating a shared secret key K between Alice and Bob.

The process, shown in figure 3, begins by Alice choosing a large prime number, p, a base, g, and a secret integer, a. Alice will calculate a public number $A = g^a \bmod p$. She will then send p, g, and A to Bob. Bob will then pick his own secret b, and send $B = g^b \bmod p$ back to Alice. Finally, Alice calculates the secret key K as $K = B^a \bmod p$, and Bob calculates it as $K = A^b \bmod p$. They both now have the same key with which to encode messages to each other.

We've seen that Diffie-Hellman is a simple way to exchange a key; yet, but it's a bit cumbersome in practice. What we'd really like is a public-key protocol that involves fewer messages back and forth—and in which only one person, not two, needs to create public and private keys.

3 RSA

RSA (together with its variants) is probably the most widely-used cryptographic protocol in modern electronic commerce. Much like Diffie-Hellman, it is built on modular arithmetic.

3.1 How It Works

As shown in Figure 4, the process is more direct than with Diffie-Hellman. Let's suppose you want to send your credit card number to Amazon.com. Then in the simplest variant, Amazon picks two large prime numbers, p and q, with the condition that neither $p-1$ nor $q-1$ is divisible by 3. It then multiplies them together to get $N = pq$ and sends N to you. On retrieving N, you calculate $y = x^3 \bmod N$, where x is your credit card number, and send y back to Amazon.

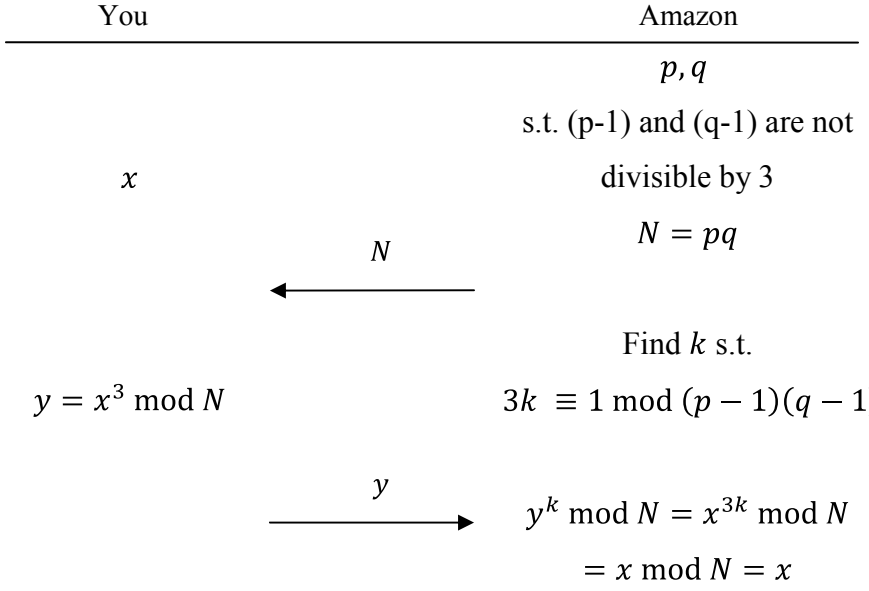


Figure 4: RSA uses modular arithmetic to retrieve x efficiently from an encoded message. An eavesdropper will only see N and $x^3 \bmod N$.

Amazon then faces the problem of how to recover x given y. In other words, how does it take a *cube root* modulo N? Fortunately, it can do that given using its knowledge of the prime factors p and q, together with the following formula discovered by the mathematician Leonhard Euler in the 1700's:

$$x^{(p-1)(q-1)} = 1 \bmod N$$

(Why is this formula true? Basically, because $(p-1)(q-1)$ is the order of the *multiplicative group* mod N, consisting of all numbers from 1 to N that are relatively prime to N. We won't give a more detailed proof here.)

Euler's formula implies that, if Amazon can only find an integer k such that $3k = 1 \bmod (p-1)(q-1)$, then

$$y^k = x^{3k} = x^{c(p-1)(q-1)+1} = x \bmod N,$$

where c is some integer. But the fact that neither $p-1$ nor $q-1$ is divisible by 3 implies that such an integer k must exist – and furthermore k can be found in polynomial time given p and q, for example by using Euclid's algorithm. And once Amazon has k, it can also compute $y^k \bmod N = x$ in polynomial time using repeated squaring. It can thereby recover your credit card number x, as desired.

The obvious question is, how secure is this system? Well, any adversary who could factor N into pq could obviously decrypt the message x, by using the same algorithm that Amazon itself uses. Hence this whole system is predicated on the presumed intractability of factoring large integers (an assumption that would be violated if, for example, we built large-scale quantum computers). And of course, any proof that factoring is hard would also prove $P \neq NP$.

In the other direction, you might wonder: *assuming* the factoring problem is hard, is RSA secure? Alas, that's been an open problem for 30 years! Yet despite its uncertain theoretical foundations, the RSA system has withstood all attacks thus far (unlike many other proposed cryptosystems), and today millions of people rely on it.

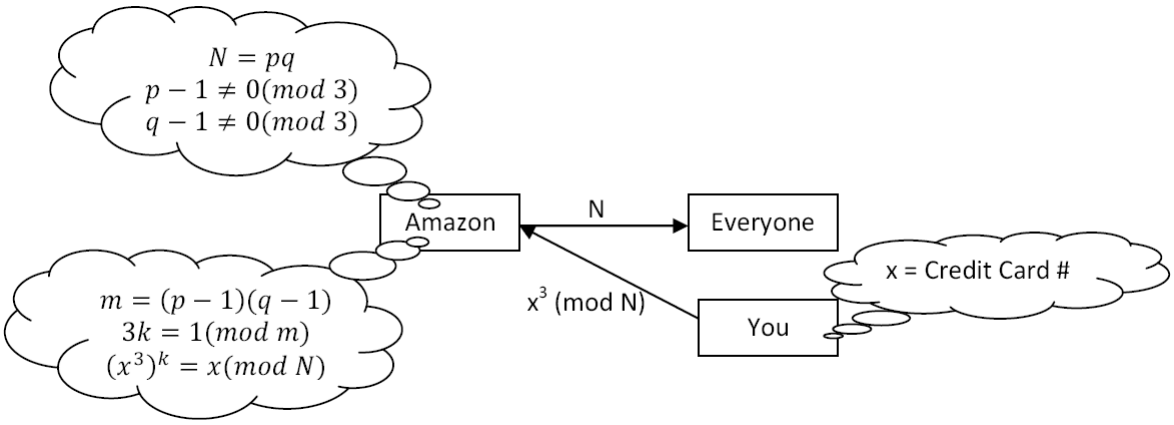
6.080/6.089 GITCS April 17, 2008

Lecture 18

Lecturer: Scott Aaronson *Scribe: Hristo Paskov*

1 Recap

Last time we talked about public key cryptography which falls in the realm of accomplishing bizarre social goals using number theory. Our first example of a public-key cryptosystem, in which two people exchanging messages did not have to meet beforehand, was Diffie-Hellman. We then talked about the RSA cryptosystem, which is probably the most widely used today. Here are the basics of how it works:



The first step is taken by the recipient of the message, by generating two giant prime numbers p and q and setting $N = pq$. Note that p and q must be chosen such that $p-1$ and $q-1$ are not divisible by 3. The recipient keeps p and q a closely-guarded secret, but gives out N to anyone who asks. Suppose a sender has a secret message x that she wants to send to the recipient. The sender calculates x^3 mod N and sends it to the recipient. Now it's the recipient's turn to recover the message. He can use some number theory together with the fact that he knows p and q, the factors of N. The recipient first finds an integer k such that $3k = 1 \bmod (p-1)(q-1)$, which can be done in polynomial time via Euclid's algorithm, and then takes $(x^3)^k \bmod N = x^{3k} \bmod N = x$. The exponentiation can be done in polynomial time by using the trick of repeated squaring. Voila!

When you look at this procedure, you might wonder why are we cubing as opposed to raising to another power; is there anything special about 3? As it turns out, 3 is just the first choice that's convenient. Squaring would lead to a ciphertext that had multiple decryptions (corresponding to the multiple square roots mod N), while we want the decryption to be unique. Indeed, if we wanted the square root to be unique, then we'd need $p-1$ and $q-1$ to not divisible by 2, which is a problem since p and q (being large prime numbers) are odd!

You could, however, raise to a power higher than 3, and in fact that's what people usually do. If the other components of the cryptosystem—such as the padding out of messages with random garbage—aren't implemented properly, then there's a class of attacks called "small-exponent attacks" which break RSA with small exponents though not with large ones. On the other hand, if everything else is implemented properly, then as far as we know x^3 mod N is already secure.

(Just like in biology, everything in cryptography is always more complicated than what you said, *whatever* you said. In particular, as soon as you leave a clean mathematical model and enter the real world, where code is buggy, hardware inadvertently leaks information, etc. etc., there's always further scope for paranoia. And cryptographers are extremely paranoid people.)

As mentioned in the last lecture, we know that a fast factoring algorithm would lead to a break of RSA. However, we don't know the opposite direction: could you break RSA without factoring? In 1979, Rabin showed that if you squared the plaintext x instead of cubing it, then recovering x *would* be as hard as factoring. But as discussed earlier, in that case you'd lose the property that every decryption is unique. This problem is what's prevented widespread adoption of Rabin's variant of RSA.

2 Trapdoor One-Way Functions

The operation $x^3 \bmod N$ in RSA is an example of what's called *trapdoor one way function*, or TDOWF. A trapdoor one-way function is a one-way function with the additional property that if you know some secret "trapdoor" information then you can efficiently invert it. So for example, the function $f(x) = x^3 \bmod N$ is believed to be a one-way function, yet is easy to invert by someone who knows the prime factors of N.

2.1 Different Classes of TDOWF's

Question from the floor: Are there any candidate TDOWF's *not* based on modular arithmetic (like RSA is)?

Answer: One class that's been the subject of much recent research is based on lattices. (Strictly speaking, the objects in this class are not TDOWF's, but something called *lossy TDOWF's*, but they still suffice for public-key encryption.) Part of the motivation for studying this class is that the cryptosystems based on modular arithmetic could all be broken by a quantum computer, if we had one. By contrast, even with a quantum computer we don't yet know how to break lattice cryptosystems. Right now, however, lattice cryptosystems are not used much. Part of the problem is that, while the message and key lengths are polynomial in n, there are large polynomial blowups. Thus, these cryptosystems aren't considered to be as practical as RSA. On the other hand, in recent years people have come up with better constructions, so it's becoming more practical.

There's also a third class of public-key cryptosystems based on elliptic curves, and elliptic-curve cryptography *is* currently practical. Like RSA, elliptic-curve cryptography is based on abelian groups, and like RSA it can be broken by a quantum computer. However, elliptic-curve cryptography has certain nice properties that are not known to be shared by RSA.

In summary, we only know of a few classes of candidate TDOWF's, and all of them are based on *some* sort of interesting math. When you ask for a trapdoor that makes your one-way function easy to invert again, you're really asking for something mathematically special. It almost seems like an accident that plausible candidates exist at all! By contrast, if you just want an ordinary, *non*-trapdoor OWF, then as far as we know, all sorts of "generic" computational processes that scramble up the input will work.

3 NP-completeness and Cryptography

An open problem for decades has been to base cryptography on an NP-complete problem. There are strong heuristic arguments, however, that suggest that if this is possible, it'll require very

different ideas from what we know today. One reason (discussed last time) is that cryptography requires average-case hardness rather than worst-case. A second reason is that many problems in cryptography actually belong to the class $NP \cap coNP$. For example, given an encrypted message, we could ask if the first bit of the plaintext is 1. If it is, then a short proof is to decrypt the message. If it's not, then a short proof is *also* to decrypt the message. However, problems in $NP \cap coNP$ can't be NP-complete under the most common reductions unless $NP = coNP$.

3.1 Impagliazzo's Five Worlds

A famous paper by Impagliazzo discusses five possible worlds of computational complexity and cryptography, corresponding to five different assumptions you can make. You don't need to remember the names of the worlds, but I thought you might enjoy seeing them.

1. Algorithmica - $P = NP$ or at the least fast probabalistic algorithms exist to solve all NP problems.
2. Heuristica - $P \neq NP$, but while NP problems are hard in the worst case, they are easy on average.
3. Pessiland - NP-complete problems are hard on average *but* one-way functions don't exist, hence no cryptography
4. Minicrypt - One-way functions exist (hence private-key cryptography, pseudorandom number generators, etc.), but there's no public-key cryptography
5. Cryptomania - Public-key cryptography exists; there are TDOWF's

The reigning belief is that we live in Cryptomania, or at the very least in Minicrypt.

4 Fun with Encryption

4.1 Message Authentication

Besides encrypting a message, can you prove that a message actually came from you? Think back to the one-time pad, the first decent cryptosystem we saw. On its face, the one-time pad seems to provide authentication as a side benefit. Recall that this system involves you and a friend sharing a secret key k, you transmitting a message x securely by sending $y = x \oplus k$, and your friend decoding the message by computing $x = y \oplus k$. Your friend might reason as follows: if it was anyone other than you who sent the message, then why would $y \oplus k$ yield an intelligible message as opposed to gobbledygook?

There are some holes in this argument (see if you can spot any), but the basic idea is sound. However, to accomplish this sort of authentication, you do need the other person to share a secret with you, in this case the key. It's like a secret handshake of fraternity brothers.

Going with the analogy of private vs. public key cryptography, we can ask whether there's such a thing public-key authentication. That is, if a person trusts that some public key N came from you, he or she should be able to trust any further message that you send as *also* coming from you. As a side benefit, RSA gives you this ability to authenticate yourself, but we won't go into the details.

4.2 Computer Scientists and Dating

Once you have cryptographic primitives like the RSA function, there are all sorts of games you can play. Take, for instance, the problem of Alice and Bob wanting to find out if they're both interested in dating each other. Being shy computer scientists, however, they should only find out they like each other if they're both interested; if one of them is *not* interested, then that one shouldn't be able to find out the other is interested.

An obvious solution (sometimes used in practice) would be to bring in a trusted mutual friend, Carl, but then Alice and Bob wold have to trust Carl not to spill the beans. Apparently there are websites out there that give this sort of functionality. However, ideally we would like to not have to rely on a third party.

Suggestion from the floor: Alice and Bob could face each other with their eyes closed, and each open their eyes only if they're interested.

Response: If neither one is interested, then there seems to be a termination problem! Also, we'd like a protocol that doesn't require physical proximity – remember that they're shy computer scientists!

4.2.1 The Dating Protocol

So let's suppose Alice and Bob are at their computers, just sending messages back and forth. If we make no assumptions about computational complexity, then the dating task is clearly impossible. Why? Intuitively it's "obvious": because eventually one of them will have to say something, without yet knowing whether his or her interest will be reciprocated or not! And indeed one can make this intuitive argument more formal.

So we're going to need a cryptographic assumption. In particular, let's assume RSA is secure. Let's also assume, for the time being, that Alice and Bob are what the cryptographers call *honest but curious.* In other words, we'll assume that they can both be trusted to follow the protocol correctly, but that they'll also try to gain as much information as possible from whatever messages they see. Later we'll see how to remove the honest-but-curious assumption, to get a protocol that's valid even if one player is trying to cheat.

Before we give the protocol, three further remarks might be in order. First, the very fact that Alice and Bob are carrying out a dating protocol in the first place, might be seen as *prima facie* evidence that they're interested! So you should imagine, if it helps, that Alice and Bob are at a singles party where *every* pair of people has to carry out the protocol. Second, it's an unavoidable feature of any protocol that if one player is interested and the other one isn't, then the one who's interested will learn that the other one isn't. (Why?) Third, it's also unavoidable that one player could *pretend* to be interested, and then after learning of the other player's interest, say "ha ha! I wasn't serious. Just wanted to know if you were interested."

In other words, we can't ask cryptography to solve the problem of heartbreak, or of people being jerks. All we can ask it to do is ensure that each player can't learn whether the other player has stated an interest in them, without stating interest themselves.

Without further ado, then, here's how Alice and Bob can solve the dating problem:

1. Alice goes through the standard procedure of picking two huge primes, p and q, such that $p-1$ and $q-1$ are not divisible by 3, and then taking $N = pq$. She keeps p and q secret, but sends Bob N together with $x^3 \bmod N$ and $y^3 \bmod N$ for some x and y. If she's not interested, then x and y are both 0 with random garbage padded onto them. If she *is* interested, then x is again 0 with random garbage, but y is 1 with random garbage.

2. Assuming RSA is secure, Bob (not knowing the prime factors of N) doesn't know how to take cube roots mod N efficiently, so $x^3 \bmod N$ and $y^3 \bmod N$ both look completely random to him. Bob does the following: he first picks a random integer r from 0 to $N-1$. Then, if he's not interested in Alice, he sends her $x^3 r^3 \bmod N$. If he *is* interested, he sends her $y^3 r^3 \bmod N$.

3. Alice takes the cube root of whatever number Bob sent. If Bob wasn't interested, this cube root will be $xr \bmod N$, while if he was interested it will be $yr \bmod N$. Either way, the outcome will look completely random to Alice, since she doesn't know r (which was chosen randomly). She then sends the cube root back to Bob.

4. Since Bob knows r, he can divide out r. We see that if Bob was not interested, he simply gets x which reveals nothing about Alice's interest. Otherwise he gets y which is 1 if and only if Alice is interested.

So there we have it. It seems that, at least in principle, computer scientists have solved the problem of flirting for shy people (assuming RSA is secure). This is truly nontrivial for computer scientists. However, this is just one example of what's called *secure multiparty computation*; a general theory to solve essentially all such problems was developed in the 1980's. So for example: suppose two people want to find out who makes more money, but without either of them learning anything else about the other's wealth. Or a group of friends want to know how much money they have *in total*, without any individual revealing her own amount. All of these problems, and many more, are known to be solvable cryptographically.

5 Zero-Knowledge Proofs

5.1 Motivation

In our dating protocol, we made the critical assumption that Alice and Bob were "honest but curious," i.e. they both followed the protocol correctly. We'd now like to move away from this assumption, and have the protocol work even if one of the players is cheating. (Naturally, if they're *both* cheating then there's nothing we can do.)

As discussed earlier, we're not concerned with the case where Bob pretends that he likes Alice just to find out whether she likes him. There's no cryptographic protocol that helps with Bob being a jerk, and we can only hope he'll get caught being one. Rather, the situation we're concerned with is when one of the players *looks* like they're following the protocol, but are actually just trying to find out the other player's interest.

What we need is for Alice and Bob to *prove* to each other at each step of the protocol that they're correctly following the protocol—i.e., sending whatever message they're supposed to send, given whether they're interested or not. The trouble is, they have to do this without *revealing* whether they're interested or not! Abstractly, then, the question is how it's possible to prove something to someone without revealing a crucial piece of information on which the proof is based.

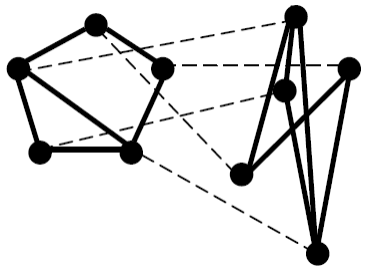
5.2 History

Zero-knowledge proofs have been a major idea in cryptography since the 1980's. They were introduced by Goldwasser, Micali, and Rackoff in 1985. Interestingly, their paper was rejected multiple times before publication but is now one of the foundational papers of theoretical computer science.

5.3 Interactive Proofs

For thousands of years, the definition of a proof accepted by mathematicians was a sequence of logical deductions that could be shared with anyone to convince them of a mathematical truth. But developments in theoretical computer science over the last few decades have required generalizing the concept of proof, to *any sort of computational process or interaction* that can terminate a certain way only if the statement to be proven is true. Zero-knowledge proofs fall into the latter category, as we'll see next.

5.4 Simple Example: Graph Nonisomorphism



To explain the concept of zero-knowledge proofs, it's easiest to start with a concrete example. The simplest example concerns the Graph Isomorphism problem. Here we're given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, which are defined by lists of their edges and vertices. The graphs are called *isomorphic* if there's a way to permute their vertices so that they are the same graph.

5.4.1 Complexity

It's clear that the Graph Isomorphism problem is in NP, since a short proof that G_1 and G_2 are isomorphic is just to specify the isomorphism (i.e., a mapping between the vertices of G_1 and G_2).

Is Graph Isomorphism in P? Is it NP-complete? We don't yet know the answer to either question, though we do have strong evidence that it isn't NP-complete. Specifically, we know that if Graph Isomorphism is NP-complete then $NP^{NP} = coNP^{NP}$, or "the polynomial hierarchy collapses" (proving this statement is beyond the scope of the course). Some computer scientists conjecture that Graph Isomorphism is intermediate between P and NP-complete, just as we believe Factoring to be. Others conjecture that Graph Isomorphism is in P, and we simply don't know enough about graphs yet to give an algorithm. (Note that we have efficient algorithms for Graph Isomorphism that work extremely well *in practice* – just not any that we can prove will work in all cases.)

As an amusing side note, it's said that the reason Levin wasn't the first to publish on NP-completeness is that he got stuck trying to show the Graph Isomorphism problem was NP-complete.

5.4.2 Proving No Isomorphism Exists

We said before that Graph Isomorphism is in NP. But is it in coNP? That is, can you give a short proof that two graphs are *not* isomorphic? Enumerating all the possibilities obviously won't work, since it's exponentially inefficient (there are $n!$ possible mappings). To this day, we don't know whether Graph Isomorphism is in coNP (though there are some deep recent results suggesting that it is).

Still, let's see an incredibly simple way that an all-knowing prover could convince a polynomial-time verifier that two graphs are not isomorphic. To illustrate, consider Coke and Pepsi. Suppose you claim that the two drinks are different but I maintain they're the same. How can you convince me you're right, short of giving me the chemical formula for both? By doing a blind taste test! If I blindfold you and you can reliably tell which is which, then you'll have convinced me that they must be different, even if I don't understand how.

The same idea applies to proving that G_1 and G_2 are not isomorphic. Suppose you're some wizard who has unlimited computational power, but the person you are trying to convince does not. The person can pick one of the two graphs at random and permute the vertices in a random way to form a new graph G', then send you G' and ask which graph she started with. If the graphs are indeed not isomorphic, then you'll be able to answer correctly every time, whereas if G_1 and G_2 *are* isomorphic, then your chance of guessing correctly will be at most $1/2$ (since a random permutation of G_1 is the same as a random permutation of G_2). If the verifier repeats this test 100 times and you answer correctly every time, then she can be sure to an extremely high confidence (namely $1 - 2^{-100}$) that the graphs are not isomorphic.

But notice something interesting: even though the verifier became convinced, she did so without gaining any new knowledge about G_1 and G_2 (by which, for example, she could convince someone else that they're not isomorphic)! In other words, if she'd merely trusted you, then she could have simulated her entire interaction with you on her own, without ever involving you at all. Any interactive proof system that has this property – that the prover only tells the verifier things that the latter "already knew" – is called a *zero-knowledge proof system*.

(Admittedly, it's only obvious that the verifier doesn't learn anything if she's "honest" – that is, if she follows the protocol correctly. Conceivably a *dishonest* verifier who violated the protocol could learn something she didn't know at the start. This is a distinction we'll see again later.)

5.5 The General Case

How can we extend this notion of a zero-knowledge proof to arbitrary problems, besides Graph Isomorphism? For example, suppose that you've proven the Riemann Hypothesis, but are paranoid and do not want anyone else to know your proof just yet. That might sound silly, but it's essentially how mathematicians worked in the Middle Ages: each knew how to solve some equation but didn't want to divulge the general method for solving it to competitors.

So suppose you have a proof of some arbitrary statement, and you want to convince people you have a proof without divulging any of the details. It turns out that there's a way to convert *any* mathematical proof into zero-knowledge form; what's more, the conversion can even be done in polynomial time. However, we'll need to make cryptographic assumptions.

5.6 Goldreich-Micali-Wigderson

In what follows, we'll assume that your proof is written out in machine-checkable form, in some formal system like Zermelo-Fraenkel set theory. We know that THEOREM, the problem of proving a theorem in at most n symbols, is an NP-complete problem, and is therefore efficiently reducible to any other NP-complete problem. Thus, we just need to find *some* NP-complete problem for which we can prove that we have a solution, without divulging the solution. Out of the thousands of known NP-complete problems, it turns out that the most convenient for our purpose will be the problem of 3-coloring a graph.

5.6.1 3-Coloring Proof

Suppose we have a 3-coloring of a graph and we want to prove that we have this 3-coloring without divulging it. Also, suppose that for each vertex of the graph, there's a magical box in which we can store the color of that vertex. What makes these boxes magical is that we can open them but the verifier can't. The key point is that, by storing colors in the boxes, we can "commit" to them: that is, we can assure the verifier that we've picked the color of each vertex beforehand, and are not just making them up based on which questions she asks.

Using these boxes, we can run the following protocol:

1. Start with a 3-coloring of the graph; then randomly permute the colors of the vertices. There are $3! = 6$ ways to permute the colors. For example, red⇒green, green⇒red, blue stays the same.

2. Write the color of each vertex on a slip of paper and place it in the magic box that's labeled with that vertex's number. Give all of the magic boxes to the verifier.

3. Let the challenger pick any two neighboring vertices, and open the boxes corresponding to those vertices.

4. Throw away the boxes and repeat the whole protocol as many times as desired.

If we really have a 3-coloring of the graph, then the verifier will see two different colors every time she chooses two neighboring vertices. On the other hand, suppose we were lying and didn't have a 3-coloring. Then eventually the verifier will find a conflict. Note that there are $O(n^2)$ edges, where n is the number of vertices of the graph. Therefore, since we commit to the colors in advance, there's a $\Omega(1/n^2)$ chance of catching us if we were lying. By repeating the whole protocol, say, n^3 times, the verifier can boost the probability of catching a lie exponentially close to 1, and can therefore (assuming everything checks out) become exponentially confident that we were telling the truth.

On the other hand, since we permute the colors randomly and reshuffle every time, the verifier learns nothing about the actual 3-coloring; she just sees two different random colors every time and thereby gains no knowledge!

Of course, the whole protocol relied on the existence of "magic boxes." So what if we don't have the magic boxes available? Is there any way we could *simulate* their functionality, if we were just sending messages back and forth over the Internet?

Yes: using cryptography! Instead of locking each vertex's color in a box, we can *encrypt* each color and send the verifier the encrypted messages. Then, when the verifier picks two adjacent vertices and asks us for their colors, we can decrypt the corresponding messages (though not the encrypted messages for any other vertices). For this to work, we just need to ensure two things:

1. A polynomial-time verifier shouldn't be able to learn *anything* from the encrypted messages. In particular, this means that even if two vertices are colored the same, the corresponding encrypted messages should look completely different. Fortunately, this is easy to arrange, for example by padding out the color data with random garbage prior to encrypting it.

2. When, in the last step, we decrypt two chosen messages, we should be able to *prove* to the verifier that the messages were decrypted correctly. In other words, every encrypted message should have one and only one decryption. As discussed earlier, the most popular public-key cryptosystems, like RSA, satisfy this property by construction. But even with more

"generic" cryptosystems (based on arbitrary one-way functions), it's known how to simulate the unique-decryption property by adding in more rounds of communication.

5.6.2 Back to Dating

Recall our original goal in discussing zero-knowledge: we wanted to make the dating protocol work correctly, even if Alice or Bob might be cheating. How can we do that? Well, first have Alice and Bob send each other encrypted messages that encode whether or not they're interested in each other, as well as their secret numbers p, q, and r. Then have them follow the dating protocol exactly as before, but with one addition: *at each step, a player not only sends the message that's called for in the protocol, but also provides a zero-knowledge proof that that's exactly the message they were supposed to send—given the sequence of previous messages, whether or not they're interested, and* p, q, r. Note that this is possible, since decrypting all the encrypted messages and verifying that the protocol is being followed correctly is an NP problem, which is therefore reducible to SAT and thence to 3-Coloring. And by definition, a zero-knowledge proof leaks no information about Alice and Bob's private information, so the protocol remains secure.

To clarify one point, it's not known how to implement this dating protocol using an arbitrary OWF—only how to implement the GMW part of it (the part that makes the protocol secure against a cheating Alice or Bob). To implement the protocol itself, we seem to need a stronger assumption, like the security of RSA or something similar. (Indeed, it's not even known how to implement the dating protocol using an arbitrary *trapdoor* OWF, although if we know further that the trapdoor OWF is a permutation, then it's possible.)

6.080/6.089 GITCS April 24, 2008

Lecture 19

Lecturer: Scott Aaronson *Scribe: Michael Fitzgerald*

1 Recap And Discussion Of Previous Lecture

In the previous lecture, we discussed different cryptographic protocols. People asked: "In the RSA cryptosystem, why do people raise to a power greater than three?" Raising to a power greater than three is an extra precaution; it's like adding a second lock on your door. If everything has been implemented correctly, the RSA we discussed (cubing the message $\pmod n$) should be fine. This assumes that your message has been padded appropriately, however. If your message hasn't been padded, "small-exponent attacks" can be successful at breaking RSA; sending the message to a bunch of different recipients with different public keys can let the attacker take advantage of the small exponent. Raising to a power greater than three mitigates this risk.

There are a couple of other attacks that can be successful. "Timing attacks" look at the length of time the computer takes to generate numbers to get hints as to what those numbers are. Other attacks can look at the electromagnetic waves coming from the computer to try and get hints about the number. Then there are attacks that abuse the cryptosystem with constructed inputs and try to determine some information about the system based on the error messages they receive. In general, modern cryptosystems are most often defeated when attackers find bugs in the *implementations* of the systems, not in the systems themselves. Social engineering remains the most successful way of breaching security; often just calling someone on the phone, pretending to be a company tech support person, and asking for their password will get you a response.

We also talked about zero-knowledge proofs and general interactive protocols in the last lecture. Twenty years ago, a revolution in the notion of "proof" drove home the point that a proof doesn't have to be just a static set of symbols that someone checks for accuracy. For example, a proof can be an interactive process that ends with you being convinced of a statement's truth, without learning much of anything else. We gave two examples of so-called *zero-knowledge protocols*: one that convinces a verifier that two graphs are not isomorphic, and another that proves *any* statement with a short conventional proof, assuming the existence of one-way functions.

2 More Interactive Proofs

It turns out that this notion of an interactive proof is good for more than just cryptography. It was discovered in the early 1990s that interactive proofs can convince you of solutions to problems that we think are much harder than NP-complete ones. As an analogy, it's hard to tell that an author of a research paper knows what he's talking about just from reading his paper. If you get a chance to ask him questions off the cuff and he's able to respond correctly, it's much more convincing. Similarly, if you can send messages back and forth with a prover, can you use that to convince yourself of more than just the solution to an NP-complete problem? To study this in the 1980s, people defined a complexity class called IP, which stands for "interactive proof." The details of this story are beyond the scope of the class, but it's being mentioned because it's important.

Consider the following scenario. Merlin and Arthur are communicating. Merlin has infinite computational power, but he is not trustworthy. Arthur is a PPT (probabilistic polynomial time)

king; he can flip coins, generate random numbers, send messages back and forth with Merlin, etc. What we want from a good protocol is this: if Merlin is telling the truth, then there should be some strategy for Merlin that causes Arthur to accept with probability 1. On the other hand, if Merlin is lying, then Arthur should reject with probability greater than 1/2, *regardless* of Merlin's strategy. These correspond to the properties of completeness and soundness that we discussed a while ago.

How big is the class IP, then? It certainly contains NP, as Merlin's strategy could just be to send a solution over to Arthur for the latter to check and approve. Is IP bigger than NP, though? Does interaction let you verify more statements than just a normal proof would? In 1990, Lund, Fortnow, Karloff, and Nisan showed that IP contains $coNP$ as well. This isn't obvious; the key idea in the proof involves how polynomials over finite fields can be judged as equal by testing them at random points. This theorem takes advantage of that fact, along with the fact that you can reinterpret a Boolean formula as a polynomial over a finite field. An even bigger bombshell came a month later, when Shamir showed that IP contains the entire class $PSPACE$, of problems solvable with polynomial memory. Since it was known that IP is *contained* in $PSPACE$, this yields the famous result $IP = PSPACE$.

What does this result mean, in intuitive terms? Suppose an alien comes to earth and says, "I can play perfect chess." You play the alien and it wins. But this isn't too surprising, since you're not very good at chess (for the purposes of this example, at least). The alien then plays again your local champion, then Kasparov, then Deep Blue, etc., and it beats them all. But just because the alien can beat anyone on earth, doesn't mean that it can beat anything in the universe! Is there any way for the alien to prove the stronger claim?

Well, remember that earlier we mentioned that a generalized $n \times n$ version of chess is a $PSPACE$ problem. Because of that, we can transform chess to a game about polynomials over finite fields. In this transformed game, the best strategy for one of the players is going to be to move randomly. Thus, if you play randomly against the alien in this transformed game and it wins, you can be certain (with only an exponentially small probability of error) that it has an optimal strategy, and could beat anyone.

You should be aware of this result, as well as the zero-knowledge protocol for the 3-Coloring, since they're two of the only examples we have in computational complexity theory where you take an NP-complete or $PSPACE$-complete problem, and do something with it that actually exploits its *structure* (as opposed to just treating it as a generic search problem). And it's known that exploiting structure in this sort of way—no doubt, at an astronomically more advanced level—will someday be needed to solve the $P = NP$ problem.

3 Machine Learning

Up to this point, we've only talked about problems where all the information is explicitly given to you, and you just have to do something with it. It's like being handed a grammar textbook and asked if a sentence is grammatically correct. Give that textbook to a baby, however, and it will just drool on it; humans learn to speak and walk and other incredibly hard things (harder than anything taught at MIT) without ever being explicitly told how to do them. This is obviously something we'll need to grapple with if we ever want to understand the human brain. We talked before about how computer science grew out of this dream people had of eventually understanding the process of thought: can you reduce it to something mechanical, or automate it? At some point, then, we'll have to confront the problem of learning: inferring a general rule from specific examples when the rule is never explicitly given to you.

3.1 Philosophy Of Learning

As soon as we try to think about learning, we run into some profound philosophical problems. The most famous of these is the *Problem of Induction*, proposed by 18$^{\text{th}}$-century Scottish philosopher David Hume. Consider two hypotheses:

1. The sun rises every morning.
2. The sun rises every morning until tomorrow, when it will turn into the Death Star and crash into Jupiter.

Hume makes the point that both of these hypotheses are completely compatible with all the data we have up until this point. They both explain the data we have equally well. We clearly believe the first over the second, but what grounds do we have for favoring one over the other? Some people say they believe the sun will rise because they believe in the laws of physics, but then the question becomes why they believe the laws of physics will continue.

To give another example, here's a "proof" of why it's not possible to learn a language, due to Quine. Suppose you're an anthropologist visiting a native tribe and trying to learn their language. One of the tribesmen points to a rabbit and says "gavagai." Can you infer that "gavagai" is their word for rabbit? Maybe gavagai is their word for food or dinner, or "little brown thing." By talking to them longer you could rule those out, but there are other possibilities that you haven't ruled out, and there will always be more. Maybe it means "rabbit" on weekdays but "deer" on weekends, etc.

Is there any way out of this? Right, we can go by Occam's Razor: if there are different hypotheses that explain the data equally well, we choose the simplest one.

Here's a slightly different way of saying it. What the above thought experiments really show is not the impossibility of learning, but rather the impossibility of learning in a theoretical vacuum. Whenever we try to learn something, we have some set of hypotheses in mind which is vastly smaller than the set of all logically conceivable hypotheses. That "gavagai" would mean "rabbit" is a plausible hypothesis; the weekday/weekend hypothesis does *not* seem plausible, so we can ignore it until such time as the evidence forces us to.

How, then, do we separate plausible hypotheses from hypotheses that aren't plausible? Occam's Razor seems related to this question. In particular, what we want are hypotheses that are *simpler than the data they explain*, ones that take fewer bits to write down than just the raw data. If your hypothesis is extremely complicated, and if you have to revise your hypothesis for every new data point that comes along, then you're probably doing something wrong.

Of course, it would be nice to have a theory that makes all of this precise and quantitative.

3.2 From Philosophy To Computer Science

It's a point that's not entirely obvious, but the problem of learning and prediction is related to the problem of data compression. Part of predicting the future is coming up with a succinct description of what has happened in the past. A philosophical person will ask why that should be so, but there might not be an answer. The belief that there are simple laws governing the universe has been a pretty successful assumption, so far at least. As an example, if you've been banging on a door for five minutes and it hasn't opened, a sane person isn't going to expect it to open on the next knock. This could almost be considered the definition of sanity.

If we want to build a machine that can make reasonable decisions and learn and all that good stuff, what we're really looking for is a machine that can create simple, succinct descriptions and

hypotheses to explain the data it has. What exactly is a "simple" description, then? One good way to define this is by Kolmogorov complexity; a simple description is one that corresponds to a Turing machine with few states. This is an approach that many people take. The fundamental problem with this is that Kolmogorov complexity is not computable, so we can't really use this in practice. What we want is a quantitative theory that will let us deal with *any* definition of "simple" we might come up with. The question will then be: "given some class of hypotheses, if we want to be able to predict 90% of future data, how much data will we need to have seen?" This is where theoretical computer science really comes in, and in particular the field of *computational learning theory*. Within this field, we're going to talk about a model of learning due to Valiant from 1984: the PAC (Probably Approximately Correct) model.

3.3 PAC Learning

To understand what this model is all about, it's probably easiest just to give an example. Say there's a hidden line on the chalk board. Given a point on the board, we need to classify whether it's above or below the line. To help, we'll get some sample data, which consists of random points on the board and whether each point is above or below the line. After seeing, say, twenty points, you won't know *exactly* where the line is, but you'll probably know roughly where it is. And using that knowledge, you'll be able to predict whether most future points lie above or below the line.

Suppose we've agreed that predicting the right answer "most of the time" is okay. Is any random choice of twenty points going to give you that ability? No, because you could get really unlucky with the sample data, and it could tell you almost nothing about where the line is. Hence the "Probably" in PAC.

As another example, you can speak a language for your whole life, and there will still be edge cases of grammar that you're not familiar with, or sentences you construct incorrectly. That's the "Approximately" in PAC. To continue with that example, if as a baby you're really unlucky and you only ever hear one sentence, you're not going to learn much grammar at all (that's the "Probably" again).

Let's suppose that instead of a hidden line, there's a hidden squiggle, with a side 1 and a side 2. It's really hard to predict where the squiggle goes, just from existing data. If your class of hypotheses is arbitrary squiggles, it seems impossible to find a hypothesis that's even probably approximately correct. But what is the difference between lines and squiggles, that makes one of them learnable and the other one not learnable?

Well, no matter how many points there are, you can always cook up a squiggle that works for those points, whereas the same is not true for lines. That seems related to the question somehow, but why?

What computational learning theory lets you do is delineate mathematically what it is about a class of hypotheses that makes it learnable or not learnable (we'll get to that later).

3.4 Framework

Here's the basic framework of Valiant's PAC Learning theory, in the context of our line-on-the-chalkboard example:

S: Sample Space - The set of all the points on the blackboard.

D: Sample Distribution - The probability distribution from which the points are drawn (the uniform distribution in our case).

Concept - A function $h : S \to \{0, 1\}$ that maps each point to either 0 or 1. In our example, each concept corresponds to a line.

C: Concept Class - The set of all the possible lines.

"True Concept" $c \in C$: The actual hidden line; the thing you're trying to learn.

In this model, you're given a bunch of sample points drawn from S according to D, and each point comes with its classification. Your goal is to find a hypothesis $h \in C$ that classifies future points correctly almost all of the time:

$$\Pr_{x \in D}[h(x) = c(x)] \geq 1 - \epsilon$$

Note that the future points that you test on should be drawn from the same probability distribution D as the sample points. This is the mathematical encoding of the "future should follow from the past" declaration in the philosophy; it also encodes the well-known maxim that "nothing should be on the test that wasn't covered in class."

As discussed earlier, we won't be able to achieve our goal with certainty, which is why it's called *Probably* Approximate Correct learning. Instead, we only ask to succeed in finding a good classifier with probability at least $1 - \delta$ over the choice of sample points.

One other question: does the hypothesis h have to belong to the concept class C? There are actually two notions, both of which we'll discuss: *proper learning* (h must belong to C) and *improper learning* (h can be arbitrary).

These are the basic definitions for this theory.

Question from the floor: Don't some people design learning algorithms that output confidence probabilities along with their classifications?

Sure! You can also consider learning algorithms that try to predict the output of a real-valued function, etc. Binary classification is just the simplest learning scenario – and for that reason, it's a nice scenario to focus on to build our intuition.

3.5 Sample Complexity

One of the key issues in computational learning theory is *sample complexity*. Given a concept class C and a learning goal (the accuracy and confidence parameters ϵ and δ), how much sample data will you need to achieve the goal? Hopefully the number of samples m will be a finite number, but even more hopefully, it'll a *small* finite number, too.

Valiant proposed the following theorem, for use with finite concept classes, which gives an upper bound on how many samples will suffice:

$$m \geq \frac{1}{\epsilon} \log \frac{|C|}{\delta}$$

As ϵ gets smaller (i.e., as we want a more accurate hypothesis), we need to see more and more data. As there are more concepts in our concept class, we also need to see more data.

A learning method that achieves Valiant's bound is simply the following: find any hypothesis that fits all the sample data, and output it!

As long as you've seen m data points, the theorem says that with probability at least $1-\delta$, you'll have a classifier that predicts at least a $1 - \epsilon$ fraction of future data. There's only a logarithmic dependency on $\frac{1}{\delta}$, which means we can learn within an exponentially small probability of error using only a polynomial number of samples. There's also a log dependence on the number of concepts $|C|$, which means that even if there's an exponential number of concepts in our concept class, we

can still do the learning with a polynomial amount of data. If that weren't true we'd really be in trouble.

Next time: proof of Valiant's bound, VC-dimension, and more...

6.080/6.089 GITCS 1 April 2008

Lecture 20

Lecturer: Scott Aaronson *Scribe: Geoffrey Thomas*

Probably Approximately Correct Learning

In the last lecture, we covered Valiant's model of "Probably Approximately Correct" (PAC) learning. This involves:

S:	A *sample space* (e.g., the set of all points)
D:	A *sample distribution* (a probability distribution over points in the sample space)
$c : S \to \{0, 1\}$:	A *concept*, which accepts or rejects each point in the sample space
C:	A *concept class*, or collection of concepts

For example, we can take our sample space to be the set of all points on the blackboard, our sample distribution to be uniform, and our concept class to have one concept corresponding to each line (where a point is accepted if it's above the line and rejected if it's below it). Given a set of points, as well as which points are accepted or rejected, our goal is to *output a hypothesis* that explains the data: e.g., draw a line that will correctly classify most of the future points.

A bit more formally, there's some "true concept" $c \in C$ that we're trying to learn. Given sample points $x_1, \ldots, x_m$, which are drawn independently from D, together with their classifications $c(x_1), \ldots, c(x_m)$, our goal is to find a hypothesis $h \in C$ such that

$$\mathbf{Pr}\left[h(x) = c(x)\right] \geq 1 - \epsilon$$

. Furthermore, we want to succeed at this goal with probability at least $1 - \delta$ over the choice of x_i's. In other words, with high probability we want to output a hypothesis that's approximately correct (hence "Probably Approximately Correct").

How many samples to learn a finite class?

The first question we can ask concerns *sample complexity*: how many samples do we need to have seen to learn a concept effectively? It's not hard to prove the following theorem: after we see

$$m = O\left(\frac{1}{\epsilon} \log \frac{|C|}{\delta}\right)$$

samples drawn from D, *any* hypothesis $h \in C$ we can find that agrees with all of these samples (i.e., such that $h(x_i) = c(x_i)$ for all i) will satisfy

$$\mathbf{Pr}\left[h(x) = c(x)\right] \geq 1 - \epsilon$$

with probability at least $1 - \delta$ over the choice of $x_1, \ldots, x_m$.

We can prove this theorem by the contrapositive. Let $h \in C$ be any "bad" hypothesis: that is, such that $\mathbf{Pr}\left[h(x) = c(x)\right] < 1 - \epsilon$. Then if we independently pick m points from the sample distribution D, the hypothesis h will be correct on all of these points with probability at most $(1 - \epsilon)^m$. So by the union bound, the probability that there *exists* a bad hypothesis in C that nevertheless agrees with all our sample data is at most $|C|\,(1 - \epsilon)^m$ (the number of hypotheses,

good or bad, times the maximum probability of each bad hypothesis agreeing with the sample data). Now we just do algebra:

$$\begin{aligned} \delta &= |C|\,(1-\epsilon)^m \\ m &= \log_{1-\epsilon} \frac{\delta}{|C|} \\ &= \frac{\log \delta/|C|}{\log 1-\epsilon} \\ &\approx \frac{1}{\epsilon} \log \frac{|C|}{\delta}. \end{aligned}$$

Note that there always *exists* a hypothesis in C that agrees with c on all the sample points: namely, c itself (i.e. the truth)! So as our learning algorithm, we can simply do the following:

1. Find any hypothesis in $h \in C$ that agrees with all the sample data (i.e., such that $h(x_i) = c(x_i)$ for all $x_1, \ldots, x_m$).
2. Output h.

Such an h will always exist, and by the theorem above it will probably be a good hypothesis. All we need is to see enough sample points.

How many samples to learn an infinite class?

The formula

$$m \approx \frac{1}{\epsilon} \log \frac{|C|}{\delta}$$

works so long as $|C|$ is finite, but it breaks down when $|C|$ is infinite. How can we formalize the intuition that the concept class of lines is learnable, but the concept class of arbitrary squiggles is not? A line seems easy to guess (at least approximately), if I give you a small number of random points and tell you whether each point is above or below the line. But if I tell you that *these* points are on one side of a squiggle, and *those* points are on the other side, then no matter how many points I give you, it seems impossible to predict which side the next point will be on.

So what's the difference between the two cases? It can't be the number of lines versus the number of squiggles, since they're both infinite (and be taken to have the same infinite cardinality).

From the floor: Isn't the difference just that you need two parameters to specify a line, but infinitely many parameters to specify a squiggle?

That's getting closer! The trouble is that the notion of a "parameter" doesn't occur anywhere in the theory; it's something we have to insert ourselves. To put it another way, it's possible to come up with silly parameterizations where even a line takes infinitely many parameters to specify, as well as clever parameterizations where a squiggle can be specified with just one parameter.

Well, the answer isn't obvious! The idea that finally answered the question is called *VC-dimension* (after two of its inventors, Vapnik and Chervonenkis). We say the set of points $x_1, \ldots, x_m$ is *shattered* by a concept class C if for all 2^m possible settings of $c(x_1), \ldots, c(x_m)$ to 0 or 1 (reject or accept), there is some concept $c \in C$ that agrees with those values. Then the VC-dimension of C, denoted $\mathrm{VCdim}(C)$, is the size of the largest set of points shattered by C. If we can find arbitrarily large (finite) sets of points that can be shattered, then $\mathrm{VCdim}(C) = \infty$.

If we let C be the concept class of lines in the plane, then $\text{VCdim}(C) = 3$. Why? Well, we can put three points in a triangle, and each of the eight possible classifications of those points can be realized by a single line. On the other hand, there's no set of four points such that all sixteen possible classifications of those points can be realized by a line. Either the points form a quadrilateral, in which case we can't make opposite corners have the same classification; or they form a triangle and an interior point, in which case we can't make the interior point have a different classification from the other three points; or three of the points are collinear, in which case we certainly can't classify those points with a line.

Blumer et al.[1] proved that a concept class is PAC-learnable if and only if its VC-dimension is finite, and that

$$m = O\left(\frac{\text{VCdim}(C)}{\epsilon}\log\frac{1}{\delta\epsilon}\right)$$

samples suffice. Once again, a learning algorithm that works is just to output *any* hypothesis h in the concept class that agrees with all the data. Unfortunately we don't have time to prove that here.

A useful intuition is provided by a corollary of Blumer et al.'s result called the *Occam's Razor Theorem*: whenever your hypothesis has sufficiently fewer bits of information than the original data, it will probably correctly predict most future data drawn from the same distribution.

Computational Complexity of Learning

We've seen that given a finite concept class—or even an infinite class with finite VC-dimension—after seeing enough sample points, you can predict the future just by finding any hypothesis in the class that fits the data. But how hard is it *as a computational problem* to find a hypothesis that fits the data? This has the general feel of something that might be NP-complete! In particular, it feels similar to satisfiability—find some hypothesis that satisfies certain fixed outputs—though it's not quite the same.

Here we need to make a subtle distinction. For *proper* learning—where the goal is to output a hypothesis in some fixed format (like a DNF expression), it's indeed possible to prove in some cases that finding a hypothesis that fits the data is NP-complete. For *improper* learning—where the hypothesis can be any polynomial-time algorithm so long as it predicts the data—to this day we don't know whether finding a hypothesis is NP-complete.

On the other hand, the learning problem is certainly *in* NP, since given a hypothesis it's easy to check whether it fits the data or not. This means that if $P = NP$, then all learning problems are in P and are computationally tractable. Think about what that means: we could ask our computer to find the shortest efficient description of the stock market, the patterns of neural firings in a human brain, etc., and thereby solve many of the hardest problems of AI! This is yet another reason to believe $P \neq NP$.

[1] Blumer, Ehrenfeucht, Haussler, Warmuth, "Learnability and the Vapnik-Chervonenkis dimension", JACM, 1989

6.080/6.089 GITCS Feb 5, 2008

Lecture 21

Lecturer: Scott Aaronson *Scribe: Scott Aaronson / Chris Granade*

1 Recap and Discussion of Previous Lecture

Theorem 1 (Valiant) $m = O\left(\frac{1}{\epsilon}\log\left(|C|/\delta\right)\right)$ *samples suffice for* (ϵ, δ)*-learning.*

Theorem 2 (Blumer et al.) $m = O\left(\frac{1}{\epsilon}\,\mathrm{VCdim}\,(C)\log\frac{1}{\delta\epsilon}\right)$ *samples suffice.*

In both cases, the learning algorithm that achives the bound is just "find any hypothesis h compatible with all the sample data, and output it."

You asked great, probing questions last time, about what these theorems really mean. For example, "why can't I just draw little circles around the 'yes' points, and expect that I can therefore predict the future?" It's unfortunately a bit hidden in the formalism, but what these theorems are "really" saying is that to predict the future, it suffices to find a succinct description of the past–a description that takes many fewer bits to write down than the past data itself. Hence the dependence on $|C|$ or $\mathrm{VCdim}\,(C)$: the size or dimension of the concept class from which our hypothesis is drawn.

We also talked about the computational problem of finding a small hypothesis that agrees with the data. Certainly we can always solve this problem in polynomial time if $\mathsf{P} = \mathsf{NP}$. But what if $\mathsf{P} \neq \mathsf{NP}$? Can we show that "learning is NP-hard"? Here we saw that we need to distinguish two cases:

Proper learning problems (where the hypothesis has to have a certain form): Sometimes we can show these are NP-hard. Example: Finding a DNF expression that agrees with the data.

Improper learning problems (where the hypothesis can be any Boolean circuit): It's an open problem whether any of these are NP-hard. (Incidentally, why do we restrict the hypothesis to be a Boolean circuit? It's equivalent to saying, we should be able to compute in polynomial time what a given hypothesis predicts.)

So, if we can't show that improper (or "representation-independent") learning is NP-complete, what other evidence might there be for its hardness? The teaser from last time: we could try to show that finding a hypothesis that explains past data is at least as hard as breaking some cryptographic code!

But how would we actually do this? How would we reduce a cryptanalysis problem to a learning problem? To be concrete, let's just consider the RSA cryptosystem. Can any of you give me a PAC-learning problem, such that if you could solve it, then you could also break RSA?

How about this: our concept class C will have one concept c for each product of prime numbers $N = pq$, with $p - 1$ and $q - 1$ not divisible by 3. (Technically, for each N expressible with at most n bits.)

Our sample space S will consist of pairs of the form (y, i), where $1 \leq y \leq N-1$ and $1 \leq i \leq \log N$. Here's the trick: (y, i) will be in c if and only if the i^{th} bit of $y^{1/3} \bmod N$ is a 1. The sample distribution D will be uniform over S.

So basically, you (the learner) will be given a bunch of encrypted messages of the form $x^3 \bmod N$, and for each one, you'll also told some bit of the plaintext x. Based on this "training" data, you need to infer the general rule for going from $x^3 \bmod N$ to some given bit of x.

First question: is there such a rule, which is expressible by a polynomial-size circuit? Sure there is! Namely, the rule that someone who knew the trapdoor information, who knew p and q, would use to decrypt the messages!

On the other hand, if you don't already know this rule, is there an efficient algorithm to infer it from sample data? Well, not if RSA is secure! The sample data–the set of (y, i) pairs–is stuff that an eavesdropper could not only plausibly have access to, but could actually generate itself! So if, by examining that data, the adversary could gain the ability to go from $x^3 \bmod N$ to a desired bit of x—well, then RSA is toast. (Today, it's actually known how to base the hardness of improper learning on the existence of any one-way function, not just the RSA function.) A beautiful connection between learning theory and cryptography—typical of the connections that abound in theoretical computer science.

1.1 RSA and Language Learning In Infants: The Argument Chomsky Should've Made

What is Noam Chomsky famous for, besides hating America? Linguistics, of course—among other things, what's called the "Poverty of the Stimulus Argument." This is an argument that tries to show, more or less from first principles, that many of the basic ingredients of grammar (nouns, verbs, verb conjugation, etc.) must be hardwired into the human brain. They're not just taught to children by their parents: the children are "pre-configured" to learn grammar.

The argument says, suppose that weren't the case; suppose instead that babies started out as blank slates. Before it has to start speaking, a baby will hear, what, a hundred thousand sentences? Chomsky claims, with a bit of handwaving, that isn't nearly enough sentences to infer the general rules of grammar, the rules separating grammatical sentences from ungrammatical ones. The sample data available to the baby are just too impoverished for it to learn a language from scratch.

But here's the problem: the sample complexity bounds we saw earlier today should make us skeptical of any such argument! These bounds suggested that, in principle, it really *is* possible to predict the future given a surprisingly small amount of sample data. As long as the VC-dimension of your concept class is small—and I know I haven't convinced you of this, but in "most" practical cases it is—the amount of data needed for learning will be quite reasonable.

So the real stumbling block would seem to be not sample complexity, but computational complexity! In other words, if the basic ingredients of language weren't hardwired into every human baby, then even if in principle a baby has heard enough sentences spoken by its parents to infer the rules of the English language, how would the baby actually do the computation? It's just a baby!

More concretely, let's say I give you a list of n-bit strings, and I tell you that there's some nondeterministic finite automaton M, with much fewer than n states, such that each string was produced by following a path in M. Given that information, can you reconstruct M (probably and approximately)? It's been proven that if you can, then you can also break RSA! Now, finite automata are often considered the very simplest models of human languages. The grammar of any real human language is much too rich and complicated to be captured by a finite automaton. So this result is saying that even learning the least expressive, unrealistically simple languages is already as hard as breaking RSA!

So, using what we've learned about cryptography and computational learning theory, I submit

that we can now make the argument that Chomsky should have made but didn't. Namely: grammar must be hard-wired, since if a baby were able to pick up grammar from scratch, that baby could also break the RSA cryptosystem.[1]

2 Quantum Computing

Up to now in this course, you might have gotten the impression that we're just doing pure math—and in some sense, we are! But for most of us, the real motivation comes from the fact that computation is not just some intellectual abstraction: it's something that actually takes place in our laptops, our brains, our cell nuclei, and maybe all over the physical universe. So given any of the models we've been talking about in this course, you can ask: does this mesh with our best understanding of the laws of physics?

Consider the Turing machine or the circuit models. For at least a couple of decades, there was a serious question: will it be possible to build a general-purpose computer that will scale beyond a certain size? With vacuum tubes, the answer really wasn't obvious. Vacuum tubes fail so often that some people guessed there was a fundamental physical limit on how complex (say) a circuit or a Turing machine tape head could be before it would inevitably fail. In the 1950s, John von Neumann (who we met earlier) became interested in this question, and he proved a powerful theorem: it's possible to build a reliable computer out of unreliable components (e.g., noisy AND, OR, and NOT gates), provided the failure probability of each component is below some critical threshold, and provided the failures are uncorrelated with each other.

But who knows if those assumptions are satisfied in the physical universe? What really settled the question was the invention of the transistor in 1947—which depended on understanding semiconductors (like silicon and germanium), which in turn depended on the quantum revolution in physics eighty years ago. In that sense, every computer we use today is a quantum computer.

But you might say, this is all for the EE people. Once you've got the physical substrate, then we theorists can work out everything else sitting in our armchairs. But can we really?

Consider: what do we mean by efficiently solvable problem? Already in this course, you've seen two plausible definitions: P (the class of problems solvable by polytime deterministic algorithms) and BPP (the class solvable by polytime randomized algorithms with bounded error probability). The fact that we already had to change models once—from P to BPP—should make you suspicious, and this despite the fact that nowadays we conjecture that $P = BPP$. Could nature have *another* surprise in store for us, besides randomness?

[1] The ideas in this section are developed in much more detail in Ronald de Wolf's Masters thesis, "Philosophical Applications of Computational Learning Theory": http://homepages.cwi.nl/ rdewolf/publ/philosophy/phthesis.pdf

6.080/6.089 GITCS	May 6-8, 2008
Lecture 22/23	
Lecturer: Scott Aaronson	*Scribe: Chris Granade*

1 Quantum Mechanics

1.1 Quantum states of n qubits

If you have an object that can be in two perfectly distinguishable states $|0\rangle$ or $|1\rangle$, then it can also be in a superposition of the $|0\rangle$ and $|1\rangle$ states:

$$\alpha|0\rangle + \beta|1\rangle$$

where α and β are complex numbers such that:

$$|\alpha|^2 + |\beta|^2 = 1$$

For simplicity, let's restrict to real amplitudes only. Then, the possible states of this object–which we call a quantum bit, or qubit– lie along a circle.

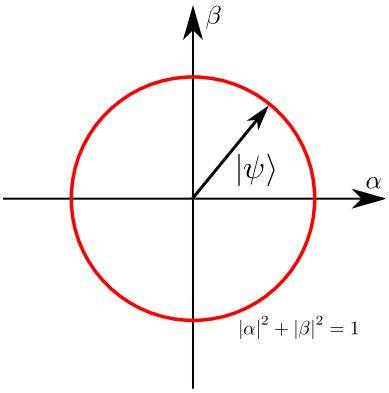


Figure 1: An arbitrary single-qubit state $|\psi\rangle$ drawn as a vector.

If you measure this object in the "standard basis," you see $|0\rangle$ with probability $|\alpha|^2$ and $|1\rangle$ with probability $|\beta|^2$. Furthermore, the object "collapses" to whichever outcome you see.

1.2 Quantum Measurements

Measurements (yielding $|x\rangle$ with probability $|\alpha_x|^2$) are *irreversible, probabilistic, and discontinuous.*

As long as you don't ask specifically what a measurement *is*—how the universe knows what constitutes a measurement and what doesn't—but just assume it as an axiom, everything is well-defined mathematically. If you do ask, you enter a no-man's land. Recently there's been an important set of ideas, known as decoherence theory, about how to explain measurement as ordinary unitary interaction, but they still don't explain where the probabilities come from.

1.3 Unitary transformations

But this is not yet interesting! The interesting part is what else we can do the qubit, besides measure it right away. It turns out that, by acting on a qubit in a suitable way–in the case of an electron, maybe shining a laser on it–we can effectively multiply the vector of amplitudes by any matrix that preserves the property that the probabilities sum to 1. By which I mean, any matrix that always maps unit vectors to other unit vectors. We call such a matrix a unitary matrix. Unitary transformations are *reversible, deterministic, and continuous.*

Examples of unitary matrices:

- The identity I.
- The NOT gate $X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$.
- The phase-i gate $\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$.
- 45-degree counterclockwise rotation.

Physicists think of quantum states in terms of the Schrödinger equation, $\frac{d|\psi\rangle}{dt} = iH|\psi\rangle$ (perhaps the third most famous equation in physics after $e = mc^2$ and $F = ma$). A unitary is just the result of leaving the Schrödinger equation "on" for a while.

> **Q:** Why do we use complex numbers?
> **Scott:** The short answer is that it works! A "deeper" answer is that if we used real numbers only, it would not be possible to divide a unitary into arbitrarily small pieces. For example, the NOT gate we saw earlier can't be written as the square of a real-valued unitary matrix. We'll see in a moment that you can do this if you have complex numbers.

For each of these matrices, what does it do? Why is it unitary? How about this one?

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Is it unitary? Given a matrix, how do you decide if it's unitary or not?

Theorem 1 *U is unitary if and only if $UU^* = I$, where U^* means you transpose the matrix and replace every entry by its complex conjugate. (A nice exercise if you've seen linear algebra.) Equivalently, $U^{-1} = U^*$. One corollary is that every unitary operation is reversible.*

As an exercise for the reader, you can apply this theorem to find which of the matricies we've already seen are unitary.

Now, let's see what happens when we take the 45-degree rotation matrix, and apply it twice to the same state.

$$\begin{aligned} |0\rangle &\rightarrow (|0\rangle + |1\rangle)/\sqrt{2} \\ |1\rangle &\rightarrow (-|0\rangle + |1\rangle)/\sqrt{2} \\ (|0\rangle + |1\rangle)/\sqrt{2} &\rightarrow \left[\frac{|0\rangle + |1\rangle}{\sqrt{2}} + \frac{-|0\rangle + |1\rangle}{\sqrt{2}}\right]/\sqrt{2} \\ &= |1\rangle \end{aligned}$$

This matrix acts as the "square root of NOT"! Another way to see that is by squaring the matrix.

$$\begin{bmatrix} \cos(45°) & -\sin(45°) \\ \sin(45°) & \cos(45°) \end{bmatrix}^2 |\psi\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} |\psi\rangle$$

Already, we have something that doesn't exist in the classical world.

We can also understand the action of this matrix in terms of interference of amplitudes.

2 Two Qubits

To describe two qubits, how many amplitudes do we need? Right, four – one for each possible two-bit string.

$$\alpha |00\rangle + \beta |01\rangle + \gamma |10\rangle + \delta |11\rangle$$

$$|\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\delta|^2 = 1$$

If you measure both qubits, you'll get $|00\rangle$ with probability $|\alpha|^2$, $|01\rangle$ with probability $|\beta|^2$, etc. And the state will collapse to whichever 2-bit string you see.

But what happens if you measure only the first qubit, not the second? With probability $|\alpha|^2 + |\beta|^2$, you get $|0\rangle$, and the state collapses to $\frac{\alpha|00\rangle+\beta|01\rangle}{\sqrt{|\alpha|^2+|\beta|^2}}$. With probability $|\gamma|^2 + |\delta|^2$, you get $|1\rangle$, and the state collapses to $\frac{\gamma|10\rangle+\delta|11\rangle}{\sqrt{|\gamma|^2+|\delta|^2}}$. Any time you ask the universe a question, it makes up its mind; any time you don't it ask a question, it puts off making up its mind for as long as it can.

What happens if you apply a NOT gate to the second qubit? Answer: You get $\beta |00\rangle + \alpha |01\rangle + \delta |10\rangle + \gamma |11\rangle$. "For every possible configuration of the other qubits, what happens if I apply the gate to this qubit?" If we consider $(\alpha, \beta, \gamma, \delta)$ as a vector of four complex numbers, what does this transformation look like as a 4×4 matrix?

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Can we always factor a two-qubit state: "here's the state of the first qubit, here's the state of the second qubit?" Sometimes we can:

- $|01\rangle = |0\rangle |1\rangle = |0\rangle \otimes |1\rangle$ (read $|0\rangle$ "tensor" $|1\rangle$).
- $|00\rangle + |01\rangle + |10\rangle + |11\rangle = \frac{1}{2}(|0\rangle + |1\rangle)(|0\rangle + |1\rangle)$.

In these cases, we say the state is **separable**. But what about $|00\rangle + |11\rangle$? This is a state that *can't* be factored. We therefore call it an **entangled** state. You might have heard about entanglement as one of the central features of quantum mechanics. Well, here it is.

Just as there are quantum states that can't be decomposed, there are also *operations* that can't be decomposed. Perhaps the simplest is the **Controlled-NOT**, which maps $|x\rangle|y\rangle$ to $|x\rangle|x \oplus y\rangle$ (i.e., flips the second bit iff the first bit is 1).

$$\begin{aligned} |00\rangle &\rightarrow |00\rangle \\ |01\rangle &\rightarrow |01\rangle \\ |10\rangle &\rightarrow |11\rangle \\ |11\rangle &\rightarrow |10\rangle \end{aligned}$$

What does this look like as a 4×4 matrix?

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Incidentally, could we have a 2-qubit operation that mapped $|x\rangle|y\rangle$ to $|x\rangle|x \text{ AND } y\rangle$? Why not?

$$\begin{aligned} |0\rangle|0\rangle &\rightarrow |0\rangle|0\rangle \\ |0\rangle|1\rangle &\rightarrow |0\rangle|0\rangle \end{aligned}$$

This is not reversible!

2.1 Obtaining Entanglement

Before we can create a quantum computer, we need some way to entangle the qubits so they're not just a bunch of particles laying around. Perhaps the simplest such operation is the CNOT gate that we saw earlier.

So how do we use CNOT to produce entanglement? We can use a Hadamard followed by a CNOT, where the Hadamard matrix $\boxed{\text{H}}$ puts a qubit into superposition by switching between the $\{|0\rangle, |1\rangle\}$ basis and the $\{|+\rangle, |-\rangle\}$ basis.

$$\begin{aligned} |+\rangle &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\ |-\rangle &= \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \\ \boxed{\text{H}} &= \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \end{aligned}$$

Applying $\boxed{\text{H}}$ to $|0\rangle$ and $|1\rangle$ results in:

$$\begin{aligned} |0\rangle &\rightarrow \frac{|0\rangle + |1\rangle}{\sqrt{2}} = |+\rangle \\ |1\rangle &\rightarrow \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |-\rangle \end{aligned}$$

Already with two qubits, we're in a position to see some profound facts about quantum mechanics that took people decades to understand.

Think again about the state $|00\rangle + |11\rangle$. What happens if you measure just the first qubit? Right, with probability 1/2 you get $|00\rangle$, with probability 1/2 you get $|11\rangle$. Now, why might that be disturbing? Right: because the second qubit might be light-years away from the first one! For a measurement of the first qubit to *affect the second qubit* would seem to require faster-than-light communication! This is what Einstein called "spooky action at a distance."

But think about it more carefully. Can you actually use this effect to send a message faster than light? What would happen if you tried? Right, the result would be random! In fact, we're not going to prove it here, but there's something called the *no-communication theorem*, which says *nothing* you do to the first qubit only can affect the probability of any measurement outcome on the second qubit only.

But in that case, why can't we just imagine that at the moment these two qubits were created, they flipped a coin, and said, "OK, if anyone asks, we'll both be 1." Well, because in 1964, John Bell proved there are certain experiments where no explanation of that kind can possibly agree with quantum mechanics. And in the 1980s, the experiments were actually done, and they vindicated quantum mechanics and in most physicists' view, dashed Einstein's hope for a "completion" of quantum mechanics. That's on your problem set.

2.2 No-Cloning Theorem

Is it possible to duplicate a quantum state? This would be very nice, since we know we only have one chance to measure a quantum state. Here is what such a duplication would look like:

$$\alpha|0\rangle + \beta|1\rangle \rightarrow (\alpha|0\rangle + \beta|1\rangle)(\alpha|0\rangle + \beta|1\rangle) = \alpha^2|00\rangle + \alpha\beta|01\rangle + \alpha\beta|10\rangle + \beta^2|11\rangle$$

This operation is not possible because it is not linear. The final amplitudes α^2, β^2 and $\alpha\beta$ don't depend linearly on α and β. That's the **no-cloning theorem**, and it's really as simple as it looks.

3 n Qubits

For 60 years, *these* were the sorts of examples that drove people's intuitions about quantum mechanics: one particle, occasionally two particles. Rarely did people think abstractly about hundreds or thousands of particles all entangled with one another. But within the last 15 years, we've realized that's where things get *really* crazy. And that brings us to quantum computing. It goes without saying that I'm going to present just the theory at first. Later we can discuss where current experiments are.

How many amplitudes would we need to describe the state of 1000 qubits? Right, 2^{1000}. One for every possible string of 1000 bits:

$$\sum_{x \in \{0,1\}^{1000}} \alpha_x |x\rangle$$

Think about what this *means*. To keep track of the state of 1000 numbers, Nature, off to the side somewhere, apparently has to write down this list of 2^{1000} complex numbers. That's more numbers than there are atoms in the visible universe. Think about how much *computing power* Nature must be expending for that. What a colossal waste! The next thought: we might as well try and take advantage of it!

Q: Doesn't a single qubit already require an infinite amount of information to specify?

Scott: The answer is yes, but there is always noise and error in the real world, so we only care about approximating the amplitudes to some finite precision. In some sense, the "infinite amount of information" is just an artifact of our mathematical description of the qubit's state. By contrast, the exponent in the description of n entangled particles is not an artifact; it's real (if quantum mechanics is the right description of Nature).

3.1 Exploiting Interference

What's an immediate difficulty with taking advantage of this computational power? Well, if we simply measure n qubits, all we get is a classical n-bit string; everything else disappears. It's like the instant we look, nature tries to "hide" the fact that it's doing an exponential amount of computation.

But luckily for us, Nature doesn't always do a good job of hiding. A good example of this is the double-slit experiment: we don't measure which of the two slits the photon passed through, but rather the resulting interference pattern. In particular, we saw that the different paths taken by a quantum system can *interfere destructively* and cancel each other out.

So *that's* what we want to exploit in quantum computing. The goal is to choreograph things so that the different computational paths leading to a given wrong answer interfere destructively and cancel each out, while the different paths leading to a given right answer interfere constructively, hence the right answers are observed with high probability when we measure. You can see how this is gonna be tricky, if it's possible at all.

A key point about interference is that for two computation paths to destructively interfere with each other, they must lead to outcomes that are identical in *every respect*. To calculate the amplitude of a given outcome, you add up the amplitudes for all of the paths leading to that outcome; destructive interference is when the amplitudes cancel each other out.

3.2 Universal Set of Quantum Gates

Concretely, in a quantum computer we have n qubits, which we assume for simplicity start out all in the $|0\rangle$ state. Given these qubits, we apply a sequence of unitary transformation called "quantum gates." These gates form what's called a *quantum circuit*.

An example of such a circuit is shown below, where we apply the Hadamard to the first qubit, then do a CNOT with the second qubit acting as the control bit. Written out, the effect is $\left(\frac{|0\rangle+|1\rangle}{\sqrt{2}}\right)|0\rangle \xrightarrow{\text{CNOT}} \frac{|00\rangle+|11\rangle}{\sqrt{2}}$, the result being entangled qubits, as we discussed before. A crucial

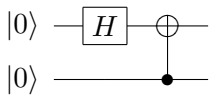


Figure 2: Entangling two qubits

point: each individual gate in a quantum circuit has to be extremely "simple", just like a classical circuit is built of AND, OR, NOT gates, the simplest imaginable building blocks. What does "simple" mean in the quantum case? Basically, that each quantum gate acts on at most (say) 2

or 3 qubits, and is the identity on all the other qubits. Why do we need to assume this? *Because physical interactions are local.*

To work with this constraint, we want a *universal set of quantum gates* that we can use to build more complex circuits, just like AND, OR, and NOT in classical computers. This universal set must contain 1-, 2-, and 3-qubit gates that can be combined to produce any unitary matrix.

We have to be careful when we say *any* unitary matrix, since there are uncountably infinitely many unitary matrices (you can rotate by any real-number angle, for instance). However, there are small sets of quantum gates that can be used to *approximate* any unitary matrix to arbitrary precision. As a technical note, the word "universal" has different meanings; for example, we usually call a set of gates universal if it can be used to approximate any unitary matrix *involving real numbers only*; this certainly suffices for quantum computation.

We've already seen the Hadamard and CNOT gates, but unfortunately these aren't sufficient to be a universal set of quantum gates. According to the Gottesman-Knill Theorem, any circuit constructed with just Hadamard and CNOT gates can be simulated efficiently with a classical computer. However, the Hadamard matrix paired with another gate called the **Toffoli gate** (also called controlled-controlled-NOT, or CCNOT) *is* sufficient to be used as a universal set of gates (for real-valued matrices).

The Toffoli gate will act similarly to the CNOT gate, except that we will control based on the first *two* qubits:

$$|x\rangle\,|y\rangle\,|z\rangle \to |x\rangle\,|y\rangle\,|z \oplus xy\rangle$$

where xy indicates the Boolean AND of x and y.

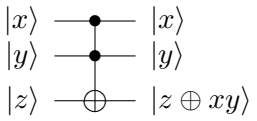


Figure 3: The Toffoli Gate diagram

Note, however, that these are not the only two gates whose combination allows for universal quantum computation. Another example of a universal pair of gates is the CNOT gate taken with the $\pi/8$ gate. We represent the $\pi/8$ gate using the following unitary:

$$T = \begin{bmatrix} \cos(\pi/8) & \sin(\pi/8) \\ -\sin(\pi/8) & \cos(\pi/8) \end{bmatrix}$$

But how many of these gates would be needed to approximate a random n-qubit unitary? Well, you remember Shannon's counting argument? What if we tried something similar in the quantum world? An n-qubit unitary has roughly $2^n \times 2^n$ degrees of freedom. On the other hand, the number of quantum circuits of size T is "merely" exponential in T. Hence, we need $T = \exp(n)$.

We, on the other hand, are only interested in the tiny subset of unitaries that can be built up out of a *polynomial* number of gates. Polynomial time is still our gold standard.

So, a quantum circuit has this polynomial number of gates, and then, at the end, *something* has to be measured. For simplicity, we assume a single qubit is measured. (Would it make a difference if there were intermediate measurements? No? Why not? Because we can simulate measurements using CNOTs.) Just like with BPP, we stipulate that if $x \in L$ (the answer is "yes"), then the

measurement outcome should be $|1\rangle$ with probability at least 2/3, while if $x \notin L$ (the answer is "no"), then the measurement outcome should be $|1\rangle$ with probability at most 1/3.

There's a final, technical requirement. We have to assume there's a classical polynomial-time algorithm to *produce* the quantum circuit in the first place. Otherwise, how do we find the circuit?

The class of all decision problems L that can be solved by such a family of quantum circuits is called BQP (Bounded-Error Quantum Polynomial Time).

4 Bounded-Error Quantum Polynomial Time (BQP)

Bounded-Error Quantum Polynomial Time (BQP) is, informally, the class of problems that can be efficiently solved by a quantum computer.

Incidentally: the idea of quantum computing occurred independently to a bunch of people in the 70s and 80s, but is usually credited to Richard Feynman and David Deutsch. BQP was defined by Bernstein and Vazirani in 1993.

4.1 Requirements for a BQP circuit

To be in BQP, a problem has to satisfy a few requirements:

Polynomial Size. How many of our building-block circuits (e.g., Hadamard and Toffoli) do we need to approximate an arbitrary n-qubit unitary? The answer is the quantum analogue to Shannon's counting argument. An n-qubit unitary has $2^n \times 2^n$ degrees of freedom, and there are doubly-exponentially many of them. On the other hand, the number of quantum circuits of size T is "merely" exponential in T. Hence, "almost all" unitaries will require an exponential number of quantum gates.

However, we are only interested in the small subset of unitaries that can be built using a *polynomial* number of gates. Polynomial time is still the gold standard.

Output. For simplicity, we assume that we measure a single qubit at the end of a quantum circuit. Just like with BPP, we stipulate that:

$$\text{Output} = \begin{cases} \text{if } x \in L: & |1\rangle \text{ with probability} \geq \frac{2}{3} \\ \text{if } x \notin L: & |1\rangle \text{ with probability} \leq \frac{1}{3} \end{cases}$$

Circuit Construction. There is a final technical requirement to constructing quantum circuits. We have to assume that there is a classical polynomial-time algorithm to *produce* the quantum circuit in the first place. Otherwise, how do we find the circuit?

4.2 BQP's Relation to Other Algorithm Families

P $\subseteq$ BQP: A quantum computer can always simulate a classical one (like using an airplane to drive down the highway). We can use the CNOT gate to simulate the NOT gate, and the Toffoli gate to simulate the AND gate.

BPP $\subseteq$ BQP: Loosely speaking, in quantum mechanics we "get randomness for free." More precisely, any time we need to make a random decision, all we need to do is apply a Hadamard

to some $|0\rangle$ qubit, putting it into an equal superposition of $|0\rangle$ and $|1\rangle$ states. Then we can CNOT that bit wherever we needed a random bit. We're not exploiting interference here; we're just using quantum mechanics as a source of random numbers.

BQP $\subseteq$ EXP: In exponential time, we can always write out a quantum state as an exponentially long vector of amplitudes, then explicitly calculate the effect of each gate in a quantum circuit.

BQP $\subseteq$ PSPACE: We can calculate the probability of each measurement outcome $|x\rangle$ by summing the amplitudes of all paths that lead to $|x\rangle$, which only takes polynomial space, as was shown by Bernstein and Vazirani. We won't give a detailed proof here.

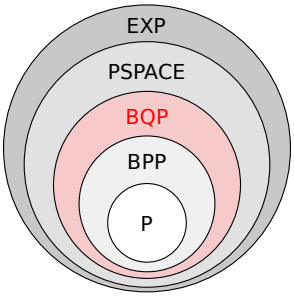


Figure 4: BQP inclusion diagram

We can draw a crucial consequence from this diagram, the first major contribution that complexity theory makes to quantum computing. Namely: in our present state of knowledge, there's little hope of proving unconditionally that quantum computers are more powerful than classical ones, since any proof of $\mathsf{P} \neq \mathsf{BQP}$ would also imply $\mathsf{P} \neq \mathsf{PSPACE}$.

5 Next Time: Quantum Algorithms

Next class we'll see some examples of quantum algorithms that actually outperform their classical counterparts:

- The Deutsch-Jozsa Algorithm
- Simon's Algorithm
- Shor's Algorithm

6.080/6.089 GITCS May 13, 2008

Lecture 24

Lecturer: Scott Aaronson *Scribe: Chris Granade*

1 Quantum Algorithms

Of course the real question is: can quantum computers actually do something more efficiently than classical computers? In this lecture, we'll see why the modern consensus is that they can.

1.1 Computing the XOR of Two Bits

We'll first see an algorithm due to Deutsch and Jozsa. Even though this algorithm is trivial by modern standards, it gave the first example where a quantum algorithm could provably solve a problem using fewer resources than a classical algorithm.

Suppose we're given access to a Boolean function $f : \{0,1\} \rightarrow \{0,1\}$. And suupose we want to compute $f(0) \oplus f(1)$, the XOR of $f(0)$ and $f(1)$. Classically, how many times would we need to evaluate f? It's clear that the answer is twice: knowing only $f(0)$ or $f(1)$ tells us exactly nothing about their XOR.

So what about in the quantum case? Well, first we need to say what it even *means* to evaluate f. Since this is a quantum algorithm we're talking about, we should be able to evaluate both inputs, $f(0)$ and $f(1)$ in quantum superposition. But we have to do so in a reversible way. For example, we can't map the state $|x, b\rangle$ to $|x, f(x)\rangle$ (overwriting b), since that wouldn't be unitary.

The standard solution is that querying f means applying a unitary transformation that maps $|x, y\rangle \rightarrow |x, y \oplus f(x)\rangle$. Is it reversible? Yeah. Applying it twice gets you back to where you started. I claim we can compute $f(0) \oplus f(1)$ using just a single one of these operations. How?

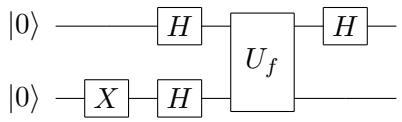


Figure 1: Finding $f(0) \oplus f(1)$ in one query.

In the circuit above, the effect of the gates before U_f is to prepare an initial state $|\psi_0\rangle$:

$$|\psi_0\rangle = |+\rangle\,|-\rangle = \frac{1}{2}\left[|0\rangle + |1\rangle\right]\left[|0\rangle - |1\rangle\right]$$

If you think of the effect of U_f on the first qubit in this state, it's just to negate the amplitude if $f(0) \neq f(1)$! Thus, U_f produces $|+\rangle\,|-\rangle$ if $f(0) = f(1)$ and $|-\rangle\,|-\rangle$ otherwise. The final Hadamard gate transforms the first qubit back into the computational basis, so that we measure 1 if and only if $f(0) \neq f(1)$.

In particular, this means that if you want to compute the XOR of N bits with a quantum computer, you can do so using $N/2$ queries, as follows: first divide the bits into $N/2$ pairs of bits,

then run the above algorithm on each pair, and finally output the XOR of the results. Of course, this is only a constant-factor speedup, but it's a harbinger of much more impressive speedups to come.

1.2 Simon's Algorithm

Say you're given a Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}^n$. You're promised there exists a "secret string" s such that $f(x) = f(y)$ if and only if $y = x \oplus s$, where $\oplus$ denotes a sum mod 2. The problem is to find s by querying f as few times as possible.

How many queries would a classical randomized algorithm need to solve this problem? Something very similar was on your problem set! Right, $2^{n/2}$. This is basically just the birthday paradox. Until it happens to find an x, y pair such that $f(x) = f(y)$, your algorithm is basically just "shooting in the dark"; it has essentially no information about s. And after T queries, the probability of having found an x, y pair such that $f(x) = f(y)$ is at most $T^2/(2^n - 1)$ (why?).

On the other hand, in 1993 Daniel Simon gave a quantum algorithm that solves this problem in polynomial time, in fact using only $O(n)$ queries. This was the first example of a problem that a quantum computer can solve exponentially faster than a classical one. Admittedly, it's a contrived example (and probably for that reason, Simon's paper was originally rejected!). But it's good to see for two reasons: first, it led directly to Shor's factoring algorithm. And second, the easiest way to *understand* Shor's algorithm is to understand Simon's algorithm, and then see Shor's algorithm as the same thing with a different underlying group!

Before proceeding further, though, there's one thing I want to clear up. I said that Simon's problem was the first known example where quantum computers provably give an exponential speedup over classical computers. How is that consistent with what I said before, that we can't prove $\mathsf{P} \neq \mathsf{BQP}$ unconditionally?

Right, Simon's problem involves the function f as a "black-box." In the black-box setting, we *can* prove unconditionally that quantum computers give an exponential speedup over classical ones.

1.3 RSA

Alright, so let's say you want to break the RSA cryptosystem, in order to rob some banks, read your ex's email, whatever. We all know that breaking RSA reduces to finding the prime factors of a large integer N. Unfortunately, we also know that "trying all possible divisors in parallel," and then instantly picking the right one, isn't going to work. Hundreds of popular magazine articles notwithstanding, trying everything in parallel just isn't the sort of thing that a quantum computer can do. Sure, in some sense you can "try all possible divisors" – but if you then measure the outcome, you'll get a random potential divisor, which almost certainly won't be the one you want.

What this means is that, if we want a fast quantum factoring algorithm, we're going to have to exploit some *structure* in the factoring problem: in other words, some mathematical property of factoring that it *doesn't* share with just a generic problem of finding a needle in a haystack.

Fortunately, the factoring problem has oodles of special properties. What are some examples we discussed in class? Right: if I give you a positive integer, you might not know its prime factorization, but you do know that it has exactly *one* factorization! By contrast, if I gave you (say) a Sudoku puzzle and asked you to solve it, *a priori* you'd have no way of knowing whether it had exactly one solution, 200 million solutions, or no solutions at all. Of course, knowing that there's exactly one needle in a haystack is still not much help in finding the needle! But this uniqueness is a hint that

the factoring problem might have *other* nice mathematical properties lying around for the picking. As it turns out, it does.

The property we'll exploit is the reducibility of factoring to another problem, called period-finding. OK, time for a brief number theory digression. Let's look at the powers of 2 mod 15:

$$2,\ 4,\ 8,\ 1,\ 2,\ 4,\ 8,\ 1,\ 2,\ 4,\ \ldots$$

As you can see, taking the powers of 2 mod 15 gives us a *periodic sequence*, whose period (i.e., how far you have to go before it starts repeating) is 4. For another example, let's look at the powers of 2 mod 21:

$$2,\ 4,\ 8,\ 16,\ 11,\ 1,\ 2,\ 4,\ 8,\ 16,\ \ldots$$

This time we get a periodic sequence whose period is 6.

What's a general rule that governs what the period will be? We discussed this earlier, when we were talking about the RSA cryptosystem! The beautiful pattern, discovered by Euler in the 1760s, is this. Let N be a product of two prime numbers, p and q, and consider the sequence:

$$x \bmod N,\ x^2 \bmod N,\ x^3 \bmod N,\ x^4 \bmod N,\ \ldots$$

Then, provided that x is not divisible by p or q, the above sequence will repeat with some period that divides $(p-1)(q-1)$. So, for example, if $N = 15$, then the prime factors of N are $p = 3$ and $q = 5$, so $(p-1)(q-1) = 8$. And indeed, the period of the sequence is 4, which divides 8. If $N = 21$, then $p = 3$ and $q = 7$, so $(p-1)(q-1) = 12$. And indeed, the period is 6, which divides 12.

Now, I want you to step back and think about what this means. It means that *if* we can find the period of the sequence of powers of $x \bmod N$, *then* we can learn something about the prime factors of N. In particular, we can learn a divisor of $(p-1)(q-1)$. Now, I'll admit that's not as good as learning p and q themselves, but grant me that it's something. Indeed, it's more than something: it turns out that if we could learn several random divisors of $(p-1)(q-1)$ (for example, by trying different random values of x), then with high probability we could put those divisors together to learn $(p-1)(q-1)$ itself. And once we knew $(p-1)(q-1)$, we could then use some more little tricks to recover p and q, the prime factors we wanted. (This is again in your problem set.)

So what's the fly in the ointment? Well, even though the sequence of powers mod$N$ will *eventually* start repeating itself, the number of steps before it repeats could be almost as large as N itself – and N might have hundreds or thousands of digits! This is why finding the period doesn't seem to lead to a fast *classical* factoring algorithm.

Aha, but we have a quantum computer! (Or at least, we're *imagining* that we do.) So maybe there's still hope. In particular, suppose we could create an enormous quantum superposition over all the numbers in our sequence:

$$\sum_r |r\rangle\,|x^r \bmod N\rangle$$

Then maybe there's some quantum operation we could perform on that superposition that would reveal the period.

The key point is that we're no longer trying to find a needle in an exponentially-large haystack, something we *know* is hard even for a quantum computer. Instead, we're now trying to find the

period of a sequence, which is a *global* property of all the numbers in the sequence taken together. And that makes a big difference.

Look: if you think about quantum computing in terms of "parallel universes" (and whether you do or don't is up to you), there's no feasible way to detect a *single* universe that's different from all the rest. Such a lone voice in the wilderness would be drowned out by the vast number of suburb-dwelling, Dockers-wearing conformist universes. What one can hope to detect, however, is a joint property of *all* the parallel universes together – a property that can only be revealed by a computation to which all the universes contribute [1].

So, the task before us is not hopeless! But if we want to get this period-finding idea to work, we'll have to answer two questions:

1. Using a quantum computer, can we quickly create a superposition over $x \bmod N$, $x^2 \bmod N$, $x^3 \bmod N$, ...?

2. Supposing we did create such a superposition, how would we figure out the period?

Let's tackle the first question first. We can certainly create a superposition over all integers r, from 1 up to N^2 or so. The trouble is, given an r, how do we quickly compute $x^r \bmod N$? We've already seen the answer: repeated squaring!

OK, so we can efficiently create a quantum superposition over all pairs of integers of the form $(r, x^r mod N)$, where r ranges from 1 up to N or so. But then, given a superposition over all the elements of a periodic sequence, how do we extract the period of the sequence?

Well, we've finally come to the heart of the matter – the one part of Shor's quantum algorithm that actually depends on quantum mechanics. To get the period out, Shor uses something called the *quantum Fourier transform*, or QFT. My challenge is, how can I explain the QFT to you without going through the math? Hmmmm...

OK, let me try this. Like many computer scientists, I keep extremely odd hours. You know that famous experiment where they stick people for weeks in a sealed room without clocks or sunlight, and the people gradually shift from a 24-hour day to a 25- or 26- or 28-hour day? Well, that's just ordinary life for me. One day I'll wake up at 9am, the next day at 11am, the day after that at 1pm, etc. Indeed, I'll happily 'loop all the way around' if no classes or appointments intervene.

Now, here's my question: let's say I tell you that I woke up at 5pm this afternoon. From that fact alone, what can you conclude about how long my "day" is: whether I'm on a 25-hour schedule, or a 26.3-hour schedule, or whatever?

The answer, of course, is not much! I mean, it's a pretty safe bet that I'm not on a 24-hour schedule, since otherwise I'd be waking up in the morning, not 5pm. But almost any other schedule – 25 hours, 26 hours, 28 hours, etc. – will necessarily cause me to "loop all around the clock," so that it'd be no surprise to see me get up at 5pm on some particular afternoon.

Now, though, I want you to imagine that my bedroom wall is covered with analog clocks. These are very strange clocks: one of them makes a full revolution every 17 hours, one of them every 26 hours, one of them every 24.7 hours, and so on for just about every number of hours you can imagine. (For simplicity, each clock has only an hour hand, no minute hand.) I also want you to imagine that beneath each clock is a posterboard with a thumbtack in it. When I first moved into my apartment, each thumbtack was in the middle of its respective board. But now, whenever I

[1] For safety reasons, please don't explain the above to popular writers of the "quantum computing = exponential parallelism" school. They might shrivel up like vampires exposed to sunlight.

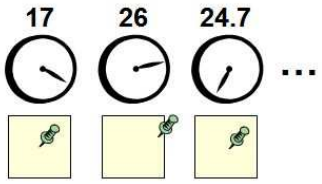


Figure 2: A possible configuration of clocks and pegboards.

wake up in the "morning," the first thing I do is to go around my room, and *move each thumbtack exactly one inch in the direction that the clock hand above it is pointing.*

Now, here's my new question: *by examining the thumbtacks in my room, is it possible to figure out what sort of schedule I'm keeping?*

I claim that it *is* possible. As an example, suppose I was keeping a 26-hour day. Then what would happen to the thumbtack below the 24-hour clock? It's not hard to see that it would undergo periodic motion: sure, it would drift around a bit, but after every 12 days it would return to the middle of the board where it had started. One morning I'd move the thumbtack an inch in this direction, another morning an inch in that, but eventually all these movements in different directions would cancel each other out.

On the other hand – again supposing I was keeping a 26-hour day – what would happen to the thumback below the 26-hour clock? Here the answer is different. For as far as the 26-hour clock is concerned, I've been waking up at exactly the same time each "morning"! Every time I wake up, the 26-hour clock is pointing the same direction as it was the last time I woke up. So I'll keep moving the thumbtack one more inch in the same direction, until it's not even on the posterboard at all!

It follows, then, that just by seeing which thumbtack traveled the farthest from its starting point, you could figure out what sort of schedule I was on. In other words, you could infer the "period" of the periodic sequence that is my life.

And that, basically, is the quantum Fourier transform. Well, a little more precisely, the QFT is a *linear transformation* (indeed a unitary transformation) that maps one vector of complex numbers to another vector of complex numbers. The input vector has a nonzero entry corresponding to every time when I wake up, and zero entries everywhere else. The output vector records the positions of the thumbtacks on the posterboards (which one can think of as points on the complex plane). So what we get, in the end, is a linear transformation that maps a quantum state encoding a periodic sequence, to a quantum state encoding the *period* of that sequence.

Another way to think about this is in terms of *interference*. I mean, the key point about quantum mechanics – the thing that makes it different from classical probability theory – is that, whereas probabilities are always non-negative, *amplitudes* in quantum mechanics can be positive, negative, or even complex. And because of this, the amplitudes corresponding to different ways of getting a particular answer can "interfere destructively" and cancel each other out.

And that's exactly what's going on in Shor's algorithm. Every "parallel universe" corresponding to an element of the sequence contributes *some* amplitude to every "parallel universe" corresponding to a possible period of the sequence. The catch is that, for all periods other than the "true" one, these contributions point in different directions and therefore cancel each other out. Only for the "true" period do the contributions from different universes all point in the *same* direction. And

that's why, when we measure at the end, we'll find the true period with high probability.

Questions for next time:

1. Can QCs be built?
2. What are the limits of QCs?
3. Anything beyond QCs?